

C Programming

CONTENTS

INTRODUCTION	1
WELCOME TO C	2
<i>Target Audience</i>	2
<i>Expected Knowledge</i>	2
<i>Advantageous Knowledge</i>	2
COURSE OBJECTIVES	3
PRACTICAL EXERCISES	4
FEATURES OF C	5
<i>High Level Assembler</i>	5
<i>(Processor) Speed Comes First!</i>	5
<i>Systems Programming</i>	5
<i>Portability</i>	5
<i>Write Only Reputation</i>	5
THE HISTORY OF C	6
<i>Brian Kernighan, Dennis Ritchie</i>	6
<i>Standardization</i>	7
<i>ANSI</i>	7
<i>ISO</i>	7
STANDARD C VS. K&R C	8
A C PROGRAM	9
<i>#include</i>	9
<i>Comments</i>	9
<i>main</i>	9
<i>Braces</i>	9
<i>printf</i>	9
<i>\n</i>	9
<i>return</i>	9
THE FORMAT OF C	10
<i>Semicolons</i>	10
<i>Free Format</i>	10
<i>Case Sensitivity</i>	10
<i>Random Behavior</i>	10

ANOTHER EXAMPLE	11
<i>int</i>	11
<i>scanf</i>	11
<i>printf</i>	11
<i>Expressions</i>	11
VARIABLES	12
<i>Declaring Variables</i>	12
<i>Valid Names</i>	12
<i>Capital Letters</i>	12
PRINTF AND SCANF	13
<i>printf</i>	13
<i>scanf</i>	13
&.....	13
INTEGER TYPES IN C.....	14
<i>limits.h</i>	14
<i>Different Integers</i>	14
<i>unsigned</i>	14
<i>%hi</i>	14
INTEGER EXAMPLE.....	15
<i>INT_MIN, INT_MAX</i>	15
CHARACTER EXAMPLE.....	16
<i>char</i>	16
<i>CHAR_MIN, CHAR_MAX</i>	16
<i>Arithmetic With char</i>	16
<i>%c</i> vs <i>%i</i>	16
INTEGERS WITH DIFFERENT BASES	17
<i>Decimal, Octal and Hexadecimal</i>	17
<i>%d</i>	17
<i>%o</i>	17
<i>%x</i>	17
<i>%X</i>	17
REAL TYPES IN C	18
<i>float.h</i>	18
<i>float</i>	18
<i>double</i>	18
<i>long double</i>	18
REAL EXAMPLE.....	19
<i>%lf</i>	19
<i>%le</i>	19
<i>%lg</i>	19
<i>%7.2lf</i>	19
<i>%.2le</i>	19
<i>%.4lg</i>	19
CONSTANTS	20
<i>Typed Constants</i>	20
WARNING!	21
NAMED CONSTANTS	22
<i>const</i>	22
<i>Lvalues and Rvalues</i>	22
PREPROCESSOR CONSTANTS	23
TAKE CARE WITH PRINTF AND SCANF!.....	24
<i>Incorrect Format Specifiers</i>	24

INTRODUCTION PRACTICAL EXERCISES.....	27
INTRODUCTION SOLUTIONS.....	29
OPERATORS IN C.....	33
OPERATORS IN C	34
ARITHMETIC OPERATORS	35
+, -, *, /	35
%.....	35
USING ARITHMETIC OPERATORS	36
THE CAST OPERATOR	37
INCREMENT AND DECREMENT.....	38
PREFIX AND POSTFIX	39
<i>Prefix ++, --</i>	39
<i>Postfix ++, --</i>	39
<i>Registers</i>	39
TRUTH IN C.....	40
<i>True</i>	40
<i>False</i>	40
<i>Testing Truth</i>	40
COMPARISON OPERATORS.....	41
LOGICAL OPERATORS	42
<i>And, Or, Not</i>	42
LOGICAL OPERATOR GUARANTEES.....	43
<i>C Guarantees</i>	43
<i>and Truth Table</i>	43
<i>or Truth Table</i>	43
WARNING!	44
<i>Parentheses</i>	44
BITWISE OPERATORS	45
& vs &&.....	45
vs 	45
^	45
<i>Truth Tables For Bitwise Operators</i>	45
BITWISE EXAMPLE.....	46
<i>Arithmetic Results of Shifting</i>	46
Use <i>unsigned</i> When Shifting Right.....	46
ASSIGNMENT	47
<i>Assignment Uses Registers</i>	47
WARNING!	48
<i>Test for Equality vs. Assignment</i>	48
OTHER ASSIGNMENT OPERATORS	49
+=, -=, *=, /=, %= etc.....	49
SIZEOF OPERATOR	50
CONDITIONAL EXPRESSION OPERATOR.....	51
<i>Conditional expression vs. if/then/else</i>	51
PRECEDENCE OF OPERATORS	52
ASSOCIATIVITY OF OPERATORS.....	53
<i>Left to Right Associativity</i>	53
<i>Right to Left Associativity</i>	53
PRECEDENCE/ASSOCIATIVITY TABLE	54
OPERATORS IN C PRACTICAL EXERCISES.....	57

OPERATORS IN C SOLUTIONS.....	59
CONTROL FLOW	63
CONTROL FLOW	64
DECISIONS IF THEN	65
WARNING!	66
<i>Avoid Spurious Semicolons After if.....</i>	66
IF THEN ELSE	67
NESTING IFS	68
<i>Where Does else Belong?</i>	68
SWITCH	69
<i>switch vs. if/then/else.....</i>	69
MORE ABOUT SWITCH	70
<i>switch Less Flexible Than if/then/else.....</i>	70
A SWITCH EXAMPLE.....	71
<i>Twelve Days of Christmas.....</i>	71
WHILE LOOP	72
(ANOTHER) SEMICOLON WARNING!.....	73
<i>Avoid Semicolons After while</i>	73
<i>Flushing Input</i>	73
WHILE , NOT UNTIL!	74
<i>There Are Only “While” Conditions in C.....</i>	74
DO WHILE	75
FOR LOOP.....	76
<i>for And while Compared</i>	76
FOR IS NOT UNTIL EITHER!	77
<i>C Has While Conditions, Not Until Conditions</i>	77
STEPPING WITH FOR	78
<i>math.h.....</i>	78
EXTENDING THE FOR LOOP.....	79
<i>Infinite Loops</i>	79
BREAK	80
<i>break is Really Goto!</i>	80
<i>break, switch and Loops.....</i>	80
CONTINUE	81
<i>continue is Really Goto.....</i>	81
<i>continue, switch and Loops</i>	81
SUMMARY.....	82
CONTROL FLOW PRACTICAL EXERCISES.....	83
CONTROL FLOW SOLUTIONS	87
FUNCTIONS	95
FUNCTIONS	96
THE RULES	97
WRITING A FUNCTION - EXAMPLE	98
<i>Return Type.....</i>	98
<i>Function Name</i>	98
<i>Parameters</i>	98
<i>Return Value.....</i>	98

CALLING A FUNCTION - EXAMPLE	99
<i>Prototype</i>	99
<i>Call</i>	99
<i>Ignoring the Return</i>	99
CALLING A FUNCTION - DISASTER!	100
<i>Missing Prototypes</i>	100
PROTOTYPES.....	101
<i>When a Prototype is Missing</i>	101
PROTOTYPING IS NOT OPTIONAL.....	102
<i>Calling Standard Library Functions</i>	102
WRITING PROTOTYPES	103
<i>Convert The Function Header Into The Prototype</i>	103
<i>Added Documentation</i>	103
TAKE CARE WITH SEMICOLONS	104
<i>Avoid Semicolons After The Function Header</i>	104
EXAMPLE PROTOTYPES.....	105
EXAMPLE CALLS	106
RULES OF VISIBILITY	107
<i>C is a Block Structured Language</i>	107
CALL BY VALUE.....	108
CALL BY VALUE - EXAMPLE	109
C AND THE STACK.....	110
STACK EXAMPLE	111
STORAGE	112
<i>Code Segment</i>	112
<i>Stack</i>	112
<i>Data Segment</i>	112
<i>Heap</i>	112
AUTO	113
<i>Stack Variables are “Automatic”</i>	113
<i>Stack Variables are Initially Random</i>	113
<i>Performance</i>	113
STATIC	114
<i>static Variables are Permanent</i>	114
<i>static Variables are Initialized</i>	114
<i>static Variables Have Local Scope</i>	114
REGISTER.....	115
<i>register Variables are Initially Random</i>	115
<i>Slowing Code Down</i>	115
GLOBAL VARIABLES.....	116
<i>Global Variables are Initialized</i>	116
FUNCTIONS PRACTICAL EXERCISES	119
FUNCTIONS SOLUTIONS.....	121
POINTERS.....	127
POINTERS.....	128
POINTERS - WHY?	129
DECLARING POINTERS	130
EXAMPLE POINTER DECLARATIONS	131
<i>Pointers Have Different Types</i>	131
<i>Positioning the “*”</i>	131

THE “&” OPERATOR	132
<i>Pointers Are Really Just Numbers.....</i>	<i>132</i>
<i>Printing Pointers</i>	<i>132</i>
RULES.....	133
<i>Assigning Addresses</i>	<i>133</i>
THE “*” OPERATOR	134
WRITING DOWN POINTERS	135
INITIALIZATION WARNING!.....	136
<i>Always Initialize Pointers</i>	<i>136</i>
INITIALIZE POINTERS!.....	137
<i>Understanding Initialization</i>	<i>137</i>
NULL	138
<i>NULL and Zero.....</i>	<i>138</i>
A WORLD OF DIFFERENCE!.....	139
<i>What is Pointed to vs the Pointer Itself</i>	<i>139</i>
FILL IN THE GAPS	140
TYPE MISMATCH.....	141
CALL BY VALUE - REMINDER.....	142
CALL BY REFERENCE	143
POINTERS TO POINTERS	144
POINTERS PRACTICAL EXERCISES	147
POINTERS SOLUTIONS.....	151
ARRAYS IN C.....	155
ARRAYS IN C.....	156
DECLARING ARRAYS	157
EXAMPLES	158
<i>Initializing Arrays.....</i>	<i>158</i>
ACCESSING ELEMENTS	159
<i>Numbering Starts at Zero.....</i>	<i>159</i>
ARRAY NAMES.....	160
<i>A Pointer to the Start.....</i>	<i>160</i>
<i>Cannot Assign to an Array.....</i>	<i>160</i>
PASSING ARRAYS TO FUNCTIONS	161
<i>Bounds Checking Within Functions</i>	<i>161</i>
EXAMPLE.....	162
<i>A Pointer is Passed.....</i>	<i>162</i>
<i>Bounds Checking.....</i>	<i>162</i>
USING POINTERS	163
<i>Addition With Pointers.....</i>	<i>163</i>
POINTERS GO BACKWARDS TOO	164
<i>Subtraction From Pointers.....</i>	<i>164</i>
POINTERS MAY BE SUBTRACTED.....	165
USING POINTERS - EXAMPLE.....	166
* AND ++.....	167
<i>In “*p++” Which Operator is Done First?.....</i>	<i>167</i>
<i>(*p)++</i>	<i>167</i>
<i>*++p.....</i>	<i>167</i>
WHICH NOTATION?.....	168
<i>Use What is Easiest!</i>	<i>168</i>
STRINGS	169

Character Arrays vs. Strings.....	170
Null Added Automatically.....	170
Excluding Null.....	170
PRINTING STRINGS	171
printf “%s” Format Specifier.....	171
NULL REALLY DOES MARK THE END!	172
ASSIGNING TO STRINGS	173
POINTING TO STRINGS	174
Strings May be Stored in the Data Segment	174
MULTIDIMENSIONAL ARRAYS	177
REVIEW	178
ARRAYS PRACTICAL EXERCISES.....	181
ARRAYS SOLUTIONS	185
STRUCTURES IN C.....	197
STRUCTURES IN C.....	198
CONCEPTS	199
SETTING UP THE TEMPLATE	200
Structures vs. Arrays.....	200
CREATING INSTANCES	201
Instance?.....	201
INITIALIZING INSTANCES	202
STRUCTURES WITHIN STRUCTURES	203
Reminder - Avoid Leading Zeros.....	203
ACCESSING MEMBERS	204
Accessing Members Which are Arrays.....	204
Accessing Members Which are Structures.....	204
UNUSUAL PROPERTIES	205
Common Features Between Arrays and Structures.....	205
Differences Between Arrays and Structures	205
INSTANCES MAY BE ASSIGNED.....	206
Cannot Assign Arrays.....	206
Can Assign Structures Containing Arrays.....	206
PASSING INSTANCES TO FUNCTIONS	207
Pass by Value or Pass by Reference?.....	207
POINTERS TO STRUCTURES	208
WHY (*p) . NAME?.....	209
A New Operator.....	209
USING p->NAME	210
PASS BY REFERENCE - WARNING.....	211
const to the Rescue!.....	211
RETURNING STRUCTURE INSTANCES	212
LINKED LISTS.....	213
A Recursive Template?	213
EXAMPLE.....	214
Creating a List.....	214
PRINTING THE LIST.....	215
SUMMARY.....	217
STRUCTURES PRACTICAL EXERCISES.....	219

STRUCTURES SOLUTIONS.....	223
READING C DECLARATIONS.....	233
READING C DECLARATIONS.....	234
INTRODUCTION	235
SOAC	236
TYPEDEF.....	245
SUMMARY.....	252
READING C DECLARATIONS PRACTICAL EXERCISES	253
READING C DECLARATIONS SOLUTIONS.....	255
HANDLING FILES IN C.....	261
HANDLING FILES IN C.....	262
INTRODUCTION	263
<i>The Standard Library.....</i>	<i>263</i>
STREAMS	264
stdin, stdout and stderr.....	264
WHAT IS A STREAM?.....	265
<i>Fast Programs Deal with Slow Hardware</i>	<i>265</i>
<i>Caches and Streams.....</i>	<i>265</i>
WHY STDOUT AND STDERR ?.....	266
STDIN IS LINE BUFFERED.....	268
<i>Signaling End of File.....</i>	<i>268</i>
int not char	268
OPENING FILES.....	269
<i>The Stream Type.....</i>	<i>269</i>
DEALING WITH ERRORS	270
<i>What Went Wrong?.....</i>	<i>270</i>
FILE ACCESS PROBLEM.....	271
DISPLAYING A FILE.....	272
<i>Reading the Pathname but Avoiding Overflow</i>	<i>272</i>
<i>The Program's Return Code</i>	<i>272</i>
EXAMPLE - COPYING FILES	273
<i>Reading and Writing Files</i>	<i>273</i>
<i>Closing files.....</i>	<i>273</i>
<i>Transferring the data.....</i>	<i>273</i>
<i>Blissful Ignorance of Hidden Buffers</i>	<i>274</i>
<i>Cleaning up</i>	<i>274</i>
<i>Program's Return Code</i>	<i>274</i>
CONVENIENCE PROBLEM	275
<i>Typing Pathnames</i>	<i>275</i>
<i>No Command Line Interface</i>	<i>275</i>
ACCESSING THE COMMAND LINE	276
argc	276
argv	276
USEFUL ROUTINES.....	278
fscanf	278
fgets	278
fprintf	279
fputs	279

<i>fgets</i> Stop Conditions	280
BINARY FILES	281
<i>fopen</i> “wb”	282
The Control Z Problem	282
The Newline Problem.....	283
The Movement Problem	284
Moving Around Files	284
<i>fsetpos</i> vs. <i>fseek</i>	284
SUMMARY	286
HANDLING FILES IN C PRACTICAL EXERCISES.....	287
HANDLING FILES IN C SOLUTIONS	289
MISCELLANEOUS THINGS	301
MISCELLANEOUS THINGS.....	302
UNIONS.....	303
Size of <i>struct</i> vs. Size of <i>union</i>	303
REMEMBERING	304
A Member to Record the Type.....	304
ENUMERATED TYPES	306
USING DIFFERENT CONSTANTS	307
Printing <i>enums</i>	307
THE PREPROCESSOR	308
INCLUDING FILES	309
PATHNAMES.....	310
Finding <i>#include</i> Files	310
PREPROCESSOR CONSTANTS	311
<i>#if</i>	311
<i>#endif</i>	311
<i>#define</i>	311
<i>#undef</i>	311
AVOID TEMPTATION!	312
PREPROCESSOR MACROS	313
A DEBUGGING AID.....	315
WORKING WITH LARGE PROJECTS	316
DATA SHARING EXAMPLE.....	317
Functions are Global and Sharable	317
DATA HIDING EXAMPLE	318
<i>static</i> Before Globals.....	318
Errors at Link Time	318
DISASTER!	319
Inconsistencies Between Modules	319
USE HEADER FILES	320
GETTING IT RIGHT.....	321
Place Externs in the Header	321
BE AS LAZY AS POSSIBLE	322
SUMMARY	323
MISCELLANEOUS THINGS PRACTICAL EXERCISES	325
MISCELLANEOUS THINGS SOLUTIONS	327

C AND THE HEAP	329
C AND THE HEAP	330
WHAT IS THE HEAP?	331
<i>The Parts of an Executing Program</i>	331
<i>Stack</i>	332
<i>Heap and Stack “in Opposition”</i>	332
HOW MUCH MEMORY?	333
<i>Simple Operating Systems</i>	333
<i>Advanced Operating Systems</i>	333
<i>Future Operating Systems</i>	333
DYNAMIC ARRAYS	334
USING DYNAMIC ARRAYS	335
<i>One Pointer per Dynamic Array</i>	335
<i>Calculating the Storage Requirement</i>	335
USING DYNAMIC ARRAYS (CONTINUED)	336
<i>Insufficient Storage</i>	336
<i>Changing the Array Size</i>	336
<i>When realloc Succeeds</i>	336
<i>Maintain as Few Pointers as Possible</i>	337
<i>Requests Potentially Ignored</i>	337
<i>Releasing the Storage</i>	337
CALLOC/MALLOC EXAMPLE	338
REALLOC EXAMPLE	339
REALLOC CAN DO IT ALL	340
<i>realloc can Replace malloc</i>	340
<i>realloc can Replace free</i>	340
ALLOCATING ARRAYS OF ARRAYS	341
<i>Pointers Access Fine with Dynamic Arrays</i>	341
<i>Pointers to Pointers are not Good with Arrays of Arrays</i>	342
<i>Use Pointers to Arrays</i>	342
DYNAMIC DATA STRUCTURES	343
LINKING THE LIST	344
SUMMARY	345
C AND THE HEAP PRACTICAL EXERCISES.....	347
C AND THE HEAP SOLUTIONS.....	349
APPENDICES	353
PRECEDENCE AND ASSOCIATIVITY OF C OPERATORS:	354
SUMMARY OF C DATA TYPES.....	355
MAXIMA AND MINIMA FOR C TYPES	356
PRINTF FORMAT SPECIFIERS.....	357
TABLE OF ESCAPE SEQUENCES	358
ASCII TABLE	359
BIBLIOGRAPHY	361
<i>The C Puzzle Book</i>	361
<i>The C Programming Language 2nd edition</i>	361
<i>The C Standard Library</i>	361
<i>C Traps and Pitfalls</i>	361

Introduction

C Programming

Welcome to C

Target Audience

This course is intended for people with previous programming experience with another programming language. It does not matter what the programming language is (or was). It could be a high level language like Pascal, FORTRAN, BASIC, COBOL, etc. Alternatively it could be an assembler, 6502 assembler, Z80 assembler etc.

Expected Knowledge

You are expected to understand the basics of programming:

- What a variable is
- The difference between a variable and a constant
- The idea of a decision ("*if* it is raining, *then* I need an umbrella, *else* I need sunblock")
- The concept of a loop

Advantageous Knowledge

It would be an advantage to understand:

- Arrays, data structures which contain a number of slots of the same type. For example an array of 100 exam marks, 1 each for 100 students.
- Records, data structures which contain a number of slots of different types. For example a patient in database maintained by a local surgery.

It is not a problem if you do not understand these last two concepts since they are covered in the course.

Course Objectives

- ☞ **Be able to read and write C programs**
- ☞ **Understand all C language constructs**
- ☞ **Be able to use pointers**
- ☞ **Have a good overview of the Standard Library**
- ☞ **Be aware of some of C's traps and pitfalls**

Course Objectives

Obviously in order to be a competent C programmer you must be able to write C programs. There are many examples throughout the notes and there are practical exercises for you to complete.

The course discusses **all** of the C language constructs. Since C is such a small language there aren't that many of them. There will be no dark or hidden corners of the language left after you have completed the course.

Being able to use pointers is something that is absolutely essential for a C programmer. You may not know what a pointer is now, but you will by the end of the course.

Having an understanding of the Standard Library is also important to a C programmer. The Standard Library is a toolkit of routines which if weren't provided, you'd have to invent. In order to use what is provided you need to know its there - why spend a day inventing a screwdriver if there is one already in your toolkit.

Practical Exercises

- ⌘ **Practical exercises are a very important part of the course**
- ⌘ **An opportunity to experience some of the traps first hand!**
- ⌘ **Solutions are provided, discuss these amongst yourselves and/or with the tutor**
- ⌘ **If you get stuck, ask**
- ⌘ **If you can't understand one of the solutions, ask**
- ⌘ **If you have an alternative solution, say**

Practical Exercises

Writing C is Important!

There are a large number of practical exercises associated with this course. This is because, as will become apparent, there are things that can go wrong when you write code. The exercises provide you with an opportunity to “go wrong”. By making mistakes first hand (and with an instructor never too far away) you can avoid these mistakes in the future.

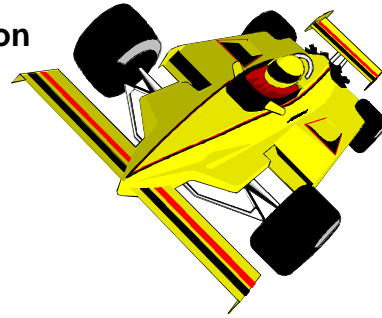
Solutions to the practical exercises are provided for you to refer to. It is not considered “cheating” for you to use these solutions. They are provided for a number of reasons:

- You may just be stuck and need a “kick start”. The first few lines of a solution may give you the start you need.
- The solution may be radically different to your own, exposing you to alternative coding styles and strategies.

You may think your own solution is better than the one provided. Occasionally the solutions use one line of code where three would be clearer. This doesn't make the one line “better”, it just shows you how it *can* be done.

Features of C

- § C can be thought of as a “high level assembler”
- § Designed for maximum processor speed
- § Safety a definite second!
- § THE system programming language
- § (Reasonably) portable
- § Has a “write only” reputation



Features of C

High Level Assembler

Programmers coming to C from high level languages like Pascal, BASIC etc. are usually surprised by how “low level” C is. It does very little for you, if you want it done, it expects you to write the code yourself. C is really little more than an assembler with a few high level features. You will see this as we progress through the course.

(Processor) Speed Comes First!

The reason C exists is to be fast! The execution speed of your program is everything to C. Note that this does **not** mean the development speed is high. In fact, almost the opposite is true. In order to run your program as quickly as possible C throws away all the features that make your program “safe”. C is often described as a “racing car without seat belts”. Built for ultimate speed, people are badly hurt if there is a crash.

Systems Programming

C is the systems programming language to use. Everything uses it, UNIX, Windows 3.1, Windows 95, NT. Very often it is the first language to be supported. When Microsoft first invented Windows years back, they produced a C interface with a promise of a COBOL interface to follow. They did so much work on the C interface that we’re still waiting for the COBOL version.

Portability

One thing you are probably aware of is that assembler is *not* portable. Although a Pascal program will run more or less the same anywhere, an assembler program will not. If C is nothing more than an assembler, that must imply its portability is just about zero. This depends entirely on how the C is written. It can be written to work specifically on one processor and one machine. Alternatively, providing a few rules are observed, a C program can be as portable as anything written in any other language.

Write Only Reputation

C has a fearsome reputation as a “write only” language. In other words it is possible to *write* code that is impossible to *read*. Unfortunately some people take this as a challenge.

History of C

- § **Developed by Brian Kernighan and Dennis Ritchie of AT&T Bell Labs in 1972**
- § **In 1983 the American National Standards Institute began the standardisation process**
- § **In 1989 the International Standards Organisation continued the standardisation process**
- § **In 1990 a standard was finalised, known simply as “Standard C”**
- § **Everything before this is known as “K&R C”**

The History of C

**Brian
Kernighan,
Dennis Ritchie**

C was invented primarily by Brian Kernighan and Dennis Ritchie working at AT&T Bell Labs in the United States. So the story goes, they used to play an “asteroids” game on the company mainframe. Unfortunately the performance of the machine left a lot to be desired. With the power of a 386 and around 100 users, they found they did not have sufficient control over the “spaceship”. They were usually destroyed quickly by passing asteroids.

Taking this rather personally, they decided to re-implement the game on a DEC PDP-7 which was sitting idle in the office. Unfortunately this PDP-7 had no operating system. Thus they set about writing one.

The operating system became a larger project than the asteroids game. Some time later they decided to port it to a DEC PDP-11. This was a mammoth task, since everything was hand-crafted in assembler.

The decision was made to re-code the operating system in a high level language, so it would be more portable between different types of machines. All that would be necessary would be to implement a compiler on each new machine, then compile the operating system.

The language that was chosen was to be a variant of another language in use at the time, called B. B is a word oriented language ideally suited to the PDP-7, but its facilities were not powerful enough to take advantage of the PDP-11 instruction set. Thus a new language, C, was invented.

The History of C

Standardization C turned out to be very popular and by the early 1980s hundreds of implementations were being used by a rapidly growing community of programmers. It was time to standardize the language.

ANSI In America, the responsibility for standardizing languages is that of the American National Standards Institute, or ANSI. The name of the ANSI authorized committee that developed the standard for C was X3J11. The language is now defined by ANSI Standard X3.159-1989.

ISO In the International arena, the International Standards Organization, or ISO, is responsible for standardizing computer languages. ISO formed the technical committee JTC1/SC22/WG14 to review the work of X3J11. Currently the ISO standard for C, ISO 9889:1990, is essentially identical to X3.159. The Standards differ only in format and in the numbering of the sections. The wording differs in a few places, but there are no substantive changes to the language definition.

The ISO C Standard is thus the final authority on what constitutes the C programming language. It is referred to from this point on as just "The Standard". What went before, i.e. C as defined by Brian Kernighan and Dennis Ritchie is known as "K&R C".

Standard C vs K&R C

- ⌘ **Parameter type checking added**
- ⌘ **Proper floating point support added**
- ⌘ **Standard Library covered too**
- ⌘ **Many “grey areas” addressed**
- ⌘ **New features added**

- ⌘ **Standard C is now the choice**
- ⌘ **All modern C compilers are Standard C**
- ⌘ **The course discusses Standard C**

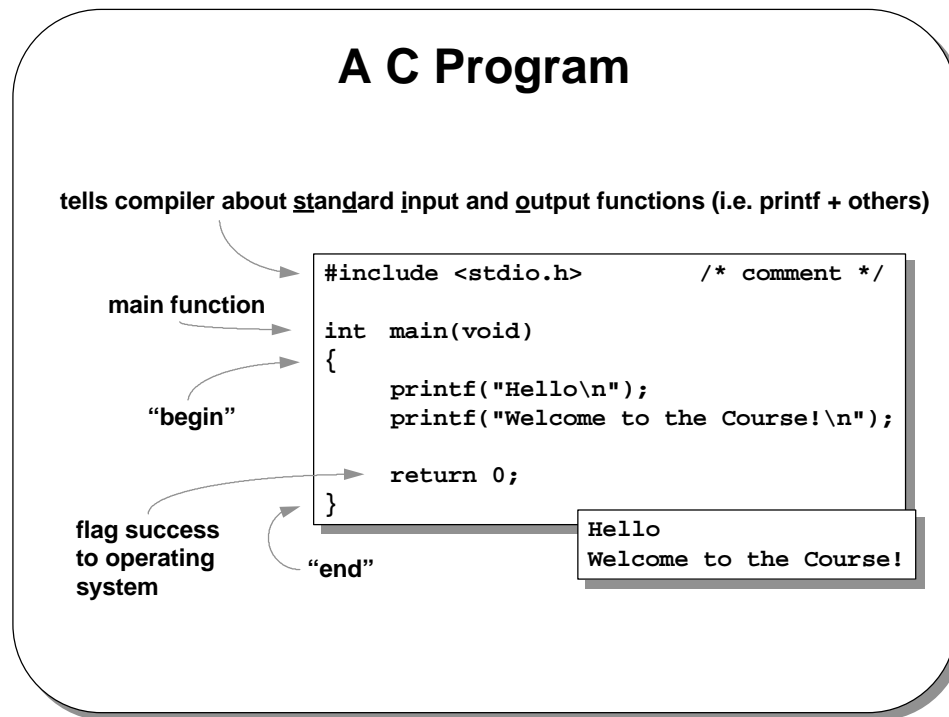
Standard C vs. K&R C

The C language has benefited enormously from the standardization processes. As a result it is much more usable than what went before. In K&R C there was no mechanism for checking parameters passed to functions. Neither the number, nor the types of the parameters were checked. As a programmer, if you were ever so reckless as to call any function anywhere you were totally responsible for reading the manual and ensuring the call was correct. In fact a separate utility, called *lint*, was written to do this.

Floating point calculations were always somewhat of a joke in K&R C. All calculations were carried out using a data type called *double*. This is despite there being provision for smaller floating point data type called *float*. Being smaller, floats were supposed to offer faster processing, however, converting them to double and back often took longer!

Although there had been an emerging Standard Library (a collection of routines provided with C) there was nothing standard about what it contained. The same routine would have different names. Sometimes the same routine worked in different ways.

Since Standard C is many times more usable than its predecessor, Standard C and not K&R C, is discussed on this course.



A C Program

#include	The #include directive instructs the C Preprocessor (a non interactive editor which will be discussed later) to find the text file " stdio.h ". The name itself means "standard input and output" and the ".h" means it is a header file rather than a C source file (which have the ".c" suffix). It is a text file and may be viewed with any text editor.
Comments	Comments are placed within /* and */ character sequences and may span any number of lines.
main	The main function is most important. This defines the point at which your program starts to execute. If you do not write a main function your program will not run (it will have no starting point). In fact, it won't even compile.
Braces	C uses the brace character " { " to mean "begin" and " } " to mean "end". They are much easier to type and, after a while, a lot easier to read.
printf	The printf function is the standard way of producing output. The function is defined within the Standard Library, thus it will always be there and always work in the same way.
\n	The sequence of two characters "\n" followed by "n" is how C handles new lines. When printed it causes the cursor to move to the start of the next line.
return	return causes the value, here 0, to be passed back to the operating system. How the operating system handles this information is up to it. MS-DOS, for instance, stores it in the ERRORLEVEL variable. The UNIX Bourne and Korn shells store it in a temporary variable, \$?, which may be used within shell scripts. "Tradition" says that 0 means success. A value of 1, 2, 3 etc. indicates failure. All operating systems support values up to 255. Some support values up to 65535, although if portability is important to you, only values of 0 through 255 should be used.

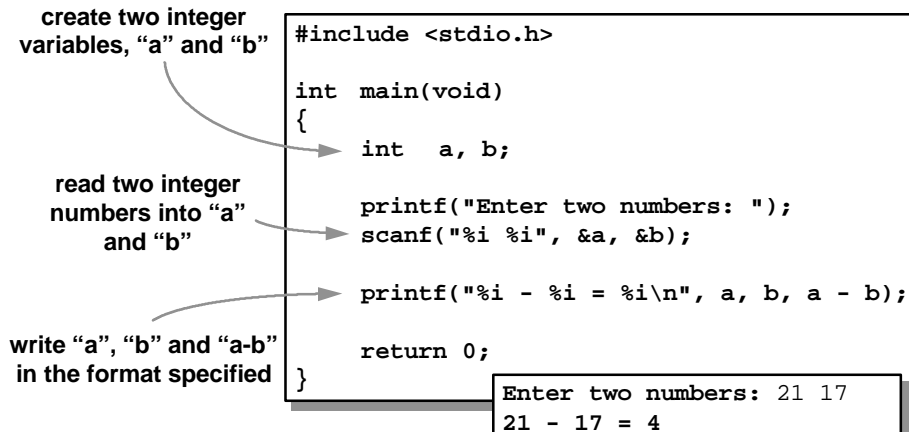
The Format of C

- ❧ **Statements are terminated with semicolons**
- ❧ **Indentation is ignored by the compiler**
- ❧ **C is case sensitive - all keywords and Standard Library functions are lowercase**
- ❧ **Strings are placed in double quotes**
- ❧ **Newlines are handled via `\n`**
- ❧ **Programs are capable of flagging success or error, those forgetting to do so have one or other chosen randomly!**

The Format of C

Semicolons	Semicolons are very important in C. They form a statement terminator - they tell the compiler where one statement ends and the next one begins. If you fail to place one after each statement, you will get compilation errors.
Free Format	C is a free format language. This is the up-side of having to use semicolons everywhere. There is no problem breaking a statement over two lines - all you need do is not place a semicolon in the middle of it (where you wouldn't have anyway). The spaces and tabs that were so carefully placed in the example program are ignored by the compiler. Indentation is entirely optional, but should be used to make the program more readable.
Case Sensitivity	C is a case sensitive language. Although <code>int</code> compiles, "Int", "INT" or any other variation will not. All of the 40 or so C keywords are lowercase. All of the several hundred functions in the Standard Library are lowercase.
Random Behavior	Having stated that <code>main</code> is to return an integer to the operating system, forgetting to do so (either by saying <code>return</code> only or by omitting the <code>return</code> entirely) would cause a random integer to be returned to the operating system. This random value could be zero (success) in which case your program may randomly succeed. More likely is a non zero value which would randomly indicate failure.

Another Example



Another Example

int

The `int` keyword, seen before when defining the return type for `main`, is used to create integer variables. Here two are created, the first "a", the second called "b".

scanf

The `scanf` function is the "opposite" of `printf`. Whereas `printf` produces output on the screen, `scanf` reads from the keyboard. The sequence "`%i`" instructs `scanf` to read an integer from the keyboard. Because "`%i %i`" is used two integers will be read. The first value typed placed into the variable "a", the second into the variable "b".

The space between the two "`%i`"s in "`%i %i`" is important: it instructs `scanf` that the two numbers typed at the keyboard may be separated by spaces. If "`%i,%i`" had been used instead the user would have been forced to type a comma between the two numbers.

printf

This example shows that `printf` and `scanf` share the same format specifiers. When presented with "`%i`" they both handle integers. `scanf`, because it is a reading function, reads integers from the keyboard. `printf`, because it is a writing function, writes integers to the screen.

Expressions

Note that C is quite happy to calculate "a-b" and print it out as an integer value. It would have been possible, but unnecessary, to create another variable "c", assign it the value of "a-b" and print out the value of "c".

Variables

- ⌘ Variables must be declared before use immediately after “{”
- ⌘ Valid characters are letters, digits and “_”
- ⌘ First character cannot be a digit
- ⌘ 31 characters recognised for local variables (more can be used, but are ignored)
- ⌘ Some implementations recognise only 6 characters in global variables (and function names)!
- ⌘ Upper and lower case letters are distinct

Variables

Declaring Variables

In C, all variables must be declared before use. This is not like FORTRAN, which if it comes across a variable it has never encountered before, declares it and gives it a type based on its name. In C, you the programmer must declare all variables and give each one a type (and preferably an initializing value).

Valid Names

Only letters, digits and the underscore character may be validly used in variable names. The first character of a variable may be a letter or an underscore, although The Standard says to avoid the use of underscores as the first letter. Thus the variable names “temp_in_celsius”, “index32” and “sine_value” are all valid, while “32index”, “temp-in-celsius” and “sine\$value” are not. Using variable name like “_sine” would be frowned upon, although not syntactically invalid.

Variable names may be quite long, with the compiler sorting through the first 31 characters. Names may be longer than this, but there must be a difference within the first 31 characters.

A few implementations (fortunately) require distinctions in *global* variables (which we haven't seen how to declare yet) and function names to occur within the first 6 characters.

Capital Letters

Capital letters may be used in variable names if desired. They are usually used as an alternative to the underscore character, thus “temp_in_celcius” could be written as “templnCelsius”. This naming style has become quite popular in recent years and the underscore has fallen into disuse.

printf and scanf

- § `printf` *writes* integer values to screen when `%i` is used
- § `scanf` *reads* integer values from the keyboard when `%i` is used
- § “&” VERY important with `scanf` (required to *change* the parameter, this will be investigated later) - absence will make program very ill
- § “&” not necessary with `printf` because current value of parameter is used

printf and scanf

printf

The `printf` function writes output to the screen. When it meets the format specifier `%i`, an integer is output.

scanf

The `scanf` function reads input from the keyboard. When it meets the format specifier `%i` the program waits for the user to type an integer.

&

The “&” is very important with `scanf`. It allows it to change the variable in question. Thus in:

```
scanf("%i", &j)
```

the “&” allows the variable “j” to be changed. Without this rather mysterious character, C prevents `scanf` from altering “j” and it would retain the *random* value it had previously (unless you'd remembered to initialize it).

Since `printf` does not need to change the value of any variable it prints, it does not need any “&” signs. Thus if “j” contains 15, after executing the statement:

```
printf("%i", j);
```

we would confidently expect 15 in the variable because `printf` would have been incapable of altering it.

Integer Types in C

- § C supports different kinds of integers
- § maxima and minima defined in “limits.h”

type	format	bytes	minimum	maximum
char	%c	1	CHAR_MIN	CHAR_MAX
signed char	%c	1	SCHAR_MIN	SCHAR_MAX
unsigned char	%c	1	0	UCHAR_MAX
short [int]	%hi	2	SHRT_MIN	SHRT_MAX
unsigned short	%hu	2	0	USHRT_MAX
int	%i	2 or 4	INT_MIN	INT_MAX
unsigned int	%u	2 or 4	0	UINT_MAX
long [int]	%li	4	LONG_MIN	LONG_MAX
unsigned long	%lu	4	0	ULONG_MAX

Integer Types in C

limits.h

This is the second standard header file we have met. This contains the definition of a number of constants giving the maximum and minimum sizes of the various kinds of integers. It is a text file and may be viewed with any text editor.

Different Integers

C supports integers of different sizes. The words **short** and **long** reflect the amount of memory allocated. A **short** integer theoretically occupies less memory than a **long** integer.

If you have a requirement to store a “small” number you could declare a **short** and sit back in the knowledge you were perhaps using less memory than for an **int**. Conversely a “large” value would require a **long**. It uses more memory, but your program could cope with very large values indeed.

The problem is that the terms “small number” and “large value” are rather meaningless. Suffice to say that SHRT_MAX is very often around 32,767 and LONG_MAX very often around 2,147,483,647. Obviously these aren’t the only possible values, otherwise we wouldn’t need the constants.

The most important thing to notice is that the size of **int** is either 2 or 4 bytes. Thus we cannot say, for a particular implementation, whether the largest value an integer may hold will be 32 thousand or 2 thousand million. For this reason, truly portable programs never use **int**, only **short** or **long**.

unsigned

The **unsigned** keyword causes **all** the available bits to be used to store the number - rather than setting aside the top bit for the sign. This means an **unsigned**’s greatest value may be twice as large as that of an **int**. Once **unsigned** is used, negative numbers cannot be stored, only zero and positive ones.

%hi

The “h” by the way is supposed to stand for “half” since a **short** is sometimes half the size of an **int** (on machines with a 2 byte **short** and a 4 byte **int**).

Integer Example

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    unsigned long big = ULONG_MAX;

    printf("minimum int = %i, ", INT_MIN);
    printf("maximum int = %i\n", INT_MAX);
    printf("maximum unsigned = %u\n", UINT_MAX);
    printf("maximum long int = %li\n", LONG_MAX);
    printf("maximum unsigned long = %lu\n", big);

    return 0;
}
```

```
minimum int = -32768, maximum int = 32767
maximum unsigned = 65535
maximum long int = 2147483647
maximum unsigned long = 4294967295
```

Integer Example

INT_MIN,
INT_MAX

The output of the program shows the code was run on a machine where an **int** was 16 bits, 2 bytes in size. Thus the largest value is 32767. It can also be seen the maximum value of an **unsigned int** is exactly twice that, at 65535.

Similarly the maximum value of an **unsigned long int** is exactly twice that of the maximum value of a **signed long int**.

Character Example

Note: print integer value of character

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    char lower_a = 'a';
    char lower_m = 'm';

    printf("minimum char = %i, ", CHAR_MIN);
    printf("maximum char = %i\n", CHAR_MAX);

    printf("after '%c' comes '%c'\n", lower_a, lower_a + 1);
    printf("uppercase is '%c'\n", lower_m - 'a' + 'A');

    return 0;
}
```

minimum char = 0, maximum char = 255
after 'a' comes 'b'
uppercase is 'M'

Character Example

char

C has the **char** data type for dealing with characters. Characters values are formed by placing the required value in *single* quotes. Thus:

```
char lower_a = 'a';
```

places the ASCII value of lowercase “a”, 97, into the variable “lower_a”. When this value of 97 is printed using %c, it is converted back into lowercase “a”. If this were run on an EBCDIC machine the value stored would be different, but would be converted so that “a” would appear on the output.

CHAR_MIN, CHAR_MAX

These two constants give the maximum and minimum values of characters. Since char is guaranteed to be 1 byte you may feel these values are always predictable at 0 and 255. However, C does not define whether **char** is signed or unsigned. Thus the minimum value of a char could be -128, the maximum value +127.

Arithmetic With char

The program shows the compiler is happy to do arithmetic with characters, for instance:

```
lower_a + 1
```

which yields 97 + 1, i.e. 98. This prints out as the value of lowercase “b” (one character immediately beyond lowercase “a”). The calculation:

```
lower_m - 'a' + 'A'
```

which gives rise to “M” would produce different (probably meaningless) results on an EBCDIC machine.

%c vs %i

Although you will notice here that **char** may be printed using %i, do not think this works with other types. You could not print an **int** or a **short** using %li.

Integers With Different Bases

§ It is possible to work in octal (base 8) and hexadecimal (base 16)

```
#include <stdio.h>
int main(void)
{
    int    dec = 20, oct = 020, hex = 0x20;
    printf("dec=%d, oct=%d, hex=%d\n", dec, oct, hex);
    printf("dec=%d, oct=%o, hex=%x\n", dec, oct, hex);
    return 0;
}
```

zero puts compiler into octal mode!

zero "x" puts compiler into hexadecimal mode

dec=20, oct=16, hex=32
dec=20, oct=20, hex=20

Integers With Different Bases

Decimal, Octal and Hexadecimal

C does not require you to work in decimal (base 10) all the time. If it is more convenient you may use octal or hexadecimal numbers. You may even mix them together in the same calculation.

Specifying octal constants is done by placing a leading zero before a number. So although 8 is a perfectly valid decimal eight, 08 is an invalid sequence. The leading zero places the compiler in octal mode but 8 is not a valid octal digit. This causes confusion (but only momentary) especially when programming with dates.

Specifying zero followed by "x" places the compiler into hexadecimal mode. Now the letters "a", "b", "c", "d", "e" and "f" may be used to represent the numbers 10 through 15. The case is unimportant, so 0x15AE, 0x15aE and 0x15ae represent the same number as does 0X15AE.

- %d** Causes an integer to be printed in *decimal* notation, this is effectively equivalent to %i
- %o** Causes an integer to be printed in *octal* notation.
- %x** Causes an integer to be printed in *hexadecimal* notation, "abcdef" are used.
- %X** Causes an integer to be printed in *hexadecimal* notation, "ABCDEF" are used.

Real Types In C

- ☞ C supports different kinds of reals
- ☞ maxima and minima are defined in “float.h”

type	format	bytes	minimum	maximum
float	%f %e %g	4	FLT_MIN	FLT_MAX
double	%lf %le %lg	8	DBL_MIN	DBL_MAX
long double	%Lf %Le %Lg	10	LDBL_MIN	LDBL_MAX

Real Types In C

- float.h** This is the third standard header file seen and contains only constants relating to C's floating point types. As can be seen here, maximum and minimum values are defined, but there are other useful things too. There are constants representing the accuracy of each of the three types.
- float** This is the smallest and least accurate of C's floating point data types. Nonetheless it is still good for around 6 decimal places of accuracy. Calculations using **float** are faster, but less accurate. It is relatively easy to overflow or underflow a **float** since there is comparatively little storage available. A typical minimum value is 10^{-38} , a typical maximum value 10^{+38} .
- double** This is C's mid-sized floating point data type. Calculations using **double** are slower than those using **float**, but more accurate. A **double** is good for around 12 decimal places. Because there is more storage available (twice as much as for a float) the maximum and minimum values are larger. Typically 10^{+308} or even 10^{+1000} .
- long double** This is C's largest floating point data type. Calculations using **long double** are the slowest of all floating point types but are the most accurate. A **long double** can be good for around 18 decimal places. Without employing mathematical “tricks” a **long double** stores the largest physical value C can handle. Some implementations allow numbers up to 10^{+4000} .
-

Real Example

```
#include <stdio.h>
#include <float.h>

int main(void)
{
    double f = 3.1416, g = 1.2e-5, h = 5000000000.0;

    printf("f=%lf\tg=%lf\tg=%lf\n", f, g, h);
    printf("f=%le\tg=%le\tg=%le\n", f, g, h);
    printf("f=%lg\tg=%lg\tg=%lg\n", f, g, h);

    printf("f=%7.2lf\tg=%.2le\tg=%.4lg\n", f, g, h);

    return 0;
}
```

f=3.141600	g=0.000012	h=5000000000.000000
f=3.141600e+00	g=1.200000e-05	h=5.000000e+09
f=3.1416	g=1.2e-05	h=5e+09
f= 3.14	g=1.20e-05	h=5e+09

Real Example

- %lf** This format specifier causes **printf** to display 6 decimal places, regardless of the magnitude of the number.
- %le** This format specifier still causes **printf** to display 6 decimal places, however, the number is displayed in “exponential” notation. For instance 1.200000e-05 indicates that 1.2 must be multiplied by 10^{-5} .
- %lg** As can be seen here, the “g” format specifier is probably the most useful. Only “interesting” data is printed - excess unnecessary zeroes are dropped. Also the number is printed in the shortest format possible. Thus rather than 0.000012 we get the slightly more concise 1.2e-05.
- %7.2lf** The 7 indicates the total width of the number, the 2 indicates the desired number of decimal places. Since “3.14” is only 4 characters wide and 7 was specified, 3 leading spaces are printed. Although it cannot be seen here, rounding is being done. The value 3.148 would have appeared as 3.15.
- %.2le** This indicates 2 decimal places and exponential format.
- %.4lg** Indicates 4 decimal places (none are printed because they are all zero) and shortest possible format.

Constants

- ⌘ **Constants have types in C**
- ⌘ **Numbers containing “.” or “e” are double: 3.5, 1e-7, -1.29e15**
- ⌘ **For float constants append “F”: 3.5F, 1e-7F**
- ⌘ **For long double constants append “L”: -1.29e15L, 1e-7L**
- ⌘ **Numbers without “.”, “e” or “F” are int, e.g. 10000, -35 (some compilers switch to long int if the constant would overflow int)**
- ⌘ **For long int constants append “L”, e.g. 9000000L**

Constants

Typed Constants

When a variable is declared it is given a type. This type defines its size and how it may be used. Similarly when a constant is specified the compiler gives it a type. With variables the type is obvious from their declaration. Constants, however, are not declared. Determining their type is not as straightforward.

The rules the compiler uses are outlined above. The constant “12”, for instance, would be integer since it does not contain a “.”, “e” or an “F” to make it a floating point type. The constant “12.” on the other hand would have type **double**. “12.L” would have type **long double** whereas “12.F” would have type **float**.

Although “12.L” has type **long double**, “12L” has type **long int**.

Warning!

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    double f = 5000000000.0;
```

```
    double g = 5000000000;
```

```
    printf("f=%lf\n", f);
```

```
    printf("g=%lf\n", g);
```

```
    return 0;
```

```
}
```

double precision constant
created because of "."

constant is integer or long
integer but 2,147,483,647 is
the maximum!

```
f=5000000000.000000  
g=705032704.000000
```

OVERFLOW



Warning!

The program above shows one of the problems of not understanding the nature of constants in C. Although the ".0" at the end of the 5000000000 would appear to make little difference, its absence makes 5000000000 an integral type (as in the case of the value which is assigned to "g"). Its presence (as in the case of the value which is assigned to "f") makes it a **double**.

The problem is that the largest value representable by most integers is around 2 thousand million, but this value is around 2½ times as large! The integer value overflows and the overflowed value is assigned to g.

Named Constants

§ Named constants may be created using `const`

creates an
integer
constant

error!

```
#include <stdio.h>

int main(void)
{
    const long double pi = 3.141592653590L;
    const int days_in_week = 7;
    const sunday = 0;

    days_in_week = 5;

    return 0;
}
```

Named Constants

`const`

If the idea of full stops, “e”s, “f”s and “L”s making a difference to the type of your constants is all a bit too arbitrary for you, C supports a `const` keyword which can be used to create constants with types.

Using `const` the type is explicitly stated, except with `const sunday` where the integer type is the default. This is consistent with existing rules, for instance `short` really means `short int`, `long` really means `long int`.

Lvalues and Rvalues

Once a constant has been created, it becomes an *rvalue*, i.e. it can only appear on the *right* of “=”. Ordinary variables are *lvalues*, i.e. they can appear on the left of “=”. The statement:

```
days_in_week = 5;
```

produces the rather unfriendly compiler message “invalid lvalue”. In other words the value on the left hand side of the “=” is not an lvalue it is an rvalue.

Preprocessor Constants

§ Named constants may also be created using the Preprocessor

- Needs to be in “search and replace” mode
- Historically these constants consist of capital letters

search for “PI”, replace with 3.1415....
Note: no “=”
and no “;”

```
#include <stdio.h>

#define PI 3.141592653590L
#define DAYS_IN_WEEK 7
#define SUNDAY 0

int day = SUNDAY;
long flag = USE_API;
```

“PI” is NOT substituted here

Preprocessor Constants

The preprocessor is a rather strange feature of C. It is a non interactive editor, which has been placed on the “front” of the compiler. Thus the compiler never sees the code you type, only the output of the preprocessor. This handles the **#include** directives by physically inserting the named file into what the compiler will eventually see.

As the preprocessor is an editor, it can perform search and replace. To put it in this mode the **#define** command is used. The syntax is simply:

```
#define search_text replace_text
```

Only whole words are replaced (the preprocessor knows enough C syntax to figure word boundaries). Quoted strings (i.e. everything within quotation marks) are left alone.

Take Care With `printf` And `scanf`!

“%c” fills one byte of “a” which is two bytes in size

“%f” expects 4 byte float in IEEE format, “b” is 2 bytes and NOT in IEEE format



```
#include <stdio.h>

int main(void)
{
    short a = 256, b = 10;

    printf("Type a number: ");
    scanf("%c", &a);

    printf("a = %hi, b = %f\n", a, b);

    return 0;
}
```

Type a number: 1
a = 305 b = Floating support not loaded

Take Care With `printf` and `scanf`!

Incorrect Format Specifiers

One of the most common mistakes for newcomers to C is to use the wrong format specifiers to `printf` and `scanf`. Unfortunately the compiler does not usually check to see if these are correct (as far as the compiler is concerned, the formatting string is just a string - as long as there are double quotes at the start and end, the compiler is happy).

It is vitally important to match the correct format specifier with the type of the item. The program above attempts to manipulate a 2 byte `short` by using `%c` (which manipulates 1 byte `chars`).

The output, `a=305` can just about be explained. The initial value of “a” is 256, in bit terms this is:

0000 0001 0000 0000

When prompted, the user types 1. As `printf` is in character mode, it uses the ASCII value of 1 i.e. 49. The bit pattern for this is:

0011 0001

This bit pattern is written into the first byte of a, but because the program was run on a byte swapped machine the value appears to be written into the bottom 8 bits, resulting in:

0000 0001 0011 0001

which is the bit pattern corresponding to 305.

Summary

- § **K&R C vs Standard C**
- § **main, printf**
- § **Variables**
- § **Integer types**
- § **Real types**
- § **Constants**
- § **Named constants**
- § **Preprocessor constants**
- § **Take care with printf and scanf**

Review Questions

1. What are the integer types?
2. What are the floating point types?
3. What format specifier would you use to read or write an **unsigned long int**?
4. If you made the assignment

```
char c = 'a';
```

then printed “c” as an integer value, what value would you see (providing the program was running on an ASCII machine).

Introduction Practical Exercises

Directory: **INTRO**

1. Write a program in a file called "**MAX.C**" which prints the maximum and minimum values of an integer. Use this to determine whether your compiler uses 16 or 32 bit integers.
 2. Write a program in a file called "**AREA.C**" which reads a real number (you can choose between float, double or long double) representing the radius of a circle. The program will then print out the area of the circle using the formula: $\text{area} = \pi r^2$
 π to 13 decimal places is 3.1415926535890. The number of decimal places you use will depend upon the use of float, double or long double in your program.
 3. Cut and paste your area code into "**CIRCUMF.C**" and modify it to print the circumference using the formula: $\text{circum} = 2\pi r$
 4. When both of these programs are working try giving either one invalid input. What answers do you see, "sensible" zeroes or random values?
What would you deduce **scanf** does when given invalid input?
 5. Write a program "**CASE**" which reads an upper case character from the keyboard and prints it out in lower case.
-

Introduction Solutions

1. Write a program in a file called "MAX.C" which prints the maximum and minimum values of an integer. Use this to determine whether your compiler uses 16 or 32 bit integers.

This task is made very easy by the constants defined in the header file "limits.h" discussed in the chapter notes. If the output of the program is in the region of ± 32 thousand then the compiler uses 16 bit integers. If the output is in the region of ± 2 thousand million the compiler uses 32 bit integers.

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    printf("minimum int = %i, ", INT_MIN);
    printf("maximum int = %i\n", INT_MAX);

    return 0;
}
```

2. Write a program in a file called "AREA.C" which reads a real number representing the radius of a circle. The program will then print out the area of the circle using the formula: $\text{area} = \pi r^2$

In the following code note:

- Long doubles are used for maximum accuracy
- Everything is initialized. This slows the program down slightly but does solve the problem of the user typing invalid input (scanf bombs out, but the variable radius is left unchanged at 0.0)
- There is no C operator which will easily square the radius, leaving us to multiply the radius by itself
- The `%nLf` in the printf allows the number of decimal places output to be specified

```
#include <stdio.h>

int main(void)
{
    long double      radius = 0.0L;
    long double      area = 0.0L;
    const long double pi = 3.1415926353890L;

    printf("please give the radius ");
    scanf("%Lf", &radius);

    area = pi * radius * radius;

    printf("Area of circle with radius %.3Lf is %.12Lf\n", radius, area);

    return 0;
}
```


3. Cut and paste your area code into “CIRCUMF.C” and modify it to print the circumference using the formula: $\text{circum} = 2\pi r$

The changes to the code above are trivial.

```
#include <stdio.h>

int main(void)
{
    long double      radius = 0.0L;
    long double      circumf = 0.0L;
    const long double pi = 3.1415926353890L;

    printf("please give the radius ");
    scanf("%Lf", &radius);

    circumf = 2.0L * pi * radius;

    printf("Circumference of circle with radius %.3Lf is %.12Lf\n",
           radius, circumf);

    return 0;
}
```

4. When both of these programs are working try giving either one invalid input. What answers do you see, “sensible” zeroes or random values?
What would you deduce `scanf` does when given invalid input?

*When `scanf` fails to read input in the specified format it abandons processing leaving the variable **unchanged**. Thus the output you see is entirely dependent upon how you have initialized the variable “radius”. If it is not initialized its value is random, thus “area” and “circumf” will also be random.*

5. Write a program “CASE” which reads an upper case character from the keyboard and prints it out in lower case.

Rather than coding in the difference between 97 and 65 and subtracting this from the uppercase character, get the compiler to do the hard work. Note that the only thing which causes `printf` to output a character is `%c`, if `%i` had been used the output would have been the ASCII value of the character.

```
#include <stdio.h>

int main(void)
{
    char  ch;

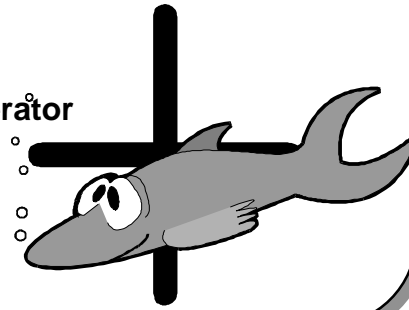
    printf("Please input a lowercase character ");
    scanf("%c", &ch);
    printf("the uppercase equivalent is '%c'\n", ch - 'a' + 'A');

    return 0;
}
```

Operators in C

Operators in C

- § **Arithmetic operators**
- § **Cast operator**
- § **Increment and Decrement**
- § **Bitwise operators**
- § **Comparison operators**
- § **Assignment operators**
- § **sizeof operator**
- § **Conditional expression operator**



Operators in C

The aim of this chapter is to cover the full range of diverse operators available in C. Operators dealing with pointers, arrays and structures will be left to later chapters.

Arithmetic Operators

§ C supports the arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo (remainder)

§ “%” may not be used with reals

Arithmetic Operators

+, -, *, /

C provides the expected mathematical operators. There are no nasty surprises. As might be expected, “+” and “-” may be used in a unary sense as follows:

or

x	=	+y;
x	=	-y;

The first is rather a waste of time and is exactly equivalent to “**x = y**” The second multiplies the value of “y” by -1 before assigning it to “x”.

%

C provides a modulo, or “remainder after dividing by” operator. Thus 25/4 is 6, 25%4 is 1. This calculation only really makes sense with integer numbers where there can be a remainder. When dividing floating point numbers there isn’t a remainder, just a fraction. Hence this operator cannot be applied to reals.

Using Arithmetic Operators

§ The compiler uses the types of the operands to determine how the calculation should be done

The diagram illustrates how the compiler determines the type of division based on the operand types. It shows a code snippet with three annotations pointing to specific lines:

- Annotation 1:** "i" and "j" are ints, integer division is done, 1 is assigned to "k". This points to the line `k = i / j;`.
- Annotation 2:** "f" and "g" are double, double division is done, 1.25 is assigned to "h". This points to the line `h = f / g;`.
- Annotation 3:** integer division is still done, despite "h" being double. Value assigned is 1.00000. This points to the line `h = i / j;`.

```
int main(void)
{
    int    i = 5,    j = 4,    k;
    double f = 5.0, g = 4.0, h;

    k = i / j;
    h = f / g;
    h = i / j;

    return 0;
}
```

Using Arithmetic Operators

One operator "+" must add integers together and add reals together. It might almost have been easier to provide two, then the programmer could carefully choose whenever addition was performed. But why stop with two versions? There are, after all, different kinds of integer and different kinds of real. Suddenly we can see the need for many different "+" variations. Then there are the numerous combinations of **int** and **double**, **short** and **float** etc. etc.

C gets around the problem of having many variations, by getting the "+" operator to choose itself what sort of addition to perform. If "+" sees an integer on its left and its right, integer addition is performed. With a real on the left and right, real addition is performed instead.

This is also true for the other operators, "-", "*", and "/". The compiler examines the types on either side of each operator and does whatever is appropriate. Note that this is literally true: **the compiler is only concerned with the types of the operands**. No account whatever is taken of the type being assigned to. Thus in the example above:

```
h = i / j;
```

It is the types of "i" and "j" (int) cause integer division to be performed. The fact that the result is being assigned to "h", a double, has no influence at all.

The Cast Operator

§ The cast operator *temporarily* changes the type of a variable

if either operand is a double,
the other is automatically
promoted

integer division is done here,
the result, 1, is changed to a
double, 1.00000

```
int main(void)
{
    int    i = 5, j = 4;
    double f;

    f = (double)i / j;
    f = i / (double)j;
    f = (double)i / (double)j;
    f = (double)(i / j);

    return 0;
}
```

The Cast Operator

Clearly we face problems with assignments like:

```
f = i / j;
```

if the compiler is just going to proceed with integer division we would be forced to declare some real variables, assign the integer values and divide the reals.

However, the compiler allows us to “change our mind” about the type of a variable or expression. This is done with the cast operator. The cast operator *temporarily* changes the type of the variable/expression it is applied to. Thus in:

```
f = i / j;
```

Integer division would normally be performed (since both “i” and “j” are integer). However the cast:

```
f = (double)i / j;
```

causes the type of “i” to be temporarily changed to **double**. In effect 5 becomes 5.0. Now the compiler is faced with dividing a **double** by an integer. It automatically promotes the integer “j” to a double (making it 4.0) and performs division using **double** precision maths, yielding the answer 1.25.

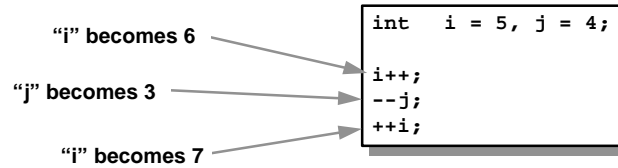
Increment and Decrement

- § C has two special operators for adding and subtracting one from a variable

++ increment

-- decrement

- § These may be either prefix (before the variable) or postfix (after the variable):



Increment and Decrement

C has two special, dedicated, operators which add one to and subtract one from a variable. How is it a minimal language like C would bother with these operators? They map directly into assembler. All machines support some form of "inc" instruction which increments a location in memory by one. Similarly all machines support some form of "dec" instruction which decrements a location in memory by one. All that C is doing is mapping these instructions directly.

Prefix and Postfix

§ The prefix and postfix versions are different

```
#include <stdio.h>

int main(void)
{
    int    i, j = 5;

    i = ++j;
    printf("i=%d, j=%d\n", i, j);

    j = 5;
    i = j++;
    printf("i=%d, j=%d\n", i, j);

    return 0;
}
```

equivalent to:

1. j++;
2. i = j;

equivalent to:

1. i = j;
2. j++;

i=6, j=6
i=5, j=6

Prefix and Postfix

The two versions of the ++ and -- operators, the prefix and postfix versions, are different. Both will add one or subtract one regardless of how they are used, the difference is in the assigned value.

Prefix ++, --

When the prefix operators are used, the increment or decrement happens **first**, the **changed** value is then assigned. Thus with:

```
i = ++j;
```

The current value of "j", i.e. 5 is changed and becomes 6. The 6 is copied across the "=" into the variable "i".

Postfix ++, --

With the postfix operators, the increment or decrement happens **second**. The **unchanged** value is assigned, then the value changed. Thus with:

```
i = j++;
```

The current value of "j", i.e. 5 is copied across the "=" into "i". Then the value of "j" is incremented becoming 6.

Registers

What is actually happening here is that C is either using, or not using, a temporary register to save the value. In the prefix case, "**i = ++j**", the increment is done and the value transferred. In the postfix case, "**i = j++**", C loads the current value (here "5") into a handy register. The increment takes place (yielding 6), then C takes the value stored in the register, 5, and copies that into "i". Thus the increment does take place before the assignment.

Truth in C

- § To understand C's comparison operators (less than, greater than, etc.) and the logical operators (and, or, not) it is important to understand how C regards truth
- § There is no boolean data type in C, integers are used instead
- § The value of 0 (or 0.0) is false
- § Any other value, 1, -1, 0.3, -20.8, is true

```
if(32)
    printf("this will always be printed\n");

if(0)
    printf("this will never be printed\n");
```

Truth in C

C has a very straightforward approach to what is true and what is false.

True Any non zero value is true. Thus, 1 and -5 are both true, because both are non zero. Similarly 0.01 is true because it, too, is non zero.

False Any zero value is false. Thus 0, +0, -0, 0.0 and 0.00 are all false.

Testing Truth Thus you can imagine that testing for truth is a very straightforward operation in C. Load the value to be tested into a register and see if any of its bits are set. If even a single bit is set, the value is immediately identified as true. If no bit is set anywhere, the value is identified as false.

The example above does cheat a little by introducing the if statement before we have seen it formally. However, you can see how simple the construct is:

```
if(condition)
    statement-to-be-executed-if-condition-was-true ;
```

Comparison Operators

§ C supports the comparison operators:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	is equal to
!=	is not equal to

§ These all give 1 (non zero value, i.e. true) when the comparison succeeds and 0 (i.e. false) when the comparison fails

Comparison Operators

C supports a full set of comparison operators. Each one gives one of two values to indicate success or failure. For instance in the following:

```
int i = 10, j, k;  
  
j = i > 5;  
k = i <= 1000;
```

The value 1, i.e. true, would be assigned to "j". The value 0, i.e. false, would be assigned to "k".

Theoretically any arbitrary non zero integer value could be used to indicate success. 27 for instance is non zero and would therefore "do". However C guarantees that 1 and only 1 will be used to indicate truth.

Logical Operators

§ C supports the logical operators:

&&	and
	or
!	not

§ These also give 1 (non zero value, i.e. true) when the condition succeeds and 0 (i.e. false) when the condition fails

```
int i, j = 10, k = 28;  
i = ((j > 5) && (k < 100)) || (k > 24);
```

Logical Operators

And, Or, Not

C supports the expected logical operators “and”, “or” and “not”. Unfortunately although the use of the words themselves might have been more preferable, symbols “&&”, “||” and “!” are used instead.

C makes the same guarantees about these operators as it does for the comparison operators, i.e. the result will only ever be 1 or 0.

Logical Operator Guarantees

- § C makes two important guarantees about the evaluation of conditions
- § Evaluation is left to right
- § Evaluation is “short circuit”

“i < 10” is evaluated first, if false the whole statement is false (because false AND anything is false) thus “a[i] > 0” would not be evaluated

```
if(i < 10 && a[i] > 0)
    printf("%i\n", a[i]);
```

Logical Operator Guarantees

C Guarantees

C makes further guarantees about the logical operators. Not only will they produce 1 or 0, they are will be evaluated in a well defined order. The left-most condition is always evaluated first, even if the condition is more complicated, like:

```
if(a && b && c && d || e)
```

Here “a” will be evaluated first. If true, “b” will be evaluated. If true, “c” will be evaluated and so on.

The next guarantee C makes is that as soon as it is decided whether a condition is true or false, no further evaluation is done. Thus if “b” turned out to be false, “c” and “d” would not be evaluated. The next thing evaluated would be “e”.

This is probably a good time to remind you about truth tables:

and Truth Table

&&	false	true
false	false	false
true	false	true

or Truth Table

	false	true
false	false	true
true	true	true

Warning!

- ⚠ **Remember to use parentheses with conditions, otherwise your program may not mean what you think**

in this attempt to say “i not equal to five”, “!i” is evaluated first. As “i” is 10, i.e. non zero, i.e. true, “!i” must be false, i.e. zero. Zero is compared with five

```
int    i = 10;

if(!i == 5)
    printf("i is not equal to five\n");
else
    printf("i is equal to five\n");
```

i is equal to five

Warning!

Parentheses

An extra set of parentheses (round brackets) will always help to make code easier to read and easier to understand. Remember that code is written once and maintained thereafter. It will take only a couple of seconds to add in extra parentheses, it may save several minutes (or perhaps even hours) of debugging time.

Bitwise Operators

☞ C has the following bit operators which may only be applied to integer types:

&	bitwise and
	bitwise inclusive or
^	bitwise exclusive or
~	one's compliment
>>	right shift
<<	left shift

Bitwise Operators

As Brian Kernighan and Dennis Ritchie needed to manipulate hardware registers in their PDP-11, they needed the proper tools (i.e. bit manipulation operators) to do it.

& vs &&

You will notice that the bitwise and, **&**, is related to the logical and, **&&**. As Brian and Dennis were doing more bitwise manipulation than logical condition testing, they reserved the single character for bitwise operation.

| vs ||

Again the bitwise (inclusive) or, **|**, is related to the logical or, **||**.

^

A bitwise exclusive or is also provided.

Truth Tables For Bitwise Operators

or	0	1
0	0	1
1	1	1

and	0	1
0	0	0
1	0	1

xor	0	1
0	0	1
1	1	0

The ones compliment operator "**~**" flips all the bits in a value, so all 1s are turned to 0s, while all 0s are turned to 1s.

Bitwise Example

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    short a = 0x6eb9;
```

```
    short b = 0x5d27;
```

```
    unsigned short c = 7097;
```

```
    printf("0x%x, ", a & b);
```

```
    printf("0x%x, ", a | b);
```

```
    printf("0x%x\n", a ^ b);
```

```
    printf("%u, ", c << 2);
```

```
    printf("%u\n", c >> 1);
```

```
    return 0;
```

```
}
```

```
0x4c21, 0x7fbf, 0x339e
28388, 3548
```

```
0x6eb9 0110 1110 1011 1001
0x5d27 0101 1101 0010 0111
0x4c21 0100 1100 0010 0001
```

```
0x6eb9 0110 1110 1011 1001
0x5d27 0101 1101 0010 0111
0x7fbf 0111 1111 1011 1111
```

```
0x6eb9 0110 1110 1011 1001
0x5d27 0101 1101 0010 0111
0x339e 0011 0011 1001 1110
```

```
7097 0001 1011 1011 1001
28388 0110 1110 1110 0100
```

```
7097 0001 1011 1011 1001
3548 0000 1101 1101 1100
```

Bitwise Example

This example shows bitwise manipulation. **short** integers are used, because these can be relied upon to be 16 bits in length. If **int** had been used, we may have been manipulating 16 or 32 bits depending on machine and compiler.

Arithmetic Results of Shifting

Working in hexadecimal makes the first 3 examples somewhat easier to understand. The reason why the 7097 is in decimal is to show that "**c<<2**" multiplies the number by 4 (shifting one place left multiplies by 2, shifting two places multiplies by 4), giving 28388. Shifting right by one divides the number by 2. Notice that the right-most bit is lost in this process. The bit cannot be recovered, once gone it is gone forever (there is no access to the carry flag from C). The missing bit represents the fraction (a half) truncated when integer division is performed.

Use **unsigned** When Shifting Right

One important aspect of right shifting to understand is that if a *signed* type is right shifted, the most significant bit is inserted. If an *unsigned* type is right shifted, 0s are inserted. If you do the maths you'll find this is correct. If you're not expecting it, however, it can be a bit of a surprise.

Assignment

- § Assignment is more flexible than might first appear
- § An assigned value is always made available for subsequent use

```
int i, j, k, l, m, n;  
i = j = k = l = m = n = 22;  
printf("%i\n", j = 93);
```

"n = 22" happens first, this makes 22 available for assignment to "m". Assigning 22 to "m" makes 22 available for assignment to "l" etc.

"j" is assigned 93, the 93 is then made available to printf for printing

Assignment

Assignment Uses Registers

Here is another example of C using registers. Whenever a value is assigned in C, the assigned value is left lying around in a handy register. This value in the register may then be referred to subsequently, or merely overwritten by the next statement.

Thus in the assignment above, 22 is placed both into "n" and into a machine register. The value in the register is then assigned into "m", and again into "l" etc.

With: `printf("%i\n", j = 93);`

93 is assigned to "j", the value of 93 is placed in a register. The value saved in the register is then printed via the "%i".

Warning!

- § One of the most frequent mistakes is to confuse test for equality, “==”, with assignment, “=”

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    if(i = 0)
        printf("i is equal to zero\n");
    else
        printf("somehow i is not zero\n");

    return 0;
}
```



somehow i is not zero

Warning!

Test for Equality vs. Assignment

A frequent mistake made by newcomers to C is to use assignment when test for equality was intended. The example above shows this. Unfortunately it uses the *if* then *else* construct to illustrate the point, something we haven't formally covered yet. However the construct is very straightforward, as can be seen.

Here “i” is initialized with the value of zero. The test isn't really a test because it is an assignment. The compiler overwrites the value stored in “i” with zero, this zero is then saved in a handy machine register. It is this value, saved in the register, that is tested. Since zero is always false, the *else* part of the construct is executed. The program would have worked differently if the test had been written “i == 0”.

Other Assignment Operators

§ There is a family of assignment operators:

+=	-=	*=	/=	%=
&=	=	^=		
<<=	>>=			

§ In each of these:

expression1 *op*= expression2

is equivalent to:

(expression1) = (expression1) op (expression2)

a += 27;

a = a + 27;

f /= 9.2;

f = f / 9.2;

i *= j + 2;

i = i * (j + 2);

Other Assignment Operators

+=, -=, *=, /=, %= etc.

There is a whole family of assignment operators in C, not just “=”. They all look rather unfamiliar and therefore rather frightening at first, but they really are very straightforward. Take, for instance, the statement “**a -= b**”. All this means is “**a = a - b**”. The only other thing to remember is that C evaluates the right hand expression first, thus “**a *= b + 7**” definitely means “**a = a * (b + 7)**” and NOT “**a = a * b + 7**”.

If they appear rather strange for a minimalist language like C, they used to make a difference in the K&R days before compiler optimizers were written.

If you imagine the assembler statements produced by “**a = a + 7**”, these could be as involved as “take value in ‘a’ and load into register”, “take value in register and add 7”, “take value in register and load into ‘a’”. Whereas the statement “**a += 7**” could just involve “take value in ‘a’ and add 7”.

Although there was a difference in the K&R days (otherwise these operators would never have been invented) a modern optimizing compiler should produce exactly the same code. Really these operators are maintained for backwards compatibility.

sizeof Operator

§ C has a mechanism for determining how many bytes a variable occupies

```
#include <stdio.h>

int main(void)
{
    long big;

    printf("\"big\" is %u bytes\n", sizeof(big));
    printf("a short is %u bytes\n", sizeof(short));
    printf("a double is %u bytes\n", sizeof double);

    return 0;
}
```

```
"big" is 4 bytes
a short is 2 bytes
a double is 8 bytes
```

sizeof Operator

The C Standard does not fix the size of its data types between implementations. Thus it is possible to find one implementation using 16 bit **ints** and another using 32 bit **ints**. It is also, theoretically, possible to find an implementation using 64 bit **long** integers. Nothing in the language, or Standard, prevents this.

Since C makes it so difficult to know the size of things in advance, it compensates by providing a built in operator **sizeof** which returns (usually as an **unsigned int**) the number of bytes occupied by a data type or a variable.

You will notice from the example above that the parentheses are optional:

sizeof(double)

and

sizeof double

are equivalent.

Because **sizeof** is a keyword the parentheses are optional. **sizeof** is NOT a Standard Library function.

Conditional Expression Operator

- § The conditional expression operator provides an in-line if/then/else
- § If the first expression is true, the second is evaluated
- § If the first expression is false, the third is evaluated

```
int i, j = 100, k = -1;
i = (j > k) ? j : k;
```

```
if(j > k)
    i = j;
else
    i = k;
```

```
int i, j = 100, k = -1;
i = (j < k) ? j : k;
```

```
if(j < k)
    i = j;
else
    i = k;
```

Conditional Expression Operator

C provides a rather terse, ternary operator (i.e. one which takes 3 operands) as an alternative to the *if* then *else* construct. It is rather like:

condition ? value-when-true : value-when-false

The condition is evaluated (same rules as before, zero false, everything else true). If the condition were found to be true the value immediately after the "?" is used. If the condition were false the value immediately after the ":" is used.

The types of the two expressions must be the same. It wouldn't make much sense to have one expression evaluating to a double while the other evaluates to an unsigned char (though most compilers would do their best to cope).

Conditional expression vs. if/then/else

This is another of those C operators that you must take at face value and decide whether to ever use it. If you feel *if* then *else* is clearer and more maintainable, use it. One place where this operator is useful is with pluralisation, for example:

```
if(dir == 1)
    printf("1 directory\n");
else
    printf("%i directories\n", dir);
```

may be expressed as:

```
printf("%i director%s\n", (dir == 1) ? "y" : "ies");
```

It is a matter of personal choice as to whether you find this second form more acceptable. Strings, printed with "%s", will be covered later in the course.

Precedence of Operators

- § C treats operators with different importance, known as *precedence*
- § There are 15 levels
- § In general, the unary operators have higher precedence than binary operators
- § Parentheses can always be used to improve clarity

```
#include <stdio.h>

int main(void)
{
    int j = 3 * 4 + 48 / 7;
    printf("j = %i\n", j);
    return 0;
}
```

j = 18

Precedence of Operators

C assigns different “degrees of importance” or “precedence” to its 40 (or so) operators. For instance the statement

$3 * 4 + 48 / 7$

could mean:

$((3 * 4) + 48) / 7$

or maybe:

$(3 * 4) + (48 / 7)$

or maybe even:

$3 * ((4 + 48) / 7)$

In fact it means the second, “ $(3 * 4) + (48 / 7)$ ” because C attaches more importance to “ $*$ ” and “ $/$ ” than it does to “ $+$ ”. Thus the multiplication and the divide are done before the addition.

Associativity of Operators

- § For two operators of equal precedence (i.e. same importance) a second rule, “associativity”, is used
- § Associativity is either “left to right” (left operator first) or “right to left” (right operator first)

```
#include <stdio.h>

int main(void)
{
    int i = 6 * 4 / 7;
    printf("i = %d\n", i);
    return 0;
}
```

i = 3

Associativity of Operators

Precedence does not tell us all we need to know. Although “*****” is more important than “**+**”, what happens when two operators of equal precedence are used, like “*****” and “**/**” or “**+**” and “**-**”? In this case C resorts to a second rule, associativity.

Associativity is either “left to right” or “right to left”.

Left to Right Associativity

This means the left most operator is done first, then the right.

Right to Left Associativity

The right most operator is done first, then the left.

Thus, although “*****” and “**/**” are of equal precedence in “**6 * 4 / 7**”, their associativity is left to right. Thus “*****” is done first. Hence “**6 * 4**” first giving 24, next “**24 / 7**” = 3.

If you are wondering about an example of right to left associativity, consider:

a = b += c;

Here both “**=**” and “**+=**” have the same precedence but their associativity is right to left. The right hand operator “**+=**” is done first. The value of “**c**” modifies “**b**”, the modified value is then assigned to “**a**”.

Precedence/Associativity Table

<u>Operator</u>	<u>Associativity</u>
() [] -> .	left to right
! ~ ++ -- - + (cast) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= >= >	left to right
== !=	left to right
&	left to right
	left to right
^	left to right
&&	left to right
	left to right
? :	right to left
= += -= *= /= %= etc	right to left
,	left to right

Precedence/Associativity Table

The table above shows the precedence and associativity of C's operators. This chapter has covered around 37 operators, the small percentage of remaining ones are concerned with pointers, arrays, structures and calling functions.

Review

```
#include <stdio.h>

int main(void)
{
    int i = 0, j, k = 7, m = 5, n;

    j = m += 2;
    printf("j = %d\n", j);

    j = k++ > 7;
    printf("j = %d\n", j);

    j = i == 0 & k;
    printf("j = %d\n", j);

    n = !i > k >> 2;
    printf("n = %d\n", n);

    return 0;
}
```

Review

Consider what the output of the program would be if run? Check with your colleagues and the instructor to see if you agree.

Operators in C Practical Exercises

Directory: **OPERS**

1. Write a program in "**SUM.C**" which reads two integers and prints out the sum, the difference and the product. Divide them too, printing your answer to two decimal places. Also print the remainder after the two numbers are divided.

Introduce a test to ensure that when dividing the numbers, the second number is not zero.

What happens when you add two numbers and the sum is too large to fit into the data type you are using? Are there friendly error messages?

2. Cut and paste your "**SUM.C**" code into "**BITOP.C**". This should also read two integers, but print the result of bitwise anding, bitwise oring and bitwise exclusive oring. Then either use these two integers or prompt for two more and print the first left-shifted by the second and the first right-shifted by the second. You can choose whether to output any of these results as decimal, hexadecimal or octal.

What happens when a number is left shifted by zero? If a number is left shifted by -1, does that mean it is right shifted by 1?

3. Write a program in a file called "**VOL.C**" which uses the area code from "**AREA.C**". In addition to the radius, it prompts for a height with which it calculates the volume of a cylinder. The formula is volume = area * height.
-

Operators in C Solutions

1. Write a program in "SUM.C" which reads two integers and prints out the sum, the difference and the product. Divide them too, printing your answer to two decimal places. Also print the remainder after the two numbers are divided.

Introduce a test to ensure that when dividing the numbers, the second number is not zero.

A problem occurs when dividing the two integers since an answer to two decimal places is required, but dividing two integers yields an integer. The solution is to cast one or other (or both) of the integers to a double, so that double precision division is performed. The minor problem of how to print "%" is overcome by placing "%%" within the string.

```
#include <stdio.h>

int main(void)
{
    int    first, second;

    printf("enter two integers ");
    scanf("%i %i", &first, &second);

    printf("%i + %i = %i\n", first, second, first + second);
    printf("%i - %i = %i\n", first, second, first - second);
    printf("%i * %i = %i\n", first, second, first * second);

    if(second != 0) {
        printf("%i / %i = %.2lf\n", first, second,
               (double)first / second);
        printf("%i %% %i = %i\n", first, second,
               first % second);
    }

    return 0;
}
```

What happens when you add two numbers and the sum is too large to fit into the data type you are using? Are there friendly error messages?

C is particularly bad at detecting overflow or underflow. When two large numbers are entered the addition and multiplication yield garbage.

-
2. Cut and paste your “SUM.C” code into “BITOP.C”. This should also read two integers, but print the result of bitwise anding, bitwise oring and bitwise exclusive oring. Then either use these two integers or prompt for two more and print the first left-shifted by the second and the first right-shifted by the second. You can choose whether to output the results as decimal, hexadecimal or octal.

```
#include <stdio.h>

int main(void)
{
    int    first, second;

    printf("enter two integers ");
    scanf("%i %i", &first, &second);

    printf("%x & %x = %x\n", first, second, first & second);
    printf("%x | %x = %x\n", first, second, first | second);
    printf("%x ^ %x = %x\n", first, second, first ^ second);

    printf("enter two more integers ");
    scanf("%i %i", &first, &second);

    printf("%i << %i = %i\n", first, second, first << second);
    printf("%i >> %i = %i\n", first, second, first >> second);

    return 0;
}
```

What happens when a number is left shifted by zero? If a number is left shifted by -1, does that mean it is right shifted by 1?

When a number is shifted by zero, it should remain unchanged. The effects of shifting by negative amounts are undefined.

3. Write a program in a file called "VOL.C" which uses the area code from "AREA.C". In addition to the radius, it prompts for a height with which it calculates the volume of a cylinder. The formula is volume = area * height.

Here notice how an especially long string may be broken over two lines, providing double quotes are placed around each part of the string.

```
#include <stdio.h>

int main(void)
{
    long double      radius = 0.0L;
    long double      height = 0.0L;
    long double      volume = 0.0L;
    const long double pi = 3.1415926353890L;

    printf("please give the radius and height ");
    scanf("%Lf %Lf", &radius, &height);

    volume = pi * radius * radius * height;

    printf("Volume of cylinder with radius %.3Lf "
           "and height %.3Lf is %.12Lf\n",
           radius, height, volume);

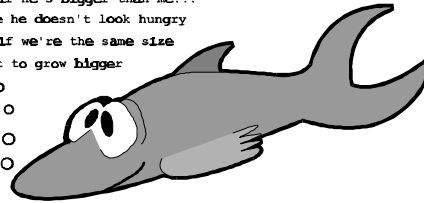
    return 0;
}
```

Control Flow

Control Flow

- ⌘ **Decisions - if then else**
- ⌘ **More decisions - switch**
- ⌘ **Loops - while, do while, for**
- ⌘ **Keyword break**
- ⌘ **Keyword continue**

```
if I'm bigger than him and I'm hungry...  
    then it's mealtime  
else, if he's bigger than me...  
    hope he doesn't look hungry  
else, if we're the same size  
    wait to grow bigger  
    o  
    o  
    o  
    o
```



Control Flow

This chapter covers all the decision making and looping constructs in C.

Decisions - **if** then

- ⌘ **Parentheses surround the test**
- ⌘ **One statement becomes the “then part”**
- ⌘ **If more are required, braces must be used**

```
scanf("%i", &i);  
  
if(i > 0)  
    printf("a positive number was entered\n");  
  
if(i < 0) {  
    printf("a negative number was entered\n");  
    i = -i;  
}
```

Decisions **if** then

This formally introduces C's **if** then construct which was seen a few times in the previous chapter. The most important thing to remember is to surround the condition with parentheses. These are mandatory rather than optional. Notice there is no keyword *then*. It is implied by the sense of the statement.

If only one statement is to be executed, just write the statement, if many statements are to be executed, use the begin and end braces “{” and “}” to group the statements into a block.

Warning!

⚠ **A semicolon after the condition forms a “do nothing” statement**

```
printf("input an integer: ");  
scanf("%i", &j);  
  
if(j > 0);  
    printf("a positive number was entered\n");
```

```
input an integer: -6  
  
a positive number was entered
```



Warning!

**Avoid Spurious
Semicolons
After `if`**

Having become used to the idea of placing semicolon characters after each and every statement in C, we start to see that the word “statement” is not as straightforward as might appear.

A semicolon has been placed after the condition in the code above. The compiler considers this placed for a reason and makes the semicolon the *then* part of the construct. A “do nothing” or a “no op” statement is created (each machine has an instruction causing it to wait for a machine cycle). Literally if “j” is greater than zero, nothing will be done. After the machine cycle, the next statement is always arrived at, regardless of the no op execution.

if then else

- § An optional **else** may be added
- § One statement by default, if more are required, braces must be used

```
if(i > 0)
    printf("i is positive\n");
else
    printf("i is negative\n");
```

```
if(i > 0)
    printf("i is positive\n");
else {
    printf("i is negative\n");
    i = -i;
}
```

if then else

Optionally an **else** statement, which is executed if the condition is false, may be added. Again, begin and end braces should be used to block together a more than one statement.

You may wish to always use braces as in:

```
if(i > 0) {
    printf("i is positive\n");
} else {
    printf("i is negative\n");
}
```

This is perhaps a suitable point to mention the braces have no clear, fixed position in C. Being a free format language you may feel happier with:

```
if(i > 0)
{
    printf("i is positive\n");
}
else
{
    printf("i is negative\n");
}
```

or:

```
if(i > 0)
{
    printf("i is positive\n");
}
else
{
    printf("i is negative\n");
}
```

All are acceptable to the compiler, i.e. the positioning of the braces makes no difference at all.

Nesting ifs

§ **else associates with the nearest if**

```
int i = 100;
if(i > 0)
    if(i > 1000)
        printf("i is big\n");
    else
        printf("i is reasonable\n");
```

i is reasonable

```
int i = -20;
if(i > 0) {
    if(i > 1000)
        printf("i is big\n");
} else
    printf("i is negative\n");
```

i is negative

Nesting ifs

**Where Does
else Belong?**

C, along with other high level languages, has a potential ambiguity with nested **if** then **else** statements. This arises in trying to determine where an **else** clause belongs. For instance, consider:

```
if it is a weekday
if it is raining    catch the bus to work
else                walk to work
```

Does this mean “if it is a weekday and it is not raining” walk to work, or does it mean “if it is not a weekday” then walk to work. If the latter, we could end up walking to work at weekends, whether or not it is raining.

C resolves this ambiguity by saying that all *elses* belong to the nearest *if*. Applying these rules to the above would mean “if it is a weekday and it is not raining” walk to work. Fortunately we will not end up walking to work at weekends.

switch

§ C supports a **switch** for multi-way decision making

```
switch(c) {
    case 'a': case 'A':
        printf("area = %.2f\n", r * r * pi);
        break;
    case 'c': case 'C':
        printf("circumference = %.2f\n", 2 * r * pi);
        break;
    case 'q':
        printf("quit option chosen\n");
        break;
    default:
        printf("unknown option chosen\n");
        break;
}
```

switch

switch vs.
if/then/else

C supports a multi-way decision making construct called **switch**. The code above is an alternative to the nested if then else construct:

```
if(c == 'a' || c == 'A')
    printf("area = %.2f\n", r * r * pi);
else if(c == 'c' || c == 'C')
    printf("circumference = %.2f\n", 2 * r * pi);
else if(c == 'q')
    printf("quit option chosen\n");
else
    printf("unknown option chosen\n");
```

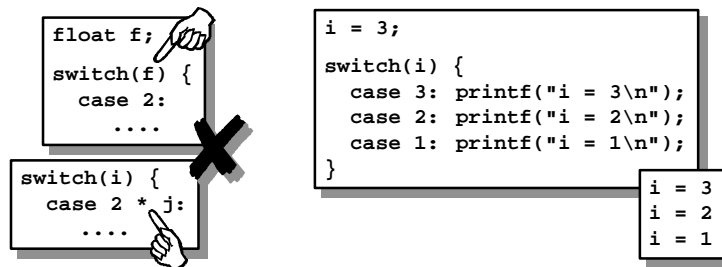
The conditions may be placed in any order:

```
switch(c) {
    default:
        printf("unknown option chosen\n");
        break;
    case 'q':
        printf("quit option chosen\n");
        break;
    case 'c': case 'C':
        printf("circumference = %.2f\n", 2 * r * pi);
        break;
    case 'a': case 'A':
        printf("area = %.2f\n", r * r * pi);
        break;
}
```

Placing **default** first does not alter the behavior in any way.

More About switch

- ⌘ Only integral constants may be tested
- ⌘ If no condition matches, the `default` is executed
- ⌘ If no `default`, nothing is done (not an error)
- ⌘ The `break` is important



More About **switch**

switch Less
Flexible Than
if/then/else

The **switch** is actually a little less flexible than an **if** then **else** construct. **switch** may only test integer types and not any of the reals, whereas

```
if(f == 0.0)
    printf("f is zero\n");
```

is quite valid,

```
switch(f) {
    case 0.0:
        printf("f is zero\n");
        break;
}
```

will not compile. Also, the **switch** can test only against constants, not against the values of other variables. Whereas

```
if(i == j)
    printf("equal\n");
```

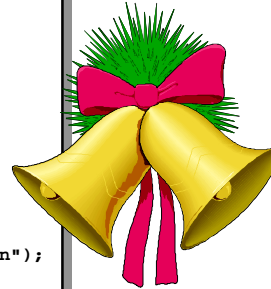
is valid:

```
switch(i) {
    case j:
        printf("equal\n");
        break;
}
```

is not.

A switch Example

```
printf("On the ");
switch(i) {
    case 1:    printf("1st");    break;
    case 2:    printf("2nd");    break;
    case 3:    printf("3rd");    break;
    default:   printf("%ith", i); break;
}
printf(" day of Christmas my true love sent to me ");
switch(i) {
    case 12:   printf("twelve lords a leaping, ");
    case 11:   printf("eleven ladies dancing, ");
    case 10:   printf("ten pipers piping, ");
    case 9:    printf("nine drummers drumming, ");
    case 8:    printf("eight maids a milking, ");
    case 7:    printf("seven swans a swimming, ");
    case 6:    printf("six geese a laying, ");
    case 5:    printf("five gold rings, ");
    case 4:    printf("four calling birds, ");
    case 3:    printf("three French hens, ");
    case 2:    printf("two turtle doves and ");
    case 1:    printf("a partridge in a pear tree\n");
}
```



A switch Example

Twelve Days of Christmas

This example shows the effects of the presence or absence of the **break** keyword on two **switch** statements. With the first, only one statement in the **switch** will be executed. For example, say "i" is set to 2, the first **switch** calls **printf** to print "2nd". The **break** is encountered causing the **switch** to finish and control be transferred to the line:

```
printf("day of Christmas my true love sent to me");
```

Then the second **switch** is entered, with "i" still set to 2. The **printf** corresponding to the "two turtle doves" is executed, but since there is no **break**, the **printf** corresponding to the "partridge in the pear tree" is executed. The absence of **breaks** in the second **switch** statement means that if "i" were, say, 10 then 10 **printf** statements would be executed.

while Loop

- ⌘ The simplest C loop is the **while**
- ⌘ Parentheses must surround the condition
- ⌘ One statement forms the body of the loop
- ⌘ Braces must be added if more statements are to be executed

```
int j = 5;
while(j > 0)
    printf("j = %i\n", j--);
```

```
while(j > 0) {
    printf("j = %i\n", j);
    j--;
}
```

```
j = 5
j = 4
j = 3
j = 2
j = 1
```

while Loop

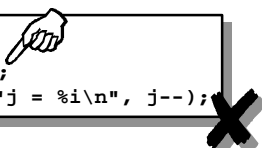
C has three loops, **while** is the simplest of them all. It is given a condition (in parentheses, just like with the **if** statement) which it evaluates. If the condition evaluates to true (non zero, as seen before) the body of the loop is executed. The condition is evaluated again, if still true, the body of the loop is executed again. This continues until the condition finally evaluates to false. Then execution jumps to the first statement that follows on after the loop.

Once again if more than one statement is required in the body of the loop, begin and end braces must be used.

(Another) Semicolon Warning!

§ A semicolon placed after the condition forms a body that does nothing

```
int j = 5;
while(j > 0);
    printf("j = %i\n", j--);
```



program disappears
into an infinite loop

- Sometimes an empty loop body is required

```
int c, j;
while(scanf("%i", &j) != 1)
    while((c = getchar()) != '\n')
        ;
```

placing semicolon
on the line below
makes the
intention obvious

(Another) Semicolon Warning!

Avoid
Semicolons
After **while**

We have already seen that problems can arise if a semicolon is placed after an **if** statement. A similar problem exists with loops, although it is more serious. With **if** the no op statement is potentially executed only once. With a loop it may be executed an infinite number of times. In the example above, instead of the loop body being:

```
printf("j = %i\n", j--);
```

causing "j" to be decremented each time around the loop, the body becomes "do nothing". Thus "j" remains at 5. The program loops infinitely doing nothing. No output is seen because the program is so busily "doing nothing" the printf statement is never reached.

Flushing Input

Occasionally doing nothing is exactly what we want. The practical exercises have already illustrated that there is a problem with scanf buffering characters. These characters may be thrown away with the **while** loop shown above. This employs some of the features we investigated in the last chapter. When the value is assigned to "c", that value (saved in a register) may be tested against "\n".

To be honest this **scanf** loop above leaves something to be desired. While **scanf** is failing there is no indication that the user should type anything else (the terminal seems to hang), **scanf** just waits for the next thing to be typed. Perhaps a better construction would be:

```
printf("enter an integer: ");
while(scanf("%i", &j) != 1) {
    while((ch = getchar()) != '\n')
        ;
    printf("enter an integer: ");
}
```

while, Not Until!

§ Remember to get the condition the right way around!

user probably intends “until j is equal to zero”, however this is NOT the way to write it

```
int j = 5;
printf("start\n");
while(j == 0)
    printf("j = %i\n", j--);
printf("end\n");
```

start
end



while, Not Until!

**There Are Only
“While”
Conditions in C**

One important thing to realize is that all of C's conditions are *while* conditions. The loops are executed *while* the condition is true.

do while

§ **do while** guarantees execution at least once

```
int j = 5;
printf("start\n");
do
    printf("j = %i\n", j--);
while(j > 0);
printf("stop\n");
```

```
start
j = 5
j = 4
j = 3
j = 2
j = 1
stop
```

```
int j = -10;
printf("start\n");
do {
    printf("j = %i\n", j);
    j--;
} while(j > 0);
printf("stop\n");
```

```
start
j = -10
stop
```

do while

The **do while** loop in C is an “upside down” version of the **while** loop. Whereas **while** has the condition followed by the body, **do while** has the body followed by the condition. This means the body must be executed before the condition is reached. Thus the body is guaranteed to be executed at least once. If the condition is false the loop body is never executed again.

for Loop

§ **for** encapsulates the essential elements of a loop into one statement

```
for(initial-part; while-condition; update-part)
    body;
```

```
int j;
for(j = 5; j > 0; j--)
    printf("j = %i\n", j);
```

```
j = 5
j = 4
j = 3
j = 2
j = 1
```

```
for(j = 5; j > 0; j--) {
    printf("j = %i ", j);
    printf("%s\n", ((j%2)==0)?"even":"odd");
}
```

```
j = 5 odd
j = 4 even
j = 3 odd
j = 2 even
j = 1 odd
```

for Loop

The **for** loop is syntactically the most complicated of C's 3 loops. Essentially though, it is similar to the **while** loop, it even has a while type condition. The C **for** loop is one of the most concise expressions of a loop available in any language. It brings together the starting conditions, the loop condition and all update statements that must be completed before the loop can be executed again.

for And while Compared

The construct:

```
for(initial-part; while-condition; update-part)
    body;
```

is equivalent to:

```
initial-part;
while(while-condition) {
    body;
    update-part;
}
```

Essentially all you need is to remember the two semicolon characters that must separate the three parts of the construct.

for Is Not Until Either!

§ Remember to get the `for` condition the right way around (it is really a *while* condition)

user probably intends "until j is equal to zero", however this is NOT the way to write it either!

```
int j;
printf("start\n");
for(j = 5; j == 0; j--)
    printf("j = %i\n", j);
printf("end\n");
```

start
end



for Is Not Until Either!

C Has While Conditions, Not Until Conditions

This slide is here to remind you once again there are no "until" conditions in C. Even though there are 3 kinds of loop, they all depend on while conditions - the loops continue *while* the conditions are true NOT *until* they become false.

The loop in the program above never really gets started. "j" is initialized with 5, then "j" is tested against zero. Since "j" is not zero, C jumps over the loop and lands on the `printf("end\n")` statement.

One point worth making is that the `for` is a cousin of the `while` not a cousin of the `do while`. Here we see, just like the `while` loop, the `for` loop body can execute zero times. With the `do while` loop the body is guaranteed to execute once.

Stepping With `for`

- § Unlike some languages, the `for` loop is not restricted to stepping up or down by 1

```
#include <math.h>

int main(void)
{
    double angle;

    for(angle = 0.0; angle < 3.14159; angle += 0.2)
        printf("sine of %.1lf is %.2lf\n",
               angle, sin(angle));

    return 0;
}
```

Stepping With `for`

Some languages, like Pascal and Ada, only allow for loops to step up or down by one. If you want to step by 2 you end up having to use a while construct. There is no similar restriction in C. It is possible to step up or down in whole or fractional steps.

Here the use of `+=` is illustrated to increment the variable “angle” by 0.2 each time around the loop.

`math.h`

This is the fourth Standard header file we have met. It contains declarations of various mathematical functions, particularly the sine (`sin`) function which is used in the loop.

Extending the `for` Loop

- § The initial and update parts may contain multiple comma separated statements

```
int i, j, k;
for(i = 0, j = 5, k = -1; i < 10; i++, j++, k--)
```

- § The initial, condition and update parts may contain no statements at all!

```
for( ; i < 10; i++, j++, k--)
```

```
for( ; i < 10; )
```

use of a while loop
would be clearer here!

```
for(;;)
```

creates an infinite loop

Extending the `for` Loop

The `for` loop would seem ideal only so long as one initial statement and one loop update statement are required. If two or more should need executing it would seem as though an alternative construct would be needed. However this is not the case, using the special comma operator, several statements may be executed in the initial and/or update parts of the loop.

The comma operator guarantees sequential execution of statements, thus “`i = 0`” is guaranteed to be executed before “`j = 5`” which is guaranteed to be executed before “`k = -1`”.

If you have no need for an initial or an update condition, leave the corresponding part of the loop empty, but remember the semicolon. In the example above:

```
for( ; i < 10; )
```

would probably be better replaced with:

```
while(i < 10)
```

Infinite Loops

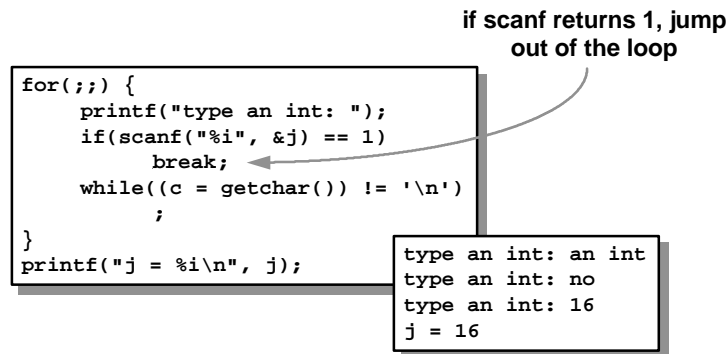
The strange looking construct:

```
for(;;)
```

creates an infinite loop and is read as “for ever”.

break

- § The **break** keyword forces immediate exit from the nearest enclosing loop
- § Use in moderation!



break

It must seem strange that C has a construct to deliberately create an infinite loop. Such a loop would seem something to avoid at all costs! Nonetheless it is possible to put infinite loops to work in C by jumping out of them. Any loop, no matter what the condition, can be jumped out of using the C keyword **break**.

We saw the loop below earlier:

```
printf("enter an integer: ");
while(scanf("%i", &j) != 1) {
    while((ch = getchar()) != '\n')
        ;
    printf("enter an integer: ");
}
```

This loop has the **printf** repeated. If the **printf** were a more complicated statement, prone to frequent change and the loop many hundreds of lines long, it may be a problem keeping the two lines in step. The **for(;;)** loop addresses this problem by having only one **printf**.

break is Really Goto!

It doesn't necessarily address the problem very well because it now uses the equivalent of a goto statement!

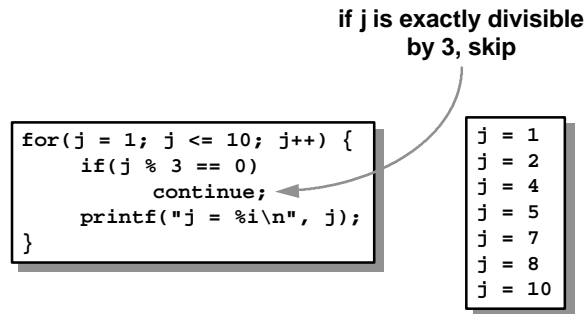
The goto is the scourge of modern programming, because of its close relationship some companies ban the use of **break**. If it is to be used at all, it should be used in moderation, overuse is liable to create spaghetti.

break, **switch** and Loops

This is exactly the same **break** keyword as used in the **switch** statement. If a **break** is placed within a **switch** within a loop, the **break** forces an exit from the **switch** and NOT the loop. There is no way to change this.

continue

- § The **continue** keyword forces the next iteration of the nearest enclosing loop
- § Use in moderation!



continue

Whereas **break** forces an immediate exit from the nearest enclosing loop the **continue** keyword causes the next iteration of the loop. In the case of **while** and **do while** loops, it jumps straight to the condition and re-evaluates it. In the case of the **for** loop, it jumps onto the update part of the loop, executes that, then re-evaluates the condition.

continue is Really Goto

Statements applying to the use of **break** similarly apply to **continue**. It is just another form of goto and should be used with care. Excessive use of **continue** can lead to spaghetti instead of code. In fact the loop above could just as easily be written as:

```
for(j = 1; j <= 10; j++)
    if(j % 3 != 0)
        printf("j = %i\n", j);
```

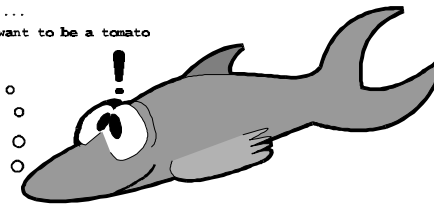
continue, switch and Loops

Whereas **break** has an effect on the switch statement, **continue** has no such effect. Thus a **continue** placed within a **switch** within a loop would effect the loop.

Summary

- ⌘ **if (then) else** - watch the semicolons
- ⌘ **switch** can test integer values
- ⌘ **while, do while, for** - watch the semicolons again
- ⌘ **break**
- ⌘ **continue**

```
else ...  
I want to be a tonato
```



Summary

Control Flow Practical Exercises

Directory: **FLOW**

1. Write a program in "**QUANT.C**" which "quantifies" numbers. Read an integer "x" and test it, producing the following output:

x greater than or equal to 1000	print "hugely positive"
x from 999 to 100 (including 100)	print "very positive"
x between 100 and 0	print "positive"
x exactly 0	print "zero"
x between 0 and -100	print "negative"
x from -100 to -999 (including -100)	print "very negative"
x less than or equal to -1000	print "hugely negative"

Thus -10 would print "negative", -100 "very negative" and 458 "very positive".

2. Cut and paste your **AREA**, **RADIUS** and **VOL** programs into a file called "**CIRC.C**" which accepts four options. The option 'A' calculates the area of a circle (prompting for the radius), the option 'C' calculates the circumference of a circle (prompting for the radius), the option 'V' calculates the volume of a cylinder (prompting for the radius and height), while the option 'Q' quits the program.

The program should loop until the quit option is chosen.

3. Improve the error checking of your "**CIRC**" program such that the program will loop until the user enters a valid real number.
4. Write a program in "**POW.C**" which reads two numbers, the first a real, the second an integer. The program then outputs the first number raised to the power of the second.

Before you check, there is no C operator to raise one number to the power of another. You will have to use a loop.

5. Write a program in "**DRAWX.C**" which draws an "x" of user specified height. If the user typed 7, the following series of '*' characters would be drawn (without the column and line numbers):

	1	2	3	4	5	6	7
1	*						*
2		*				*	
3			*		*		
4				*			
5			*		*		
6		*				*	
7	*						*

and if the user typed 6 would draw '*' characters as follows:

	1	2	3	4	5	6
1	*					*
2		*			*	
3			*	*		
4			*	*		
5		*			*	
6	*					*

6. Write a program in "**BASES.C**" which offers the user a choice of converting integers between octal, decimal and hexadecimal. Prompt the user for either 'o', 'd' or 'h' and read the number in the chosen format. Then prompt the user for the output format (again 'o', 'd' or 'h') and print the number out accordingly.

A nice enhancement would be to offer the user only the *different* output formats, i.e. if 'o' is chosen and an octal number read, the user is offered only 'd' and 'h' as output format.

Control Flow Solutions

1. Write a program in “**QUANT.C**” which “quantifies” numbers. Read an integer “x” and test it, producing the following output:

x greater than or equal to 1000	print “hugely positive”
x from 999 to 100 (including 100)	print “very positive”
x between 100 and 0	print “positive”
x exactly 0	print “zero”
x between 0 and -100	print “negative”
x from -100 to -999 (including -100)	print “very negative”
x less than or equal to -1000	print “hugely negative”

Thus -10 would print “negative”, -100 “very negative” and 458 “very positive”.

In the following solution the words “very” and “hugely” are printed separately from “positive” and “negative”.

```
#include <stdio.h>

int main(void)
{
    int    i;

    printf("Enter an integer ");
    scanf("%i", &i);

    if(i >= 1000 || i <= -1000)
        printf("hugely ");
    else if(i >= 100 || i <= -100)
        printf("very ");

    if(i > 0)
        printf("positive\n");
    else if(i == 0)
        printf("zero\n");
    else if(i < 0)
        printf("negative\n");

    return 0;
}
```

2. Cut and paste your **AREA**, **RADIUS** and **VOL** programs into a file called “**CIRC.C**” which accepts four options. The program should loop until the quit option is chosen.

3. Improve the error checking of your “CIRC” program such that the program will loop until the user enters a valid real number.

*Notice the getchar loop to discard unread input. When the character is entered via getchar, or the real number read via scanf, the return key is **saved** (it is buffered as we shall see in a later chapter). Although this doesn't cause a problem the first time through the loop, it does cause a problem the second and subsequent times.*

The value returned by scanf is important. When told to read one thing (as with “%Lf”) scanf returns one on success, if the input is not in the correct format zero is returned. If this is the case, the getchar loop is entered to discard this unwanted input.

```
#include <stdio.h>

int main(void)
{
    int          ch;
    int          still_going = 1;
    long double  radius = 0.0L;
    long double  answer = 0.0L;
    long double  height = 0.0L;
    const long double pi = 3.1415926353890L;

    while(still_going) {

        printf( "Area          A\n"
               "Circumference C\n"
               "Volume         V\n"
               "Quit           Q\n\n"
               "Please choose ");

        ch = getchar();

        if(ch == 'A' || ch == 'C' || ch == 'V') {
            do {
                printf("please give the radius ");
                while(getchar() != '\n')
                    ;
            }
            while(scanf("%Lf", &radius) != 1);
        }
        if(ch == 'V') {
            do {
                printf("please give the height ");
                while(getchar() != '\n')
                    ;
            }
            while(scanf("%Lf", &height) != 1);
        }
        if(ch != 'A' && ch != 'C' && ch != 'V')
            while(getchar() != '\n')
                ;

        if(ch == 'A') {
            answer = pi * radius * radius;
            printf("Area of circle with radius %.3Lf is %.12Lf\n",
                  radius, answer);
        }
    }
}
```

```

    } else if(ch == 'C') {
        answer = 2.0 * pi * radius;
        printf("Circumference of circle with radius "
               "%.3Lf is %.12Lf\n",
               radius, answer);

    } else if(ch == 'V') {
        answer = pi * radius * radius * height;
        printf("Volume of cylinder with radius "
               "%.3Lf and height %.3Lf is %.12Lf\n",
               radius, height, answer);

    } else if(ch == 'Q')
        still_going = 0;
    else
        printf("Unknown option '%c' ignored\n\n", ch);
}
return 0;
}

```

4. Write a program in “**POW.C**” which reads two numbers, the first a real, the second an integer. The program then outputs the first number raised to the power of the second.

Careful consideration must be given to the initialization of “answer” and the loop condition “count < p” in the program below. Initializing “answer” with zero and or a loop condition of “count <= p” would have yielded very different (i.e. wrong) results.

```

#include <stdio.h>

int main(void)
{
    int          count = 1;
    int          p = 0;
    double       n = 0.0L;
    long double  answer = 0.0L;

    printf("enter the number ");
    scanf("%lf", &n);

    printf("enter the power ");
    scanf("%d", &p);

    for(answer = n; count < p; count++)
        answer = answer * n;

    printf("%.3lf to the power of %d is %.9Lf\n", n, p, answer);

    return 0;
}

```

5. Write a program in "DRAWX.C" which draws an "x" of user specified height.

Drawing the top left/bottom right diagonal is easy since '' occurs when the row and column numbers are equal. For the other diagonal, '*' occurs when the column number is equal to the height less the row number plus one.*

```
#include <stdio.h>

int main(void)
{
    int    height;
    int    row;
    int    column;

    printf("Enter height of 'x' ");
    scanf("%i", &height);

    for(row = 1; row <= height; row++) {
        for(column = 1; column <= height; column++) {
            if(row == column || column == height - row + 1)
                printf("*");
            else
                printf(" ");
        }
        printf("\n");
    }

    return 0;
}
```

6. Write a program in “**BASES.C**” which offers the user a choice of converting integers between octal, decimal and hexadecimal. Prompt the user for either ‘o’, ‘d’ or ‘h’ and read the number in the chosen format. Then prompt the user for the output format (again ‘o’, ‘d’ or ‘h’) and print the number out accordingly.

A nice enhancement would be to offer the user only the *different* output formats, i.e. if ‘o’ is chosen and an octal number read, the user is offered only ‘d’ and ‘h’ as output format.

```
#include <stdio.h>

int main(void)
{
    int    input;
    int    i_option;
    int    o_option;
    int    keep_going;

    do {
        printf("Input options:\n"
               "Octal input      o\n"
               "Decimal input    d\n"
               "Hexadecimal input x      ");

        i_option = getchar();

        keep_going = 0;

        switch(i_option) {
            case 'o':
                printf("enter octal number ");
                scanf("%o", &input);
                break;
            case 'd':
                printf("enter decimal number ");
                scanf("%d", &input);
                break;
            case 'x':
                printf("enter hexadecimal number ");
                scanf("%x", &input);
                break;
            default:
                keep_going = 1;
                break;
        }
        while(getchar() != '\n')
            ;
    }
```

```
    } while(keep_going);

do {
    if(i_option != 'o')
        printf("\nOctal output      o");

    if(i_option != 'd')
        printf("\nDecimal output    d");

    if(i_option != 'x')
        printf("\nHexadecimal output x");

    printf("      ");

    o_option = getchar();
    while(getchar() != '\n')
        ;

    keep_going = 0;

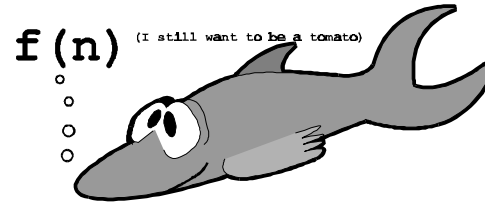
    switch(o_option) {
        case 'o':
            printf("%o\n", input);
            break;
        case 'd':
            printf("%d\n", input);
            break;
        case 'x':
            printf("%x\n", input);
            break;
        default:
            keep_going = 1;
            break;
    }
} while(keep_going);

return 0;
}
```

Functions

Functions

- § Rules of functions
- § Examples - writing a function, calling a function
- § Function prototypes
- § Visibility
- § Call by value
- § The stack
- § auto, static and register



Functions

This chapter examines functions in C.

The Rules

- § A function may accept as many parameters as it needs, or no parameters (like `main`)
- § A function may return either one or no values
- § Variables declared inside a function are only available to that function, unless explicitly passed to another function

The Rules

Functions in C may take as many parameters as they need. An example of this is `printf` which may take an arbitrary number of parameters, here 7:

```
printf("%i %.2lf %.2lg %c %u %o\n", j, f, g, c, pos, oct);
```

Alternatively functions may take no parameters at all, like `main`

```
int main(void)
```

A function may either return a value or not. In particular it may return ONE value or not.

C is a block structured language and variables declared within a function block may only be used within that block.

Writing a Function - Example

this is the TYPE of the value handed back

accept 3 doubles when called

```
int print_table(double start, double end, double step)
{
    double d;
    int lines = 1;

    printf("Celsius\tFahrenheit\n");
    for(d = start; d <= end; d += step, lines++)
        printf("%.11f\t%.11f\n", d, d * 1.8 + 32);

    return lines;
}
```

this is the ACTUAL value handed back

Writing a Function - Example

There are a number of essential elements involved in writing functions:

- | | |
|----------------------|---|
| Return Type | If a function is to return a value, the value must have a type. The type of the return value must be specified first. |
| Function Name | Obviously each function must have a unique name to distinguish it from the other functions in the program. |
| Parameters | A type and a name must be given to each parameter. |
| Return Value | If the function is to return a value, the actual value (corresponding to the type already specified) must be passed back using the return keyword. |
-

Calling a Function - Example

IMPORTANT: this tells the compiler how print_table works

```
#include <stdio.h>

int print_table(double, double, double);

int main(void)
{
    int    how_many;
    double end = 100.0;

    how_many = print_table(1.0, end, 3);
    print_table(end, 200, 15);

    return 0;
}
```

the compiler knows these
should be doubles and
converts them automatically

here the function's return value is ignored - this
is ok, if you don't want it, you don't have to use it

Calling a Function - Example

There are a number of essential elements when calling functions:

Prototype

A prototype informs the compiler how a function works. In this case:

```
int print_table(double, double, double);
```

tells the compiler that the print_table function accepts three doubles and returns an integer.

Call

The function is called (executed, run) by sending three parameters, as in:

```
print_table(1.0, end, 3);
```

even though the third parameter "3" is not of the correct type it is automatically converted by the compiler. If necessary the returned value may be assigned to a variable as in:

```
how_many = print_table(1.0, end, 3);
```

Ignoring the Return

It is not necessary to use the returned value as in:

```
print_table(end, 200, 15);
```

any return value that is not used is discarded. Note that here the 200 and the 15 are automatically converted from int to double.

Calling a Function - Disaster!

now the compiler does not know how the function works

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int how_many;
```

```
    double end = 100.0;
```

```
    how_many = print_table(1.0, end, 3);
```

```
    print_table(end, 200, 15);
```

```
    return 0;
```

```
}
```

the compiler does NOT
convert these ints to
doubles. The function
picks up doubles
anyway!



Calling a Function - Disaster!

Missing Prototypes

The only difference between this and the previous example is the lack of the prototype:

```
int print_table(double, double, double);
```

This missing line causes serious problems. Now the compiler does not have the information it needs at the two points of call:

```
how_many = print_table(1.0, end, 3);
```

and

```
print_table(end, 200, 15);
```

The compiler assumes that all the parameters are correct. Thus the third parameter "3" is NOT converted from an integer to a double. Neither are the "200" or the "15". This is a major problem since integers and doubles are not the same size and not the same format.

When the function picks up the parameters they are not what was intended and the function behaves strangely.

Prototypes

§ The (optional) line

```
int    print_table(double, double, double);
```

is known as a *prototype*

§ If the compiler meets a call to an unknown function it “guesses”

- Guess 1: the function returns an int, even if it doesn't
- Guess 2: you have passed the correct number of parameters and made sure they are all of the correct type, even if you haven't

§ The prototype provides the compiler with important information about the return type and parameters

Prototypes

The all important missing line is called the function “prototype”. The compiler needs one of these for every single function called from your program. If you forget to provide a prototype and go ahead and call a function anyway, the compiler will assume some defaults.

When a Prototype is Missing

First: the compiler will assume the function returns an integer. This is rather curious. A safer assumption might be that the function returns nothing, that way any attempt to use the returned value would generate an error. However, `int` it is. This is a special problem with the mathematical functions, `sin`, `cos`, `tan`, etc. which return `double`. By not prototyping these functions the compiler incorrectly truncates the `doubles`, using only 2 or 4 bytes of the 8 bytes that are returned.

Second: the compiler assumes that the correct number of parameters have been passed and that the type of each one is correct. This was clearly not the case in the previous program, whereas the number of parameters was correct, although the types were not.

Prototyping is Not Optional

- § To achieve working programs the compiler is best given a prototype for each function called
- § When calling a Standard Library function, `#include` the file specified in the help page(s) - this file will contain the prototype
- § When calling one of your own functions, write a prototype by hand

Prototyping is Not Optional

Since prototypes can make such a difference to whether a program works, it is curious C regards them as optional. Even though this is the case, we should regard them as compulsory.

We must ensure that *each* function called is properly prototyped. This is more straightforward than it sounds since most C compilers come equipped with a “warning level”. Although the compiler will not complain if a call is made to an unprototyped function at a low warning level, turning the warning level up does cause a message to appear. Various programming standards employed by large software houses state that programs should compile without a single warning at the highest warning level.

Calling Standard Library Functions

If we wish to call a Standard Library function, a prototype for it will already have been written and made available in one of the Standard header files. All we need to do is `#include` the relevant header file - its name will be given us by the on-line help or text manual.

If we wish to call one of our own functions, we must write a prototype by hand.

Writing Prototypes

§ Prototype:

```
int print_table(double, double, double);
```

§ Function header:

```
int print_table(double start, double end, double step)
{
```

§ The function prototype may optionally include variable names (which are ignored)

```
int print_table(double start, double end, double step);
```

```
int print_table(double x, double y, double z);
```

Writing Prototypes

Convert The Function Header Into The Prototype

Writing a function involves writing the function header. Once that's been done only a quick "cut and paste" is necessary to create the function prototype. You can either slice out the whole header, including the names of the parameters, or edit them out.

Parameter Names Ignored

In fact the compiler completely ignores parameter names in function prototypes, if provided, the names don't have to relate to the ones used in the function itself. The example above shows the `print_table` prototype using the name "start" or "x" for its first parameter. Either of these are ok, even if the name of the first parameter in fact turns out to be "artichoke".


Added Documentation

In fact this begs the question as to why parameter names should be put in at all, if the compiler is just going to ignore them. The answer is that a function prototype which includes *meaningful* names is far more helpful than one which does not. The names "start", "stop" and "end" provide meaning and save us having to find either the code for `print_table` or the manual which describes it.

Take Care With Semicolons


- ❗ The prototype has a semicolon

```
int print_table(double start, double end, double step);
```




- ❗ The function header has an open brace

```
int print_table(double start, double end, double step)
```



- ❗ Don't confuse the compiler by adding a semicolon into the function header!

```
int print_table(double start, double end, double step);  
{
```



Take Care With Semicolons

Avoid Semicolons After The Function Header

We have seen that the prototype can be an almost exact copy of the function header. If they are so similar, how exactly does the compiler tell them apart? It is all done by the character that follows the closing parenthesis. If that character is an opening brace, the compiler knows the text forms the function header, if the character is a semicolon, the compiler knows the text forms the function prototype.

Adding a semicolon into the function header is particularly fatal. Meeting the semicolon first, the compiler assumes it has met the function prototype. After the prototype comes the beginning of a block, but what block?

Example Prototypes

```
/* no parameters, int return value */
int get_integer(void);

/* no parameters, double return value */
double get_double(void);

/* no parameters, no return value */
void clear_screen(void);

/* three int parameters, int return value */
int day_of_year(int day, int month, int year);

/* three int parameters, long int return value */
long day_since_1_jan_1970(int, int, int);

/* parameter checking DISABLED, double return value */
double k_and_r_function();

/* short int parameter, (default) int return value */
transfer(short int s);
```

Example Prototypes

Above are examples of function prototypes. Notice that **void** must be used to indicate the absence of a type. Thus in:

```
int get_integer(void);
```

void for the parameter list indicates there are no parameters. This is NOT the same as saying:

```
int get_integer();
```

which would have the effect of *disabling* parameter checking to the `get_integer` function. With this done, far from passing no parameters into the function, any user could pass two, fourteen or fifty parameters with impunity!

C makes no distinction between functions (lumps of code that return a value) and procedures (lumps of code that execute, but return no value) as do languages like Pascal. In C there are just functions, functions which return things and functions which don't return anything. An example of a prototype for a function which does not return anything is:

```
void clear_screen(void);
```

The first **void** indicates no return value, the second indicates (as before) no parameters.

The `day_of_year` and `day_since_1_jan_1970` function prototypes indicate the difference between naming parameters and not. With the `day_of_year` function it is obvious that the day, month and year must be provided in that order without resorting to any additional documentation. If `day_since_1_jan_1970` were written by an American, the month might be required as the first parameter. It is impossible to tell without further recourse to documentation.

The prototype for the `transfer` function demonstrates a rather curious C rule. If the return type is omitted, **int** is assumed.

Example Calls

```

int      i;
double   d;
long     l;
short int s = 5;

i = get_integer();
d = get_double();
clear_screen();

i = day_of_year(16, 7, 1969);
l = day_since_1_jan_1970(1, 4, 1983);

d = k_and_r_function();
d = k_and_r_function(19.7);
d = k_and_r_function("hello world");

i = transfer(s);

```

no mention of "void"
when calling these
functions

the compiler cannot tell
which of these (if any) is
correct - neither can we
without resorting to
documentation!

Example Calls

The most important thing to realize is that when calling a function with a prototype like

```
int get_integer(void);
```

it is NOT correct to say: `i = get_integer(void);`

whereas it IS correct to say: `i = get_integer();`

The compiler just doesn't expect `void` at the point of call.

The examples above also illustrate a call to `clear_screen`. If you thought it would be pointless to call a function which takes no parameters and returns no value, here is an example. The `clear_screen` function does not need to be passed a parameter to tell it how many times to clear the screen, just once is enough. Similarly it does not need to return an integer to say whether it succeeded or failed. We assume it succeeds.


It is difficult to say what date `day_since_1_jan_1970` is dealing with in the code above, it could be the 1st of April, or just as easily the 4th of January.


Rules of Visibility

§ C is a block structured language, variables may only be used in functions declaring them

```
int main(void)
{
    int i = 5, j, k = 2;
    float f = 2.8F, g;
    d = 3.7;
}

void func(int v)
{
    double d, e = 0.0, f;
    i++; g--;
    f = 0.0;
}
```

 compiler does not know about "d"

 "i" and "g" not available here

func's "f" is used, not main's

Rules of Visibility

C is a Block Structured Language

Variables allocated within each function block may only be used within that function block. In fact C allows variables to be allocated wherever an opening brace is used, for instance:

```
void func(int v)
{
    double d, e = 0.0, f;

    if(e == 0.0) {
        int i, j = 5;

        i = j - 1;
        printf("i=%i, e=%lg\n", i, e);
    }
    d = 0.1;
}
```

The two variables "i" and "j" are created only in the *then* part of the **if** statement. If the variable "e" didn't compare with zero, these variables would never be created. The variables are only available up until the "}" which closes the block they are allocated in. An attempt to access "i" or "j" on or after the line "**d = 0.1**" would cause an error.

The variables "d", "e" and "f" are all available within this "if" block since the block lives inside the function block.

Call by Value

- § When a function is called the parameters are ***copied*** - “call by value”
- § The function is unable to change any variable passed as a parameter
- § In the next chapter ***pointers*** are discussed which allow “call by reference”
- § We have already had a sneak preview of this mechanism with `scanf`

Call by Value

C is a “call by value” language. Whenever a parameter is passed to a function a *copy* of the parameter is made. The function sees this copy and the original is protected from change.

This can be a advantage and a disadvantage. If we wanted a `get_current_date` function, for instance, we would want three “returns”, the day, month and year, but functions may only return one value. Three functions `get_current_day`, `get_current_month` and `get_current_year` would be needed. Clearly this is inconvenient!

In fact, C supports “call by reference” too. This is a mechanism by which the parameter becomes not a copy of the variable but its address. We have already seen this mechanism with `scanf`, which is able to alter its parameters easily. This is all tied up with the use of the mysterious “&” operator which will be explained in the next chapter.

Call by Value - Example

```
#include <stdio.h>
void change(int v);
int main(void)
{
    int var = 5;
    change(var);
    printf("main: var = %i\n", var);
    return 0;
}
void change(int v)
{
    v *= 100;
    printf("change: v = %i\n", v);
}
```

the function
was not able
to alter "var"

the function is
able to alter "v"

change: v = 500
main: var = 5

Call by Value - Example

This program shows an example of call by value. The **main** function allocates a variable "var" of type **int** and value 5. When this variable is passed to the **change** function a copy is made. This copy is picked up in the parameter "v". "v" is then changed to 500 (to prove this, it is printed out). On leaving the **change** function the parameter "v" is thrown away. The variable "var" still contains 5.

C and the Stack

☞ **C uses a stack to store local variables (i.e. those declared in functions), it is also used when passing parameters to functions**

The calling function pushes the parameters

The function is called

The called function picks up the parameters

The called function pushes its local variables

When finished, the called function pops its local variables and jumps back to the calling function

The calling function pops the parameters

The return value is handled

C and the Stack

C is a stack-based language. Conceptually a stack is like a pile of books. New books must be added to the pile only at the top. If an attempt is made to add a new book to the middle of the pile, the whole thing will collapse. Similarly when books are removed from the pile, they must only be removed from the top since removing one from the middle or bottom of the pile would cause a collapse.

Thus: a stack may only have a new item added to the top
 a stack may only have an existing item removed from the top

You can imagine that while the books are in the pile, the spines of the books (i.e. the title and author) could be easily read. Thus there is no problem *accessing* items on the stack, it is only the addition and removal of items which is rigorously controlled.

The list above shows the rules that C employs when calling functions. When a variable is passed as a parameter to a function a copy of the variable is placed on the stack. The function picks up this copy as the parameter. Since the function may only access the parameter (because the original variable was allocated in another function block) the original variable *cannot* be changed.

If a function allocates any of its own variables, these too are placed on the stack. When the function finishes it is responsible for destroying these variables.

When the function returns to the point of call, the calling function destroys the parameters that it copied onto the stack.

Stack Example

```
#include <stdio.h>
double power(int, int);
int main(void)
{
    int    x = 2;
    double d;

    d = power(x, 5);
    printf("%lf\n", d);

    return 0;
}
double power(int n, int p)
{
    double result = n;
    while(--p > 0)
        result *= n;

    return result;
}
```

32.0	power: result
2	power: n
5	power: p
?	main: d
2	main: x

Stack Example

When the **main** function starts, it allocates storage for two variables “x” and “d”. These are placed on the bottom of the otherwise empty stack. When **main** calls the **power** function as in

```
d = power(x, 5);
```

“5” is copied onto the stack, then the value of “x”, which is “2”. The power function is called. It immediately picks up two parameters “n” and “p”. The “n” happens to be where the “2” was copied, the “p” where the “5” was copied.

The function requires its own local variable “result” which is placed on the stack above “n” and is initialized with the value “2.0”. The loop executes 4 times, multiplying result by 2 each time. The value of “p” is now zero. The value stored in “result” by this stage is “32.0”. This value is returned. Different compilers have different strategies for returning values from functions. Some compilers return values on the stack, others return values in registers. Some do both depending on wind direction and phase of the moon. Let us say here that the return value is copied into a handy register.

The **return** keyword causes the **power** function to finish. Before it can truly finish, however, it is responsible for the destruction of the variable “result” which it created. This is removed (popped) from the stack.

On return to the **main** function, the two parameters “n” and “p” are destroyed. The return value, saved in a handy register is transferred into the variable “d” (which is then printed on the next line).

The **return 0** causes **main** to finish. Now “0” is stored in a handy register, ready for the operating system to pick it up. Before things can truly finish, **main** must destroy its own variables “x” and “d”.

Storage

- § **C stores local variables on the stack**
- § **Global variables may be declared. These are not stack based, but are placed in the data segment**
- § **Special keywords exist to specify where local variables are stored:**
 - `auto` - place on the stack (default)
 - `static` - place in the data segment
 - `register` - place in a CPU register
- § **Data may also be placed on the heap, this will be discussed in a later chapter**

Storage

When a program is running in memory it consists of a number of different parts:

- | | |
|---------------------|--|
| Code Segment | This is where all the code lives, <code>main</code> , <code>printf</code> , <code>scanf</code> etc. etc. This segment is definitely <i>read only</i> (otherwise you could write self-modifying code!). |
| Stack | This is where all the local variables are stored. We have seen how it becomes deeper as functions are called and shallower as those functions return. The <i>stack alters size continuously</i> during the execution of a program and is definitely NOT read only. |
| Data Segment | This is a <i>fixed sized</i> area of the program where global variables are stored. Since global variables in a program are always there (not like local variables which are created and destroyed) there are always a fixed number of fixed sized variables - thus the data segment is fixed size. |
| Heap | The last and strangest part of the executing program, the heap, can vary in size during execution. Its size is controlled by calls to the four <i>dynamic memory allocation</i> routines that C defines: <code>malloc</code> , <code>calloc</code> , <code>realloc</code> and <code>free</code> . The heap, and these routines, are discussed later in the course. |

Local variables which have been seen thus far have been stack based. Global variables may also be created (though we have not yet seen how this is done). Mixed in with locals and globals these notes have also hinted at the occasional use of registers.

In fact keywords exist in C which allow us to control where local variables are placed - on the stack (which they are by default), in the data segment along with the global variables, or in registers.

auto

- § Local variables are automatically allocated on entry into, and automatically deallocated on exit from, a function
- § These variables are therefore called “automatic”
- § Initial value: random
- § Initialisation: recommended

```
int table(void)
{
    int      lines = 13;
    auto int  columns;
```

auto keyword
redundant

auto

Stack Variables are “Automatic”

To be honest, C’s **auto** keyword is a complete waste of space. Local variables are, by default, placed on the stack. When a function starts, stack storage is allocated. When the function ends, the stack storage is reclaimed. Since this happens totally automatically, stack based variables are called “automatic” variables.

“**int columns**” and “**auto int columns**” are exactly identical. In other words the **auto** keyword does nothing, it makes the automatic variable automatic (which it is anyway).

Stack Variables are Initially Random

An important thing to understand about automatic variables is although the compiler is happy to allocate storage from the stack it will NOT initialize the storage (unless explicitly instructed to do so). Thus automatic variables initially contain whatever that piece of stack last contained. You may see a quarter of a double, half of a return address, literally anything. Certainly whatever it is will make little sense. The upshot is that if you need a value in an automatic variable (including zero) it is vital to put that value in there by assignment.

Performance

It is possible to imagine a scenario where a function is called, say one thousand times. The function allocates one stack based variable. Thus one thousand times the variable must be created, one thousand times the variable must be destroyed. If you are worried about the last nanosecond of performance, that may be something you might want to worry about.

static

- § The **static** keyword instructs the compiler to place a variable into the data segment
- § The data segment is *permanent* (static)
- § A value left in a **static** in one call to a function will still be there at the next call
- § Initial value: 0
- § Initialisation: unnecessary if you like zeros

```
int running_total(void)
{
    static int rows;
    rows++;
}
```

permanently allocated,
but local to this
function

static

static Variables are Permanent

By default, a variable is stack based, random and continually goes through an automatic creation and destruction process whenever the function declaring it is called. Adding **static** into a variable declaration causes the variable to be stored in the data segment. This is the same part of the program where the global variables are stored. Thus the variable is *permanently* allocated.

static Variables are Initialized

This means the first time **running_total** is called, the storage for the variable “rows” has already been allocated. It has also already been initialized (to zero). If a value of 1 is left in the variable and the function returns, the next time the function is called the 1 will be seen. If 2 is left in the variable, the next time the 2 will be seen, etc. Since there is no creation and destruction a function containing one static variable should execute faster than one having to allocate and deallocate a stack based one.

static Variables Have Local Scope

Although the variable is permanently allocated, its scope is local. The “rows” variable cannot be seen outside the **running_total** function. It is perfectly possible to have two static variables of the same name within two different functions:

```
int func1(void)          int func2(void)
{
    static int i = 30;    {
                          static int i = -30;
                          i++;
    }                    i--;
}
```

The variable “i” in the function **func1** will steadily increase every time the function is called. The variable “i” in the function **func2** will steadily decrease. These two variables are permanent, separate and inaccessible by the other function.

register

- ⌘ The `register` keyword tells the compiler to place a variable into a CPU register (you cannot specify which)
- ⌘ If a register is unavailable the request will be ignored
- ⌘ Largely redundant with optimising compilers
- ⌘ Initial value: random
- ⌘ Initialisation: recommended

```
void speedy_function(void)
{
    register int i;
    for(i = 0; i < 10000; i++)
```

register

The `register` keyword *requests* a variable be placed in a CPU register. The compiler is under no obligation to satisfy this request. The keyword goes back to K&R C, when there were no optimizers. Thus various optimization features were added to the language itself.

`register` Variables are Initially Random

If a register is available, it will be allocated. However, C will not clear it out, thus it will contain whatever value happened to be in there previously.

Slowing Code Down

Optimizers for C have now been written and these are best left to decide which variables should be placed in registers. In fact is it possible to imagine a scenario where code actually runs *slower* as a result of the use of this keyword.

Imagine the best strategy for optimizing a function is to place the first declared variable "i" into a CPU register. This is done for the first 10 lines of the function, then the variable "j" becomes the one most frequently used and thus "i" is swapped out of the register and "j" swapped in. The programmer tries to optimize and places "i" into a register. If only one register is available and the optimizer feels obliged to satisfy the request the second part of the function will run more slowly (since "j" needs to be placed in a register, but cannot).

The optimizer is almost certainly better at making these decisions (unless you have written the optimizer and you know how it works) and should be left to its own devices.

Global Variables

- § **Global variables are created by placing the declaration outside all functions**
- § **They are placed in the data segment**
- § **Initial value: 0**
- § **Initialisation: unnecessary if you like zeros**

```
#include <stdio.h>
double d;
int main(void)
{
    int i;
    return 0;
}
```

variable "d" is global
and available to all
functions defined
below it

Global Variables

You should be aware that many programming standards for C ban the use of global variables. Since access to a global variable is universal and cannot be controlled or restricted it becomes difficult to keep track of who is modifying it and why.

Nonetheless global variables may be easily created, by placing the variable declaration outside any function. This places the variable in the data segment (along with all the static local variables) where it is permanently allocated throughout the execution of the program.

Global Variables are Initialized

Just as with static local variables, initialization to zero is performed by the operating system when the program is first loaded into memory.

Review

- ⌘ **Writing and calling functions**
- ⌘ **The need for function prototypes**
- ⌘ **Visibility**
- ⌘ **C is “call by value”**
- ⌘ **Local variables are stack based, this can be changed with the `static` and `register` keywords**
- ⌘ **Global variables may be created, they are stored in the data segment**

Review Questions

1. Which two characters help the compiler determine the difference between the function prototype and the function header?
 2. What is automatic about an automatic variable?
 3. What is the initial value of a register variable?
 4. What are the names of the four parts of an executing program?
-

Functions Practical Exercises

Directory: **FUNCS**

1. By now you have probably experienced problems with **scanf** insofar as when an invalid character is typed things go drastically wrong. In **"GETVAL.C"** write and test two functions:

```
double  get_double(void);
int     get_int(void);
```

which loop, prompting the user, until a valid double/valid integer are entered.

2. Copy **"POW.C"** from the **FLOW** directory. Turn your power calculation into a function with the following prototype:

```
long double  power(double first, int second);
```

Use your **get_double** and **get_int** functions from part 1 to read the double and integer from the user.

3. Copy **"CIRC.C"** from the **FLOW** directory. Write functions with the following prototypes:

```
double  volume(double radius, double height);
double  area(double radius);
double  circumf(double radius);
char    get_option(void);
```

Use the **get_double** function written in part 1 to read the radius (and height if necessary). The **get_option** function should accept only 'a', 'A', 'c', 'C', 'v', 'V', 'q' or 'Q' where the lowercase letters are the same as their uppercase equivalents.

If you **#include <ctype.h>**, you will be able to use the **tolower** function which converts uppercase letters to their lowercase equivalent. This should make things a little easier. Look up **tolower** in the help.

Functions Solutions

1. In "GETVAL.C" write and test two functions:

```
double  get_double(void);
int     get_int(void);
```

which loop, prompting the user, until a valid double/valid integer are entered.

```
#include <stdio.h>

int  get_int(void);
double get_double(void);

int  main(void)
{
    int i;
    double d;

    printf("type an integer ");
    i = get_int();
    printf("the integer was %i\n", i);

    printf("type an double ");
    d = get_double();
    printf("the double was %lg\n", d);

    return 0;
}

int  get_int(void)
{
    int result;

    printf("> ");
    while(scanf("%i", &result) != 1) {
        while(getchar() != '\n')
            ;
        printf("> ");
    }

    return result;
}

double get_double(void)
{
    double result;

    printf("> ");
    while(scanf("%lf", &result) != 1) {
        while(getchar() != '\n')
            ;
        printf("> ");
    }

    return result;
}
```

2. Copy "POW.C" from the FLOW directory. Turn your power calculation into a function. Use your `get_double` and `get_int` functions from part 1.

In the function "power" the parameter may be treated "destructively" since call by value is used, and altering the parameter will have no effect on the main program.

```
#include <stdio.h>

int      get_int();
double   get_double();
long double power(double, int);

int  main(void)
{
    int    p = 0;
    double n = 0.0;

    printf("enter the number ");
    n = get_double();

    printf("enter the power ");
    p = get_int();

    printf("%.3lf to the power of %d is %.9Lf\n", n, p, power(n, p));

    return 0;
}

long double power(double n, int p)
{
    long double answer = n;

    for(--p; p > 0; p--)
        answer *= n;

    return answer;
}

int  get_int(void)
{
    int    result;

    printf("> ");
    while(scanf("%i", &result) != 1) {
        while(getchar() != '\n')
            ;
        printf("> ");
    }

    return result;
}
```

```
double get_double(void)
{
    double result;

    printf("> ");
    while(scanf("%lf", &result) != 1) {
        while(getchar() != '\n')
            ;
        printf("> ");
    }

    return result;
}
```

3. Copy "CIRC.C" from the FLOW directory. Write functions with the following prototypes....

Use the `get_double` function written in part 1 to read the radius (and height if necessary). The `get_option` function should accept only 'a', 'A', 'c', 'C', 'v', 'V', 'q' or 'Q' where the lowercase letters are the same as their uppercase equivalents.

Using `tolower` should make things a little easier.

The version of `get_double` used here differs slightly from previous ones. Previously, if a double was entered correctly the input buffer was not emptied. This causes `scanf("%c")` in the `get_option` function to read the newline left behind in the input buffer (`getchar` would do exactly the same). Thus whatever the user types is ignored. This version always flushes the input buffer, regardless of whether the double was successfully read.

```
#include <stdio.h>
#include <ctype.h>

double    get_double(void);
double    area(double radius);
double    circumf(double radius);
double    volume(double radius, double height);
char      get_option(void);

const double    pi = 3.1415926353890;

int main(void)
{
    int    ch;
    int    still_going = 1;
    double radius = 0.0;
    double height = 0.0;

    while(still_going) {

        ch = get_option();

        if(ch == 'a' || ch == 'c' || ch == 'v') {
            printf("enter the radius ");
            radius = get_double();
        }
        if(ch == 'v') {
            printf("enter the height ");
            height = get_double();
        }
    }
}
```

```
        if(ch == 'a')
            printf("Area of circle with radius %.3lf is %.12lf\n",
                    radius, area(radius));
        else if(ch == 'c')
            printf("Circumference of circle with radius "
                    "%.3lf is %.12lf\n", radius, circumf(radius));
        else if(ch == 'v')
            printf("Volume of cylinder radius %.3lf, height %.3lf "
                    "is %.12lf\n", radius, height,
                    volume(radius, height));
        else if(ch == 'q')
            still_going = 0;
        else
            printf("Unknown option '%c'\n\n", ch);
    }
    return 0;
}

double    get_double(void)
{
    int    got;
    double result;

    do {
        printf("> ");
        got = scanf("%lf", &result);
        while(getchar() != '\n')
            ;
    }
    while(got != 1);

    return result;
}

double    area(double radius)
{
    return pi * radius * radius;
}

double    circumf(double radius)
{
    return 2.0 * pi * radius;
}

double    volume(double radius, double height)
{
    return area(radius) * height;
}
```

```
char          get_option(void)
{
    char  ch;

    do {
        printf( "Area          A\n"
                "Circumference C\n"
                "Volume         V\n"
                "Quit           Q\n\n"
                "Please choose ");

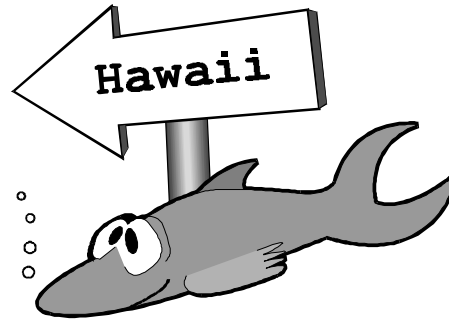
        scanf("%c", &ch);
        ch = tolower(ch);
        while(getchar() != '\n')
            ;
    }
    while(ch != 'a' && ch != 'c' && ch != 'v' && ch != 'q');

    return ch;
}
```

Pointers

Pointers

- § Declaring pointers
- § The “&” operator
- § The “*” operator
- § Initialising pointers
- § Type mismatches
- § Call by reference
- § Pointers to pointers



Pointers

This chapter deals with the concepts and some of the many uses of pointers in the C language.

Pointers - Why?

§ Using pointers allows us to:

- Achieve call by reference (i.e. write functions which change their parameters)
- Handle arrays efficiently
- Handle structures (records) efficiently
- Create linked lists, trees, graphs etc.
- Put data onto the heap
- Create tables of functions for handling Windows events, signals etc.

§ Already been using pointers with `scanf`

§ Care must be taken when using pointers since there are no safety features

Pointers - Why?

As C is such a low level language it is difficult to do anything without pointers. We have already seen that it is impossible to write a function which alters any of its parameters.

The next two chapters, dealing with arrays and dealing with structures, would be very difficult indeed without pointers.

Pointers can also enable the writing of linked lists and other such data structures (we look into linked lists at the end of the structures chapter).

Writing into the heap, which we will do towards the end of the course, would be impossible without pointers.

The Standard Library, together with the Windows, Windows 95 and NT programming environments use pointers to functions quite extensively.

One problem is that pointers have a bad reputation. They are supposed to be difficult to use and difficult to understand. This is, however, not the case, pointers are quite straightforward.

Declaring Pointers

☞ Pointers are declared by using “*”

☞ Declare an integer:

```
int i;
```

☞ Declare a pointer to an integer:

```
int *p;
```

☞ There is some debate as to the best position of the “*”

```
int* p;
```

Declaring Pointers

The first step is to know how to declare a pointer. This is done by using C's multiply character “*” (which obviously doesn't perform a multiplication). The “*” is placed at some point between the keyword `int` and the variable name. Instead of creating an integer, a *pointer to an integer* is created.

There has been, and continues to be, a long running debate amongst C programmers regarding the best position for the “*”. Should it be placed next to the type or next to the variable?

Example Pointer Declarations

```
int      *pi;          /* pi is a pointer to an int */
long int *p;           /* p is a pointer to a long int */
float*    pf;          /* pf is a pointer to a float */
char      c, d, *pc;    /* c and d are a char
                        pc is a pointer to char */
double*   pd, e, f;     /* pd is pointer to a double
                        e and f are double */
char*     start;        /* start is a pointer to a char */
char*     end;          /* end is a pointer to a char */
```

Example Pointer Declarations

Pointers Have Different Types

The first thing to notice about the examples above is that C has different *kinds* of pointer. It has pointers which point to **ints** and pointers which point to **long ints**. There are also pointers which point at **floats** and pointers to **chars**.

This concept is rather strange to programmers with assembler backgrounds. In assembler there are just pointers. In C this is not possible, only pointers to certain types exist. This is so the compiler can keep track of how much valid data exists on the end of a pointer. For instance, when looking down the pointer "start" only 1 byte would be valid, but looking down the pointer "pd" 8 bytes would be valid and the data would be expected to be in IEEE format.

Positioning the "*"

Notice that in: **char c, d, *pc;**

it seems reasonable that "c" and "d" are of type **char**, and "pc" is of type pointer to **char**. However it may seem less reasonable that in:

double* pd, e, f;

the type of "e" and "f" is **double** and NOT pointer to **double**. This illustrates the case for placing the "*" next to the variable and not next to the type.

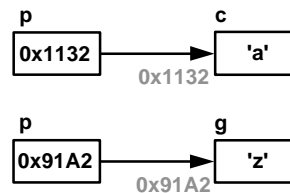
The last two examples show how supporters of the "place the * next to the type" school of thought would declare two pointers. One is declared on each line.

The “&” Operator

- § The “&”, “address of” operator, generates the address of a variable
- § All variables have addresses except register variables

```
char g = 'z';
int main(void)
{
    char c = 'a';
    char *p;

    p = &c;
    p = &g;
    return 0;
}
```



The “&” Operator

The “&” operator, which we have been using all along with **scanf**, generates the address of a variable. You can take the address of any variable which is stack based or data segment based. In the example above the variable “c” is stack based. Because the variable “g” is global, it is placed in the data segment. It is not possible to take the address of any **register** variable, because CPU registers do not have addresses. Even if the request was ignored by the compiler, and the variable is stack based anyway, its address still cannot be taken.

Pointers Are Really Just Numbers

You see from the program above that pointers are really just numbers, although we cannot say or rely upon the number of bits required to hold the number (there will be as many bits as required by the hardware). The variable “p” contains not a character, but the address of a character. Firstly it contains the address of “c”, then it contains the address of “g”. The pointer “p” may only point to one variable at a time and when pointing to “c” it is not pointing anywhere else.

By “tradition” addresses are written in hexadecimal notation. This helps to distinguish them from “ordinary” values.

Printing Pointers


The value of a pointer may be seen by calling **printf** with the **%p** format specifier.

Rules

- § **Pointers may only point to variables of the same type as the pointer has been declared to point to**
- § **A pointer to an `int` may only point to an `int`**
 - not to `char`, `short int` or `long int`, certainly not to `float`, `double` or `long double`
- § **A pointer to a `double` may only point to a `double`**
 - not to `float` or `long double`, certainly not to `char` or any of the integers
- § **Etc.....**

```
int    *p;           /* p is a pointer to an int */
long   large = 27L;  /* large is a long int,
                      initialised with 27 */

p = &large;          /* ERROR */
```



Rules

Assigning Addresses

The compiler is very firm with regard to the rule that a pointer can only point at the type it is declared to point to.

Let us imagine a machine where an `int` and a `short int` are the same size, (presumably 2 bytes). It would seem safe to assume that if we declared a pointer to an `int` the compiler would allow us to point it at an `int` and a `short int` with impunity. This is definitely not the case. The compiler disallows such behavior because of the possibility that the next machine the code is ported to has a 2 byte `short int` and a 4 byte `int`.

How about the case where we are guaranteed two things will be the same size? Can a pointer to an `int` be used to point to an `unsigned int`? Again the answer is no. Here the compiler would disallow the behavior because using the `unsigned int` directly and in an expression versus the value at the end of the pointer (which would be expected to be `int`) could give very different results!

The “*” Operator

§ The “*”, “points to” operator, finds the value at the end of a pointer

```
#include <stdio.h>
```

```
char g = 'z';
```

```
int main(void)
```

```
{
```

```
    char c = 'a';
```

```
    char *p;
```

```
    p = &c;
```

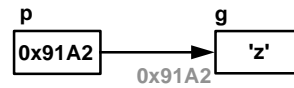
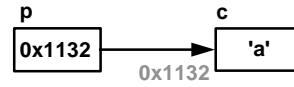
```
    printf("%c\n", *p);
```

```
    p = &g;
```

```
    printf("%c\n", *p);
```

```
    return 0;
```

```
}
```



print “what p points to”

a
z

The “*” Operator

The “*” operator is in a sense the opposite of the “&” operator. “&” generates the address of a variable, the “*” uses the address that is stored in a variable and finds what is at that location in memory.

Thus, in the example above, the pointer “p” is set to point to the variable “c”. The variable “p” contains the number 0x1132 (that’s 4402 in case you’re interested). “*p” causes the program to find what is stored at location 0x1132 in memory. Sure enough stored in location 0x1132 is the value 97. This 97 is converted by “%c” format specifier and ‘a’ is printed.

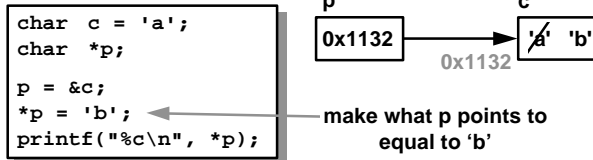
When the pointer is set to point to “g”, the pointer contains 0x91A2 (that is 37282 in decimal). Now the pointer points to the other end of memory into the data segment. Again when “*p” is used, the machine finds out what is stored in location 0x91A2 and finds 122. This is converted by the “%c” format specifier, printing ‘z’.

Writing Down Pointers

- It is not only possible to *read* the values at the end of a pointer as with:

```
char c = 'a';
char *p;
p = &c;
printf("%c\n", *p);
```

- It is possible to *write* over the value at the end of a pointer:



Writing Down Pointers

We have just seen an example of reading the value at the end of a pointer. But it is possible not only to *read* a value, but to *write* over and thus *change* it. This is done in a very natural way, we change variables by using the assignment operator, “=”. Similarly the value at the end of a pointer may be changed by placing “*pointer” (where “pointer” is the variable containing the address) on the left hand side of an assignment.

In the example above: ***p = 'b';**

literally says, take the value of 98 and write it into wherever “p” points (in other words write into memory location 0x1132, or the variable “c”).

Now you’re probably looking at this and thinking, why do it that way, since

c = 'b';

would achieve the same result and be a lot easier to understand. Consider that the variables “p” and “c” may live in different blocks and you start to see how a function could alter a parameter passed down to it.

Initialisation Warning!

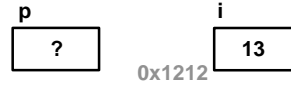
§ The following code contains a horrible error:

```
#include <stdio.h>

int main(void)
{
    short i = 13;
    short *p;

    *p = 23;
    printf("%hi\n", *p);

    return 0;
}
```



Initialization Warning!

Always Initialize Pointers

The code above contains an all too common example of a pointer bug. The user presumably expects the statement:

```
*p = 23;
```

to overwrite the variable "i". If this is what is desired it would help if the pointer "p" were first set to point to "i". This could be easily done by the single statement:

```
p = &i;
```

which is so sadly missing from this program. "p" is an automatic variable, stack based and *initialized with a random value*. All automatic variables are initialized with random values, pointers are no exception. Thus when the statement:

```
*p = 23;
```

is executed we take 23 and randomly overwrite the two bytes of memory whose address appears in "p". These two random bytes are very unlikely to be the variable "i", although it is theoretically possible. We could write anywhere in the program. Writing into the code segment would cause us to crash immediately (because the code segment is read only). Writing into the data segment, the stack or the heap would "work" because we are allowed to write there (though some machines make parts of the data segment read only).

General Protection Fault

There is also a possibility that this random address lies outside the bounds of our program. If this is the case and we are running under a protect mode operating system (like Unix and NT) our program will be killed before it does any real damage. If not (say we were running under MS DOS) we would corrupt not our own program, but another one running in memory. This could produce unexpected results in *another* program. Under Windows this error produces the famous "GPF" or General Protection Fault.

Initialise Pointers!

- § Pointers are best initialised!
 - § A pointer may be declared and initialised in a single step
- ```
short i = 13;
short *p = &i;
```
- § This does NOT mean “make what p points to equal to the address of i”
  - § It DOES mean “declare p as a pointer to a short int, make p equal to the address of i”

```
short *p = &i;
```

```
short *p = &i;
short *p = &i;
```

## Initialize Pointers!

Hours of grief may be saved by ensuring that all pointers are initialized before use. Three extra characters stop the program on the previous page from destroying the machine and transforms it into a well behaved program.

### Understanding Initialization

In the line: `short *p = &i;`

it is very important to understand that the “\*” is not the “find what is pointed to” operator. Instead it ensures we do not declare a `short int`, but a *pointer to a short int* instead.

This is the case for placing the “\*” next to the type, if we had written

```
short* p = &i;
```

It would have been somewhat more obvious that we were declaring “p” to be a pointer to a `short int` and that we were initializing “p” to point to “i”.

## NULL

- § A special invalid pointer value exists #defined in various header files, called NULL
- § When assigned to a pointer, or when found in a pointer, it indicates the pointer is invalid

```
#include <stdio.h>

int main(void)
{
 short i = 13;
 short *p = NULL;

 if(p == NULL)
 printf("the pointer is invalid!\n");
 else
 printf("the pointer points to %hi\n", *p);

 return 0;
}
```

---

## NULL

We have already seen the concept of preprocessor constants, and how they are **#defined** into existence. A special define exists in the “stdio.h” header file (and a few other of the Standard headers just in case), called NULL. It is a special *invalid* value of a pointer.

The value may be placed in any kind of pointer, regardless of whether it points to **int**, **long**, **float** or **double**.

### NULL and Zero

You shouldn't enquire too closely into what the value of NULL actually is. Mostly it is defined as zero, but you should never assume this. On some machines zero is a legal pointer and so NULL will be defined as something else.

*Never* write code assuming NULL and zero are the same thing, otherwise it will be non portable.

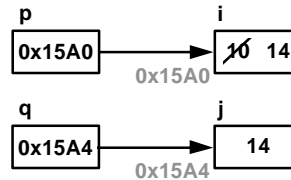
---

## A World of Difference!

§ There is a great deal of difference between:

```
int i = 10, j = 14;
int *p = &i;
int *q = &j;

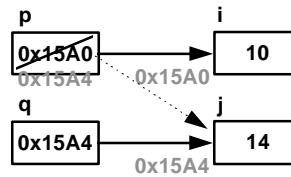
*p = *q;
```



and:

```
int i = 10, j = 14;
int *p = &i;
int *q = &j;

p = q;
```



## A World of Difference!

**What is Pointed  
to vs the  
Pointer Itself**

It is important to understand the difference between:

`*p = *q;`

and

`p = q;`

In the first, “`*p = *q`”, what is pointed to by “`p`” is overwritten with what is pointed to by “`q`”. Since “`p`” points to “`i`”, and “`q`” points to “`j`”, “`i`” is overwritten by the value stored in “`j`”. Thus “`i`” becomes 14.

In the second statement, “`p = q`” there are no “`*`”s. Thus the value contained in “`p`” itself is overwritten by the value in “`q`”. The value in `q` is `0x15A4` (which is 5540 in decimal) which is written into “`p`”. If “`p`” and “`q`” contain the same address, `0x15A4`, they must point to the same place in memory, i.e. the variable “`j`”.

## Fill in the Gaps

```
int main(void)
{
 int i = 10, j = 14, k;
 int *p = &i;
 int *q = &j;

 *p += 1;

 p = &k;

 *p = *q;

 p = q;

 *p = *q;

 return 0;
}
```

i   
0x2100

j   
0x2104

k   
0x1208

p   
0x120B

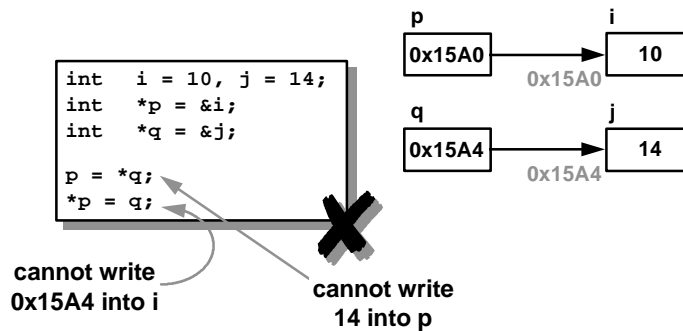
q   
0x1210

## Fill in the Gaps

Using the variables and addresses provided, complete the picture. Do not attach any significance to the addresses given to the variables, just treat them as random numbers.

## Type Mismatch

⚡ The compiler will not allow type mismatches when assigning to pointers, or to where pointers point



## Type Mismatch

The compiler checks very carefully the syntactic correctness of the pointer code you write. It will make sure when you assign to a pointer, an address is assigned. Similarly if you assign to what is at the end of a pointer, the compiler will check you assign the “pointed to” type.

There are some programming errors in the program above. The statement:

**p = \*q;**

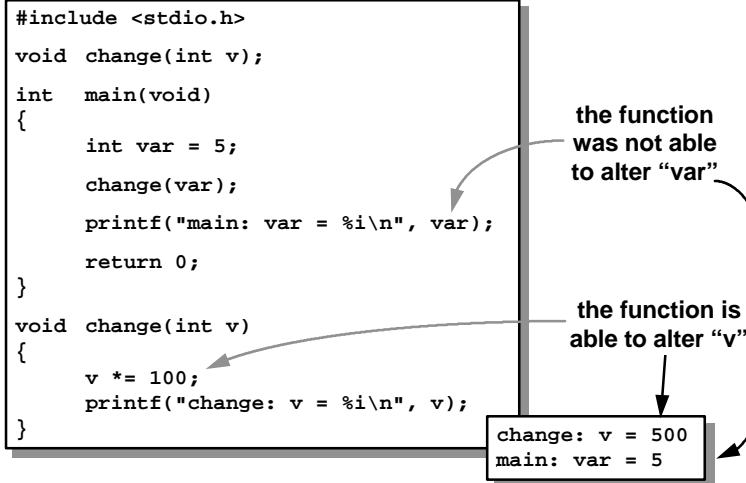
would assign what is pointed to by “q” (i.e. 14), into “p”. Although this would seem to make sense (because “p” just contains a number anyway) the compiler will not allow it because the types are wrong. We are assigning an **int** into an **int\***. The valid pointer 0x15A0 (5536 in decimal) is corrupted with 14. There is no guarantee that there is an integer at address 14, or even that 14 is a valid address.

Alternatively the statement:

**\*p = q;**

takes the value stored in “q”, 0x15A4 (5540 in decimal) and writes it into what “p” points to, i.e. the variable “i”. This might seem to make sense, since 5540 is a valid number. However the address in “q” may be a different size to what can be stored in “i”. There are no guarantees in C that pointers and integers are the same size.

## Call by Value - Reminder



## Call by Value - Reminder

This is a reminder of the call by value program.

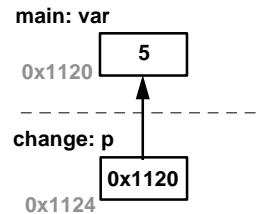
The **main** function allocates a variable "var" of type int and value 5. When this variable is passed to the **change** function a copy is made. This copy is picked up in the parameter "v". "v" is then changed to 500 (to prove this, it is printed out). On leaving the **change** function the parameter "v" is thrown away. The variable "var" still contains 5.



## Call by Reference

prototype "forces" us to pass a pointer

```
#include <stdio.h>
void change(int* p);
int main(void)
{
 int var = 5;
 change(&var);
 printf("main: var = %i\n", var);
 return 0;
}
void change(int* p)
{
 *p *= 100;
 printf("change: *p = %i\n", *p);
}
```



```
change: *p = 500
main: var = 500
```

## Call by Reference

This program demonstrates call by reference in C. Notice the prototype which requires a single pointer to **int** to be passed as a parameter.

When the **change** function is invoked, the address of "var" is passed across:

```
change(&var);
```

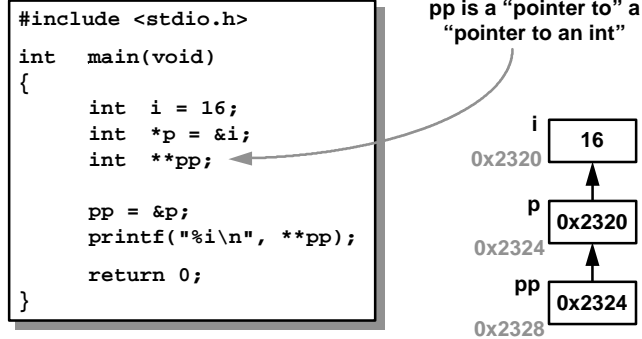
The variable "p", declared as the parameter to function **change**, thus points to the variable "var" within **main**. This takes some thinking about since "var" is not *directly* accessible to **main** (because it is declared in another function block) however "p" is and so is wherever it points.

By using the "**\*p**" notation the **change** function writes down the pointer over "var" which is changed to 500.

When the **change** function returns, "var" retains its value of 500.

## Pointers to Pointers

- ⌘ **C allows pointers to any type**
- ⌘ **It is possible to declare a pointer to a pointer**



## Pointers to Pointers

The declaration: `int i;`

declares "i" to be of type int: `int *p;`

declares "p" to be of type pointer to int. One "\*" means one "pointer to". Thus in the declaration:

`int **pp;`

two \*s must therefore declare "pp" to be of type a pointer to a pointer to `int`.

Just as "p" must point to `ints`, so "pp" must point to pointers to `int`. This is indeed the case, since "pp" is made to point to "p". "\*"p causes 16 to be printed

`printf("%p", pp);`

would print 0x2324 whereas

`printf("%p", *pp);`

would print 0x2320 (what "pp" points to).

`printf("%i", **pp);`

would cause what "0x2320 points to" to be printed, i.e. the value stored in location 0x2320 which is 16.

## Review

```
int main(void)
{
 int i = 10, j = 7, k;
 int *p = &i;
 int *q = &j;
 int **pp = &p;

 **pp += 1;

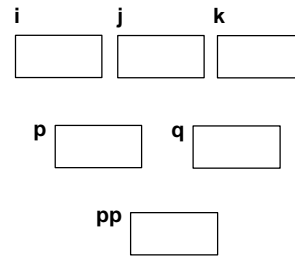
 *pp = &k;

 **pp = *q;

 i = *q***pp;

 i = *q/**pp; /* headache? */;

 return 0;
}
```



## Review Questions

What values should be placed in the boxes?



---

---

## Pointers Practical Exercises

---

---

Directory: **POINT**

1. Work your way through the following code fragments. What would be printed? When you have decided, compile and run the program “**POINTEX.C**” to check your answers. You will find it helpful, especially with some of the later exercises, to draw boxes representing the variables and arrows representing the pointers.

- a) 

```
int i = -23;
int * p = &i;

printf("*p = %i\n", *p);
```
- b) 

```
int i;
int * p = &i;

printf("*p = %i\n", *p);
```
- c) 

```
int i = 48;
int * p;

printf("*p = %i\n", *p);
```
- d) 

```
int i = 10;
int * p = &i;
int j;

j = ++*p;

printf("j = %i\n", j);
printf("i = %i\n", i);
```
- e) 

```
int i = 10, j = 20;
int * p = &i;
int * q = &j;

*p = *q;
printf("i = %i, j = %i\n", i, j);
printf("*p = %i, *q = %i\n", *p, *q);

i = 10; j = 20;

p = q;
printf("i = %i, j = %i\n", i, j);
printf("*p = %i, *q = %i\n", *p, *q);
```
- f) 

```
int i = 10, j = 0;
int * p = &i;
int * q = &j;

p = q;
printf("i = %i, j = %i\n", i, j);
printf("*p = %i, *q = %i\n", *p, *q);

*p = *q;
printf("i = %i, j = %i\n", i, j);
printf("*p = %i, *q = %i\n", *p, *q);
```
-

```

g) float ten = 10.0F;
 float hundred = 100.0F;
 float * fp0 = &ten, * fp1 = &hundred;

 fp1 = fp0;
 fp0 = &hundred;
 *fp1 = *fp0;

 printf("ten/hundred = %f\n", ten/hundred);

h) char a = 'b', b = 'c', c = 'e';
 char *l = &c, *m = &b, *n, *o = &a;

 n = &b; *m = ++*o; m = n; *l = 'a';

 printf("a = %c, b = %c, c = %c, d = %c\n", a, b, c, d);
 printf("*l = %c, *m = %c, *n = %c, *o = %c\n", *l, *m, *n, *o);

i) int i = 2, j = 3, k;
 int * p = &i, *q = &j;
 int ** r;

 r = &p;
 printf("***r = %i\n", **r);
 k = *p**q;
 printf("k = %i\n", k);
 *p = *q;
 printf("***r = %i\n", **r);
 k = **r**q;
 printf("k = %i\n", k);
 k = *p/ *q;
 printf("k = %i\n", k);

```

2. Open the file “**SWAP.C**”. You will see the program reads two integers, then calls the **swap** function to swap them. The program doesn't work because it uses call by value. Alter the function to use call by reference and confirm it works.

3. In the file “**BIGGEST.C**” two functions are called:

```
int *biggest_of_two(int*, int*);
```

and

```
int *biggest_of_three(int*, int*, int*);
```

The first function is passed pointers to two integers. The function should return whichever pointer points to the larger integer. The second function should return whichever pointer points to the largest of the three integers whose addresses are provided.

4. Open the file “**DIV.C**”. You will see the program reads two integers. Then a function with the following prototype is called:

```
void div_rem(int a, int b, int *divides, int *remains);
```

This function is passed the two integers. It divides them (using integer division), and writes the answer over wherever “divides” points. Then it finds the remainder and writes it into where “remains” points. Thus for 20 and 3, 20 divided by 3 is 6, remainder 2. Implement the **div\_rem** function.

5. The program in “CHOP.C” reads a **double** before calling the chop function, which has the following prototype:

```
void chop(double d, long *whole_part, double *fraction_part);
```

This function chops the **double** into two parts, the whole part and the fraction. So “365.25” would be chopped into “365” and “.25”. Implement and test the function.

6. The **floor** function returns, as a double, the “whole part” of its parameter (the fractional part is truncated). By checking this returned value against the maximum value of a **long** (found in **limits.h**) print an error message if the **chop** function would overflow the **long** whose address is passed.
-



---

---

## Pointers Solutions

---

---

2. Open the file “**SWAP.C**”. You will see the program reads two integers, then calls the function **swap** to swap them. Alter the function to use call by reference and confirm it works.

```
#include <stdio.h>

void swap(int*, int*);

int main(void)
{
 int a = 100;
 int b = -5;

 printf("the initial value of a is %i\n", a);
 printf("the initial value of b is %i\n", b);

 swap(&a, &b);

 printf("after swap, the value of a is %i\n", a);
 printf("and the value of b is %i\n", b);

 return 0;
}

void swap(int *i, int *j)
{
 int temp = *i;
 *i = *j;
 *j = temp;
}
```

---

3. In the file “**BIGGEST.C**” implement the two functions called:

```
int *biggest_of_two(int*, int*);

and

int *biggest_of_three(int*, int*, int*);
```

*The biggest\_of\_three function could have been implemented with a complex series of if/then/else constructs, however since the biggest\_of\_two function was already implemented, it seemed reasonable to get it to do most of the work.*

```
#include <stdio.h>

int* biggest_of_two(int*, int*);
int* biggest_of_three(int*, int*, int*);
```

---

```

int main(void)
{
 int a = 100;
 int b = -5;
 int c = 200;
 int *p;

 p = biggest_of_two(&a, &b);
 printf("the biggest of %i and %i is %i\n", a, b, *p);

 p = biggest_of_three(&a, &b, &c);
 printf("the biggest of %i %i and %i is %i\n", a, b, c, *p);

 return 0;
}

int* biggest_of_two(int * p, int * q)
{
 return (*p > *q) ? p : q;
}

int* biggest_of_three(int * p, int * q, int * r)
{
 int *first = biggest_of_two(p, q);
 int *second = biggest_of_two(q, r);

 return biggest_of_two(first, second);
}

```

---

4. In "DIV.C" implement

```

void div_rem(int a, int b, int *divides, int *remains);

#include <stdio.h>

void div_rem(int a, int b, int *divides, int *remains);

int main(void)
{
 int a, b;
 int div = 0;
 int rem = 0;

 printf("enter two integers ");
 scanf("%i %i", &a, &b);

 div_rem(a, b, &div, &rem);

 printf("%i divided by %i = %i "
 "remainder %i\n", a, b, div, rem);

 return 0;
}

```

---

```

void div_rem(int a, int b, int *divides, int *remains)
{
 *divides = a / b;
 *remains = a % b;
}

```

---

5. The program in "CHOP.C" reads a **double** before calling the chop function, which has the following prototype:

```
void chop(double d, long *whole_part, double *fraction_part);
```

6. By checking the **floor** function returned value against the maximum value of a **long** print an error message if the **chop** function would overflow the **long** whose address is passed.

*One of the most important things in the following program is to include "math.h". Without this header file, the compiler assumes floor returns an integer. Thus the truncated double actually returned is corrupted. Since it is the cornerstone of all calculations in chop, it is important this value be intact. Use of the floor function is important, since if the user types 32767.9 and the maximum value of a long were 32767, testing the double directly against LONG\_MAX would cause our overflow message to appear, despite the whole value being able to fit into a long int.*

```

#include <stdio.h>
#include <math.h>
#include <limits.h>

void chop(double d, long *whole_part, double *fraction_part);

int main(void)
{
 double d = 0.0;
 long whole = 0;
 double fraction = 0.0;

 printf("enter a double ");
 scanf("%lf", &d);

 chop(d, &whole, &fraction);

 printf("%lf chopped is %ld and %.5lg\n",
 d, whole, fraction);

 return 0;
}

void chop(double d, long *whole_part, double *fraction_part)
{
 double truncated = floor(d);

 if(truncated > LONG_MAX) {
 printf("assigning %.0lf to a long int would overflow "
 "(maximum %ld)\n", truncated, LONG_MAX);

 *whole_part = LONG_MAX;
 } else
 *whole_part = (long)truncated;

 *fraction_part = d - truncated;
}

```

---

---

---

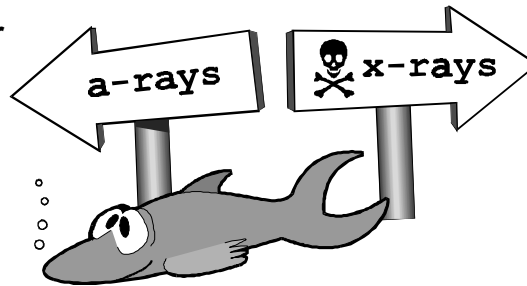
## Arrays in C

---

---

## Arrays in C

- § Declaring arrays
- § Accessing elements
- § Passing arrays into functions
- § Using pointers to access arrays
- § Strings
- § The null terminator



---

## Arrays in C

This chapter discusses all aspects of arrays in C.

---

## Declaring Arrays

- § An array is a collection of data items (called *elements*) all of the same type
- § It is declared using a type, a variable name and a **CONSTANT** placed in square brackets
- § C always allocates the array in a single block of memory
- § The size of the array, once declared, is fixed forever - there is no equivalent of, for instance, the “redim” command in BASIC

---

## Declaring Arrays

An important fact to understand about arrays is that they consist of the same type all the way through. For instance, an array of 10 **int** is a group of 10 integers all bunched together. The array doesn't change type half way through so there are 5 **int** and 5 **float**, or 1 **int**, 1 **float** followed by 1 **int** and 1 **float** five times. Data structures like these could be created in C, but an array isn't the way to do it.

Thus to create an array we merely need a type for the elements and a count. For instance:

```
long a[5];
```

creates an array called “a” which consists of 5 **long ints**. It is a rule of C that the storage for an array is physically contiguous in memory. Thus wherever, say, the second element sits in memory, the third element will be adjacent to it, the fourth next to that and so on.

---

## Examples

```
#define SIZE 10
int a[5]; /* a is an array of 5 ints */
long int big[100]; /* big is 400 bytes! */
double d[100]; /* but d is 800 bytes! */
long double v[SIZE]; /* 10 long doubles, 100 bytes */
```

```
int a[5] = { 10, 20, 30, 40, 50 };
double d[100] = { 1.5, 2.7 };
short primes[] = { 1, 2, 3, 5, 7, 11, 13 };
long n[50] = { 0 };
```

all five  
elements  
initialised

first two elements  
initialised,  
remaining ones  
set to zero

compiler fixes  
size at 7  
elements

quickest way of setting  
ALL elements to zero

```
int i = 7;
const int c = 5;

int a[i];
double d[c];
short primes[];
```



## Examples

Above are examples of declaring and initializing arrays. Notice that C can support arrays of any type, including structures (which will be covered the next chapter), except **void** (which isn't a type so much as the absence of a type). You will notice that a *constant* must appear within the brackets so:

```
long int a[10];
```

is fine, as is:

```
#define SIZE 10
long int a[SIZE];
```

But:

```
int size = 10;
long int a[size];
```

and

```
const int a_size = 10;
long int a[a_size];
```

will NOT compile. The last is rather curious since "a\_size" is obviously constant, however, the compiler will not accept it. Another thing to point out is that the number provided must be an integral type, "int a[5.3]" is obviously nonsense.

### Initializing Arrays

1. The number of initializing values is exactly the same as the number of elements in the array. In this case the values are assigned one to one, e.g.  
`int a[5] = { 1, 2, 3, 4, 5 };`
2. The number of initializing values is less than the number of elements in the array. Here the values are assigned "one to one" until they run out. The remaining array elements are initialized to zero, e.g.  
`int a[5] = { 1, 2 };`
3. The number of elements in the array has not been specified, but a number of initializing values has. Here the compiler fixes the size of the array to the number of initializing values and they are assigned one to one, e.g.  
`int a[] = { 1, 2, 3, 4, 5 };`



## Accessing Elements

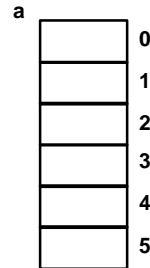
- § The elements are accessed via an integer which ranges from 0..size-1
- § There is no bounds checking

```
int main(void)
{
 int a[6];
 int i = 7;

 a[0] = 59;
 a[5] = -10;
 a[i/2] = 2;

 a[6] = 0;
 a[-1] = 5;

 return 0;
}
```



## Accessing Elements

### Numbering Starts at Zero

THE most important thing to remember about arrays in C is the scheme by which the elements are numbered. The FIRST element in the array is element number ZERO, the second element is number one and so on. The LAST element in the array "a" above is element number FIVE, i.e. the total number of elements less one.

This scheme, together with the fact that there is no bounds checking in C accounts for a great deal of errors where array bounds accessing is concerned. It is all too easy to write "**a[6] = 0**" and index one beyond the end of the array. In this case whatever variable were located in the piece of memory (maybe the variable "i") would be corrupted.

Notice that the array access **a[i/2]** is fine, since "i" is an integer and thus **i/2** causes integer division.

## Array Names

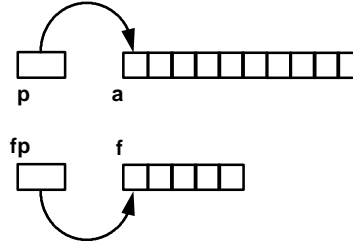
- § There is a special and unusual property of array names in C
- § The name of an array is a pointer to the start of the array, i.e. the zeroth element, thus

$a == \&a[0]$

```
int a[10];
int *p;

float f[5];
float *fp;

p = a; /* p = &a[0] */
fp = f; /* fp = &f[0] */
```



## Array Names

### A Pointer to the Start

In C, array names have a rather unusual property. The compiler treats the name of an array as an address which may be used to initialize a pointer without error. The address is that of the first element (i.e. the element with index 0).

### Cannot Assign to an Array

Note that the address is a *constant*. If you are wondering what would happen with the following:


```
int a[10];
int b[10];

a = b;
```

the answer is that you'd get a compiler error. The address that "a" yields is a constant and thus it cannot be assigned to. This makes sense. If it were possible to assign to the name of an array, the compiler might "forget" the address at which the array lived in memory.

## Passing Arrays to Functions

- § When an array is passed to a function a pointer to the zeroth element is passed across
- § The function may alter any element
- § The corresponding parameter may be declared as a pointer, or by using the following special syntax



```
int add_elements(int a[], int size)
{
```

```
int add_elements(int *p, int size)
{
```

---

## Passing Arrays to Functions

If we declare an array:

```
int a[60];
```

and then pass this array to a function:

```
function(a);
```

the compiler treats the name of the array “a” in exactly the same way it did before, i.e. as a pointer to the zeroth element of the array. This means that a pointer is passed to the function, i.e. the array is NOT passed by value.

### Bounds Checking Within Functions

One problem with this strategy is that there is no way for the function to know how many elements are in the array (all the function gets is a pointer to one integer, this could be one lone integer or there could be one hundred other integers immediately after it). This accounts for the second parameter in the two versions of the `add_elements` function above. This parameter must be provided by us as the valid number of elements in the array.

Note that there is some special syntax which makes the parameter a pointer. This is:

```
int a[]
```

This is one of very few places this syntax may be used. Try to use it to declare an array and the compiler will complain because it cannot determine how much storage to allocate for the array. All it is doing here is the same as:

```
int * a;
```

Since pointers are being used here and we can write down pointers, any element of the array may be changed.

---

## Example

```
#include <stdio.h>

void sum(long [], int);

int main(void)
{
 long primes[6] = { 1, 2,
 3, 5, 7, 11 };

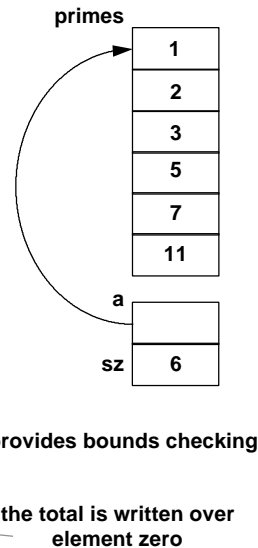
 sum(primes, 6);

 printf("%li\n", primes[0]);

 return 0;
}

void sum(long a[], int sz)
{
 int i;
 long total = 0;
 for(i = 0; i < sz; i++)
 total += a[i];

 a[0] = total;
}
```



## Example

### A Pointer is Passed

In the example above the array “primes” is passed down to the function “sum” by way of a pointer. “a” is initialized to point to primes[0], which contains the value 1.

Within the function the array access a[i] is quite valid. When “i” is zero, a[0] gives access to the value 1. When “i” is one, a[1] gives access to the value 2 and so on. Think of “i” as an offset of the number of **long ints** beyond where “a” points.

### Bounds Checking

The second parameter, “sz” is 6 and provides bounds checking. You will see the **for** loop:

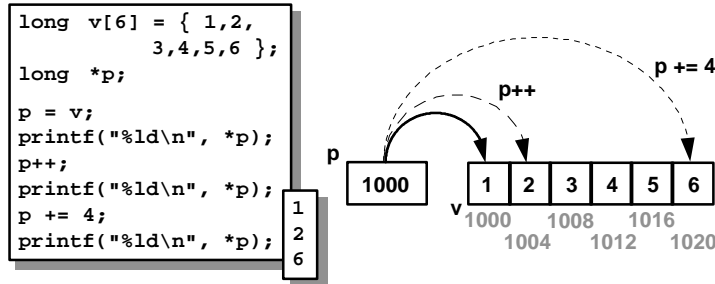
```
for(i = 0; i < sz; i++)
```

is ideally suited for accessing the array elements. a[0] gives access to the first element, containing 1. The last element to be accessed will be a[5] (because “i” being equal to 6 causes the loop to exit) which contains the 11.

Notice that because call by reference is used, the sum function is able to alter any element of the array. In this example, element a[0], in other words prime[0] is altered to contain the sum.

## Using Pointers

- § Pointers may be used to access array elements rather than using constructs involving “[ ]”
- § Pointers in C are automatically scaled by the size of the object pointed to when involved in arithmetic



## Using Pointers

Pointers in C are ideally suited for accessing the elements of an array. We have already seen how the name of an array acts like a pointer.

In the example above the array “v” starts at address 1000 in memory, i.e. the address of element zero is 1000. Since the elements are **long ints** and hence 4 bytes in size, the next element, v[1] sits at address 1004 in memory.

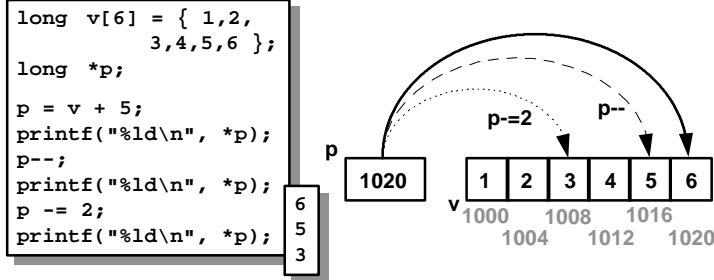
### Addition With Pointers

If a pointer to a **long int** is initialised with “v” it will contain 1000. The **printf** prints what is pointed to by “p”, i.e. 1. The most important thing to realize is that on the next line “p++” the value contained by “p” does NOT become 1001. The compiler, realizing that “p” is a pointer to a **long int**, and knowing that **longs** are 4 bytes in size makes the value 1004. **Addition to pointers is scaled by the size of the object pointed to.** **printf** now prints 2 at the end of the pointer 1004.

With the next statement “p += 4”, the 4 is scaled by 4, thus 16 is added to the pointer.  $1004 + 16 = 1020$ . This is the address of the sixth element, v[5]. Now the printf prints 6.

## Pointers Go Backwards Too

Scaling not only happens when addition is done, it happens with subtraction too



## Pointers Go Backwards Too

This scaling of pointers by the size of the object pointed to not only occurs with addition. Whenever subtraction is done on a pointer, the scaling occurs too.

So, in the assignment: `p = v + 5;`

as we have already seen, `v` gives rise to the address 1000 and the 5 is scaled by the size of a **long int**, 4 bytes to give  $1000 + 5 * 4$ , i.e. 1020. Thus the pointer "p" points to the last of the long integers within the array, element `v[5]`, containing 6.

### Subtraction From Pointers

When the statement: `p--;`

is executed the pointer does NOT become 1019. Instead the compiler subtracts one times the size of a **long int**. Thus 4 bytes are subtracted and the pointer goes from 1020 to 1016. Thus the pointer now points to the element `v[4]` containing 5.

With the statement: `p -= 2;`

the 2 is scaled by 4, giving 8.  $1016 - 8$  gives 1008, this being the address of the element "`v[2]`".

## Pointers May be Subtracted

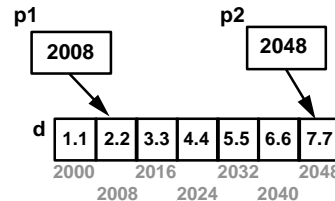
§ When two pointers *into the same array* are subtracted C scales again, giving the number of array elements separating them

```
double d[7] = { 1.1, 2.2,
 3.3, 4.4, 5.5, 6.6, 7.7 };
double *p1;
double *p2;

p1 = d + 1;
p2 = d + 6;

printf("%i\n", p2 - p1);
```

5



## Pointers May be Subtracted

We have discussed adding and subtracting integers from pointers. When this occurs the compiler scales the integer by the size of the thing pointed to and adds or subtracts the scaled amount. When two pointers are subtracted (note: two pointers may NOT be added) the compiler scales the distance between them. In the example above we are using an array of **double**, each **double** being 8 bytes in size. If the address of the first is 2000, the address of the second is 2008, the third is 2016 etc.

In the statement: **p1 = d + 1;**

“d” yields the address 2000, 1 is scaled by 8 giving 2000 + 8, i.e. 2008.

In: **p2 = d + 6;**

“d” yields the address 2000, 6 is scaled by 8 giving 2000 + 48, i.e. 2048.

When these two pointers are subtracted in:

**p2 - p1**

the apparent answer is 2048 - 2008 = 40. However, the compiler scales the 40 by the size of the object pointed to. Since these are pointers to **double**, it scales by 8 bytes, thus 40 / 8 = 5;

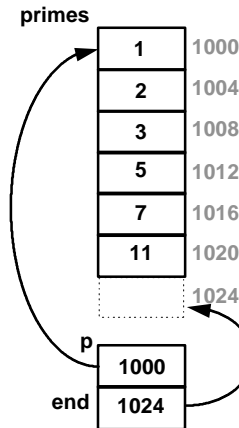
Notice there are some rules here. The first pointer “p2” must point “higher” into memory than the second pointer “p1”. If the subtraction had been written as

**p1 - p2**

the result would not have been meaningful. Also, the two pointers must point into the *same* array. If you subtract two pointers into different arrays this only gives information on where in memory the compiler has placed the arrays.

## Using Pointers - Example

```
#include <stdio.h>
long sum(long*, int);
int main(void)
{
 long primes[6] = { 1, 2,
 3, 5, 7, 11 };
 printf("%li\n", sum(primes, 6));
 return 0;
}
long sum(long *p, int sz)
{
 long *end = p + sz;
 long total = 0;
 while(p < end)
 total += *p++;
 return total;
}
```



## Using Pointers - Example

Above is an example of using pointers to handle an array. In the statement:

```
sum(primes, 6)
```

the use of the name of the array "primes" causes the address of the zeroth element, 1000, to be copied into "p". The 6 is copied into "sz" and provides bounds checking.

The initialization: 

```
long *end = p + sz;
```

sets the pointer "end" to be  $1000 + 6 * 4$  (since **long int** is 4 bytes in size), i.e. 1024. The location with address 1024 lies one beyond the end of the array, hence

```
while(p < end)
```

and NOT: 

```
while(p <= end)
```

The statement: 

```
total += *p++;
```

adds into "total" (initially zero) the value at the end of the pointer "p", i.e. 1. The pointer is then incremented, 4 is added, "p" becoming 1004. Since 1004 is less than the 1024 stored in "end", the loop continues and the value at location 1004, i.e. 2 is added in to total. The pointer increases to 1008, still less than 1024. It is only when all the values in the array have been added, i.e. 1, 2, 3, 5, 7 and 11 that the pointer "p" points one beyond the 11 to the location whose address is 1024. Since the pointer "p" now contains 1024 and the pointer "end" contains 1024 the condition:

```
while(p < end)
```

is no longer true and the loop terminates. The value stored in total, 34, is returned.



## \* and ++

**\*p++ means:**

- \*p++** find the value at the end of the pointer
- \*p++** increment the POINTER to point to the next element

**(\*p)++ means:**

- (\*p)++** find the value at the end of the pointer
- (\*p)++** increment the VALUE AT THE END OF THE POINTER (the pointer never moves)

**\*++p means:**

- \*++p** increment the pointer
- \*++p** find the value at the end of the pointer

## \* and ++

**In “\*p++”  
Which Operator  
is Done First?**

In fact “++” has a higher precedence than “\*”. If “++” gets done first, why isn’t the pointer incremented and THEN the value at the end of the pointer obtained? Clearly in the last program this didn’t happen. To understand the answer it is important to remember the register used when postfix ++ is specified. In

```
int i = 5, j;

j = i++;
```

The value of “i”, 5, is saved in a register. “i” is then incremented, becoming 6. The value in the register, 5, is then transferred into “j”. Thus the increment is done before the assignment, yet is *appears* as though the assignment happens first. Now consider:

```
x = *p++
```

and imagine that “p” contains 1000 (as before) and that “p” points to **long ints** (as before). The value of “p”, 1000, is saved in a register. “p” is incremented and becomes 1004. The pre-incremented value of 1000, saved in the register is used with “\*”. Thus we find what is stored in location 1000. This was the value 1 which is transferred into “x”.

**(\*p)++**

With “(\*p)++” the contents of location 1000, i.e. 1, is saved in the register. The contents of location 1000 are then incremented. The 1 becomes a 2 and the pointer still contains 1000. This construct is guaranteed never to move the pointer, but to continually increment at the end of the pointer, i.e. the value in element zero of the array.

**\*++p**

With “\*++p”, because *prefix* increment is used, the register is not used. The value of “p”, 1000, is incremented directly, becoming 1004. The value stored in location 1004, i.e. 2, is then accessed. This construct is guaranteed to miss the first value in the array.

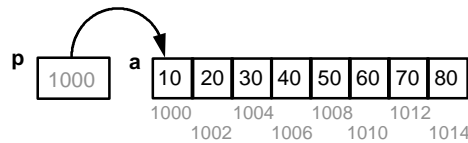
## Which Notation?

§ An axiom of C states  $a[i]$  is equivalent to  $*(a + i)$

```
short a[8] = { 10, 20, 30, 40, 50, 60, 70, 80 };
short *p = a;

printf("%i\n", a[3]);
printf("%i\n", *(a + 3));
printf("%i\n", *(p + 3));
printf("%i\n", p[3]);
printf("%i\n", 3[a]);
```

40  
40  
40  
40  
40



## Which Notation?

If both array access notation, “ $a[\text{index}]$ ”, and pointer notation, “ $*p++$ ”, may be used to access array elements which is better? First, here are all the variations:

A fundamental truth (what mathematicians call an “axiom”) in C is that any array access  $a[i]$  is equivalent to  $*(a+i)$ .

Consider  $a[3]$  which will access the element containing 40. This element is also accessed by  $*(a+3)$ . Since “ $a$ ” is the name of an array, the address 1000 is yielded giving  $*(1000+3)$ . Since the address has type pointer to **short int**, the 3 is scaled by the size of the object pointed to, i.e.  $*(1000+3*2)$ . The contents of location 1006 is the same 40 as yielded by  $a[3]$ .

Now consider  $*(p+3)$ . The pointer “ $p$ ” contains the address 1000. So  $*(p+3)$  gives  $*(1000+3)$ . Because of the type of the pointer, 3 is scaled by the size of a **short int** giving  $*(1000+3*2)$ , i.e. the contents of location 1006, i.e. 40.

The next variation,  $p[3]$ , looks strange. How can something that is clearly not an array be used on the “outside” of a set of brackets? To understand this, all that is needed is to apply the axiom above, i.e.  $a[i]$ , and hence  $p[3]$ , is equivalent to  $*(a+i)$ , hence  $*(p+3)$ . Above is an explanation of how  $*(p+3)$  works.

This last variation,  $3[a]$ , looks strangest of all. However,  $a[3]$  is equivalent to  $*(a+3)$ , but  $*(a+3)$  must be equivalent to  $*(3+a)$  since “ $+$ ” is commutative ( $10+20$  is the same as  $20+10$ ). This, reapplying the axiom, must be equivalent to  $3[a]$ . It is not generally recommended to write array accesses this way, however it not only must compile, but must access the element containing 40.

**Use What is Easiest!**

This doesn’t answer the question of which notation is best, pointer or array access. The answer is stick with what is easier to read and understand. Neither  $a[3]$  nor  $*(p+3)$  notations will have any significant speed or efficiency advantage over one another. If they both produce approximately the same speed code, why not choose the one that is clearest and makes the code most maintainable?

## Strings

- § **C has no native string type, instead we use arrays of char**
- § **A special character, called a “null”, marks the end (don’t confuse this with the NULL pointer )**
- § **This may be written as ‘\0’ (zero not capital ‘o’)**
- § **This is the only character whose ASCII value is zero**
- § **Depending on how arrays of characters are built, we may need to add the null by hand, or the compiler may add it for us**

---

## Strings

The sudden topic change may seem a little strange until you realize that C doesn't really support strings. In C, strings are just arrays of characters, hence the discussion here.

C has a special marker to denote the last character in a string. This character is called the null and is written as `'\0'`. You should not confuse this null with the NULL pointer seen in the pointers chapter. The difference is that NULL is an invalid pointer value and may be defined in some strange and exotic way. The null character is entirely different as it is always, and is guaranteed to be, zero.

Why the strange way of writing `'\0'` rather than just `0`? This is because the compiler assigns the type of `int` to `0`, whereas it assigns the type `char` to `'\0'`. The difference between the types is the number of bits, `int` gives 16 or 32 bits worth of zero, `char` gives 8 bits worth of zero. Thus, potentially, the compiler might see a problem with:

```
char c = 0;
```

since there are 16 or 32 bits of zero on the right of “=”, but room for only 8 of those bits in “c”.

---

## Example

```
char first_name[5] = { 'J', 'o', 'h', 'n', '\0' };
char last_name[6] = "Minor";
char other[] = "Tony Blurt";
char characters[7] = "No null";
```

this special case specifically  
excludes the null terminator

|            |     |     |     |     |     |     |     |     |     |     |   |
|------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---|
| first_name | 'J' | 'o' | 'h' | 'n' | 0   |     |     |     |     |     |   |
| last_name  | 'M' | 'i' | 'n' | 'o' | 'r' | 0   |     |     |     |     |   |
| other      | 'T' | 'o' | 'n' | 'y' | 32  | 'B' | 'l' | 'u' | 'r' | 't' | 0 |
| characters | 'N' | 'o' | 32  | 'n' | 'u' | 'l' | 'l' |     |     |     |   |

## Example

The example above shows the two ways of constructing strings in C. The first requires the string to be assembled by hand as in:

```
char first_name[5] = { 'J', 'o', 'h', 'n', '\0' };
```

### Character Arrays vs. Strings

Each character value occupies a successive position in the array. Here the compiler is not smart enough to figure we are constructing a string and so we must add the null character `'\0'` by hand. If we had forgotten, the array of characters would have been just that, an array of characters, not a string.

### Null Added Automatically

The second method is much more convenient and is shown by:

```
char last_name[6] = "Minor";
```

Here too the characters occupy successive locations in the array. The compiler realizes we are constructing a string and *automatically adds the null terminator*, thus 6 slots in the array and NOT 5.

As already seen, when providing an initial value with an array, the size may be omitted, as in:

```
char other[] = "Tony Blurt";
```

Here, the size deduced by the compiler is 11 which includes space for the null terminator.

### Excluding Null

A special case exists in C when the size is set to exactly the number of characters used in the initialization *not including the null character*. As in:

```
char characters[7] = "No null";
```

Here the compiler deliberately excludes the null terminator. Here is an array of characters and not a string.

## Printing Strings

- 🔗 Strings may be printed by hand
- 🔗 Alternatively `printf` supports “%s”

```
char other[] = "Tony Blurt";
```

```
char *p;
p = other;
while(*p != '\0')
 printf("%c", *p++);
printf("\n");
```

```
int i = 0;
while(other[i] != '\0')
 printf("%c", other[i++]);
printf("\n");
```

```
printf("%s\n", other);
```

## Printing Strings

### `printf` “%s” Format Specifier

Strings may be printed by hand, character by character until the null is found or by using the “%s” format specifier to `printf`. `scanf` understands this format specifier too and will read a sequence of characters from the keyboard.

Consider the way: `printf("%s\n", other);`

actually works. Being an array, “other” generates the address of the first character in the array. If this address were, say, 2010 the “%s” format specifier tells `printf` to print the character stored at location 2010. This is the character “T”.

`printf` then increments its pointer to become 2011 (because `char` is being dealt with, there is no scaling of the pointer). The value at this location “o” is tested to see if it null. Since it is not, this value is printed too. Again the pointer is incremented and becomes 2012. The character in this location “n” is tested to see if it is null, since it is not, it is printed.

This carries on right through the “y”, space, “B”, “l”, “u”, “r” and “t”. With “t” the pointer is 2019. Since the “t” is not null, it is printed, the pointer is incremented. Now its value is 2020 and the value “\0” stored at that location is tested. `printf` breaks out of its loop and returns to the caller.

Consider also the chaos that would result if the array “characters” defined previously were thrown at `printf` and the “%s” format specifier. This array of characters did not contain the null terminator and, since there is no bounds checking in C, `printf` would continue printing characters randomly from memory until by the laws of chance found a byte containing zero. This is a very popular error in C programs.

## Null Really Does Mark the End!

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
 char other[] = "Tony Blurt";
```

```
 printf("%s\n", other);
```

```
 other[4] = '\0';
```

```
 printf("%s\n", other);
```

```
 return 0;
```

```
}
```

even though the rest of the data is still there, printf will NOT move past the null terminator

Tony Blurt  
Tony

other

|     |     |     |     |    |     |     |     |     |     |   |
|-----|-----|-----|-----|----|-----|-----|-----|-----|-----|---|
| 'T' | 'o' | 'n' | 'y' | 32 | 'B' | 'l' | 'u' | 'r' | 't' | 0 |
|-----|-----|-----|-----|----|-----|-----|-----|-----|-----|---|

---

## Null Really Does Mark the End!

The example here shows how **printf** will not move past the null terminator. In the first case, 11 characters are output (including the space).

When the null terminator is written into the fifth position in the array only the four characters before it are printed. Those other characters are still there, but simply not printed.

---

## Assigning to Strings

- § Strings may be initialised with “=”, but not assigned to with “=”
- § Remember the name of an array is a **CONSTANT** pointer to the zeroth element

```
#include <stdio.h>
#include <string.h>

int main(void)
{
 char who[] = "Tony Blurt";
 who = "John Minor"; X
 strcpy(who, "John Minor");
 return 0;
}
```

## Assigning to Strings

Don't make the mistake of trying to assign values into strings at run time as in:

```
who = "John Minor";
```

By trying to assign to “who” the compiler would attempt to assign to the address at which the “T” is stored (since “who” is the name of an array and therefore the address of the zeroth element). This address is a constant. Instead the Standard Library function **strcpy** should be used as in:

```
strcpy(who, "John Minor");
```

notice how the format is: **strcpy(destination, source);**

this routine contains a loop (similar to that contained in **printf**) which walks the string checking for the null terminator. While it hasn't been found it continues copying into the target array. It ensures the null is copied too, thus making “who” a valid string rather than just an array of characters. Notice also how **strcpy** does absolutely no bounds checking, so:

```
strcpy(who, "a really very long string indeed");
```

would overflow the array “who” and corrupt the memory around it. This would very likely cause the program to crash. A safer option entirely is to use **strcpy**'s cousin **strncpy** which is count driven:

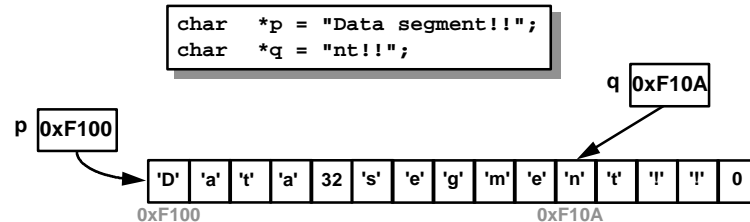
```
strncpy(who, "a really very long string indeed",
 sizeof(who));
```

This copies either up to the null terminator, or up to the count provided (here 11, the number of bytes yielded by **sizeof**). Unfortunately when **strncpy** hits the count first, it fails to null terminate. We have to do this by hand as in:

```
who[sizeof(who) - 1] = '\0';
```

## Pointing to Strings

- § To save us declaring many character arrays to store strings, the compiler can store them directly in the data segment
- § We need only declare a pointer
- § The compiler may recycle some of these strings, therefore we must NOT alter any of the characters



## Pointing to Strings

### Strings May be Stored in the Data Segment

The compiler stores strings in the data segment whenever we use double quotes and are not initializing an array of characters. For instance:

```
char str[] = "this is a string";
```

does *not* cause `str`, or the characters `"t"`, `"h"`, `"i"` etc. to be placed in the data segment. Instead we get a "normal" stack based array of characters. However, the entirely different initialization:

```
char *str = "this is a string";
```

declares a stack based pointer `"str"` pointing directly into the data segment.

The ANSI and ISO Standards committees thought it would be really neat if the compiler could optimize the storage of these strings. Thus the compiler is allowed to set more than one pointer into the same block of memory, as shown above. Obviously it can *only* do this when one string is a *substring* of another. If we had changed the initialization to read:

```
char *q = "NT!!";
```

or even:

```
char *q = "nt";
```

then the compiler would not have been able to perform this optimization. Other storage in the data segment would need to be allocated. Because we don't know how many pointers will be pointing into a block of storage it is inadvisable to write down any of these pointers. Really the declaration would be better as:

```
const char *p = "Data segment";
```

Which declares `"p"` as a pointer to a constant character. In fact it is not only `"D"` (the character to which `"p"` is set to point) which is the constant character, all the characters accessible by `"p"` become constant.



## Example

this utterly pointless statement causes the compiler to store the characters, unfortunately we forget to save the address

```
#include <stdio.h>

int main(void)
{
 char *p = "a string in the data segment\n";
 "a second string in the data segment\n";
 printf("a third string in the data segment\n");
 printf("%s", p);
 printf(p);
 return 0;
}
```

a third string in the data segment  
a string in the data segment  
a string in the data segment

## Example

The program above gives an insight into the nature of strings stored in the data segment. Each of the lines:

```
char *p = "a string in the data segment\n";
"a second string in the data segment\n";
printf("a third string in the data segment\n");
printf("%s", p);
```

and

cause strings to be stored in the data segment. The second of these is rather a waste of time (indeed most compilers will produce a warning to this effect) as although the characters are carefully stored by the compiler we forget to provide a variable to store the address. Thus the address is forgotten and unless we trawl the data segment looking for them we'd have a hard task finding them. A smart compiler may decide not to store these characters at all. Although the third statement:

```
printf("a third string in the data segment\n");
```

does not provide a variable to store the address, it does pass the address as the first and only parameter to **printf** (remember the address will be that of the "a" at the front of the string). **printf** takes this address and prints the characters stored at successive locations until the null is encountered.

The fourth line also causes characters to be placed in the data segment, though perhaps not as obviously as with the previous statements. Here only three characters are stored, "%", "s" and the null, "\0". As before the address of the first character, the "%", is passed to `printf`, which encounters the %s format specifier. This instructs it to take the address stored in "p" and to walk down the array of characters it finds there, stopping when the null is encountered.

The statement:

```
printf(p);
```

passes the pointer "p" directly to `printf`. Instead of having to wade through a "%s", it is handed a pointer to the character "a" on the front of "a second string in the data segment\n".

---

## Multidimensional Arrays

- ❗ **C does not support multidimensional arrays**
- ❗ **However, C does support arrays of any type including arrays of arrays**

```
float rainfall[12][365];
```

“rainfall” is an array of 12 arrays of 365 float

```
short exam_marks[500][10];
```

“exam\_marks” is an array of 500 arrays of 10 short int

```
const int brighton = 7;
int day_of_year = 238;

rainfall[brighton][day_of_year] = 0.0F;
```

## Multidimensional Arrays

Sometimes a simple array just isn't enough. Say a program needed to store the rainfall for 12 places for each of the 365 days in the year (pretend it isn't a leap year). 12 arrays of 365 reals would be needed. This is exactly what:

```
float rainfall[12][365];
```

gives. Alternatively imagine a (big) college with up to 500 students completing exams. Each student may sit up to 10 exams. This would call for 500 arrays of 10 integers (we're not interested in fractions of a percent, so whole numbers will do). This is what:

```
short exam_marks[500][10];
```

gives. Although it may be tempting to regard these variables as multi dimensional arrays, C doesn't treat them as such. Firstly, to access the 5th location's rainfall on the 108th day of the year we would write:

```
printf("rainfall was %f\n", rainfall[5][108]);
```

and NOT (as in some languages):

```
printf("rainfall was %f\n", rainfall[5, 108]);
```

which wouldn't compile. In fact, C expects these variables to be initialized as arrays of arrays. Consider:

```
int rainfall_in_mm_per_month[4][12] = {
 { 17, 15, 20, 25, 30, 35, 48, 37, 28, 19, 18, 10 },
 { 13, 13, 18, 20, 27, 29, 29, 26, 20, 15, 11, 8 },
 { 7, 9, 11, 11, 12, 14, 16, 13, 11, 8, 6, 3 },
 { 29, 35, 40, 44, 47, 51, 59, 57, 42, 39, 35, 28 },
};
```

where each of the four arrays of twelve floats are initialized separately.

## Review

§ How many times does the following program loop?

```
#include <stdio.h>

int main(void)
{
 int i;
 int a[10];

 for(i = 0; i <= 10; i++) {
 printf("%d\n", i);
 a[i] = 0;
 }

 return 0;
}
```

---

## Review

Time for a break. How many times will the loop execute?

---

## Summary

- § **Arrays are declared with a type, a name, “[ ]” and a CONSTANT**
- § **Access to elements by array name, “[ ]” and an integer**
- § **Arrays passed into functions by pointer**
- § **Pointer arithmetic**
- § **Strings - arrays of characters with a null terminator**
- § **Sometimes compiler stores null for us (when double quotes are used) otherwise we have to store it ourselves**

---

## Summary

---



---

---

## Arrays Practical Exercises

---

---

**Directory:        ARRAYS**

1. In the file “**ARRAY1.C**”. There is a call to a function:

```
void print_array(int a[], int count);
```

Implement this function using either pointer or array index notation. This file also contains a call to the preprocessor macro **ASIZE** which determines the number of elements in an array. Don't worry about this, the macro works and how it works will be discussed in a later chapter.

2. In the file “**ARRAY2.C**” there is a call to the function:

```
float average(int a[], int count);
```

Implement this function which averages the “count” values in the array “a” and returns the answer as a float. You will need to cut and paste your **print\_array** function from the previous exercise.

3. In “**ARRAY3.C**” you need to write the **copy\_array** function, which has the prototype:

```
void copy_array(int to[], int from[], int count);
```

which copies the array passed as its second parameter into the array passed as its first parameter. The third parameter is a count of the number of elements to be copied. You should assume the target array has a number of elements greater than or equal to that of the source array.

Write this routine using either pointers or array index notation. Once again, you will need to cut and paste your **print\_array** function.

4. In “**ARRAY4.C**”, implement the function

```
int *biggest(int *a, int count);
```

such that the function returns a pointer to the largest element in the array pointed to by “a”.

5. In “**ARRAY5.C**”, there is a call to the **print\_in\_reverse** function which has the following prototype:

```
void print_in_reverse(float *a, int count);
```

**Using pointers**, write this function to print the array in reverse order.

6. Open the file “**STRING1.C**”. There are two strings declared and a call to the function **len** for each one. The function has the prototype

```
int len(char *str);
```

and returns the number of characters in the string. Implement this function by walking down the array searching for the null terminator character.

7. Open the file “**STRING2.C**”. Implement the **count\_char** function with the prototype:

```
int count_char(char *str, char what);
```

which returns the number of occurrences of the character *what* within the string *str*.

---



8. Open the file “**STRING3.C**”. There is a call to the **copy\_string** function which has the prototype:

```
void copy_string(char *to, char *from);
```

Notice that unlike the **copy\_array** function, there is no third parameter to indicate the number of characters to be copied. Always assume there is enough storage in the target array to contain the data from the source array.

9. Open the file “**LOTTERY.C**”. If you run the program you will see that 6 random numbers in the range 1..49 are stored in the selected array before it is printed. No checking is done to see if the same number occurs more than once.

Add the required checking and as a final touch, sort the numbers before you print them.

Could you think of a better strategy for generating the 6 different numbers?



---

---

## Arrays Solutions

---

---

1. In the file "ARRAY1.C" implement the function

```
void print_array(int a[], int count);
```

*This solution uses array index notation*

```
#include <stdio.h>

#define A_SIZE(A) sizeof(A)/sizeof(A[0])

void print_array(int a[], int count);

int main(void)
{
 int values[] = { 17, 27, 34, 52, 79,
 87, 103, 109, 187, 214 };

 printf("The array contains the following values\n");
 print_array(values, A_SIZE(values));

 return 0;
}

void print_array(int a[], int count)
{
 int i;

 for(i = 0; i < count; i++)
 printf("%i\t", a[i]);

 printf("\n");
}
```

---

2. In the file "ARRAY2.C" implement the function

```
float average(int a[], int count);
```

*The only problem here is to ensure that the average is calculated using floating point arithmetic. This will not necessarily happen since the routine deals with an array of integers. By declaring the sum as a float, when the sum is divided by the number of elements, floating point division is achieved.*

```
#include <stdio.h>

#define A_SIZE(A) sizeof(A)/sizeof(A[0])

void print_array(int a[], int count);
float average(int a[], int count);
```

---

```
int main(void)
{
 int values[] = { 17, 27, 34, 52, 79,
 87, 103, 109, 187, 214 };

 printf("The array contains the following values\n");
 print_array(values, A_SIZE(values));

 printf("and has an average of %.2f\n",
 average(values, A_SIZE(values)));

 return 0;
}

void print_array(int a[], int count)
{
 int i;

 for(i = 0; i < count; i++)
 printf("%i\t", a[i]);

 printf("\n");
}

float average(int a[], int count)
{
 float av = 0.0F;
 int i;

 for(i = 0; i < count; i++)
 av += a[i];

 return av / count;
}
```

---

3. In "ARRAY3.C" implement the function:

```
void copy_array(int to[], int from[], int count);

#include <stdio.h>

#define A_SIZE(A) sizeof(A)/sizeof(A[0])

void print_array(int a[], int count);
void copy_array(int to[], int from[], int count);
```

---

```

int main(void)
{
 int orig[6] = { 17, 27, 37, 47, 57, 67 };
 int copy[6] = { -1, -1, -1, -1, -1, -1 };

 copy_array(copy, orig, A_SIZE(copy));

 printf("The copy contains the following values\n");
 print_array(copy, A_SIZE(copy));

 return 0;
}

/* This function is as before
*/
void print_array(int a[], int count)
{
 int i;

 for(i = 0; i < count; i++)
 printf("%i\t", a[i]);

 printf("\n");
}

void copy_array(int to[], int from[], int count)
{
 int i;

 for(i = 0; i < count; i++)
 to[i] = from[i];
}

```

---

4. In "ARRAY4.C", implement the function

```
int *biggest(int *a, int count);
```

The function "biggest" initializes a pointer "current\_biggest" to the first element of the array. It then starts searching one beyond this element (since it is pointless to compare the first element with itself).

```

#include <stdio.h>

#define A_SIZE(A) sizeof(A)/sizeof(A[0])

int* biggest(int *a, int count);

```

---

```

int main(void)
{
 int values[16] = { 47, 17, 38, 91, 33, 24, 99, 35, 42, 10,
 11, 43, 32, 97, 108, -8 };
 int *p;

 p = biggest(values, A_SIZE(values));

 printf("the biggest element in the array is %i\n", *p);

 return 0;
}

int* biggest(int *a, int count)
{
 int *current_biggest = a;
 int *p = a + 1;
 int *end = a + count;

 while(p < end) {
 if(*current_biggest < *p)
 current_biggest = p;
 p++;
 }
 return current_biggest;
}

```

---

5. In "ARRAY5.C" implement the `print_in_reverse` function which has the following prototype:

```
void print_in_reverse(float *a, int count);
```

The -1 in the initialization of "end" is important, since without it, "end" points one beyond the end of the array and this element is printed within the loop. Where no -1 is used, "\*end--" would need to be changed to "--end".

```

#include <stdio.h>

#define A_SIZE(A) sizeof(A)/sizeof(A[0])

void print_in_reverse(float a[], int count);

int main(void)
{
 float values[6] = { 12.1F, 22.2F, 32.3F,
 42.4F, 52.5F, 62.6F };

 printf("The array in reverse\n");
 print_in_reverse(values, A_SIZE(values));

 return 0;
}

```

---

```

void print_in_reverse(float a[], int count)
{
 float * end = a + count - 1;

 while(end >= a)
 printf("%.1f\t", *end--);

 printf("\n");
}

```

---

6. In "STRING1.C" implement the function `len` which has the prototype

```
int len(char *str);
```

*Although the while loop within `slen` is already concise, it would be possible to write "`while(*str++)`" which would achieve the same results. This would rely on the ASCII values of the characters being non zero (true). When the null terminator is encountered, it has a value of zero (false).*

```

#include <stdio.h>

int slen(char *str);

int main(void)
{
 char s1[] = "Question 6.";
 char s2[] = "Twenty eight characters long";

 printf("The string \"%s\" is %i characters long\n",
 s1, slen(s1));

 printf("The string \"%s\" is %i characters long\n",
 s2, slen(s2));

 return 0;
}

int slen(char* str)
{
 int count = 0;

 while(*str++ != '\0')
 count++;

 return count;
}

```

---



7. In "STRING2.C" implement the `count_char` function which has the prototype:

```
int count_char(char *str, char what);
```

The solution uses the tricky, yet popular, construct "`n += first == second`". This relies on the guaranteed result of a boolean expression being 1 or 0. If first and second are not alike false, i.e. 0, results. When added into the running total, no difference is made. If first and second are the same true, i.e. 1, results. When added to the running total, the total is increased by one more. By the end of the loop we have counted all the trues. This is a count of the matching characters.

```
#include <stdio.h>

int count_char(char *str, char what);

int main(void)
{
 char s1[] = "Twenty eight characters long";
 char s2[] = "count_char";

 printf("The string \"%s\" contains '%c' %i times\n",
 s1, 'e', count_char(s1, 'e'));

 printf("The string \"%s\" contains '%c' %i times\n",
 s2, 'c', count_char(s2, 'c'));

 return 0;
}

int count_char(char *str, char what)
{
 int count = 0;

 while(*str != '\0') {
 count += *str == what;
 str++;
 }

 return count;
}
```

---

8. In "STRING3.C" implement:

```
void copy_string(char *to, char *from);
```

The `copy_string` function uses one of the most concise C constructs imaginable. Here the "`=`" is not a mistake (normally "`==`" would be intended). One byte at a time is copied via the "`=`", both pointers being moved to the next byte by the "`++`" operators. The byte that has just been copied is then tested. C treats any **non zero** value as **true**. Thus if we had copied 'A' its ASCII value would be 65 and thus true. Copying the next character gives another ASCII value and so on. At the end of the "from" string is a null terminator. This is the only character whose ASCII value is zero. **Zero** always **tests false**. Don't forget the assignment must complete before the value may be tested.

```
#include <stdio.h>

void copy_string(char to[], char from[]);
```

---

```

int main(void)
{
 char s1[] = "Twenty eight characters long";
 char s2[] = "Important data";

 copy_string(s1, s2);

 printf("The string s1 now contains \"%s\"\n", s1);

 return 0;
}

void copy_string(char to[], char from[])
{
 while(*to++ = *from++)
 ;
}

```

---

9. In “**LOTTERY.C**” 6 random numbers in the range 1..49 are stored in the selected array before printing. No checking is done to see if the same number occurs more than once. Add the required checking and as a final touch, sort the numbers before you print them.

*The search function checks to see if the new number to be added is already present in the array. Although it is a “brute force” approach, there are only a maximum of 6 numbers so this is not a problem. Once chosen, the Standard Library routine qsort is used to sort the numbers. This routine requires the int\_compare function. Look up qsort in the help to understand what is going on here.*

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TOTAL_NUMBER 6

void seed_generator(void);
int get_rand_in_range(int from, int to);
int search(int target, int array[], int size);
int int_compare(const void* v_one, const void* v_two);

int main(void)
{
 int i;
 int r;
 int selected[TOTAL_NUMBER];

 seed_generator();

 for(i = 0; i < TOTAL_NUMBER; i++) {
 do
 r = get_rand_in_range(1, 49);
 while(search(r, selected, i));

 selected[i] = r;
 }
}

```

---

```

 qsort(selected, TOTAL_NUMBER, sizeof(int), int_compare);

 for(i = 0; i < TOTAL_NUMBER; i++)
 printf("%i\t", selected[i]);

 printf("\n");

 return 0;
 }

 int get_rand_in_range(int from, int to)
 {
 int min = (from > to) ? to : from;

 return rand() % abs(to - from + 1) + min;
 }

 void seed_generator(void)
 {
 time_t now;

 now = time(NULL);
 srand((unsigned)now);
 }

 int search(int target, int array[], int size)
 {
 int i;

 for(i = 0; i < size; i++)
 if(array[i] == target)
 return 1;

 return 0;
 }

 int int_compare(const void* v_one, const void* v_two)
 {
 const int* one = v_one;
 const int* two = v_two;

 return *one - *two;
 }

```

Could you think of a better strategy for generating the 6 different numbers?

*This solution uses an array of "hits" with 49 slots. Say 17 is drawn, location 17 in the array is tested to see if 17 has been drawn before. If it has, the location will contain 1. If not (the array is cleared at the start) array element 17 is set to 1. We are finished when there are 6 1s in the array. The index of each slot containing "1" is printed, i.e. 17 plus the other five. Since the array is searched in ascending order there is no need for sorting.*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX 49
#define TOTAL_NUMBER 6

void seed_generator(void);
int get_rand_in_range(int from, int to);
int count_entries(int array[]);

int main(void)
{
 int i = 0;
 int r;
 int all[MAX + 1] = { 0 }; /* Nothing selected */

 seed_generator();

 while(count_entries(all) < TOTAL_NUMBER) {
 do
 r = get_rand_in_range(1, 49);
 while(all[r]);

 all[r] = 1;
 }

 for(i = 1; i <= MAX; i++)
 if(all[i])
 printf("%i\t", i);

 printf("\n");

 return 0;
}

int get_rand_in_range(int from, int to)
{
 int min = (from > to) ? to : from;

 return rand() % abs(to - from + 1) + min;
}

void seed_generator(void)
{
 time_t now;

 now = time(NULL);
 srand((unsigned)now);
}
```

---

```
int count_entries(int array[])
{
 int i;
 int total;

 for(i = 1, total = 0; i <= MAX; i++)
 total += array[i] == 1;

 return total;
}
```

---



---

---

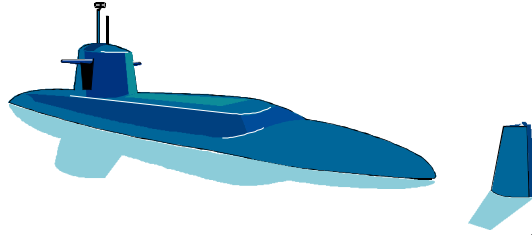
## Structures in C

---

---

## Structures in C

- § Concepts
- § Creating a structure template
- § Using the template to create an instance
- § Initialising an instance
- § Accessing an instance's members
- § Passing instances to functions
- § Linked lists



---

## Structures in C

This chapters investigates structures (records) in C.



## Concepts

- § **A structure is a collection of one or more variables grouped together under a single name for convenient handling**
- § **The variables in a structure are called *members* and may have any type, including arrays or other structures**
- § **The steps are:**
  - **set-up a template (blueprint) to tell the compiler how to build the structure**
  - **Use the template to create as many instances of the structure as desired**
  - **Access the members of an instance as desired**

---

## Concepts

Thus far we have examined arrays. The fundamental property of the array is that all of the elements are exactly the same type. Sometimes this is not what is desired. We would like to group things of potentially different types together in a tidy “lump” for convenience.

Whereas the parts of an array are called “elements” the parts of a structure are called “members”.

Just as it is possible to have arrays of any type, so it is possible to have any type within a structure (except void). It is possible to place arrays inside structures, structures inside structures and possible to create arrays of structures.

The first step is to set up a blueprint to tell the compiler how to make the kinds of structures we want. For instance, if you wanted to build a car, you’d need a detailed drawing first. Just because you possess the drawing does not mean you have a car. It would be necessary to take the drawing to a factory and get them to make one. The factory wouldn’t just stop at one, it could make two, three or even three hundred. Each car would be a single individual instance, with its own doors, wheels, mirrors etc.

---

## Setting up the Template

Structure templates are created by using the **struct** keyword

```
struct Date
{
 int day;
 int month;
 int year;
};
```

```
struct Book
{
 char title[80];
 char author[80];
 float price;
 char isbn[20];
};
```

```
struct Library_member
{
 char name[80];
 char address[200];
 long member_number;
 float fines[10];
 struct Date dob;
 struct Date enrolled;
};
```

```
struct Library_book
{
 struct Book b;
 struct Date due;
 struct Library_member *who;
};
```

## Setting up the Template

The four examples above show how the template (or blueprint) is specified to the compiler. The keyword **struct** is followed by a name (called a *tag*). The tag helps us to tell the compiler which of the templates we're interested in. Just because we have a structure template does not mean we have any structures. No stack, data segment or heap memory is allocated when we create a structure template. Just because we have a blueprint telling us that a book has a title, author, ISBN number and price does not mean we have a book.

### Structures vs. Arrays

The Date structure, consisting as it does of three integers offers advantages over an array of three integers. With an array the elements would be numbered 0, 1 and 2. This would give no clue as to which one was the day, which the month and which the year. Using a structure gives these members names so there can be no confusion.

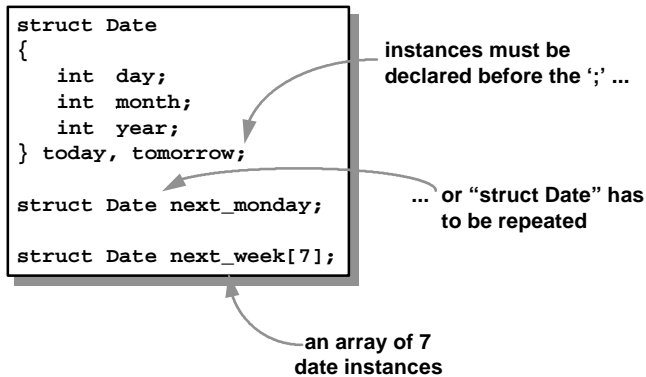
The Book structure not only contains members of different types (**char** and **float**) it also contains three arrays.

The Library\_member structure contains two Date structures, a date of birth as well as a date of enrolment within the library.

Finally the Library\_book structure contains a Book structure, a Date structure and a pointer to a Library\_member structure.

## Creating Instances

- ☞ Having created the template, an instance (or instances) of the structure may be declared



## Creating Instances

### Instance?

The template gives the compiler all the information it needs on how to build an instance or instances. An "instance" is defined in the dictionary as "an example, or illustration of". Going back to the car example, the blueprint enables us to make many cars. Each car is different and distinct. If one car is painted blue, it doesn't mean *all* cars are painted blue. Each car is an "instance". Each instance is separate from every other instance and separate from the template. There is only ever one template (unless you want to start building slightly different kinds of car).

Above, the Date template is used to create two date instances, "today" and "tomorrow". Any variable names placed after the closing brace and before the terminating semicolon are structures of the specified type.

After the semicolon of the structure template, "**struct Date**" needs to be repeated.

With the array "next\_week", each element of the array is an individual Date structure. Each element has its own distinct day, month and year members. For instance, the day member of the first (i.e. zeroth) date would be accessed with:

**next\_week[0].day**

the month of the fourth date would be accessed with:

**next\_week[3].month**

and the year of the last date with:

**next\_week[6].year**

## Initialising Instances

- Structure instances may be initialised using braces (as with arrays)

```
int primes[7] = { 1, 2, 3, 5, 7, 11, 13 };

struct Date bug_day = { 1, 1, 2000 };

struct Book k_and_r = {
 "The C Programming Language 2nd edition",
 "Brian W. Kernighan and Dennis M. Ritchie",
 31.95,
 "0-13-110362-8"
};
```

```
struct Book
{
 char title[80];
 char author[80];
 float price;
 char isbn[20];
};
```

## Initializing Instances

In the last chapter we saw how braces were used in the initialization of arrays, as in the “primes” example above. The seven slots in the array are filled with the corresponding value from the braces.

A similar syntax is used in the initialization of structures. With the initialization of “bug\_day” above, the first value 1 is assigned into bug\_day’s first member, “day”. The second value “1” is assigned into bug\_day’s second member, “month”. The 2000 is assigned into bug\_day’s third member “year”. It is just as though we had written:

```
struct Date bug_day;

bug_day.day = 1;
bug_day.month = 1;
bug_day.year = 2000;
```

With the initialization of “k\_and\_r” the first string is assigned to the member “title”, the second string assigned to the member “author” etc. It is as though we had written:

```
struct Book k_and_r;

strcpy(k_and_r.title, "The C Programming Language 2nd edition");
strcpy(k_and_r.author, "Brian W. Kernighan and Dennis M. Ritchie");
k_and_r.price = 31.95;
strcpy(k_and_r.isbn, "0-13-110362-8");
```

## Structures Within Structures

```

struct Library_member
{
 char name[80];
 char address[200];
 long member_number;
 float fines[10];
 struct Date dob;
 struct Date enrolled;
};

```

initialises first 4 elements of array "fines", remainder are initialised to 0.0

```

struct Library_member m = {
 "Arthur Dent",
 "16 New Bypass",
 42,
 { 0.10, 2.58, 0.13, 1.10 },
 { 18, 9, 1959 },
 { 1, 4, 1978 }
};

```

initialises day, month and year of "dob"

initialises day, month and year of "enrolled"

## Structures Within Structures

We have already seen that it is possible to declare structures within structures, here is an example of how to initialize them. To initialize a structure or an array braces are used. To initialize an array within a structure two sets of braces must be used. To initialize a structure within a structure, again, two sets of braces must be used.

It is as though we had written:

```

struct Library_member m;

strcpy(m.name, "Arthur Dent");
strcpy(m.address, "16 New Bypass");
m.member_number = 42;
m.fines[0] = 0.10; m.fines[1] = 2.58; m.fines[2] = 0.13; m.fines[3] = 1.10;
m.fines[4] = 0.00; m.fines[5] = 0.00; m.fines[6] = 0.00; m.fines[7] = 0.00;
m.fines[8] = 0.00; m.fines[9] = 0.00;
m.dob.day = 18; m.dob.month = 9; m.dob.year = 1959;
m.enrolled.day = 1; m.enrolled.month = 4; m.enrolled.year = 1978;

```

**Reminder -  
Avoid  
Leading  
Zeros**

Although a small point, notice the date initialization:

```
{ 18, 9, 1959 }
```

above. It is important to resist the temptation to write:

```
{ 18, 09, 1959 }
```

since the leading zero introduces an octal number and "9" is not a valid octal digit.

## Accessing Members

- Members are accessed using the instance name, “.” and the member name

```

struct Library_member
{
 char name[80];
 char address[200];
 long member_number;
 float fines[10];
 struct Date dob;
 struct Date enrolled;
}

struct Library_member m;

printf("name = %s\n", m.name);
printf("membership number = %li\n", m.member_number);
printf("fines: ");
for(i = 0; i < 10 && m.fines[i] > 0.0; i++)
 printf("f%.2f ", m.fines[i]);
printf("\njoined %i/%i/%i\n", m.enrolled.day,
 m.enrolled.month, m.enrolled.year);

```

## Accessing Members

Members of structures are accessed using C's “.” operator. The syntax is:

**structure\_variable.member\_name**

### Accessing Members Which are Arrays

If the member being accessed happens to be an array (as is the case with “fines”), square brackets must be used to access the elements (just as they would with any other array):

**m.fines[0]**

would access the first (i.e. zeroth) element of the array.

### Accessing Members Which are Structures

When a structure is nested inside a structure, two dots must be used as in

**m.enrolled.month**

which literally says “the member of ‘m’ called ‘enrolled’, which has a member called ‘month’”. If “month” were a structure, a third dot would be needed to access one of its members and so on.

## Unusual Properties

☞ Structures have some very “un-C-like” properties, certainly when considering how arrays are handled

|                          | <u>Arrays</u>             | <u>Structures</u>    |
|--------------------------|---------------------------|----------------------|
| Name is                  | pointer to zeroth element | the structure itself |
| Passed to functions by   | pointer                   | value or pointer     |
| Returned from functions  | no way                    | by value or pointer  |
| May be assigned with “=” | no way                    | yes                  |

## Unusual Properties

### Common Features Between Arrays and Structures

Structures and arrays have features in common. Both cause the compiler to group variables together. In the case of arrays, the variables are elements and have the same type. In the case of structures the variables are members and may have differing type.

### Differences Between Arrays and Structures

Despite this, the compiler does not treat arrays and structures in the same way. As seen in the last chapter, in C the name of an array yields the address of the zeroth element of the array. With structures, the name of a structure instance is *just* the name of the structure instance, NOT a pointer to one of the members.

When an array is passed to a function you have no choice as to how the array is passed. As the name of an array is “automatically” a pointer to the start, arrays are passed by pointer. There is no mechanism to request an array to be passed by value. Structures, on the other hand may be passed either by value or by pointer.

An array cannot be returned from a function. The nature of arrays makes it possible to return a pointer to a particular element, however this is not be the same as returning the whole array. It could be argued that by returning a pointer to the first element, the whole array is returned, however this is a somewhat weak argument. With structures the programmer may choose to return a structure or a pointer to the structure.

Finally, arrays cannot be assigned with C’s assignment operator. Since the name of an array is a constant pointer to the first element, it may not appear on the left hand side of an assignment (since no constant may be assigned to). Two structures may be assigned to one another. The values stored in the members of the right hand structure are copied over the members of the left hand structure, even if these members are arrays or other structures.

## Instances may be Assigned

- § Two structure instances may be assigned to one another via “=”
- § All the members of the instance are copied (including arrays or other structures)

```
struct Library_member m = {
 "Arthur Dent",

};
struct Library_member tmp;
tmp = m;
```

copies array “name”, array “address”, long integer “member\_number”, array “fines”, Date structure “dob” and Date structure “enrolled”

## Instances May be Assigned

### Cannot Assign Arrays

It is not possible to assign arrays in C, consider:

```
int a[10];
int b[10];

a = b;
```

The name of the array “a” is a constant pointer to the zeroth element of “a”. A constant may not be assigned to, thus the compiler will throw out the assignment “a = b”.

### Can Assign Structures Containing Arrays

Consider:

```
struct A {
 int array[10];
};
struct A a, b;

a = b;
```

Now both instances “a” and “b” contain an array of 10 integers. The ten elements contained in “b.array” are copied over the ten elements in “a.array”. Not only does this statement compile, it also works! All the members of a structure are copied, no matter how complicated they are. Members which are arrays are copied, members which are nested structures are also copied.



## Passing Instances to Functions

- § An instance of a structure may be passed to a function by value or by pointer
- § Pass by value becomes less and less efficient as the structure size increases
- § Pass by pointer remains efficient regardless of the structure size

```
void by_value(struct Library_member);
void by_reference(struct Library_member *);

by_value(m);
by_reference(&m);
```

compiler writes a pointer  
(4 bytes?) onto the stack

compiler writes 300+  
bytes onto the stack

## Passing Instances to Functions

### Pass by Value or Pass by Reference?

As a programmer you have a choice of passing a structure instance either by value or by pointer. It is important to consider which of these is better. When passing an array to a function there is no choice. There isn't a choice for one important reason, it is invariably *less* efficient to pass an array by value than it is by pointer. Consider an array of 100 **long int**. Since a **long int** is 4 bytes in size, and C guarantees to allocate an array in contiguous storage, the array would be a total of 400 bytes.

If the compiler used pass by value, it would need to copy 400 bytes onto the stack. This would be time consuming and we may, on a small machine, run out of stack space (remember we would need to maintain two copies - the original and the parameter). Here we are considering a "small" array. Arrays can very quickly become larger and occupy even more storage.

When the compiler uses pass by reference it copies a pointer onto the stack. This pointer may be 2 or 4 bytes, perhaps larger, but there is no way its size will compare unfavorably with 400 bytes.

The same arguments apply to structures. The `Library_member` structure is over 300 bytes in size. The choice between copying over 300 bytes vs. copying around 4 bytes is an easy one to make.

## Pointers to Structures

- § **Passing pointers to structure instances is more efficient**
- § **Dealing with an instance at the end of a pointer is not so straightforward!**

```
void member_display(struct Library_member *p)
{
 printf("name = %s\n", (*p).name);
 printf("membership number = %li\n", (*p).member_number);

 printf("fines: ");
 for(i = 0; i < 10 && (*p).fines[i] > 0.0; i++)
 printf("f%.2f ", (*p).fines[i]);

 printf("\njoined %i/%i/%i\n", (*p).enrolled.day,
 (*p).enrolled.month, (*p).enrolled.year);
}
```

---

## Pointers to Structures

Passing a pointer to a structure in preference to passing the structure by value will almost invariably be more efficient. Unfortunately when a pointer to a structure is passed, coding the function becomes tricky. The rather messy construct:

**( \*p ).name**

is necessary to access the member called "name" (an array of characters) of the structure at the end of the pointer.

---

## Why ( \*p ) . name ?

- ⌘ The messy syntax is needed because “.” has higher precedence than “\*”, thus:

**\*p.name**

means “what p.name points to” (a problem because there is no structure instance “p”)

- ⌘ As Kernighan and Ritchie foresaw pointers and structures being used frequently they invented a new operator

**p->name      =      (\*p).name**

## Why ( \*p ) . name?

The question occurs as to why:      **( \*p ) . name**

is necessary as opposed to:      **\*p . name**

The two operators “\*” and “.” live at different levels in the precedence table. In fact “.”, the structure member operator, is one of the highest precedence operators there is. The “pointer to” operator, “\*”, although being a high precedence operator is not quite as high up the table.

Thus:      **\*p . name**

would implicitly mean:      **\*( p . name )**

For this to compile there would need to be a structure called “p”. However “p” does not have type “structure”, but “pointer to structure”. Things get worse. If “p” were a structure after all, the name member would be accessed. The “\*” operator would find where “p.name” pointed. Far from accessing what we thought (a pointer to the zeroth element of the array) we would access the first character of the name. With **printf**’s fundamental inability to tell when we’ve got things right or wrong, printing the first character with the “%s” format specifier would be a fundamental error (**printf** would take the ASCII value of the character, go to that location in memory and print out all the bytes it found there up until the next byte containing zero).

### A New Operator

Since Kernighan and Ritchie foresaw themselves using pointers to structures frequently, they invented an operator that would be easier to use. This new operator consists of two separate characters “-” and “>” combined together into “->”. This is similar to the combination of divide, “/”, and multiply, “\*”, which gives the open comment sequence.

The messy **( \*p ) . name** now becomes **p->name** which is both easier to write and easier to read.

## Using `p->name`

§ Now dealing with the instance at the end of the pointer is more straightforward

```
void member_display(struct Library_member *p)
{
 printf("name = %s\n", p->name);
 printf("address = %s\n", p->address);
 printf("membership number = %li\n", p->member_number);
 printf("fines: ");
 for(i = 0; i < 10 && p->fines[i] > 0.0; i++)
 printf("f%.2f ", p->fines[i]);
 printf("\njoined %i/%i/%i\n", p->enrolled.day,
 p->enrolled.month, p->enrolled.year);
}
```

---

## Using `p->name`

As can be seen from the code above, the notation:

`p->name`

although exactly equivalent to:

`(*p).name`

is easier to read, easier to write and easier to understand. All that is happening is that the member "name" of the structure at the end of the pointer "p" is being accessed.

Note:

`p->enrolled.day`

and NOT:

`p->enrolled->day`

since "enrolled" is a structure and not a pointer to a structure.

---

## Pass by Reference - Warning

- ⚠ Although pass by reference is more efficient, the function can alter the structure (perhaps inadvertently)
- ⚠ Use a pointer to a constant structure instead

```
void member_display(struct Library_member *p)
{
 printf("fines: ");
 for(i = 0; i < 10 && p->fines[i] = 0.0; i++)
 printf("f%.2f ", p->fines[i]);
}
```

function alters  
the library  
member instance

```
void member_display(const struct Library_member *p)
{

}
```



## Pass by Reference - Warning

We have already seen how passing structure instances by reference is more efficient than pass by value. However, never forget that when a pointer is passed we have the ability to alter the thing at the end of the pointer. This is certainly true with arrays where any element of the array may be altered by a function passed a pointer to the start.

Although we may not intend to alter the structure, we may do so accidentally. Above is one of the most popular mistakes in C, confusing “=” with “==”. The upshot is that instead of testing against 0.0, we assign 0.0 into the zeroth element of the “fines” array. Thus the array, and hence the structure are changed.

### const to the Rescue!

The solution to this problem which lies with the **const** keyword (discussed in the first chapter). In C it is possible to declare a pointer to a constant. So:

```
int *p;
```

declares “p” to be a pointer to an integer, whereas:

```
const int *p;
```

declares “p” to be a pointer to a constant integer. The pointer “p” may change, so

```
p++;
```

would be allowed. However the value at the end of the pointer could not be changed, thus

```
*p = 17;
```

would NOT compile. The parameter “p” to the function **member\_display** has type “pointer to constant structure Library\_member” meaning the structure Library member on the end of the pointer cannot be changed.

## Returning Structure Instances

- § Structure instances may be returned by value from functions
- § This can be as inefficient as with pass by value
- § Sometimes it is convenient!

```

struct Complex add(struct Complex a, struct Complex b)
{
 struct Complex result = a;
 result.real_part += b.real_part;
 result.imag_part += b.imag_part;
 return result;
}

struct Complex c1 = { 1.0, 1.1 };
struct Complex c2 = { 2.0, 2.1 };
struct Complex c3;
c3 = add(c1, c2); /* c3 = c1 + c2 */

```

## Returning Structure Instances

As well as pass by value, it is also possible to return structures by value in C. The same consideration should be given to efficiency. The larger the structure the less efficient return by value becomes as opposed to return by pointer. Sometimes the benefits of return by value outweigh the inefficiencies. Take for example the code above which manipulates complex numbers. The `add` function returns the structure "result" by value. Consider this version which attempts to use return by pointer:

```

struct Complex* add(struct Complex a, struct Complex b)
{
 struct Complex result = a;
 /* as above */
 return &result;
}

```

This function contains a fatal error! The variable "result" is stack based, thus it is allocated on entry into the function and deallocated on exit from the function. When this function returns to the calling function it hands back a pointer to a piece of storage which has been deallocated. Any attempt to use that storage would be very unwise indeed. Here is a working version which attempts to be as efficient as possible:

```

void add(struct Complex *a, struct Complex *b, struct Complex
 *result)
{
 result->real_part = a->real_part + b->real_part;
 result->imag_part = a->imag_part + b->imag_part;
}

```

Pass by pointer is used for all parameters. There is no inefficient return by value, however consider how this function must be called and whether the resulting code is as obvious as the code above:

```

struct Complex c1 = { 1.0, 1.1 }, c2 = { 2.0, 2.1 }, c3;
add(&c1, &c2, &c3);

```

## Linked Lists

- § A linked list node containing a single forward pointer may be declared as follows

```
struct Node {
 int data; /* or whatever */
 struct Node *next_in_line;
};
```

pointer to next Node structure

- § A linked list node containing a forward and a backward pointer may be declared as follows

```
struct Node {
 int data;
 struct Node *next_in_line;
 struct Node *previous_in_line;
};
```

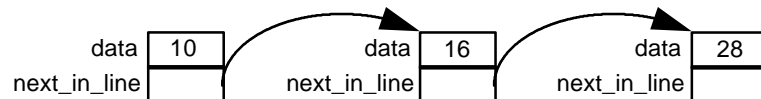
pointer to next Node structure

pointer to previous Node structure

## Linked Lists

It is possible to declare and manipulate any number of “advanced” data structures in C, like linked lists, binary trees, “red/black” trees, multi threaded trees, directed graphs and so on.

Above is the first step in manipulating linked lists, i.e. declaring the template. This particular template assumes the linked list will contain integers. The sort of picture we’re looking for is as follows:



where each structure contains one integer and one pointer to the next structure. The integer is stored in the member “data”, while the pointer is stored in the member “next\_in\_line”.

### A Recursive Template?

The structure template: 

```
struct Node {
 int data;
 struct Node* next_in_line;
};
```

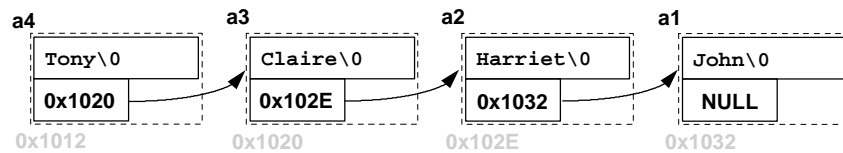
looks rather curious because the structure refers to itself. What it says is “a Node structure consists of an integer, followed by a pointer to another Node structure”. Although the compiler is not entirely sure about the “followed by a pointer to another Node structure” it is sure about pointers and how many bytes they occupy. Thus it creates a pointer sized “hole” in the structure and proceeds onwards.

## Example

```
#include <stdio.h>

struct Node {
 char name[10];
 struct Node *next_in_line;
};

struct Node a1 = { "John", NULL };
struct Node a2 = { "Harriet", &a1 },
struct Node a3 = { "Claire", &a2 }
struct Node a4 = { "Tony", &a3 };
```



## Example

In the example above, the data has changed from integers to strings. Other than that, all else is the same. A Node structure consists of data followed by a pointer to another Node structure.

### Creating a List

Four nodes are declared, "a1" through "a4". Notice that "a1" is declared first and goes at the end of the chain. "a2" is declared next and points back at "a1". This is the only way to do this, since if we attempted to make "a1" point forwards to "a2" the compiler would complain because when "a1" is initialized, "a2" doesn't exist. An alternative would be to declare the structures as follows:

```
struct Node a1 = { "John", NULL };
struct Node a2 = { "Harriet", NULL };
struct Node a3 = { "Claire", NULL };
struct Node a4 = { "Tony", NULL };
```

and then "fill in the gaps" by writing:

```
a4.next_in_line = &a3; a3.next_in_line = &a2;
a2.next_in_line = &a1;
```

Which would give exactly the same picture as above. Of course it would be just as possible to write:

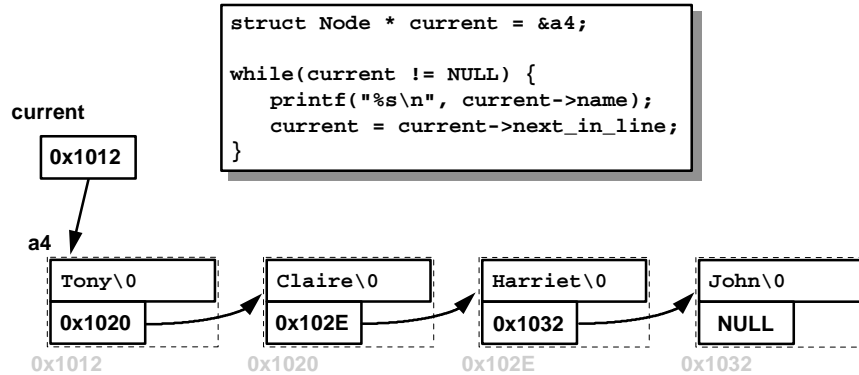
```
a1.next_in_line = &a2; a2.next_in_line = &a3;
a3.next_in_line = &a4;
```

and make the chain run the opposite way. Here "a1" would be the first node and "a4" the last node in the chain.



## Printing the List

§ The list may be printed with the following code:



## Printing the List

Above is an example of how to visit, and print the data contained in, each node in the list. A pointer is set to point at the first node in the list. This is done with:

```
struct Node *current = &a4;
```

creating a Node pointer called “current” and initializing it to point to the first node in the chain. Notice that if we had initialized this to point to, say, “a1”, we would be sunk since there is no way to get from “a1” back to “a2”.

The loop condition is: `while(current != NULL)`

let us imagine (even though it is not always true) that NULL is zero. We check the address contained in “current”, i.e. 0x1012 against zero. Clearly “current” is not zero, thus the loop is entered. The statement

```
printf("%s\n", current->name);
```

causes the “name” member of the structure at address 0x1012 to be printed, i.e. “Tony”. Then the statement

```
current = current->next_in_line;
```

is executed, causing the value of the “next\_in\_line” member, i.e. 0x1020 to be transferred into “current”. Now the pointer “current” points to the second structure instance “a3”. Once again the loop condition

```
while(current != NULL)
```

is evaluated. Now “current” is 0x1020 and is still not zero, hence the condition is still true and so the loop is entered once more.

---

## Printing the List (Continued)

The statement

```
printf("%s\n", current->name);
```

is executed, causing the “name” member of the structure at address 0x1020 to be accessed, i.e. “Claire”. Next, the statement

```
current = current->next_in_line;
```

is executed taking the value of the member “next\_in\_line”, i.e. 0x102E and transferring it into “current”. Now “current” points to the third structure instance, “a2”. Again the loop condition is evaluated:

```
while(current != NULL)
```

Since 0x102E is not zero the condition is again true and the loop body is entered. Now the statement

```
printf("%s\n", current->name);
```

prints “Harriet”, i.e. the value contained in the “name” field for the structure whose address is 0x102E. The statement

```
current = current->next_in_line;
```

causes the value in the “next\_in\_line” member, i.e. 0x1032 to be transferred into “current”. Now “current” points to the last of the structure instances “a1”. The loop condition:

```
while(current != NULL)
```

is evaluated, since 0x1032 does not contain zero, the condition is still true and the loop body is entered once more. The statement:

```
printf("%s\n", current->name);
```

prints “John” since this is the value in the “name” field of the structure whose address is 0x1032. Now the statement

```
current = current->next_in_line;
```

causes the value NULL to be transferred into current (since this is the value stored in the “next\_in\_line” member of the structure whose address is 0x1032). Now the “current” pointer is invalid. The loop condition

```
while(current != NULL)
```

is evaluated. Since “current” does contain NULL, the condition is no longer true and the loop terminates.

---

## Summary

- ❏ **Creating structure templates using struct**
- ❏ **Creating and initialising instances**
- ❏ **Accessing members**
- ❏ **Passing instances to functions by value and by reference**
- ❏ **A new operator: “->”**
- ❏ **Return by value**
- ❏ **Linked lists**

---

## Summary

---



---

---

## Structures Practical Exercises

---

---

Directory:        **STRUCT**

1. Open "**CARD1.C**" which declares and initializes two card structures. There are two functions for you to implement:

```
void print_card_by_value(struct Card which);
void print_card_by_ref(struct Card * p);
```

The first of these is passed a *copy* of the card to print out. The second is passed a *pointer* to the card. Both functions should print the same output.

2. In "**CARD2.C**" are the definitions of several cards. Implement the **is\_red** function which has the following prototype:

```
int is_red(struct Card * p);
```

This function should return true (i.e. 1) if the argument points to a red card (a heart or a diamond) and return false (i.e. 0) otherwise. You will need to copy your **print\_card\_by\_ref** function from part 1 and rename it **print\_card**.

3. Open the file "**CARD3.C**". Implement the function **may\_be\_placed** which has the following prototype:

```
int may_be_placed(struct Card * lower, struct Card * upper);
```

This function uses the rules of solitaire to return true if the card "upper" may be placed on the card "lower". The cards must be of different colors, the upper card (i.e. the one being placed) must have a value which is one less than the lower card (i.e. the one already there). You will need your **print\_card** and **is\_red** functions.

4. In "**LIST1.C**" Node structures are declared, like those in the chapter notes. Implement the function:

```
void print_list(struct Node *first_in_list);
```

which will print out all the integers in the list.

5. The file "**LIST2.C**" has an exact copy of the Nodes declared in "**LIST1.C**". Now there is a call to the function

```
void print_list_in_reverse(struct Node *first_in_list);
```

Using recursion, print the integers in reverse order. If you are unfamiliar with recursion, ask your instructor.

---

6. Linked lists enable new values to be inserted merely by altering a few pointers. "LIST3.C" creates the same list as in "LIST1.C" and "LIST2.C", but also declares three other nodes which should be inserted into the correct point in the list. Implement the function:

```
struct Node* insert(struct Node *first_in_list, struct Node *new_node);
```

which will insert each of the three nodes at the correct point in the list. Notice that one insertion occurs at the start, one in the middle and one at the end of the list. Remove the comments when you are ready to try these insertions. You will need your `print_list` function from "LIST1.C".

---





---

---

## Structures Solutions

---

---

1. In "CARD1.C" implement the functions:

```
void print_card_by_value(struct Card which);
void print_card_by_ref(struct Card * p);
```

*The print\_card\_by\_value function is straightforward, print\_card\_by\_ref more elaborate. The essential difference between the two is merely the difference between use of "." and "->". The shorter version (with one printf) is used throughout the following solutions for brevity.*

```
#include <stdio.h>

struct Card
{
 int index;
 char suit;
};

void print_card_by_value(struct Card which);
void print_card_by_ref(struct Card * p);

int main(void)
{
 struct Card king_of_spades = { 13, 's' };
 struct Card four_of_clubs = { 4, 'c' };

 print_card_by_value(king_of_spades);
 print_card_by_ref(&king_of_spades);

 print_card_by_value(four_of_clubs);
 print_card_by_ref(&four_of_clubs);

 return 0;
}

void print_card_by_value(struct Card which)
{
 printf("%i of %c\n", which.index, which.suit);
}
```

```
void print_card_by_ref(struct Card * p)
{
 switch(p->index) {
 case 14:
 case 1:
 printf("Ace");
 break;
 case 13:
 printf("King");
 break;
 case 12:
 printf("Queen");
 break;
 case 11:
 printf("Jack");
 break;
 default:
 printf("%i", p->index);
 break;
 }
 printf(" of ");
 switch(p->suit) {
 case 'c':
 printf("clubs\n");
 break;
 case 'd':
 printf("diamonds\n");
 break;
 case 's':
 printf("spades\n");
 break;
 case 'h':
 printf("hearts\n");
 break;
 }
}
```

---

2. In “CARD2.C” implement the `is_red` function which has the following prototype:

```
int is_red(struct Card * p);
```

*The value returned from `is_red` (i.e. one or zero) is already the value yielded by C’s “==” operator.*

```
#include <stdio.h>
```

---

```
#define ASIZE(A) sizeof(A)/sizeof(A[0])

struct Card
{
 int index;
 char suit;
};

int is_red(struct Card* p);
void print_card(struct Card * p);

int main(void)
{
 int i;
 struct Card hand[] = {
 { 13, 's' },
 { 4, 'c' },
 { 9, 'd' },
 { 12, 'h' },
 { 5, 'c' }
 };

 for(i = 0; i < ASIZE(hand); i++) {
 printf("the ");
 print_card(&hand[i]);

 if(is_red(&hand[i]))
 printf(" is red\n");
 else
 printf(" is not red\n");
 }

 return 0;
}

void print_card(struct Card * p)
{
 printf("%i of %c\n", p->index, p->suit);
}

int is_red(struct Card * p)
{
 return p->suit == 'h' || p->suit == 'd';
}
```

---

3. In "CARD3.C" implement the function `may_be_placed`

```
#include <stdio.h>

#define ASIZE(A) sizeof(A)/sizeof(A[0])

struct Card
{
 int index;
 char suit;
};

int is_red(struct Card* p);
void print_card(struct Card * p);
int may_be_placed(struct Card * lower, struct Card * upper);

int main(void)
{
 int i;
 struct Card lower_cards[] = {
 { 13, 's' },
 { 4, 'c' },
 { 9, 'd' },
 { 12, 'h' },
 { 5, 'c' }
 };
 struct Card upper_cards[] = {
 { 10, 'c' },
 { 3, 'd' },
 { 8, 'd' },
 { 11, 's' },
 { 4, 's' }
 };

 for(i = 0; i < ASIZE(lower_cards); i++) {

 printf("the ");
 print_card(&upper_cards[i]);

 if(may_be_placed(&lower_cards[i], &upper_cards[i]))
 printf(" may be placed on the ");
 else
 printf(" may NOT be placed on the ");

 print_card(&lower_cards[i]);
 printf("\n");
 }

 return 0;
}

void print_card(struct Card * p)
{
 printf("%i of %c\n", p->index, p->suit);
}
```

---

```

int may_be_placed(struct Card * lower, struct Card * upper)
{
 /* If both the same colour, that's bad */
 if(is_red(lower) == is_red(upper))
 return 0;

 /* Ace does not take part */
 if(lower->index == 14 || upper->index == 14)
 return 0;

 if(lower->index == upper->index + 1)
 return 1;

 return 0;
}

int is_red(struct Card * p)
{
 return p->suit == 'h' || p->suit == 'd';
}

```

---

4. In “LIST1.C” implement the function:

```
void print_list(struct Node *first_in_list);
```

*Rather than creating a local variable and assigning the value of “first\_in\_list”, this version of print\_list uses the parameter directly. Since call by value is always used, any parameter may be treated “destructively”. Note that now the parameter name used in the prototype does not correspond to that used in the function header. C doesn’t care about this and indeed this is good because the user sees “first\_in\_list” and knows the correct parameter to pass whereas the function sees “current” which is far more meaningful than changing the “first\_in\_list” pointer.*

```

#include <stdio.h>

struct Node {
 int data;
 struct Node* next_in_line;
};

void print_list(struct Node * first_in_list);

int main(void)
{
 struct Node n1 = { 100, NULL };
 struct Node n2 = { 80, NULL };
 struct Node n3 = { 40, NULL };
 struct Node n4 = { 10, NULL };

 n4.next_in_line = &n3;
 n3.next_in_line = &n2;
 n2.next_in_line = &n1;

 print_list(&n4);

 return 0;
}

```

---

```

void print_list(struct Node * current)
{
 while(current != NULL) {
 printf("%i\t", current->data);
 current = current->next_in_line;
 }
 printf("\n");
}

```

---

5. In "LIST2.C" implement the function

```
void print_list_in_reverse(struct Node *first_in_list);
```

*The first version of print\_list\_in\_reverse suffers from the problem of no trailing newline. Whereas this is not a problem with DOS (since COMMAND.COM always prints a few newlines just in case) it is an annoyance with other operating systems (like Unix).*

```
#include <stdio.h>
```

```

struct Node {
 int data;
 struct Node* next_in_line;
};

```

```
void print_list_in_reverse(struct Node * first_in_list);
```

```

int main(void)
{
 struct Node n1 = { 100, NULL };
 struct Node n2 = { 80, NULL };
 struct Node n3 = { 40, NULL };
 struct Node n4 = { 10, NULL };

 n4.next_in_line = &n3;
 n3.next_in_line = &n2;
 n2.next_in_line = &n1;

 print_list_in_reverse(&n4);

 return 0;
}

```

```

void print_list_in_reverse(struct Node * p)
{
 if(p == NULL)
 return;

 print_list_in_reverse(p->next_in_line);
 printf("%i\t", p->data);
}

```

---

---

*This second version copes with this newline problem using a static variable. Remember that all instances of the `print_list_in_reverse` function will share the same static.*

```
void print_list_in_reverse(struct Node * p)
{
 static int newline;

 if(p == NULL)
 return;

 ++newline;
 print_list_in_reverse(p->next_in_line);
 --newline;
 printf("%i\t", p->data);
 if(newline == 0)
 printf("\n");
}
```

---

6. In “**LIST3.C**” implement the function:

```
struct Node* insert(struct Node *first_in_list, struct Node *new_node);
```

*The insert function keeps the pointer “lag” one step behind the insertion point. This makes it very easy to refer to the node which must be rewired (especially as there is no way via traversing the list to return back to it). Since it is initialised to NULL, it is possible to detect when the body of the “find the insertion point” has not been entered. In this case the new node becomes the new head of the list.*

```
#include <stdio.h>
```

```
struct Node {
 int data;
 struct Node* next_in_line;
};
```

```
void print_list(struct Node * first_in_list);
struct Node*insert(struct Node *first_in_list, struct Node *new_node);
```

```
int main(void)
{
 struct Node n1 = { 100, NULL };
 struct Node n2 = { 80, NULL };
 struct Node n3 = { 40, NULL };
 struct Node n4 = { 10, NULL };
 struct Node * head;

 struct Node new_head = { 1, NULL };
 struct Node new_tail = { 200, NULL };
 struct Node new_middle = { 60, NULL };

 n4.next_in_line = &n3;
 n3.next_in_line = &n2;
 n2.next_in_line = &n1;
 head = &n4;
```

---



```
printf("Before insertions, list is ");
print_list(head);

printf("inserting %i into middle of list\n", new_middle.data);
head = insert(head, &new_middle);
print_list(head);

printf("inserting %i at end of list\n", new_tail.data);
head = insert(head, &new_tail);
print_list(head);

printf("inserting %i in front of list\n", new_head.data);
head = insert(head, &new_head);
print_list(head);

return 0;
}

void print_list(struct Node * current)
{
 while(current != NULL) {
 printf("%i\t", current->data);
 current = current->next_in_line;
 }
 printf("\n");
}

struct Node*insert(struct Node *p, struct Node *new_node)
{
 struct Node* start = p;
 struct Node* lag = NULL;

 while(p != NULL && p->data < new_node->data) {
 lag = p;
 p = p->next_in_line;
 }

 if(lag == NULL) { /* insert before list */
 new_node->next_in_line = p;
 return new_node;
 }

 lag->next_in_line = new_node;
 new_node->next_in_line = p;

 return start;
}
```

---



---

---

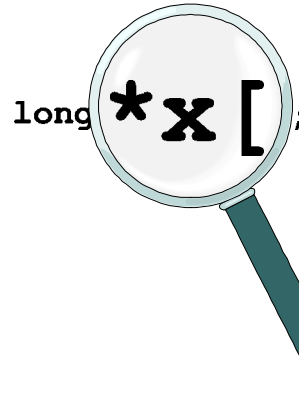
## Reading C declarations

---

---

## Reading C Declarations

- 🔗 Introduction
- 🔗 **SOAC**
- 🔗 Examples
- 🔗 typedef
- 🔗 Examples revisited



---

## Reading C Declarations

Reading declarations in C is almost impossible unless you know the rules.  
Fortunately the rules are very simple indeed and are covered in this chapter.

## Introduction

- Up until now we have seen straightforward declarations:

```
long sum;
int* p;
```

- Plus a few trickier ones:

```
void member_display(const struct Library_member *p);
```

- However, they can become *much* worse:

```
int *p[15];
float (*pfa)[23];
long (*f)(char, int);
double *(*(*n)(void))[5];
```

---

## Introduction

Thus far in the course we have seen some straightforward declarations. We have declared **ints**, **floats**, arrays of **char**, structures containing **doubles**, pointers to those structures. However, C has the capability to declare some really mind boggling things, as you can see above. Trying to understand these declarations is almost entirely hopeless until you understand the rules the compiler uses.

---

## SOAC

**Find the variable being declared**

**Spiral Outwards Anti Clockwise**

**On meeting:**

**\***

**[ ]**

**( )**

**say:**

**pointer to**

**array of**

**function taking .... and returning**



**Remember to read “struct S”, “union U” or  
“enum E” all at once**

**Remember to read adjacent collections of [ ] [ ] all  
at once**

## SOAC

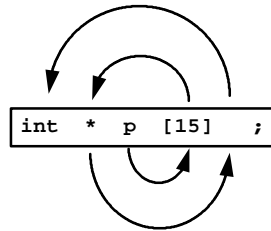
Fortunately, although mind boggling things may be declared, the rules the compiler uses are far from mind boggling. They are very straightforward and may be remembered as SOAC (most easily remembered if pronounced as “soak”). As mentioned above this stands for Spiral Outwards Anti Clockwise. Start spiraling from the variable name and if while spiraling you meet any of the characters “\*”, “[ ]” etc. mentioned above, say the corresponding thing.

The only other things to remember is that structures, enums (which we haven’t covered yet) and unions (which we also haven’t covered yet) followed by their tags should be read in one go.

Also array of array declarations (effectively multi-dimensional arrays) should be read in one go.

## Example 1.

§ What is “`int * p[15]`” ?



§ **p is an array of 15 pointers to integers**

## Example 1.

The declaration “`int * p[15]`” could declare “p” as:

1. an array of 15 pointers to integers, or
2. a pointer to an array of 15 integers

so which is it?

Always start reading at the name of the variable being declared, here “p”. Spiral outwards anti clockwise (in other words right from here). We immediately find:

**[15]**

which causes us to say “array of 15”. Carrying on spiraling again, the next thing we meet is the “**\***” which causes us to say “pointer to”, or in this case where we’re dealing with 15 of them, perhaps “pointers to”. Spiraling again, we sail between the “**]**” and the “**;**” and meet

**int**

causing us to say “integer”.

Putting all this together gives:

1. p is an
2. array of 15
3. pointers to
4. integer

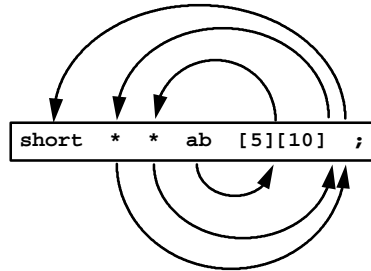
The variable “p” is therefore an array containing 15 elements, each of which is a pointer to an integer.





## Example 3.

§ What is “short \*\*ab[5][10]” ?



§ **ab is an array of 5 arrays of 10 arrays of pointers to pointers to short int**

## Example 3.

Although we're throwing in the kitchen sink here, it doesn't really make things that much more difficult.

Find the variable being declared “ab” and spiral. We find:

**[5][10]**

which we read in one go according to our special rule giving “array of 5 arrays of 10”. Spiraling again we meet the “\*” closest to “ab” and say “pointer to”. Spiraling between the “]” and the semicolon we meet the next “\*” causing us to say “pointer to” again. Spiraling once again between the “]” and the semicolon we meet

**short**

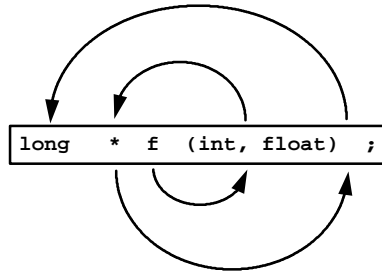
Putting this together:

1. ab is an
2. array of 5 arrays of 10
3. pointers to
4. pointers to
5. short int

Thus “ab” is a collection of 50 pointers, each pointing to a slot in memory containing an address. This address is the address of a short integer somewhere else in memory.

## Example 4.

§ What is “long \* f(int, float)” ?



§ f is a function taking an int and a float returning a pointer to a long int

## Example 4.

Here we see the “function returning” parentheses. Once again starting at “f” we spiral and find

(int, float)

and say “function (taking an int and a float as parameters) returning”, next spiral to find “\*” causing us to say “pointer to”, then spiraling between the closing parenthesis and the semicolon to finally land on “long”.

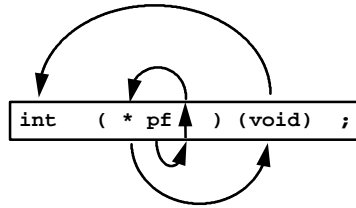
Putting this together gives:

1. f is a
2. function (taking an int and a float as parameters) returning a
3. pointer to a
4. long

Thus we find this is merely a function prototype for the function “f”.

## Example 5.

§ What is “`int (*pf)(void)`” ?



§ **pf is a pointer to a function taking no parameters and returning an int**

## Example 5.

This example shows the effect of placing parentheses around “`*pf`” when dealing with functions. The variable being declared is “`pf`”. We spiral *inside* the closing parenthesis and meet the “`*`” causing us to say “pointer to”. From there we spiral out to find:

`(void)`

which causes us to say “function (taking no parameters) and returning”. From there we spiral and find:

`int`

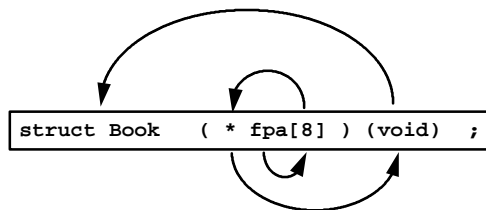
Putting this together gives:

1. `pf` is a
2. pointer to a
3. function (taking no parameters) and returning an
4. integer

Thus “`pf`” is not a function prototype, but the declaration of a single individual pointer. At the end of this pointer is a function. The course has examined the concept of pointers and seen pointers initialized to point at the stack and at the data segment. It is also possible to point pointers into the heap (which will be discussed later). “`pf`” is an example of a pointer which can point into the code segment. This is the area of the program which contains the various functions in the program, `main`, `printf`, `scanf` etc.

## Example 6.

§ What is “`struct Book (*fpa[8])(void)`” ?



§ `fpa` is an array of 8 pointers to functions, taking no parameters, returning `Book` structures

## Example 6.

Here once again, the kitchen sink has been thrown into this declaration and without our rules it would be almost impossible to understand.

Starting with “`fpa`” and spiraling we find:

`[8]`

causing us to say “array of 8”. Spiraling onwards we find “`*`” causing us to say “pointer to”. Next we encounter:

`(void)`

causing us to say “function (taking no parameters) returning”. Now we meet

`struct Book`

which, according to our special case, we read in one go.

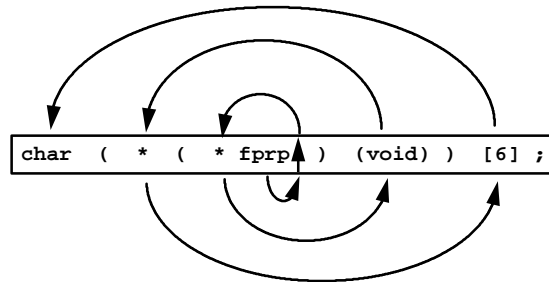
Putting this together gives:

1. `fpa` is an
2. array of 8
3. pointers to
4. functions (taking no parameters) returning
5. `Book` structures

Thus `fpa` is an array of 8 slots. Each slot contains a pointer. Each pointer points to a function. Each function returns one `Book` structure by value.

## Example 7.

§ What is “char (\*( \*fprp)(void))[6]” ?



§ fprp is a pointer to a function taking no parameters returning a pointer to an array of 6 char

## Example 7.

The declaration above is hideous and the temptation arises to start screaming. However, “fprp” is being declared. Spiraling inside the parenthesis leads us to “\*” and we say “pointer to”. Spiraling further leads us to:

(void)

causing us to say “function (taking no parameters) returning”. Spiraling beyond this leads us to the second “\*” causing us to say “pointer to”. Now we spiral to

[6]

which is an “array of 6”, and finally we alight on

char

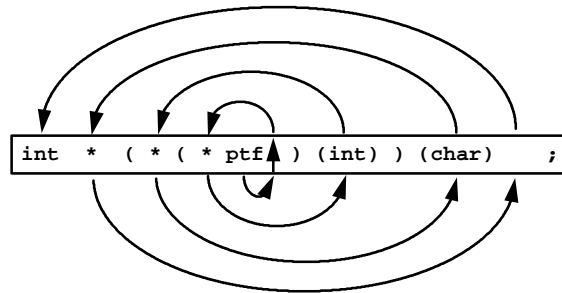
Putting this together gives:

1. fprp is a
2. pointer to a
3. function (taking no parameters) returning a
4. pointer to an
5. array of 6
6. char

Thus only one pointer is being declared here. The remainder of the declaration merely serves to tell us what type is at the end of the pointer (once it has been initialized). It is, in fact, a code pointer and points to a function. The function takes no parameters but returns a pointer. The returned pointer points to an array of 6 characters.

## Example 8.

§ What is “`int * (*(*ptf)(int))(char)`” ?



§ **ptf is a pointer to a function, taking an integer, returning a pointer to a function, taking a char, returning a pointer to an int**

## Example 8.

Although hideous, this declaration is only one degree worse than the last. Finding “ptf” and spiraling inside the parenthesis we find “\*” causing us to say “pointer to”. Now we spiral and find

**(int)**

meaning “function taking an integer and returning”. Spiraling further we find another “\*” meaning “pointer to”. Spiraling further we find

**(char)**

meaning “function taking a character and returning”. Again another “\*” meaning “pointer to”, then finally spiraling just in front of the semicolon to meet

**int**

Putting this together:

1. ptf is a
2. pointer to a
3. function taking an integer and returning a
4. pointer to a
5. function taking a character and returning an
6. integer

Thus “ptf” declares a single pointer. Again the rest of the declaration serves only to tell us what is at the end of the pointer once initialized. At the end of the pointer lives a function. This function expects an integer as a parameter. The function returns a pointer. The returned pointer points to another function which expects a character as a parameter. This function (the one taking the character) returns a single integer value.

## typedef

- § It doesn't have to be this difficult!
- § The declaration can be broken into simpler steps by using `typedef`
- § To tackle `typedef`, pretend it isn't there and read the declaration as for a variable
- § When finished remember that a *type* has been declared, not a variable

---

## typedef

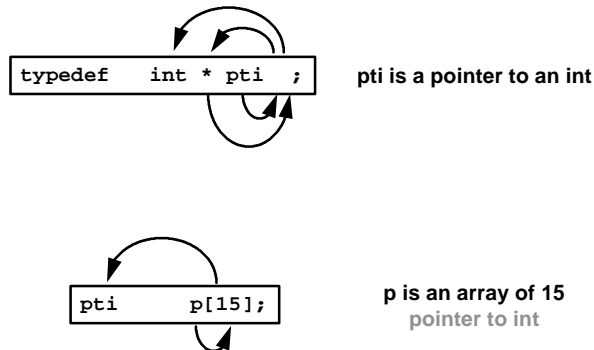
When we read a declaration, we break it down into a number of simpler steps. It is possible to give each one of these simpler steps to the compiler using the `typedef` keyword.

To understand `typedef`, ignore it. Pretend it isn't there and that a variable is being declared. Read the declaration just as for any other variable. But remember, once the declaration has been fully read the compiler has declared a *type* rather than a variable. This becomes a completely new compiler type and may be used just as validly wherever `int`, `float`, `double` etc. were used.

---

## Example 1 Revisited

⌘ Simplify “`int * p[15]`”



## Example 1 Revisited

We want to simplify the declaration “`int * p[15]`” which, you will remember, declares “p” as an array of 15 pointers to integer. Starting from the end of this, create a new type “pointer to int”. If we wrote:

```
int * pti;
```

we would declare a variable “pti” of type “pointer to integer”. By placing typedef before this, as in:

```
typedef int * pti;
```

we create a new *type* called “pti”. Wherever “pti” is used in a declaration, the compiler will understand “pointer to integer”, just as wherever `int` is used in a declaration the compiler understands “integer”. This, as a quick aside, gives a possible solution to the dilemma of where to place the “\*” in a declaration. You will remember the problems and merits of:

```
int* p;
int *p;
```

vs.

and especially the problem with `int* p, q;`

where “p” has type “pointer to `int`”, but “q” has type `int`. This typedef can solve the latter problem as in:

```
pti p, q;
```

where the type of both “p” and “q” is “pointer to `int`” without the problems mentioned above.

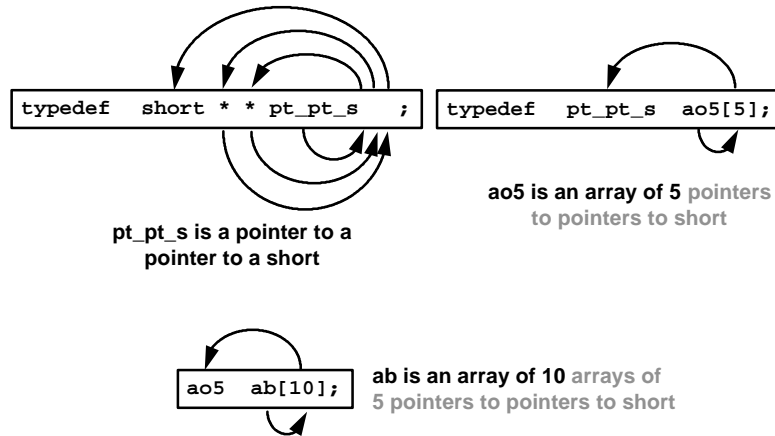
Having created this new type, declaring an array of 15 pointers to integers merely becomes:

```
pti p[15];
```



## Example 3 Revisited

2 Simplify “short \*\*ab[5][10]”



## Example 3 Revisited

We wish to simplify the declaration “**short \*\*ab[5][10]**” which as we already know declares “ab” as an array of 5 arrays of 10 pointers to pointers to **short int**.

Start from the back with the “pointers to pointers to **short int**”:

```
typedef short * * pt_pt_s;
```

creates a new type called “pt\_pt\_s” meaning “pointer to pointer to **short**”.

In fact we could stop here and define “ab” as:

```
pt_pt_s ab[5][10];
```

which is slightly more obvious than it was. However, again peeling away from the back, here is a definition for an array of 5 pointers to pointers to **short**:

```
typedef pt_pt_s ao5[5];
```

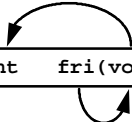
(Remember that if the typedef were covered, we would be creating a variable called “ao5” which would be an array of 5 pointers to pointers to **short**). Once this has been done, creating “ab” is easily done. We just need 10 of the ao5’s as follows:

```
ao5 ab[10];
```

## Example 5 Revised

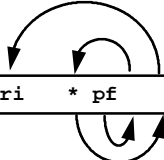
🔗 Simplify “`int (*pf)(void)`”

```
typedef int fri(void);
```



**fri** is a function, taking no parameters, returning an int

```
fri * pf ;
```



**pf** is a pointer to a function, taking no parameters, returning an int

## Example 5 Revised

Now we wish to simplify the declaration of “pf” in “`int (*pf)(void)`” which as we already know declares “pf” to be a pointer to a function taking no parameters and returning an integer.

Tackling this last part first, a new type is created, “fri” which is a function, taking no parameters, returning an integer

```
typedef int fri(void);
```

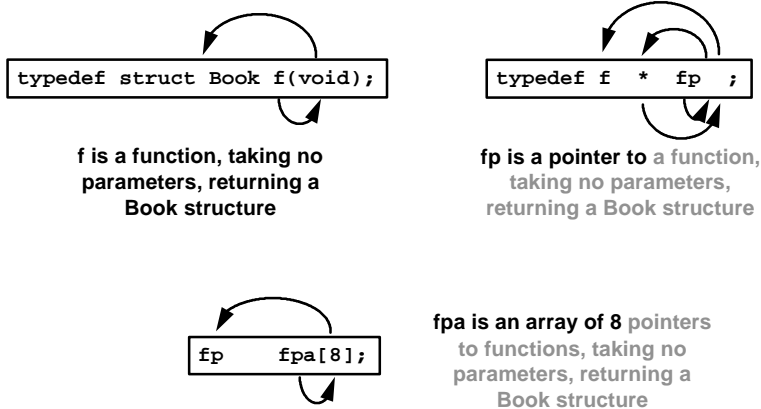
does this quite nicely. Remember that if `typedef` were covered we would be writing a function prototype for “fri”.

From here “pf” is created quite simply by declaring a pointer to an “fri” as:

```
fri * pf;
```

## Example 6 Revised

🔗 Simplify “struct Book (\*fpa[8])(void)”



## Example 6 Revised

We wish to simplify the declaration “struct Book (\*fpa[8])(void)” which as we already know declares “fpa” as an array of 8 pointers to functions, taking no parameters, returning Book structures.

We start by creating a typedef for a single function, taking no parameters, returning a Book structure. Such a function would be:

```
struct Book f(void);
```

Adding the typedef ensures that instead of “f” being the function it instead becomes the new type:

```
typedef struct Book f(void);
```

Now all we have to do is create a pointer to one of these:

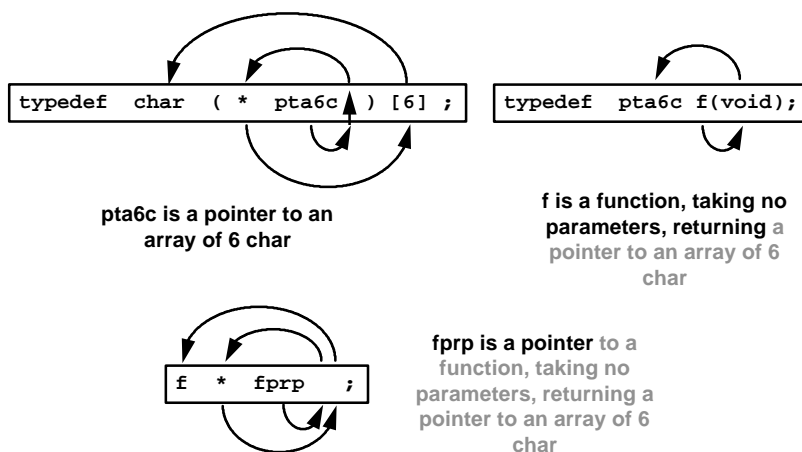
```
typedef f *fp;
```

Now all we need is an array of 8 of these:

```
fp fpa[8];
```

## Example 7 Revised

⌘ Simplify “char ((\*fprp)(void))[6]”



## Example 7 Revised

We wish to simplify the declaration “**char** (**\***(**\***fprp)(void))[6]” which, as we already know declares “fprp” as a pointer to a function, taking no parameters, returning a pointer to an array of 6 characters.

The first thing to tackle, once again, is the last part of this declaration, the pointer to an array of 6 characters. This can be done in one step as above, or in two steps as:

```
typedef char array_of_6_char[6];
typedef array_of_6_char *pta6c;
```

Now for a function, taking no parameters, that returns one of these:

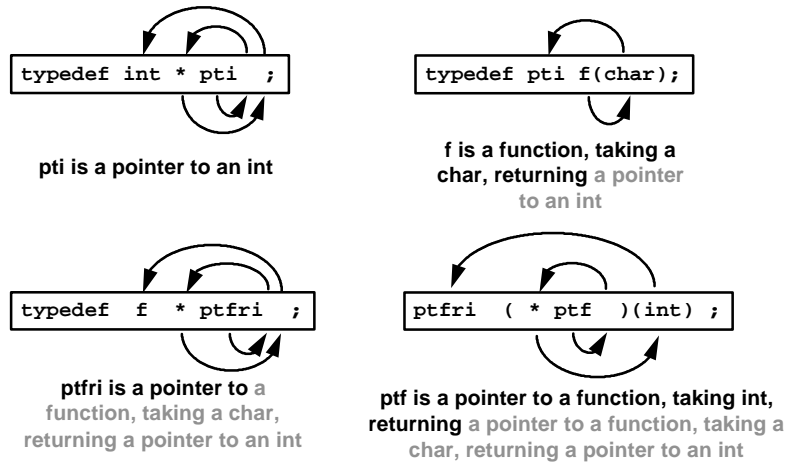
```
typedef pta6c f(void);
```

All that is left is to create “fprp” as a pointer to one of these:

```
f *fprp;
```

## Example 8 Revised

⌘ Simplify “`int * ((*ptf)(int))(char)`”



## Example 8 Revised

Finally, we wish to simplify the declaration “`int * ((*ptf)(int))(char)`” which as we already know declares “ptf” as a pointer to a function, taking an int, returning a pointer to a function, taking a char, returning a pointer to an int.

Starting at the end with the “pointer to int” part,

```
typedef int *pti;
```

creates the type “pti” which is a “pointer to an int”. Again picking away at the end, we need a function taking a char returning one of these, thus:

```
typedef pti f(char);
```

Now, a pointer to one of these:

```
typedef f *ptfri;
```

Next a function, taking an int and returning a pointer to one of these (there wasn’t room for this step above):

```
typedef ptfri func_returning_ptfri(int);
```

Now, a pointer to one of these:

```
typedef func_returning_ptfri *ptf_r_ptfri;
```

So that finally the variable “ptf” can be declared:

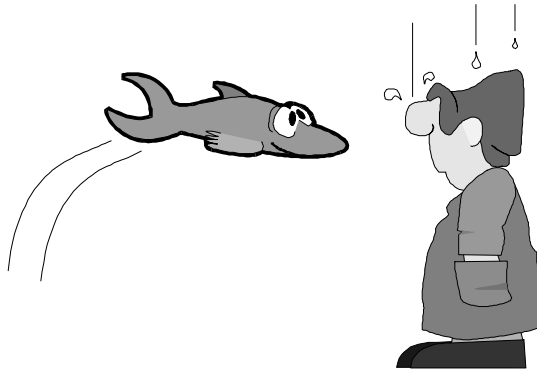
```
ptf_r_ptfri ptf;
```

Alternatively we could have used the previous typedef as in:

```
func_returning_ptfri *ptf;
```

## Summary

- ⌘ Don't Panic!
- ⌘ SOAC - Spiral Outwards Anti Clockwise
- ⌘ To simplify, use *typedef(s)*



---

## Summary

---

---

---

## Reading C Declarations Practical Exercises

---

---

1. What types do the following variables have?

```

int *a;
int b[10];
int *c[10];
int (*d)[10];
int *(*e)[10];
int (**f)[10];
int *(*g)[10];

char h(void);
char *i(void);
char (*j)(void);
char *(*k)(void);
char **l(void);
char (**m)(void);
char *(*n)(void);

float (*o(void))[6];
float *(*p(void))[6];
float (**q(void))[6];
float *(*r(void))[6];

short (*s(void))(int);
short *(*t(void))(int);
short (**u(void))(int);
short *(*v(void))(int);

long (*(*x(void))(int))[6];
long *(*(*y(void))(int))[6];
long *(*(*z(void))[7])(void);

```

2. Using `typedef`, simplify the declaration of:

```

e in 3 steps
g in 4 steps
l in 3 steps
n in 3 steps
p in 4 steps
u in 4 steps
x in 5 steps
z in 7 steps

```



---

---

## Reading C declarations Solutions

---

---

1. What types do the following variables have?

- `int *a;`

'a' is a pointer to int.

- `int b[10];`

'b' is an array of 10 int.

- `int *c[10];`

'c' is an array of 10 pointers to int.

- `int (*d)[10];`

'd' is a pointer to an array of 10 int.

- `int *(*e)[10];`

'e' is a pointer to an array of 10 pointers to int.

- `int (**f)[10];`

'f' is a pointer to a pointer to an array of 10 int.

- `int *(*g)[10];`

'g' is a pointer to a pointer to an array of 10 pointer to int.

- `char h(void);`

'h' is a function, taking no parameters, returning a char.

- `char *i(void);`

'i' is a function, taking no parameters, returning a pointer to char.

- `char (*j)(void);`

'j' is a pointer to a function, taking no parameters, returning a char.

- `char *(*k)(void);`

'k' is a pointer to a function, taking no parameters, returning a pointer to a char.

- `char **l(void);`

'l' is a function, taking no parameters, returning a pointer to a pointer to a char.

- `char (**m)(void);`

'm' is a pointer to a pointer to a function, taking no parameters, returning a char.

---

- `char        **n(void);`

'n' is a pointer to a pointer to a function, taking no parameters, returning a pointer to a char.

- `float        (*o(void))[6];`

'o' is a function, taking no parameters, returning a pointer to an array of 6 float.

- `float        *(*p(void))[6];`

'p' is a function, taking no parameters, returning a pointer to an array of 6 pointers to float.

- `float        **q(void)[6];`

'q' is a function, taking no parameters, returning a pointer to a pointer to an array of 6 float.

- `float        **r(void)[6];`

'r' is a function, taking no parameters, returning a pointer to a pointer to an array of 6 pointer to float.

- `short        (*s(void))(int);`

's' is a function, taking no parameters, returning a pointer to a function, taking an int, returning a short.

- `short        *(*t(void))(int);`

't' is a function, taking no parameters, returning a pointer to a function, taking an int, returning a pointer to a short.

- `short        **u(void)(int);`

'u' is a function, taking no parameters, returning a pointer to a pointer to a function, taking an int, returning a short.

- `short        **v(void)(int);`

'v' is a function, taking no parameters, returning a pointer to a pointer to a function, taking an int, returning a pointer to a short.

- `long         *(*x(void))(int)[6];`

'x' is a function, taking no parameters, returning a pointer to a function, taking an int, returning a pointer to an array of 6 long.

- `long         *(*y(void))(int)[6];`

'y' is a function, taking no parameters, returning a pointer to a function, taking an int, returning a pointer to an array of 6 pointers to long.

- `long         *(*z(void))[7](void);`

'z' is a pointer to a function, taking no parameters, returning a pointer to an array of 7 pointers to functions, taking no parameters, returning pointers to long.

---

2. Using `typedef`, simplify the declaration of:

- e in 3 steps. 'e' is a pointer to an array of 10 pointers to int.

i) typedef for pointer to int:

```
typedef int * int_ptr;
```

ii) typedef for 10 of "i":

```
typedef int_ptr arr_int_ptr[10];
```

iii) typedef for a pointer to "ii":

```
typedef arr_int_ptr * ptr_arr_int_ptr;
```

- g in 4 steps. 'g' is a pointer to a pointer to an array of 10 pointer to int.

Continuing from (iii) above:

iv) typedef for a pointer to "iii":

```
typedef ptr_arr_int_ptr * ptr_ptr_arr_int_ptr;
```

- l in 3 steps. 'l' is a function, taking no parameters, returning a pointer to a pointer to a char.

i) typedef for a pointer to a char:

```
typedef char * ptr_char;
```

ii) typedef for a pointer to "i":

```
typedef ptr_char * ptr_ptr_char;
```

iii) typedef of a function returning "ii":

```
typedef ptr_ptr_char func_returning_ptr_ptr_char(void);
```

- n in 3 steps. 'n' is a pointer to a pointer to a function, taking no parameters, returning a pointer to a char.

i) typedef for a pointer to a char:

```
typedef char * ptr_char;
```

ii) typedef for a function, taking no parameters, returning "i":

```
typedef ptr_char func_returning_ptr_char(void);
```

iii) typedef of a pointer to "ii":

```
typedef ptr_to_func * ptr_char func_returning_ptr_char;
```

- p in 4 steps. 'p' is a function, taking no parameters, returning a pointer to an array of 6 pointers to float.

i) typedef for a pointer to a float:

```
typedef float * ptr_float;
```

ii) typedef for an array of 6 "i"s:

```
typedef ptr_float arr_ptr_float[6];
```

---

iii) typedef of a pointer to "ii":

```
typedef arr_ptr_flt * ptr_to_arr;
```

iv) typedef of a function returning "iii":

```
typedef ptr_to_arr func_returning_ptr_to_arr(void);
```

- u in 4 steps. 'u' is a function, taking no parameters, returning a pointer to a pointer to a function, taking an int, returning a short.

i) typedef for the function taking an int and returning a short:

```
typedef short f(int);
```

ii) typedef for a pointer to "i":

```
typedef f * ptr_func;
```

iii) typedef of a pointer to "ii":

```
typedef ptr_func * ptr_ptr_func;
```

iv) typedef of a function returning "iii":

```
typedef ptr_ptr_func func_returning_ptr_ptr_func(void);
```

- x in 5 steps. 'x' is a function, taking no parameters, returning a pointer to a function, taking an int, returning a pointer to an array of 6 long.

i) typedef for an array of 6 long:

```
typedef long arr_long[6];
```

ii) typedef for a pointer to "i":

```
typedef arr_long * ptr_arr_long;
```

iii) typedef of a function taking an int and returning a "ii":

```
typedef ptr_arr_long f(int);
```

iv) typedef for a pointer to "iii":

```
typedef f * ptr_to_func;
```

v) typedef for a function, taking no parameters returning "iv":

```
typedef ptr_to_func func(void);
```

- z in 7 steps. 'z' is a pointer to a function, taking no parameters, returning a pointer to an array of 7 pointers to functions, taking no parameters, returning pointers to long.

i) typedef for a pointer to a long:

```
typedef long * ptl;
```

ii) typedef for a function, taking no parameters, returning "i":

```
typedef ptl f(void);
```

iii) typedef of a pointer to "ii":

```
typedef f * ptr_func;
```

---

iv) typedef for an array of 7 "iii":

```
typedef ptr_func arr_ptr_func[7];
```

v) typedef for a pointer to a "iv":

```
typedef arr_ptr_func * ptr_arr_ptr_func;
```

vi) typedef for a function, taking no parameters, returning "iv":

```
typedef ptr_arr_ptr_func frp(void);
```

vii) typedef for a pointer to a "vi":

```
typedef frp * ptr_frp;
```

---

---

---

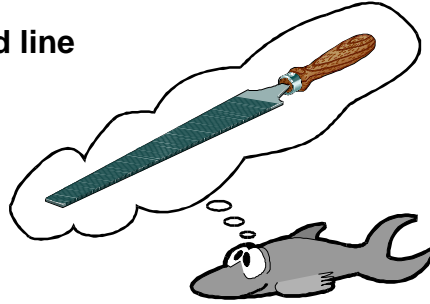
## Handling Files in C

---

---

## Handling Files in C

- § **Streams**
- § **`stdin`, `stdout`, `stderr`**
- § **Opening files**
- § **When things go wrong - `perror`**
- § **Copying files**
- § **Accessing the command line**
- § **Dealing with binary files**



---

## Handling Files in C

This chapter discusses how the Standard Library makes files accessible from the C language.

---



## Introduction

- ⌘ **File handling is not built into the C language itself**
- ⌘ **It is provided by The Standard Library (via a set of routines invariably beginning with “f”)**
- ⌘ **Covered by The Standard, the routines will always be there and work the same way, regardless of hardware/operating system**
- ⌘ **Files are presented as a sequence of characters**
- ⌘ **It is easy to move forwards reading/writing characters, it is less easy (though far from impossible) to go backwards**

---

## Introduction

### **The Standard Library**

Some languages have special keywords for dealing with files. C doesn't, instead it uses routines in the Standard Library which, because they are covered by The Standard will always work the same way despite the environment they are used in. Thus a Cray running Unix or a PC running CP/M (if there are any), the mechanism for opening a file is exactly the same.

Opening a file is rather like being presented with a large array of characters, except whereas an array provides random access to its elements a file provides sequential access to its characters. It is possible to achieve random access, but the routines are most easily driven forwards through the file character by character.

---

## Streams

- § Before a file can be read or written, a data structure known as a *stream* must be associated with it
- § A stream is usually a pointer to a structure (although it isn't necessary to know this)
- § There are three streams opened by every C program, `stdin`, `stdout` and `stderr`
- § `stdin` (standard input) is connected to the keyboard and may be read from
- § `stdout` (standard output) and `stderr` (standard error) are connected to the screen and may be written to

---

## Streams

The procedure by which files are manipulated in C is that a *stream* must be associated with a file to be read or written. A stream is a “black box” (although not in the aircraft sense) in that you don't really need to know what is going on in a stream. In fact it is best not to have to know, since there can be some headache inducing stuff happening in there.

As far as we are concerned the stream is transparent, we don't know what is it and we don't care. This is a similar idea to the “handle” concept popularized with Microsoft Windows programming. We don't know what a handle is, we just get them back from functions and pass them around to other functions that are interested in them. Same idea with a stream.

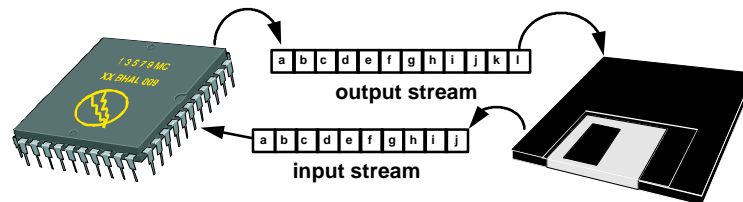
`stdin`, `stdout`  
and `stderr`

Whenever a C program runs (it doesn't matter what it does) it has 3 streams associated with it. These are:

1. the standard input, or `stdin`, connected to the keyboard. When characters are read from `stdin` the program will wait for the user to type something. `scanf`, for instance, uses `stdin`.
  2. the standard output, or `stdout`, connected to the screen. When characters are written to `stdout` characters appear on the screen. `printf`, for instance, uses `stdout`.
  3. the standard error, or `stderr`, also connected to the screen. Characters written to `stderr` will also appear on the screen. The `perror` function uses `stderr`.
-

## What is a Stream?

- Although implementations vary, a stream creates a buffer between the program running in memory and the file on the disk
- This reduces the program's need to access slow hardware devices
- Characters are *silently* read a block at a time into the buffer, or written a block at a time to the file



---

## What is a Stream?

It is all very well saying a stream must be associated with each file to be manipulated, but what *is* a stream and what does it *do*? The Standard does not say how a stream should be implemented, this is left to the compiler writers.

### Fast Programs Deal with Slow Hardware

Streams were invented in the very early days of C when devices were slow (much slower than they are today). Programs executing in memory run much faster than hardware devices can provide the information they need. It was found that when a program read characters individually from a disk, the program would have to wait excessively for the correct part of the disk to spin around. The character would be grabbed and processed, then the program would have to wait again for the disk.

### Caches and Streams

In the intervening years manufacturers have invented caches (large buffers) so the disk never reads a single character. Thus when the program requests the next character it is provided immediately from the buffer. Complex algorithms are used to determine which characters should be buffered and which should be discarded.

Streams do this buffering in software. Thus if the device you are using does not support caching, it doesn't matter because the stream will do it for you. If the device does cache requests, there is a minor duplication of effort.

---

## Why *stdout* and *stderr*?

§ There are two output streams because of *redirection*, supported by Unix, DOS, OS/2 etc.

```
#include <stdio.h>
int main(void)
{
 printf("written to stdout\n");
 fprintf(stderr, "written to stderr\n");
 return 0;
}
```

output written to  
stderr first  
because it is  
unbuffered

```
C:> outprog
written to stderr
written to stdout
C:> outprog > file.txt
written to stderr
C:> type file.txt
written to stdout
```

## Why *stdout* and *stderr*?

It seems strange to have two separate streams, *stdout* and *stderr* both going to the screen. After all, there is only *one* screen and it seems odd that a minimalist language like C would specifically attempt to cope with us having two monitors on our desks.

The real reason C has two streams going to the same place goes to the heart of Unix. Remember that C and Unix grew up together. Unix invented the idea of file redirection and of the pipe. In fact both ideas proved so popular that were adopted into other operating systems, e.g. MS-DOS, Windows 95, NT and OS/2 to name but a few.

The idea is that: *prog*

would run a program “normally” with its output going to the screen in front of us, but:

*prog > myfile.txt*

would run the program and take its screen output and write it to the file “myfile.txt” which is created in whatever directory the user is running in. Alternatively:

*prog | print*

would take the screen output and run it through the program called “print” (which I’m guessing would cause it to appear on a handy printer).

---

## Why `stdout` *and* `stderr`? (Continued)

These ideas have become fundamental to Unix, but in the early days it was discovered there was a problem. If the program “prog” needed to output any error messages these would either be mixed into the file, or printed on the line printer. What was needed was a way to write messages to the user that would be independent of the redirection currently in force. This was done by creating *two* separate streams, one for output, `stdout`, the other for errors, `stderr`. Although the standard output of the programs is redirected above, the standard error remains attached to the screen.

`stderr` guarantees the program a “direct connection” to the user despite any redirection currently in force.

---

## stdin is Line Buffered

§ Characters typed at the keyboard are buffered until Enter/Return is pressed

```
#include <stdio.h>

int main(void)
{
 int ch;

 while((ch = getchar()) != EOF)
 printf("read '%c'\n", ch);

 printf("EOF\n");

 return 0;
}
```

declared as an int, even though  
we are dealing with characters

```
C:> inprog
abc
read 'a'
read 'b'
read 'c'
read '
'
d
read 'd'
read '
'
^Z
EOF
C:>
```

## stdin is Line Buffered

Above is a program that uses the `getchar` routine. The important thing to notice is that because `getchar` uses the `stdin` stream, and `stdin` is line buffered, the characters "abc" which are typed are not processed until the enter key is pressed. Then they (and the enter key) are processed in one go as the loop executes four times.

By this time `getchar` has run out of characters and it must go back to the keyboard and wait for more. The second time around only "d" is typed, again followed by the enter key. These two characters, "d" and enter are processed in one go as the loop executes twice.

### Signaling End of File

Under MS-DOS the Control Z character is used to indicate end of file. When this is typed (again followed by enter) the `getchar` routine returns `EOF` and the loop terminates.

### int not char

It must seem curious that the variable "ch" is declared as type `int` and not `char` since we are dealing with characters, after all. The reason for this is that neither K&R C nor Standard C says whether `char` is signed or unsigned. This seems rather irrelevant until it is revealed that the value of the `EOF` define is -1. Now, if a compiler chose to implement `char` as an unsigned quantity, when `getchar` returned -1 to indicate end of file, it would cause 255 to be stored (since an unsigned variable cannot represent a negative value). When the 255 were compared with the -1 value of `EOF`, the comparison would fail. Thus the poor user would repeatedly type ^Z (or whatever your local flavour of end of file is) with no effect.

Using the type `int` guarantees that signed values may be represented properly.

## Opening Files

§ Files are opened and streams created with the **fopen** function

```
FILE* fopen(const char* name, const char* mode);
```

```
#include <stdio.h>

int main(void)
{
 FILE* in;
 FILE* out;
 FILE* append;

 in = fopen("autoexec.bat", "r");
 out = fopen("autoexec.bak", "w");
 append = fopen("config.sys", "a");
}
```

streams, you'll  
need one for each  
file you want  
open

---

## Opening Files

Before a file may be manipulated, a stream must be associated with it. This association between stream and file is made with the `fopen` routine.

All that is needed is to plug in the file name, the access mode (read, write, append) and the stream comes back. This is similar in concept to placing coins in a slot machine, pressing buttons and obtaining a chocolate bar. One kind of thing goes in (the coins, the file name) and another kind of thing comes back out (the chocolate bar, the stream).

### The Stream Type

A stream is actually declared as:

**FILE \***

i.e. a pointer to a `FILE` structure. If you want to see what this structure looks like, it is defined in the `stdio.h` header.

## Dealing with Errors

§ **fopen may fail for one of many reasons, how to tell which?**

```
void perror(const char* message);
```

```
#include <stdio.h>

int main(void)
{
 FILE* in;

 if((in = fopen("autoexec.bat", "r")) == NULL) {
 fprintf(stderr, "open of autoexec.bat failed ");
 perror("because");
 return 1;
 }

 open of autoexec.bat failed because: No such file or directory
```

## Dealing with Errors

The important thing to realize about streams is that because they are pointers it is possible for **fopen** to indicate a problem by returning NULL. This is a special definition of an invalid pointer seen previously. Thus if **fopen** returns NULL, we are guaranteed something has gone wrong.

### What Went Wrong?

The problem is that “something has gone wrong” is not really good enough. We need to know *what* has gone wrong and whether we can fix it. Is it merely that the user has spelt the filename wrong and needs to be given the opportunity to try again or has the network crashed?

The Standard Library deals with errors by manipulating a variable called “errno”, the error number. Each implementation of C assigns a unique number to each possible error situation. Thus 1 could be “file does not exist”, 2 could be “not enough memory” and so on. It is possible to access “errno” by placing:

```
extern int errno;
```

somewhere at the top of the program. After the failed call to **fopen** we could say:

```
fprintf("open of autoexec failed because %i\n", errno);
```

this would produce:

```
open of autoexec failed because 1
```

which is rather unhelpful. What **perror** does is to look up the value of 1 in a table and find a useful text message. Notice that it prints whatever string is passed to it (“because” in the program above) followed by a “:” character. If you don’t want this, invoke it as:

```
perror("");
```

In which case no text is prepended to the error text.



## File Access Problem

§ Can you see why the following will ALWAYS fail, despite the file existing and being fully accessible?

```
if((in = fopen("C:\autoexec.bat", "r")) == NULL) {
 fprintf(stderr, "open of autoexec.bat failed ");
 perror("because");
 return 1;
}
```

```
C:> dir C:\autoexec.bat
Volume in drive C is MS-DOS_62
Directory of C:\

autoexec bat 805 29/07/90 8:15
 1 file(s) 805 bytes
 1,264,183,808 bytes free

C:> myprog
open of autoexec.bat failed because: No such file or directory
```



## File Access Problem

There is a rather nasty problem waiting in the wings when C interacts with operating systems like MS-DOS, NT and OS/2 which use pathnames of the form:

**\name\text\file**

but not Unix which uses pathnames of the form:

**/name/text/file**

The problem is with the directory separator character, “\” vs. “/”. Why is this such a problem? Remember that the character sequences “\n”, “\t” and “\a” have special significance in C (as do “\f”, “\r”, “\v” and “\x”). The file we would actually be trying to open would be:

<newline>**ame**<tab>**ext**<alert>**file**

No such problem exists in Unix, because C attaches no special significance to “/n” which it sees as two characters, not one as in the case of “\n”. There are two solutions. The first: (which is rather inelegant) is to prepend “\” as follows:

**\\name\\text\\file**

The second: despite the fact we are not using Unix, specify a Unix style path. Some routine somewhere within the depths of MS-DOS, Windows, NT etc. seems to understand and switch the separators around the other way. This behavior is not covered by The Standard and thus you can’t rely upon it. The safest choice is the first solution which will always work, even though it does look messy.

## Displaying a File

```
#include <stdio.h>
int main(void)
{
 char in_name[80];
 FILE *in_stream;
 int ch;

 printf("Display file: ");
 scanf("%79s", in_name);

 if((in_stream = fopen(in_name, "r")) == NULL) {
 fprintf(stderr, "open of %s for reading failed ", in_name);
 perror("because");
 return 1;
 }

 while((ch = fgetc(in_stream)) != EOF)
 putchar(ch);

 fclose(in_stream);

 return 0;
}
```

## Displaying a File

### Reading the Pathname but Avoiding Overflow

The array “in\_name” being 80 characters in length gives the user room to specify a reasonably lengthy path and filename. Don’t think that all filenames should be 13 characters in length just because your operating system uses “eight dot three” format. The user will invariably need to specify a few directories too. The pathname that results can be almost any length.

```
scanf("%79s", in_name);
```

uses %79s to prevent the user from corrupting memory if more than 79 characters are typed (space is left for the null terminator). You will also notice this **scanf** is missing an “&”. Normally this is fatal, however here it is **not** a mistake. An array name automatically yields the address of the zeroth character. Thus we are providing the address that **scanf** needs, “&in\_name” is redundant.

### The Program’s Return Code

Once again **perror** is used when something goes wrong to produce a descriptive explanation. Notice that for the first time we are using

```
return 1;
```

to indicate the “failure” of the program. When the file has not been opened the program cannot be said to have succeeded. It thus indicates failure by returning a non zero value. In fact any value 1 up to and including 255 will do.

## Example - Copying Files

```
#include <stdio.h>

int main(void)
{
 char in_name[80], out_name[80];
 FILE *in_stream, *out_stream;
 int ch;

 printf("Source file: "); scanf("%79s", in_name);
 if((in_stream = fopen(in_name, "r")) == NULL) {
 fprintf(stderr, "open of %s for reading failed ", in_name);
 perror("because");
 return 1;
 }

 printf("Destination file: "); scanf("%79s", out_name);
 if((out_stream = fopen(out_name, "w")) == NULL) {
 fprintf(stderr, "open of %s for writing failed ", out_name);
 perror("because");
 return 1;
 }

 while((ch = fgetc(in_stream)) != EOF)
 fputc(ch, out_stream);

 fclose(in_stream);
 fclose(out_stream);

 return 0;
}
```

## Example - Copying Files

### Reading and Writing Files

Two arrays are needed for the input and output file names. The first file is, as before, opened for reading by specifying "r" to **fopen**. The second file is opened for writing by specifying "w" to **fopen**. Characters are then transferred between files until EOF is encountered within the source file.

### Closing files

Strictly speaking when we fail to open the destination file for writing, before the **return**, we should **fclose** the source file. This is not actually necessary, since on "normal" exit from a program, C closes all open files. This does **not** happen if the program "crashes".

Closing the output file will cause any operating system dependent end of file character(s) to be written.

### Transferring the data

The loop: **while((ch = fgetc(in\_stream)) != EOF) fputc(out\_stream, ch);**

uses the Standard Library routine **fgetc** to obtain the next character in sequence from the input file. Notice the call is NOT:

**ch = fgetc(in\_name)**

i.e. we use the *stream* associated with the file rather than the *name* of the file. Any attempt to pass "in\_name" to **fgetc** would produce compiler errors. The character obtained from the file is checked against **EOF** to see if we have processed all of the characters. If not, the character is written to the output file (again via the stream associated with the file, "out\_stream" and not via the name of the file in "out\_name").

---

## Example - Copying Files (Continued)

### Blissful Ignorance of Hidden Buffers

Notice that although both “in\_stream” and “out\_stream” have buffers associated with them, we need to know nothing about these buffers. We are not required to fill them, empty them, increment pointers, decrement pointers or even know the buffers exist. The `fgetc` and `fputc` routines manage everything behind the scenes.

### Cleaning up

Finally when end of file is encountered in the input file, the loop terminates and the program calls `fclose` to close the input and output files. This is really unnecessary since C will close files for us when the program finishes (which it is about to do via `return`), however it is good practice. There are only so many files you can open simultaneously (the limit usually defined by the operating system). If you can open one file, close it, then open another and close that there is no limit on the number of files your application could deal with.

There is, of course, always the danger of forgetting to close files and then turning this code into a function which would be called repeatedly. On each call, one precious file descriptor would be used up. Eventually `fopen` would fail with “too many open files”.

Once again, `fclose` deals with the stream, “in\_stream” and not the file name “in\_name”.

### Program's Return Code

Finally the `return 0;` indicates the success (i.e. successful copy) of our program.

---

## Convenience Problem

§ Although our copy file program works, it is not as convenient as the “real thing”

```
C:> copyprog
Source file: \autoexec.bat
Destination file: \autoexec.bak
C:> dir C:\autoexec.*
Volume in drive C is MS-DOS_62
Directory of C:\

autoexec bak 805 31/12/99 12:34
autoexec bat 805 29/07/90 8:15
 2 file(s) 1610 bytes
 1,264,183,003 bytes free
C:> copyprog \autoexec.bat \autoexec.000
Source file:
```

program still prompts despite begin given file names on the command line

## Convenience Problem

### Typing Pathnames

Here we can see our program working. Note that when prompted for the source and destination files it is neither necessary nor correct to type:

`\\autoexec.bat`

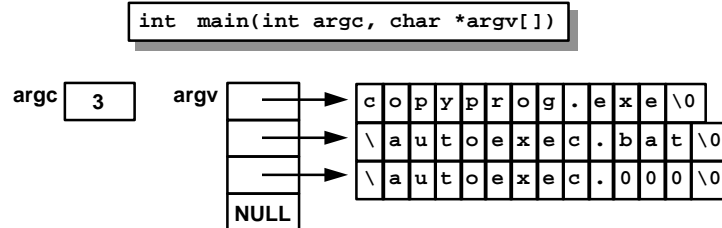
It is only the compiler (actually it's the preprocessor) which converts “\a” from the two character sequence into the alert character. Once the program has been compiled, the preprocessor is “out of the picture”, thus typing the filename is straightforward and we don't have to make a note that since the program was written in C pathnames should be typed in a strange format.

### No Command Line Interface

The fact remains that although our program works, it fails to pick up file names from the command line. It cannot be used as conveniently as the “real thing”. Clearly we would like to emulate the behavior of “supplied programs” like the “real” copy command.

## Accessing the Command Line

- ⌘ The command line may be accessed via two parameters to main, by convention these are called “argc” and “argv”
- ⌘ The first is a count of the number of words - including the program name itself
- ⌘ The second is an array of pointers to the words



## Accessing the Command Line

In environments which support C it is possible to access the command line in a clean and portable way. To do this we must change the way `main` is defined. If we use the header we have seen thus far during the course:

```
int main(void)
```

our program will ignore all words the user types on the command line (“command line parameters”). If on the other hand we use the header:

```
int main(int argc, char* argv[])
```

the program may pick up and process as many parameters (words) the user provides. Since the two variables “argc” and “argv” are parameters they are ours to name whatever we choose, for instance:

```
int main(int sky, char* blue[])
```

Providing we do not change their types the names we use are largely our choice. However, there is a convention that these parameters are always called “argc” and “argv”. This maintains consistency across all applications, across all countries, so when you see “argv” being manipulated, you know that command line parameters are being accessed. The parameters are:

**argc**  
**argv**

an integer containing a count of the number of words the user typed  
an array of pointers to strings, these strings are the actual words the user typed or an exact copy of them.

The pointers in the “argv” array are guaranteed to point to strings (i.e. null terminated arrays of characters).

## Example

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 int j;

 for(j = 0; j < argc; j++)
 printf("argv[%i] = \"%s\"\n", j, argv[j]);

 return 0;
}
```

```
C:> argprog one two three
argv[0] = "C:\cct\course\cprog\files\slideprog\argprog.exe"
argv[1] = "one"
argv[2] = "two"
argv[3] = "three"
```

---

## Example

The program above shows a C program accessing its command line. Note that element zero of the array contains a pointer to the program name, including its full path (although a few operating systems provide only the program name). This path may be used as the directory in which to find ".ini" and other data files.

Although "argc" provides a convenient count of the number of parameters the argv array itself contains a NULL terminator (a NULL pointer, not a null terminator '\0'). The loop could have been written as:

```
for(j = 0; argv[j] != NULL; j++)
 printf("argv[%i] = \"%s\"\n", j, argv[j]);
```

In fact, "argc" isn't strictly necessary, its there to make our lives slightly easier.

---

## Useful Routines

### § File reading routines:

```
int fscanf(FILE* stream, const char* format, ...);
int fgetc(FILE* stream);
char* fgets(char* buffer, int size, FILE* stream);
```

### § File writing routines:

```
int fprintf(FILE* stream, const char* format, ...);
int fputc(int ch, FILE* stream);
int fputs(const char* buffer, FILE* stream);
```

## Useful Routines

### **fscanf**

The **fscanf** routine is just like **scanf**, except for the first parameter which you will see is of the stream type. To read an int, a float and a word of not more than 39 characters into an array from the keyboard:

```
scanf("%I %f %39s", &j, &flt, word);
```

to read these things from a file:

```
fscanf(in_stream, "%I %f %39s", &j, &flt, word);
```

The **fgetc** routine has already been used in the file copy program to read individual characters from an input file.

### **fgets**

The **fgets** routine reads whole lines as strings. The only problem is fixing the length of the string. The storage used must be allocated by the user as an array of characters. Doing this involves putting a figure on how long the longest line will be. Lines longer than this magical figure will be truncated. For a “short” lines **fgets** will append the newline character, “\n” (just before the null terminator). When a “long” line is encountered, **fgets** truncates it and does not append a newline. Thus the presence or absence of the newline indicates whether the line was longer than our buffer length.

```
char line[100];

fgets(line, sizeof(line), in_stream);
if(strchr(line, '\n') == NULL)
 printf("line \"%s\" truncated at %I characters\n", line,
 sizeof(line));
```

The Standard Library routine **strchr** finds a character within a string and returns a pointer to it, if present, or NULL if absent.



---

## Useful Routines - (Continued)

### `fprintf`

The `fprintf` routine is just like `printf`, except for the first parameter which you will see is of the stream type. To write an int, a float to one decimal place and a word, left justified within a field of 39 characters:

```
printf("%i %.1f %-39s", j, flt, word);
```

to write these things to a file:

```
fprintf(out_stream, "%i %.1f %-39s", j, flt, word);
```

in fact, `printf(???)` is the equivalent of `fprintf(stdout, ???)`.

The `fputc` routine can be seen in the file copy program a few pages ago and writes single characters to a file.

### `fputs`

The `fputs` routine writes a string to a file, as follows:

```
char line[100];

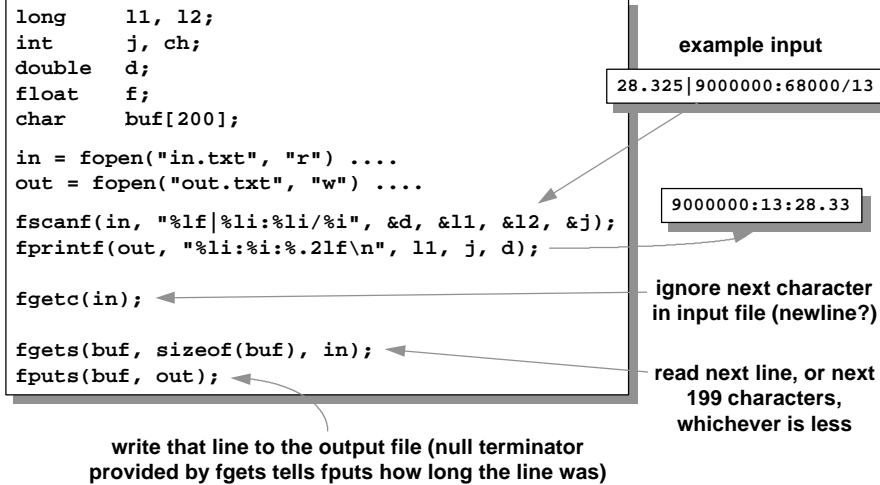
fgets(line, sizeof(line), in_stream);
fputs(line, out_stream);
```

All the characters in the character array are written, up until the null terminator “\0”. If you have a newline character at the end of the buffer this will be written out too, otherwise you will output “part of a line”. Presumably a newline character must be appended by hand at some stage.

Although `fgets` is driven by a character count (in order not to overflow the buffer), the `fputs` routine is driven by the position of the null terminator and thus does not need a count. Passing a non null terminated array of characters to `fputs` would cause serious problems.

---

## Example



## Example

The example program above shows `fscanf` reading a `double`, two `long ints` and an `int` from the file "in.txt". The `double` is separated from the first `long int` by a vertical bar "|", the two `long ints` are separated from one another by a ":", while the `long int` is separated from the `int` by "/".

When output to the file "out.txt" via the `fprintf` routine, the first `long int`, `int` and `double` are separated by ":" characters.

The next call is to `fgetc` which reads one single character from the input stream. This assumes there is a newline character immediately after the ".... 68000/13" information which we have just read. This newline character is discarded. Normally we would have said

```
ch = fgetc(in);
```

but here there seems no point assigning the newline character to anything since we're really not that interested in it.

Why all this fuss over a simple newline character? The `fgets` routine reads everything up until the next newline character. If we do not discard the one at the end of the line, `fgets` will immediately find it and read an empty line.

### `fgets` Stop Conditions

`fgets` will stop reading characters if it encounters end of file, "\n" or it reads 199 characters (it is careful to leave room for the null terminator) from the file into the buffer "buf". A null terminator is placed immediately after the last character read.

These characters are then written to the output file via the `fputs` routine.

## Binary Files

### § The Standard Library also allows binary files to be manipulated

- “b” must be added into the fopen options
- Character translation is disabled
- Random access becomes easier
- Finding the end of file can become more difficult
- Data is read and written in blocks

```
size_t fread(void* p, size_t size, size_t n, FILE* stream);
size_t fwrite(const void* p, size_t size, size_t n, FILE* stream);

int fseek(FILE* stream, long offset, int whence);
long ftell(FILE* stream);
void rewind(FILE* stream);

int fgetpos(FILE* stream, fpos_t* pos);
int fsetpos(FILE* stream, const fpos_t* pos);
```

---

## Binary Files

Thus far we have examined text files, i.e. the characters contained within each file consist entirely of ASCII (or EBCDIC) characters. Thus the file contents may be examined, edited, printed etc.

Storing, say, a double as ASCII text can be rather inefficient, consider storing the characters “3.1415926535890” (that’s 15 characters) in a file. Then some other character, perhaps space or newline would be needed to separate this from the next number. That pushes the total to 16 bytes. Storing the double itself would only cost 8 bytes. Storing another double next to it would be another 8 bytes. No separator is required since we know the exact size of each double.

This would be called a “binary file” since on opening the file we would see not recognizable characters but a collection of bits making up our double. In fact we would see 8 characters corresponding to the 8 bytes in the double. These characters would appear almost random and would almost certainly not be readable in a “human” sense.

The double containing pi could be written to a binary file as follows:

```
double pi = 3.1415926535890;
FILE* out_stream;

out_stream = fopen("out.bin", "wb");
fwrite(&pi, sizeof(double), 1, out_stream);
```

The normal checking of the return value from **fopen**, which is necessary with binary files too, has been omitted for brevity.

---

---

## Binary Files (Continued)

### `fopen` “wb”

The “wb” option to `fopen` puts the stream into “binary” write mode. This is very important, because there are a number of significant changes in the way the various routines work with binary files. Fortunately these changes are subtle and we just go ahead and write the program without needing to worry about them. Well, mostly.

### The Control Z Problem

The first change in the behavior of the routines concerns the “Control Z problem”. When MS-DOS was invented, someone decided to place a special marker at the end of each file. The marker chosen was Control-Z (whose ASCII value is 26). *Writing* a byte containing 26 to a file is no problem. *Reading* a byte containing 26 back again is a problem. If in text mode, the 26 will appear as end of file, `fgetc` will return `EOF` and you will not be able to read any further. It is therefore very important that you read binary files in binary mode. If you read a binary file in text mode you will get some small percentage of the way through the file, find a 26, and inexplicably stop.

Since MS-DOS had an influence on the design of Windows 95, NT and OS/2 they all share this problem, even though no one actually *does* store Control-Z at the end of files any more (this is because there were too many problems when an application failed to write the Control-Z. Such “EOF-less” files grew without limit and eventually ate all the disk space).

This begs the question as to how, if our end of file character has been “used up”, we can detect end of file with a binary file. This isn’t really that different a question to how with “modern” files we can detect end of file when there is no appended Control-Z. This is all done by the operating system which somewhere must maintain a count of the number of bytes in the file (the “dir” command certainly doesn’t open each file and count the number of bytes each time you run it).

---

## Binary Files (Continued)

### The Newline Problem

The second change in behavior revolves around the newline character “\n”. To understand what a newline *really* does, you need to think of the movement of a print head either for a teletype or for a printer (back in the days when printers had print heads). At the end of a line the print head returns to column one. This is called a “carriage return”. Then the paper moves up one line, called a “line feed”. Thus a single newline character would seem to do *two* things.

Under Unix there is an immense amount of heavy code within the terminal driver to make sure these two things happen whenever a “\n” is output. This behavior can even be turned off whenever appropriate.

MS-DOS is a much more simple operating system. It was decided that a newline character should do one thing and not two. It is therefore necessary to place *two* characters “\r” and “\n” at the end of each line in an MS-DOS file. The “\r”, carriage return character moves the cursor to column one, the “\n” causes the line feed to move to the next line.

Clearly we have not taken this into account thus far. In fact the Standard Library routines take care of this for us. If we do the following:

```
FILE* out_stream;

out_stream = fopen("out.txt", "w");
fprintf(out_stream, "hi\n");
```

Because “out\_stream” is opened in text mode, four characters are written to the file, “h”, “i”, “\r”, “\n”. If we had done the following:

```
FILE* out_stream;

out_stream = fopen("out.bin", "wb");
fprintf(out_stream, "hi\n");
```

then only three characters would have been written, “h”, “i”, “\n”. Without the carriage returns in the file, listing it would produce some interesting effects.

The upshot is:

|             |          |                                   |
|-------------|----------|-----------------------------------|
| text mode   | write 10 | 13 10 written                     |
| binary mode | write 10 | 10 written                        |
| text mode   | read 13  | see 13 (if 13 not followed by 10) |
|             |          | see 10 (if 13 followed by 10)     |
| binary mode | read 13  | see 13                            |

You can imagine that if a binary file were read in text mode and these 10s and 13s were embedded within, say, a **double** the value would not be pulled back out properly.

---

---

## Binary Files (Continued)

### The Movement Problem

A further problem arises with random movement around the file. Say we wish to move forwards 100 bytes in a file. If the file were opened in text mode every time we moved over a 10 it would count as two characters. Thus if there were 3 10s within the next 100 bytes would that mean we should move forwards 103 bytes instead? If the file were opened in binary mode there wouldn't be a problem since a 10 is a 10, moving 100 bytes would mean 100 bytes.

### Moving Around Files

There are two mechanisms for moving around files. As just discussed these work best with binary files. The "traditional" method is to use the `fseek` function.

```
int fseek(FILE* stream, long offset, int whence);
```

The second parameter, of type `long` is the position we wish to move to. Thus if we wished to move to the 30th byte in the file (regardless of our current position):

```
fseek(stream, 30L, SEEK_SET);
```

Where "stream" is the stream opened for reading or writing in binary mode and `SEEK_SET` is a constant defined in `stdio.h` which says "move relative to the start of the file". Two other constants `SEEK_CUR`, "move relative to the current position" and `SEEK_END`, "move relative to the end of the file", are available.

When `SEEK_SET` is used, the position specified must not be negative. When `SEEK_CUR` is used, the position may be either positive or negative, when `SEEK_END` is used the value should be negative. The `ftell` function may be used to determine the current position within the file.

### `fsetpos` vs. `fseek`

A fundamental problem with `fseek` and `ftell` is that the maximum value of a `long` is  $2^{31}-1$ . In byte terms we can move to any position within a 2.1 Gigabyte file. If the file is larger we're in trouble. To address this problem, the Standard Library defines two other routines:

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

where `fpos_t` is an implementation specific type able to hold a position within a file of arbitrary size. Unfortunately you need to visit the point you wish to return to first. In other words `fgetpos` must be called to initialize an `fpos_t` before `fsetpos` can be called using the `fpos_t`. It is not possible to say "move the position forwards by 3000 bytes" as it is with `fseek`, though you could say "move the position backwards by 3000 bytes" as long as you had remembered to save the position 3000 bytes ago.

---

## Example

```
double d;
long double lda[35];
fpos_t where;

in = fopen("binary.dat", "rb");
out = fopen("binnew.dat", "wb");

fread(&d, sizeof(d), 1, in);

fgetpos(in, &where);
fread(lda, sizeof(lda), 1, in);

fsetpos(in, &where);
fread(lda, sizeof(long double), 35, in);

fwrite(lda, sizeof(long double), 20, out);

fseek(in, 0L, SEEK_END);
```

Annotations:

- read one chunk of 8 bytes (points to `fread(&d, ...)`)
- remember current position in file (points to `fgetpos(in, &where);`)
- read one chunk of 350 bytes (points to `fread(lda, ...)`)
- return to previous position (points to `fsetpos(in, &where);`)
- read 35 chunks of 10 bytes (points to `fread(lda, ...)`)
- write 20 long doubles from lda (points to `fwrite(lda, ...)`)
- move to end of binary.dat (points to `fseek(in, 0L, SEEK_END);`)

## Example

This is an example of some of the routines mentioned. It is an example only, not a particularly coherent program. Firstly the files are opened in binary mode by appending "b" to the file mode. The first `fread` transfers `sizeof(d) == 8` bytes, multiplied by 1 (the next parameter) from the stream "in" into the variable "d".

The current position is saved using the `fgetpos` function. The second `fread` transfers `sizeof(lda) == 350` bytes, multiplied by 1, into "lda". As "lda" is an array, it is not necessary to place an "&" before it as the case with "d".

Using the `fsetpos` function, the file position is returned to the point at which the 35 long doubles are stored (just after the initial 8 byte double which was read into "d"). These long doubles are then re-read. This time the parameters to `fread` are `sizeof(long double) == 10` bytes and 35 because we need to read 35 chunks of 10 bytes. The net effect is *exactly* the same. 350 bytes are transferred from the file directly into the array "lda".

We then write the first 20 long doubles from the "lda" array to the file "out".

The call to `fseek` moves the current file position to zero bytes from the end of the file (because `SEEK_END` is used). If the call had been

```
fseek(in, 0L, SEEK_SET);
```

we would have moved back to the start of the file. This would have been equivalent to:

```
rewind(in);
```

## Summary

- ⌘ **Streams** `stdin`, `stdout`, `stderr`
- ⌘ **`fopen`** opening text files
- ⌘ **functions:** `perror`, `fprintf`, `fscanf`, `fgetc`, `fputc`
- ⌘ **variables:** `argc`, `argv`
- ⌘ **“b” option to `fopen`** to open binary files
- ⌘ **functions:** `fread`, `fwrite`, `fseek`, `ftell`

---

## Summary

---



---

---

## Handling Files in C Practical Exercises

---

---

Directory: **STDLIB**

1. Write a program called "**SHOW**" which displays text files a screenfull at a time (rather like "more"). Prompt for the name of the file to display. Assume the screen is 20 lines long.

2. Update "**SHOW**" such that it tests for a file name on the command line, so

**SHOW SHOW.C**

would display its own source code. If no file name is provided, prompt for one as before.

3. Further update "**SHOW**" so that it will display each one in a list of files:

**SHOW SHOW.C FCOPY.C ELE.TXT**

Using the prompt "press return for next file <name>" when the end of the first two files has been reached. Do not produce this prompt after "**ELE.TXT**"

4. The file "**ELE.TXT**" is a text file containing details about the elements in the Periodic Table. The format of each line in the file is:

**nm 45.234 100.95 340.98**

where "nm" is the two letter element name, 45.234 is the atomic weight (or "relative molecular mass" as it is now called), 100.95 is the melting point and 340.98 the boiling point. Write a program "**ELEMS**" to read this text file and display it, 20 lines at a time on the screen.

5. Using "**ELEMS**" as a basis, write a program "**ELBIN**" to read the text information from "**ELE.TXT**" and write it to a binary file "**ELE.BIN**". Then write a "**BINSHOW**" program to read the binary file and display the results on the screen. The results should look the same as for your "**ELEMS**" program. If you write **floats** to the file, you should notice the binary file is smaller than its text equivalent.
6. When a program is compiled, all strings that are to be loaded into the data segment at run time are written into the executable. If the executable is opened for reading in binary mode, these strings may be found and printed. Write such a program "**STRINGS.C**" which prints out sequences of 4 or more printable characters. The character classification routines from **<ctype.h>** may be helpful in determining what is printable and what is not.

---

---

## Handling Files in C Solutions

---

---

1. Write a program called **"SHOW"** which displays text files a screenful at a time (rather like "more"). Prompt for the name of the file to display. Assume the screen is 20 lines long.

*Using scanf to prompt for the filename leaves an unread newline character in the input buffer. This must be discarded or the first call to getchar in the show function will appear to do nothing (it will merely pick up the newline from the buffer). The call to getchar immediately after the call to scanf discards this newline. Notice also how the show function uses getchar within a loop. Typing "abc<return>" would cause four characters to be saved in the input buffer. If getchar is only called once each time, three other pages will zoom past. The return value from show is used as the return value from the program. Thus when show fails to open a file the return value is 1. When everything goes well, the return value is 0.*

```
#include <stdio.h>
#define STOP_LINE 20

int show(char*);

int main(void)
{
 char name[100+1];

 printf("File to show ");
 scanf("%100s", name);
 getchar();

 return show(name);
}

int show(char* filename)
{
 int ch;
 int lines = 0;
 FILE* stream;

 if((stream = fopen(filename, "r")) == NULL) {
 fprintf(stderr, "Cannot open file %s, ", filename);
 perror("");
 return 1;
 }

 while((ch = fgetc(stream)) != EOF) {
 putchar(ch);
 if(ch == '\n') {
 lines++;
 if(lines == STOP_LINE) {
 lines = 0;
 while(getchar() != '\n')
 ;
 }
 }
 }
 fclose(stream);

 return 0;
}
```

2. Update “**SHOW**” such that it tests for a file name on the command line

*By using `strncpy` to copy characters, the program is protected from overflowing the array “name”. However, `strncpy` does not guarantee to null terminate the buffer in the case where the maximum possible number of characters were copied across. Thus the program ensures the buffer is null terminated.*

```
#include <stdio.h>
#include <string.h>

#define STOP_LINE 20

int show(char*);

int main(int argc, char* argv[])
{
 char name[100+1];

 if(argc == 1) {
 printf("File to show ");
 scanf("%100s", name);
 getchar();
 } else {
 strncpy(name, argv[1], sizeof(name));
 name[sizeof(name) - 1] = '\0';
 }

 return show(name);
}

int show(char* filename)
{
 int ch;
 int lines = 0;
 FILE* stream;

 if((stream = fopen(filename, "r")) == NULL) {
 fprintf(stderr, "Cannot open file %s, ", filename);
 perror("");
 return 1;
 }

 while((ch = fgetc(stream)) != EOF) {
 putchar(ch);
 if(ch == '\n') {
 lines++;
 if(lines == STOP_LINE) {
 lines = 0;
 while(getchar() != '\n')
 ;
 }
 }
 }
 fclose(stream);

 return 0;
}
```

---

3. Further update "SHOW" so that it will display each one in a list of files.

*With this version, the character array needed if there are no command line parameters is declared within the if statement. If the array is required, the storage is allocated and used. Whereas many different possible error strategies exist (the program could exit when the first error occurs opening a file) this program "stores" the error status and continues. When the program finishes this error value is returned.*

```
#include <stdio.h>
#include <string.h>

#define STOP_LINE 20

int show(char*);

int main(int argc, char* argv[])
{
 int i;
 int err = 0;

 if(argc == 1) {
 char name[100+1];

 printf("File to show ");
 scanf("%100s", name);
 getchar();
 return show(name);
 }

 for(i = 1; i < argc; i++) {
 if(show(argv[i]))
 err = 1;

 if(i < argc - 1) {
 printf("press return for next file %s\n", argv[i + 1]);
 while(getchar() != '\n')
 ;
 }
 }

 return err;
}
```

```

int show(char* filename)
{
 int ch;
 int lines = 0;
 FILE* stream;

 if((stream = fopen(filename, "r")) == NULL) {
 fprintf(stderr, "Cannot open file %s, ", filename);
 perror("");
 return 1;
 }

 while((ch = fgetc(stream)) != EOF) {
 putchar(ch);
 if(ch == '\n') {
 lines++;
 if(lines == STOP_LINE) {
 lines = 0;
 while(getchar() != '\n')
 ;
 }
 }
 }
 fclose(stream);

 return 0;
}

```

---

4. The file format of "ELE.TXT" is

```
nm 45.234 100.95 340.98
```

Write a program "ELEMS" to read this text file and display it

```

#include <stdio.h>
#include <string.h>

#define STOP_LINE 20

int show(char*);
void processFile(FILE* s);

```

---

```
int main(int argc, char* argv[])
{
 char*p;
 char name[100+1];

 if(argc == 1) {
 printf("File to show ");
 scanf("%100s", name);
 getchar();

 p = name;
 } else
 p = argv[1];

 return show(p);
}

int show(char* filename)
{
 FILE*stream;

 if((stream = fopen(filename, "r")) == NULL) {
 fprintf(stderr, "Cannot open file %s, ", filename);
 perror("");
 return 1;
 }
 processFile(stream);
 fclose(stream);

 return 0;
}

void processFile(FILE* s)
{
 char name[3];
 float rmm;
 float melt;
 float boil;
 int count = 0;

 while(fscanf(s, "%2s %f %f %f", name, &rmm, &melt, &boil) == 4) {
 printf("Element %-2s rmm %6.2f melt %7.2f boil %7.2f\n",
 name, rmm, melt, boil);

 if(++count == STOP_LINE) {
 count = 0;
 while(getchar() != '\n')
 ;
 }
 }
}
```

---



5. Write a program to read "**ELE.TXT**" and write it to a binary file "**ELE.BIN**". Then write a "**BINSHOW**" program to read the binary file and display the results on the screen.

*The binary file generator is listed first:*

```
#include <stdio.h>

int convert(char*, char*);
void processFile(FILE*, FILE*);

int main(int argc, char* argv[])
{
 char* in;
 char* out;
 char in_name[100+1];
 char out_name[100+1];

 switch(argc) {
 case 1:
 printf("File to read ");
 scanf("%100s", in_name);
 getchar();

 printf("File to write ");
 scanf("%100s", out_name);
 getchar();

 in = in_name;
 out = out_name;
 break;
 case 2:
 printf("File to write ");
 scanf("%100s", out_name);
 getchar();

 in = argv[1];
 out = out_name;
 break;
 case 3:
 in = argv[1];
 out = argv[2];
 break;
 }

 return convert(in, out);
}

int convert(char* in, char* out)
{
 FILE* in_stream;
 FILE* out_stream;

 if((in_stream = fopen(in, "r")) == NULL) {
 fprintf(stderr, "Cannot open input file %s, ", in);
 perror("");
 return 1;
 }
}
```

---

```

 }
 if((out_stream = fopen(out, "wb")) == NULL) {
 fprintf(stderr, "Cannot open output file %s, ", out);
 perror("");
 return 1;
 }
 processFile(in_stream, out_stream);

 fclose(in_stream);
 fclose(out_stream);

 return 0;
}

void processFile(FILE* in, FILE* out)
{
 char name[3];
 float rmm;
 float melt;
 float boil;

 while(fscanf(in, "%2s %f %f %f", name, &rmm, &melt, &boil) == 4) {
 fwrite(name, sizeof(char), 2, out);
 fwrite(&rmm, sizeof(rmm), 1, out);
 fwrite(&melt, sizeof(melt), 1, out);
 fwrite(&boil, sizeof(boil), 1, out);
 }
}

```

---

*Now the binary file listing program:*

```

#include <stdio.h>

#define STOP_LINE 20

int show(char*);
void processFile(FILE* s);

int main(int argc, char* argv[])
{
 char* p;
 char name[100+1];

 if(argc == 1) {
 printf("File to show ");
 scanf("%100s", name);
 getchar();

 p = name;
 } else
 p = argv[1];

 return show(p);
}

```

---

```

int show(char* filename)
{
 FILE* stream;

 if((stream = fopen(filename, "rb")) == NULL) {
 fprintf(stderr, "Cannot open file %s, ", filename);
 perror("");
 return 1;
 }

 processFile(stream);

 fclose(stream);

 return 0;
}

void processFile(FILE* in)
{
 char name[3] = { 0 };
 float rmm;
 float melt;
 float boil;
 int count = 0;

 while(fread(name, sizeof(char), 2, in) == 2 &&
 fread(&rmm, sizeof(rmm), 1, in) == 1 &&
 fread(&melt, sizeof(melt), 1, in) == 1 &&
 fread(&boil, sizeof(boil), 1, in) == 1) {

 printf("Element %-2s rmm %6.2f melt %7.2f boil %7.2f\n",
 name, rmm, melt, boil);
 if(++count == STOP_LINE) {
 count = 0;
 while(getchar() != '\n')
 ;
 }
 }
}

```

- 
6. Write a program “**STRINGS.C**” which prints out sequences of 4 or more printable characters from an executable.

*The program buffers printable characters one through four. When a fifth is found the preceding four characters are printed followed by the fifth. The sixth and subsequent characters are printed directly. The first non printable character causes a newline to be output.*

*The program uses its own name in error messages thus if and when the user moves the executable, errors reflect the new program name and a fixed one. Notice that only the last component of argv[0] is used (the filename itself) and then only those characters before the “.” (this loses “.exe”).*

---

*As an alternative to prompting for a filename if none is provided, this program produces an error message.*

```
#include <stdio.h>
#include <string.h>

#define MAGIC_LENGTH 4

int open_file(char*, char*);
void process_file(FILE*);

int main(int argc, char* argv[])
{
 int i;
 int err = 0;
 char* dot;
 char* name;

 if((name = strrchr(argv[0], '\\')) == NULL)
 name = argv[0];
 else
 name++;

 if((dot = strchr(name, '.')) != NULL)
 *dot = '\\0';

 if(argc == 1) {
 fprintf(stderr, "usage: %s filename [filename]\\n", name);
 return 9;
 }

 for(i = 1; i < argc; i++) {
 if(open_file(argv[i], name))
 err = 1;

 if(i < argc - 1) {
 printf("press return for next file %s\\n", argv[i + 1]);
 while(getchar() != '\\n')
 ;
 }
 }

 return err;
}
```

---

```
int open_file(char* filename, char* progame)
{
 FILE* stream;

 if((stream = fopen(filename, "rb")) == NULL) {
 fprintf(stderr, "%s: Cannot open file %s, ",
 progame, filename);
 perror("");
 return 1;
 }

 process_file(stream);
 fclose(stream);

 return 0;
}

void process_file(FILE* in)
{
 int i;
 int ch;
 int count = 0;
 char buffer[MAGIC_LENGTH];

 while((ch = fgetc(in)) != EOF) {
 if(ch < ' ' || ch >= 0x7f) {
 if(count > MAGIC_LENGTH)
 putchar('\n');
 count = 0;
 } else {
 if(count < MAGIC_LENGTH)
 buffer[count] = ch;
 else if(count == MAGIC_LENGTH) {
 for(i = 0; i < MAGIC_LENGTH; i++)
 putchar(buffer[i]);
 putchar(ch);
 } else
 putchar(ch);

 ++count;
 }
 }
 if(count > MAGIC_LENGTH)
 putchar('\n');
}
```

---



---

---

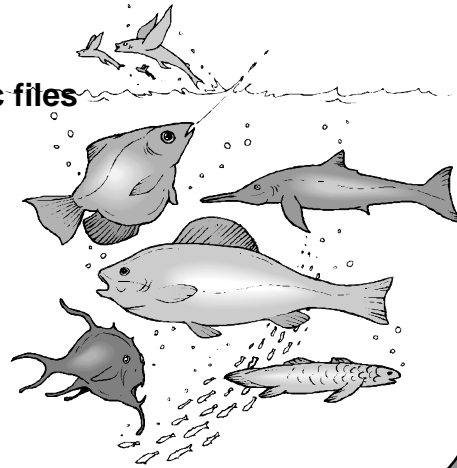
Miscellaneous Things

---

---

## Miscellaneous Things

- ☞ Unions
- ☞ Enumerated types
- ☞ The Preprocessor
- ☞ Working with multiple .c files



---

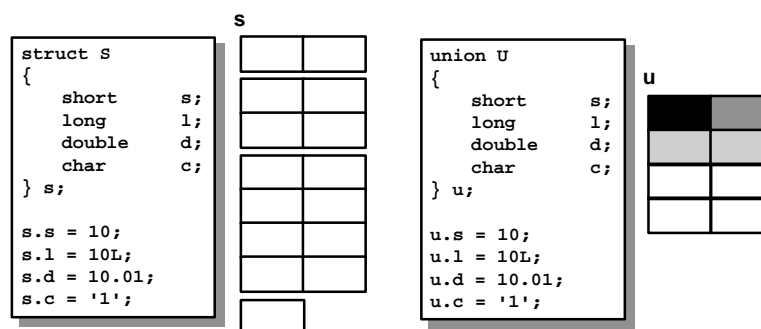
## Miscellaneous Things

This chapter covers most of the remaining important “odds and ends” in C.



## Unions

§ A union is a variable which, at different times, may hold objects of different types and sizes



## Unions

Unions provide a mechanism for overlaying variables of different types and sizes in the same memory. In the example above, the **struct** "S" arranges its members to follow one after another. The **union** "U" arranges its members to overlap and thus occupy the same region of memory.

### Size of **struct** vs. Size of **union**

The **struct** instance "s" would be 15 bytes in size (or maybe 16 when "padded"). The **union** instance "u" would be size 8 bytes in size, the size of its largest member "d".

Assigning a value to "s.s" will not effect the value stored in "s.l" or any other member of "s". Assigning a value to "u.s" will write into the first 2 of 8 bytes. "u" would store a **short**. Assigning a value to "u.l" would write into the first 4 of the 8 bytes. "u" would store a **long**. The value previously stored in "u.s" would be overwritten. Assigning a value to "u.d" would write over all 8 bytes of "u". The **long** previously stored would be overwritten and "u" would store a **double**. Assigning a value to "u.c" would write into the first byte of "u". This would be sufficient to "corrupt" the **double**, but since "u" is storing a character, we shouldn't look at the **double** anyway.

Thus, a **union** may hold values of different types here, a **short**, **long**, **double** or **char**. Unlike the structure "s", the **union** "u" may NOT hold two values at the same time.

## Remembering

- § It is up to the programmer to remember what type a union currently holds
- § Unions are most often used in structures where a member records the type currently stored

```

struct preprocessor_const
{
 char* name;
 int stored;
 union
 {
 long lval;
 double dval;
 char* sval;
 } u;
};

```

```

#define N_SIZE 10
#define PI 3.1416

```

```

struct preprocessor_const s[10000];

s[0].name = "N_SIZE";
s[0].u.lval = 10L;
s[0].stored = STORED_LONG;

s[1].name = "PI";
s[1].u.dval = 3.1416;
s[1].stored = STORED_DOUBLE;

```

## Remembering

The compiler gives no clues as to what value a **union** currently holds. Thus with the **union** "u" on the previous page we could write a value into the **long** (4 byte) part, but read a value from the **short** (2 byte) part. What is required is some mechanism for remembering what type is currently held in the **union**. All is possible, but we have to do the work ourselves.

### A Member to Record the Type

In this example a **union** is placed in a structure along with a member "stored" which records the type currently stored within the **union**. Whenever a value is written into one of the union's members, the corresponding value (probably **#defined**) is placed in the "stored" member. The example above is how a symbol table for a preprocessor might look. A simple preprocessor could deal with constants as either **longs**, **doubles** or strings. To this end, the define

```
#define N_SIZE 100
```

would cause the name "N\_SIZE" to be stored and the value 100 stored as a **long**. For the define

```
#define PI 3.1416
```

the name "PI" would be stored and the value 3.1416 stored as a **double**. For a define

```
#define DATA_FILE "c:\data\datafile1.dat"
```

the name "DATA\_FILE" would be stored and the value "c:\data\datafile1.dat" stored as a **char\***.

By overlaying the **long**, **double** and **char\*** each **preprocessor\_const** uses only the minimum amount of memory. Clearly a preprocessor constant cannot be a **long**, a **double** and a string at the same time. Using a **struct** instead of a **union** would have wasted storage (since only one out of the three members of the structure would

have been used).

## Enumerated Types

§ Enumerated types provide an automated mechanism for generating named constants

```
enum day { sun, mon, tue,
 wed, thu, fri, sat };

enum day today = sun;

if(today == mon)

```

```
#define sun 0
#define mon 1
#define tue 2
#define wed 3
#define thu 4
#define fri 5
#define sat 6

int today = sun;

if(today == mon)

```

## Enumerated Types

The enumerated type provides an “automated” **#define**. In the example above, seven different constants are needed to represent the days of the week. Using **#define** we must specify these constants and ensure that each is different from the last. With seven constants this is trivial, however imagine a situation where two or three hundred constants must be maintained.

The **enum** guarantees different values. The first value is zero, each subsequent value differs from the last by one.

The two examples above are practically identical. **enums** are implemented by the compiler as “integral types”, whether **ints** or **longs** are used is dictated by the constants (a constant larger than 32767 on a machine with 2 byte integers will cause a switch to the use of **long** integers).

## Using Different Constants

§ The constants used may be specified

```
enum day { sun = 5, mon, tue, wed, thu, fri, sat };
enum direction { north = 0, east = 90, south = 180,
 west = 270 };
```

§ What you see is all you get!

§ There are no successor or predecessor functions

---

## Using Different Constants

enum does not force the programmer to use values from 0 onwards, this is just the default. With the enum “day” above, the initial value of 5 causes “sun” to be 5, “mon” to be 6 and so on. With the enum “direction”, the value of each of the constants is specified.

### Printing `enums`

There is no mechanism for directly printing enum as text. The following is possible:

```
enum direction heading = west;

printf("your direction is currently %i\n", heading);
```

or alternatively the user may prefer:

```
enum direction heading = west;

printf("your direction is currently ");
switch(heading) {
 case north:
 printf("north\n");
 break;
 case east:
 printf("east\n");
 break;
 case west:
 printf("west\n");
 break;
 case south:
 printf("south\n");
 break;
}
```

It is also not possible to say “the direction before east”, or “the direction after south” without writing yet more code.

---

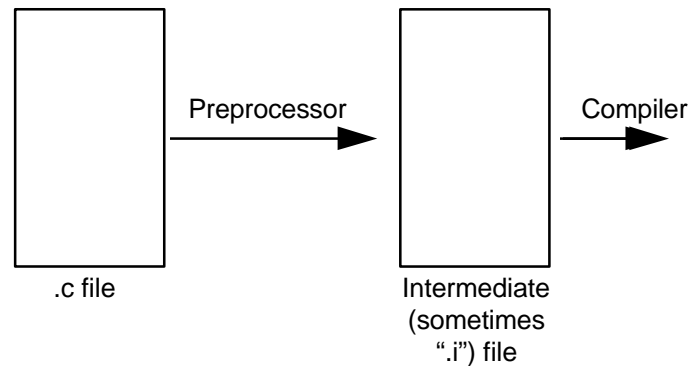
## The Preprocessor

- § Preprocessor commands start with '#' which may optionally be surrounded by spaces and tabs
- § The preprocessor allows us to:
  - include files
  - define, test and compare constants
  - write macros
  - debug

---

## The Preprocessor

We have used the preprocessor since the very first program of the course, but never looked in detail at what it can do. The preprocessor is little more than an editor placed "in front of" the compiler. Thus the compiler never sees the program you write, it only sees the preprocessor output:



As we have seen, preprocessor commands start with "#" as in:

```
#include <stdio.h>
```

which may also be written as

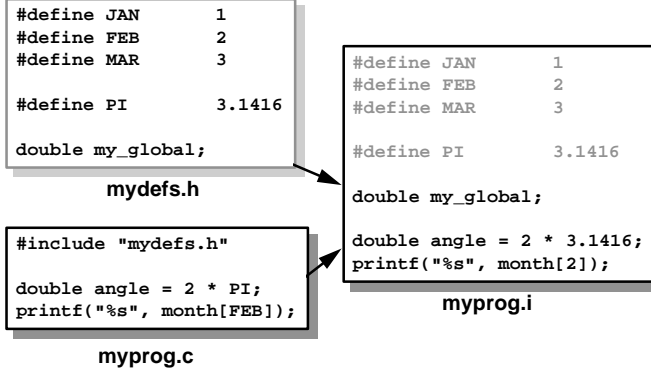
```
include <stdio.h>
```

As well as including files and defining constants, the preprocessor performs a number of other useful tasks.

---

## Including Files

§ The `#include` directive causes the preprocessor to “edit in” the entire contents of another file



---

## Including Files

The `#include` directive causes the preprocessor to physically insert (and then interpret) the entire contents of a file into the intermediate file. By the time this has been done, the compiler cannot tell the difference between what you’ve typed and the contents of the header files.

---

## Pathnames

- Full pathnames may be used, although this is not recommended

```
#include "C:\cct\course\cprog\misc\slideprog\header.h"
```

- The “I” directive to your local compiler allows code to be moved around much more easily

```
#include "header.h"
```

```
cc -I c:\cct\course\cprog\misc\slideprog myprog.c
```

---

## Pathnames

### Finding

#### #include Files

The preprocessor “knows” where to look for header files. When

```
#include <stdio.h>
```

is used, the compiler knows where the `stdio.h` file lives on the disk. In fact it examines the INCLUDE environment variable. If this contains a path, for example “c:\bc5\include” with the Borland 5.0 compiler, it opens the file “c:\bc5\include\stdio.h”.

If:

```
#include "stdio.h"
```

had been used, the preprocessor would have looked in the current directory for the file *first*, then in whatever “special” directories it knows about after (INCLUDE may specify a number of directories, separated by “;” under DOS like operating systems).

If there is a specific file in a specific directory you wish to include it might be tempting to use a full path, as in the slide above. However this makes the program difficult to port to other machines. All of the .c files using this path would have to be edited. It is much easier to use double quotes surrounding the file name only and provide the compiler with an alternative directory to search. This can be done either by updating the INCLUDE variable, or with the “I” option (they both amount to the same thing) although with the fully integrated development environments in common use today it can be a battle to find out how precisely this is done.



## Preprocessor Constants

### § Constants may be created, tested and removed

```
#if !defined(SUN)
#define SUN 0
#endif

#if SUN == MON
#undef SUN
#endif
```

if "SUN" is not defined, then begin  
define "SUN" as zero  
end

if "SUN" and "MON" are equal, then begin  
remove definition of "SUN"  
end

```
#if TUE
```

if "TUE" is defined with a non zero value

```
#if WED > 0 || SUN < 3
```

if "WED" is greater than zero or "SUN" is less than 3

```
#if SUN > SAT && SUN > MON
```

if "SUN" is greater than "SAT" and "SUN" is greater than "MON"

## Preprocessor Constants

Preprocessor constants are the "search and replace" of the editor. The constants themselves may be tested and even removed. The **defined** directive queries the preprocessor to see if a constant has been created. This is useful for setting default values. For instance:

```
#define YEAR_BASE 1970
```

```
#define YEAR_BASE 1970
```

will produce an error because the symbol "YEAR\_BASE" is defined twice (even though the value is the same). It would seem daft to do this, however the first line may be in a header file while the second in the .c file. This case may be catered for by:

```
#define YEAR_BASE 1970 (in the header)
```

```
#if defined(YEAR_BASE) (in the .c)
#undef YEAR_BASE
#endif
#define YEAR_BASE 1970
```

This would keep the preprocessor happy, since at the point at which "YEAR\_BASE" were #defined for the second time, no previous value would exist.

```
#if
#endif
#define
#undef
```

rather like "if(....) {"  
like the "}" closing the "if(....) {" block  
set up a search and replace  
forget a search and replace

## Avoid Temptation!

§ The following attempt to write Pascal at the C compiler will ultimately lead to tears

```

#define begin {
#define end ;}
#define if if(
#define then)
#define integer int

integer i;

if i > 0 then begin
 i = 17
end

```

→

```

int i;

if(i > 0) {
 i = 17
}

```

## Avoid Temptation!

The preprocessor can be used to make C look like other languages. By writing enough preprocessor constructs it is possible to make C look like your favourite language. However, ultimately this is not a good idea. No matter how hard you try it is almost inevitable that you cannot make the preprocessor understand every single construct you'd like. For instance when writing Pascal, assignments are of the form:

```
a := b;
```

It is not possible to set this up with the preprocessor, although you might think

```
#define := =
```

would work, it causes the preprocessor no end of grief. Also, declarations in Pascal are the “opposite” to C:

```
i: integer;
```

There is no way to do this either. Thus what you end up with is an unpleasant mixture of Pascal and C. However, it gets worse. To test a variable in Pascal:

```
if i = 0 then
```

would be perfect. In C assignment results. Thus our Pascal would never be Pascal only a “version” or a “variation” of it. Thus the whole idea is best avoided.

## Preprocessor Macros

§ The preprocessor supports a macro facility which should be used with care

```
#define MAX(A,B) A > B ? A : B
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))

int i = 10, j = 12, k;

k = MAX(i, j); printf("k = %i\n", k);
k = MAX(j, i) * 2; printf("k = %i\n", k);
k = MIN(i, j) * 3; printf("k = %i\n", k);
k = MIN(i--, j++); printf("i = %i\n", i);
```

```
k = 12
k = 12
k = 30
i = 8
```

## Preprocessor Macros

The preprocessor has a form of “advanced search and replace” mode in which it will replace parameters passed into macros. Macros should be used with care. The first assignment to “k” expands to:

```
k = i > j ? i : j;
```

This uses the conditional expression operator discussed earlier in the course. Since “i” is 10 and “j” is 12 “i>j” is false and so the third expression “j” is evaluated and assigned to “k”. All works as planned. The maximum value 12 is assigned to “k” as expected.

The second assignment expands to:

```
k = j > i ? j : i * 2;
```

Although “i” and “j” have been swapped, there should be little consequence. However, since “j>i” is true the second expression “j” is evaluated as opposed to “i \* 2”. The result is thus “j”, 12, and not the expected 24. Clearly an extra set of parentheses would have fixed this:

```
k = (j > i ? j : i) * 2;
```

The MIN macro with its excess of parentheses goes some way toward correcting this. The third assignment to “k” expands to:

```
k = ((i) < (j) ? (i) : (j)) * 3;
```

Now whichever value the conditional expression operator yields, “i” or “j”, will be multiplied by 3. Although the parentheses around “i” and “j” are unnecessary they make no difference to the calculation or the result. They do make a difference when the MIN macro is invoked as:

```
k = MIN(i + 3, j - 5);
```

The expansion:

```
k = ((i--) < (j++) ? (i--) : (j++))
```

causes “i” to be decremented and “j” to be incremented in the condition. 10 is tested against 12 (prefix increment and decrement) thus the condition is true. Evaluation of “i--” causes “i” to be decremented a second time. Thus “i” ends up at 8, not at the 9 the code might have encouraged us to expect.

---

## A Debugging Aid

§ Several extra features make the preprocessor an indispensable debugging tool

```
#define GOT_HERE printf("reached %i in %s\n", \
 __LINE__, __FILE__)

#define SHOW(E, FMT) printf("#E " = " FMT "\n", E)
```

```
printf("reached %i in %s\n", 17, "mysource.c");

GOT_HERE;
SHOW(i, "%x");
SHOW(f/29.5, "%lf");

printf("i = %x\n", i);
printf("f/29.5 = %lf\n", f/29.5);
```

## A Debugging Aid

The preprocessor provides many valuable debugging tools. The preprocessor constant `__LINE__` stores the current line number in the .c or .h file as an integer. The constant `__FILE__` stores the name of the current .c or .h file as a string.

The definition of `GOT_HERE` shows that preprocessor macros must be declared on one line, if more than one line is required, the lines must be glued together with `\`.

Although none of macros look particularly useful, their definition could be as follows:

```
#if defined(WANT_DEBUG)
#define GOT_HERE printf("reached %i in %s\n", __LINE__, __FILE__)
#define SHOW(E, FMT) printf("#E " = " FMT "\n", E)
#else
#define GOT_HERE
#define SHOW(E, FMT)
#endif
```

Adding the line:

```
#define WANT_DEBUG
```

above `#if defined` would **enable** all the invocations of `GOT_HERE` and `SHOW`, whereas removing this line would cause all invocations to be **disabled**.

There are two features of the `SHOW` macro worth mentioning. The first is that `#E` causes the expression to be turned into a string (a double quote is placed before the expression and another after it). The strings are then concatenated, thus:

```
SHOW(x, "%i")
```

becomes:

```
printf("x " = " "%i" "\n", x);
```

which then becomes:

```
printf("x = %i\n", x);
```

## Working With Large Projects

- § Large projects may potentially involve many hundreds of source files (*modules*)
- § Global variables and functions in one module may be accessed in other modules
- § Global variables and functions may be specifically hidden inside a module
- § Maintaining consistency between files can be a problem

---

## Working With Large Projects

All the programs examined thus far have been in a single .c file. Clearly large projects will involve many thousands of lines of code. It is impractical for a number of reasons to place all this code in one file. Firstly it would take weeks to load into an editor, secondly it would take months to compile. Most importantly only one person could work on the source code at one time.

If the source code could be divided between different files, each could be edited and compiled separately (and reasonably quickly). Different people could work on each source file.

One problem with splitting up source code is how to put it back together. Functions and global variables may be shared between the different .c files in a project. If desired, the functions in one .c may be hidden inside that file. Also variables declared globally within a .c may be hidden within that file.

---

## Data Sharing Example



```
extern float step;

void print_table(double, float);

int main(void)
{
 step = 0.15F;

 print_table(0.0, 5.5F);

 return 0;
}
```

```
#include <stdio.h>

float step;

void print_table(double start, float stop)
{
 printf("Celsius\tFahrenheit\n");
 for(;start < stop; start += step)
 printf("%.11f\t%.11f\n", start,
 start * 1.8 + 32);
}
```

## Data Sharing Example

These two modules share a variable “step” and a function `print_table`. Sharing the variable “step” is possible because the variable is declared globally within the second module. Sharing the function `print_table` is possible because the function is prototyped within the first module and declared “globally” within the second module.

### Functions are Global and Sharable

It is perhaps strange to think of functions as being global variables. Although functions are not variables (since they cannot be assigned to or otherwise altered) they are definitely global. It is possible to say:

```
extern void print_table(double, float);
```

although “extern” is implied by the prototype.

## Data Sharing Example



```
extern float step;

void print_table(double, float);

int main(void)
{
 step = 0.15F;

 print_table(0.0, 5.5F);

 return 0;
}
```

```
#include <stdio.h>

float step;

void print_table(double start, float stop)
{
 printf("Celsius\tFahrenheit\n");
 for(;start < stop; start += step)
 printf("%.11f\t%.11f\n", start,
 start * 1.8 + 32);
}
```

## Data Hiding Example

### static Before Globals

Placing the **static** keyword before a global variable or function locks that variable or function inside the .c file which declares it. The variables "entries" and "current" are hidden inside the module, the function **print** is also locked away.

### Errors at Link Time

Although there is no error when **compiling** the second module containing the statements:

```
void print(void);
extern int entries[];
```

and

```
entries[3] = 77;
print();
```

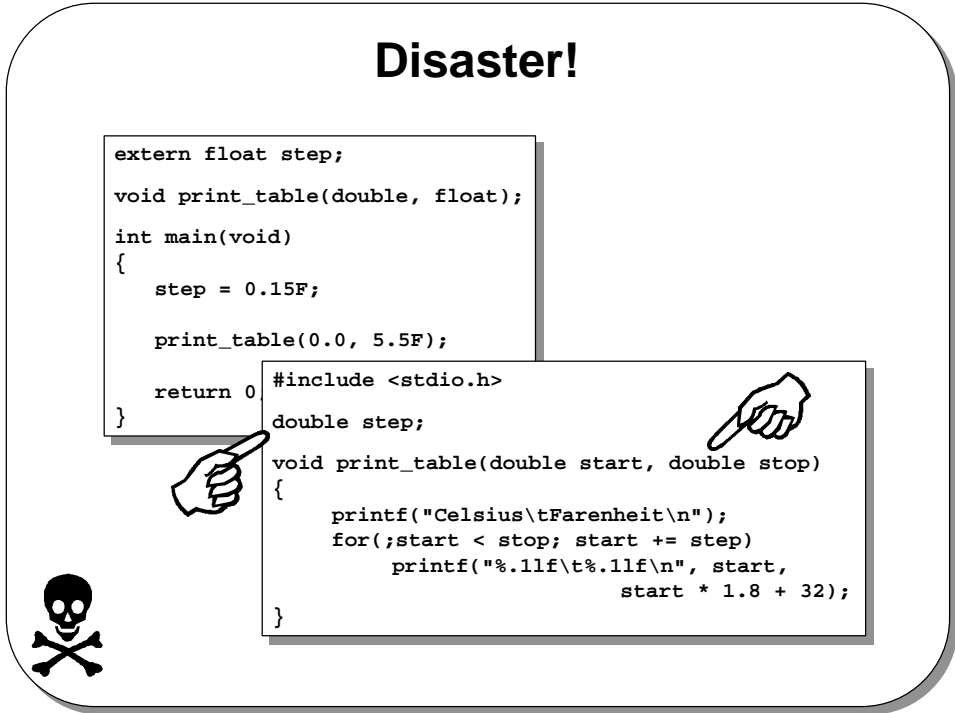
There are two errors when the program is **linked**. The errors are:

```
undefined symbol: entries
undefined symbol: print
```

The linker cannot find the symbols "entries" and "print" because they are hidden within the first module.



## Disaster!



```
extern float step;

void print_table(double, float);

int main(void)
{
 step = 0.15F;

 print_table(0.0, 5.5F);

 return 0;
}
```

```
#include <stdio.h>

double step;

void print_table(double start, double stop)
{
 printf("Celsius\tFahrenheit\n");
 for(;start < stop; start += step)
 printf("%.11f\t%.11f\n", start,
 start * 1.8 + 32);
}
```

## Disaster!

### Inconsistencies Between Modules

A few minor changes and the program no longer works. This will easily happen if two people are working on the same source code. The second module now declares the variable "step" as **double** and the second parameter "stop" as **double**.

The first module expects "step" to be **float**, i.e. a 4 byte IEEE format variable. Since "step" is actually declared as **double** it occupies 8 bytes. The first module will place 0.15 into 4 bytes of the 8 byte variable (the remaining 4 bytes having been initialized to 0 because "step" is global). The resulting value in "step" will not be 0.15.

A similar thing happens to the 5.5 assigned to "stop". It is written onto the stack as a 4 byte IEEE **float**, picked up as an 8 byte **double**.

Neither the compiler nor linker can detect these errors. The only information available to the linker are the symbol names "step" and **print\_table**. Neither of these names hold any type information.

## Use Header Files

- § **Maintain consistency between modules by using header files**
- § **NEVER** place an extern declaration in a module
- § **NEVER** place a prototype of a non static (i.e. sharable) function in a module

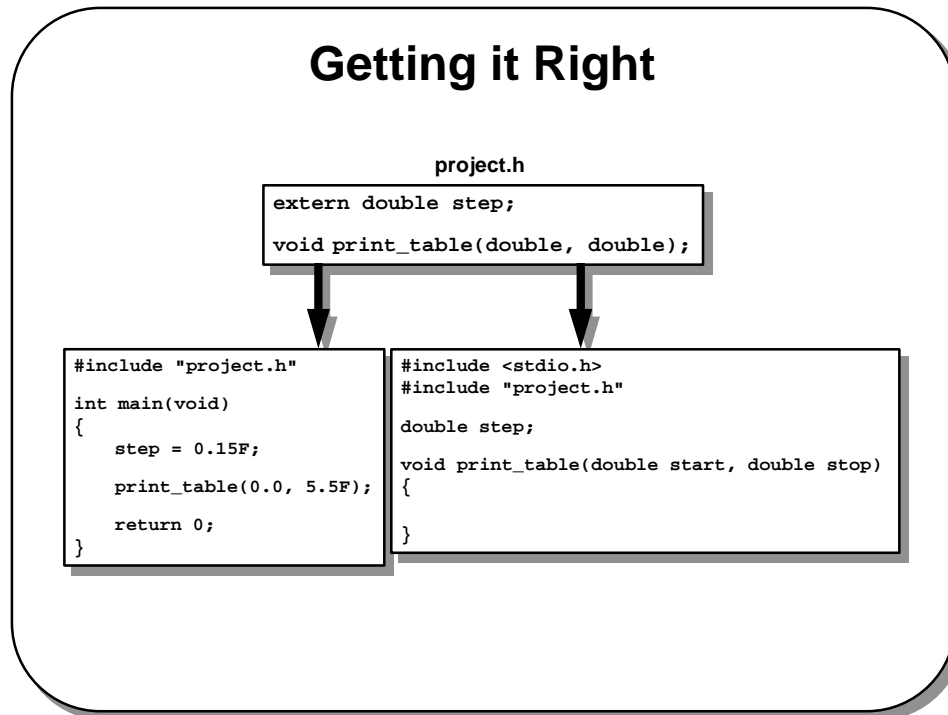
---

## Use Header Files

Although the problem of maintaining consistency may seem overwhelming, the solution is actually very simple. The preprocessor can help cross-check the contents of different modules.

An **extern** declaration should not be placed in a module, it should always be placed in a header file. Similarly function prototypes should not be placed in modules (unless **static** in which case they cannot be used anywhere but in the current module).

---



## Getting it Right

### Place Externs in the Header

By placing the **extern** declaration and the **print\_table** function prototype in the header file "project.h" the compiler can cross check the correctness of the two modules. In the first module the compiler sees:

```
extern double step;
void print_table(double, double);
```

The assignment:

```
step = 0.15F;
```

the compiler knows the type of **step** is **double**. The 4 byte **float** specified with 0.15F is automatically promoted to **double**.

When the **print\_table** function is called:

```
print_table(0.0, 5.5F);
```

the second parameter 5.5F is automatically promoted from **float** to **double**.

It may appear as though the second module would no longer compile, because:

```
extern double step;
```

is followed by:

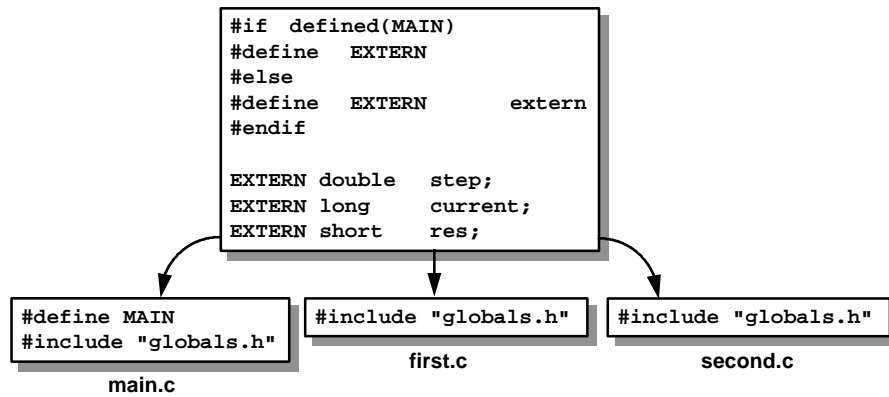
```
double step;
```

and these would appear to contradict. However, the compiler accepts these two statements providing the type **double** agrees. If either type is changed (as was the case) the compiler will produce an error message.

The compiler completes its usual cross-checking of prototype vs function header. If there is an inconsistency the compiler would report it.

## Be as Lazy as Possible

§ Get the preprocessor to declare the variables too!



## Be as Lazy as Possible

Within the module "main.c" the `#define` of `MAIN` causes `EXTERN` to be defined as nothing. Here the preprocessor performs a search and delete (as opposed to search and replace). The effect of deleting `EXTERN` means that:

```
double step;
long current;
short res;
```

results. This causes compiler to declare, and thus allocate storage for, the three variables. With the module "first.c" because `MAIN` is not defined the symbol `EXTERN` is defined as `extern`. With the preprocessor in search and replace mode the lines from `globals.h` become:

```
extern double step;
extern long current;
extern short res;
```

The only problem with this strategy is that all the globals are initialized with the same value, zero. It is not possible within `globals.h` to write:

```
EXTERN double step = 0.15;
EXTERN long current = 13;
EXTERN short res = -1;
```

because within `first.c` and `second.c` this produces the erroneous:

```
extern double step = 0.15;
extern long current = 13;
extern short res = -1;
```

An `extern` statement may not initialize the variable it declares.

## Summary

- § **A `union` may store values of different types at different times**
- § **`enum` provides an automated way of setting up constants**
- § **The preprocessor allows constants and macros to be created**
- § **Data and functions may be shared between modules**
- § **`static` stops sharing of data and functions**
- § **Use the preprocessor in large, multi module projects**

---

## Summary

---



---

---

## Miscellaneous Things Practical Exercises

---

---

Directory:     **MISC**

1. The chapter briefly outlined a possible implementation of the stack functions **push** and **pop** when discussing data and function hiding with the **static** keyword.

Open "**TEST.C**" which contains a test harness for the functions:

```
void push(int i);
int pop(void);
```

This menu driven program allows integers to be pushed and popped from a stack. Thus if 10, 20 and 30 were pushed, the first number popped would be 30, the second popped would be 20 and the last popped would be 10.

Implement these functions in the file "**STACK.C**". The prototypes for these functions are held in the header "**STACK.H**"

You should include code to check if too many values have been pushed (important since the values are stored in an array) and to see if the user attempts to pop more values than have been pushed.

---



---

---

## Miscellaneous Things Solutions

---

---

1. Open “**TEST.C**” which contains a test harness for the functions:

```
void push(int i);
int pop(void);
```

Implement these functions in the file “**STACK.C**”.

*The variable “current” and the array “the\_stack” must be shared by both push and pop. The only way to do this is to make it global, however in order for these variables not to be seen outside the module the static keyword is used.*

```
#include <stdio.h>
#include "stack.h"

#define MAX_STACK 50

static int the_stack[MAX_STACK];
static int current;

void push(int v)
{
 if(current >= MAX_STACK) {
 printf("cannot push: stack is full\n");
 return;
 }

 the_stack[current++] = v;
}

int pop(void)
{
 if(current == 0) {
 printf("cannot pop: stack is empty\n");
 return -1;
 }

 return the_stack[--current];
}
```

---

---

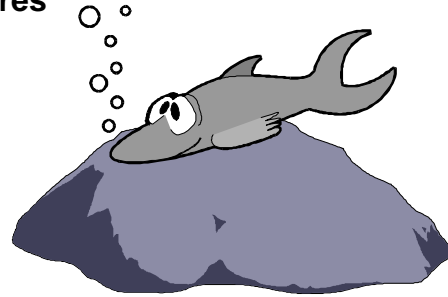
## C and the Heap

---

---

## C and the Heap

- ⌘ **What is the Heap?**
- ⌘ **Dynamic arrays**
- ⌘ **The `calloc/malloc/realloc` and `free` routines**
- ⌘ **Dynamic arrays of arrays**
- ⌘ **Dynamic data structures**



---

## C and the Heap

This chapter shows how to store data in the heap, retrieve it, enlarge it, reduce it and release it.

---

## What is the Heap?

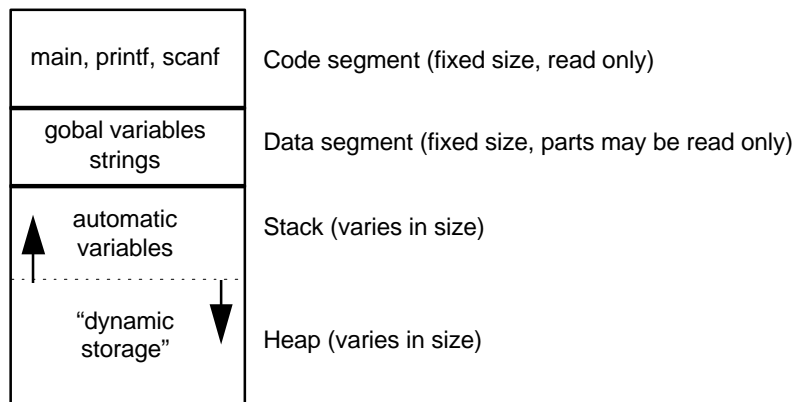
- § An executing program is divided into four parts:
- § **Stack:** provides storage for local variables, alters size as the program executes
- § **Data segment:** global variables and strings stored here. Fixed size.
- § **Code segment:** functions `main`, `printf`, `scanf` etc. stored here. Read only. Fixed size
- § **Heap:** otherwise known as “dynamic memory” the heap is available for us to use and may alter size as the program executes

---

## What is the Heap?

### The Parts of an Executing Program

A program executing in memory may be represented by the following diagram:



---

## What is the Heap? (Continued)

**Stack**                      The code and data segments are fixed size throughout the program lifetime. The stack:

1. increases in size as functions are called, parameters are pushed, local variables are created,
2. decreases in size as functions return, local variables are destroyed, parameters are popped

**Heap and Stack  
“in Opposition”**        The heap is placed in “opposition” to the stack, so that as stack usage increases (through deeply nested function calls, through the creation of large local arrays, etc.) the amount of available heap space is reduced. Similarly as heap usage increases, so available stack space is reduced.

The line between heap and stack is rather like the line between the shore and the sea. When the tide is in, there is a lot of sea and not much shore. When the tide is out there is a lot of shore and not much sea.

---

## How Much Memory?

- ⌘ With simple operating systems like MS-DOS there may only be around 64k available (depending on memory model and extended memory device drivers)
- ⌘ With complex operating systems using virtual memory like Unix, NT, OS/2, etc. it can be much larger, e.g. 2GB
- ⌘ In the future (or now with NT on the DEC Alpha) this will be a very large amount (17 thousand million GB)

---

## How Much Memory?

### Simple Operating Systems

The heap provides “dynamic memory”, but how much? With simple operating systems like MS-DOS the header in the executable file contains a number indicating the total amount of memory the program requires. This is as much memory as the program will ever get. The code and data segments are loaded in, what remains is left to be divided between heap and stack. When the stack runs into the heap the program is killed. No second chance.

### Advanced Operating Systems

With more advanced operating systems, a program is loaded into a hole in memory and left to execute. If it turns out the hole wasn't large enough (because the stack and heap collide), the operating system finds a larger hole in memory. It copies the code and data segments into the new area. The stack and heap are moved as far apart as possible within this new hole. The program is left to execute. If the stack and heap collide again the program is copied into an even larger hole and so on. There is a limit to how many times this can happen, dependent upon the amount of physical memory in the machine. If virtual memory is in use it will be the amount of virtual memory the machine may access. With “32 bit” operating systems like Unix, NT, Windows 95 etc. The limit is usually somewhere around  $2^{32}$  bytes, or 2GB. You probably won't get exactly this amount of memory, since some of the operating system must remain resident in memory, along with a few dozen megabytes of important data structures.

### Future Operating Systems

With “64 bit” operating systems, like NT running on the DEC Alpha processor, the limit is around  $2^{64}$  bytes. This is a rather large number of GB and should really be quoted in TB (terra bytes). Although most people have a “feeling” for how large one Gigabyte is, few people yet have experience of how large a Terabyte is. The ultimate limit of a program's size will be the amount of disk space. The virtual memory used by an operating system must be saved somewhere. Chances are the machine does not contain the odd billion bytes of memory, so the next best storage medium is the hard disk. The smaller the disk, the smaller the amount of the program which may be saved.

---

## Dynamic Arrays

- § Arrays in C have a fundamental problem - their size must be fixed when the program is written
- § There is no way to increase (or decrease) the size of an array once the program is compiled
- § Dynamic arrays are different, their size is fixed at run time and may be changed as often as required
- § Only a pointer is required

---

## Dynamic Arrays

Arrays in C are rather primitive data structures. No mechanism exists in the language to change the size of an array once it has been declared (like the “redim” command from BASIC). All is not lost, however. The storage for an array may be allocated on the heap. This storage must be physically contiguous (the only requirement for an array), but the routines that manage the heap guarantee this.

All the program requires is a pointer which will contain the address at which the block of memory starts.

An example. An array of 100 long integers is declared like this:

```
long a[100];
```

and is fixed in size forever (at 400 bytes). An attempt to make it larger, like:

```
long a[200];
```

will cause an error because the compiler will see the variable “a” being declared twice. If we do the following:

```
long *p;

p = malloc(100 * sizeof(long));
```

we end up with same amount of memory, 400 bytes, but stored on the heap instead of the stack or data segment. An element of the array “a” may be accessed with `a[58]`, an element of the array pointed to by “p” may be accessed with `p[58]`. The array may be made larger with:

```
long *q;

q = realloc(p, 200 * sizeof(long));
```

which increases the block of storage to hold 200 `long ints`, i.e. 800 bytes.

---



## Using Dynamic Arrays

- § The following steps create a dynamic array:
  - Declare a pointer corresponding to the desired type of the array elements
  - Initialise the pointer via `calloc` or `malloc` using the total storage required for all the elements of the array
  - Check the pointer against `NULL`
  - Increase or decrease the number of elements by calling the `realloc` function
  - Release the storage by calling `free`

---

## Using Dynamic Arrays

### One Pointer per Dynamic Array

One pointer is required for each dynamic array which is to be stored on the heap. The type of this pointer is dictated by the type of the array elements. Thus if an array of `doubles`, an array of `short` integers and an array of `Book` structures are required, three pointers would be needed:

```
double *double_array;
short *short_array;
struct Book *book_array;
```

### Calculating the Storage Requirement

The second step is to calculate how much memory will be required for each array. If 100 `doubles`, 480 `short ints` and 238 books are required:

```
double_array = malloc(100 * sizeof(double));
short_array = calloc(480, sizeof(short));
book_array = calloc(238, sizeof(struct Book));
```

There is little to choose between `malloc` and `calloc`, however the 100 `doubles` pointed to by “`double_array`” are entirely random, whereas the 480 `short ints` and the 238 books pointed to by “`short_array`” and “`book_array`” respectively are all zero (i.e. each element of the name, author and ISBN number of each book contain the null terminator, the price of each book is zero).

---

---

## Using Dynamic Arrays (continued)

### Insufficient Storage

Just calling `malloc` or `calloc` does not guarantee the memory to store the elements. The call may fail if we have already allocated a large amount of memory and there is none left (which will happen sooner rather than later under MS-DOS). The routines indicate the limit has been reached by returning the NULL pointer. Thus each of the pointers “double\_array”, “short\_array” and “book\_array” must be checked against NULL.

### Changing the Array Size

The amount of memory on the end of any one of these pointers may be changed by calling `realloc`. Imagine that 10 of the `doubles` are not required and an extra 38 books are needed:

```
da = realloc(double_array, 90 * sizeof(double));
ba = realloc(book_array, 276 * sizeof(struct Book));
```

Where “da” is of type pointer to double and “ba” is of type pointer to Book structure. Note that it is inadvisable to say:

```
book_array = realloc(book_array, 276 *
 sizeof(struct Book));
```

Since it is possible that the allocation may fail, i.e. it is not possible to find a contiguous area of dynamic memory of the required size. If this does happen, i.e. there is no more memory, `realloc` returns NULL. The NULL would be assigned to “book\_array” and the address of the 238 books is lost. Assigning to “ba” instead guarantees that “book\_array” is unchanged.

### When `realloc` Succeeds

If the allocation does **not** fail, “ba” is set to a pointer other than NULL. There are two scenarios here:

1. `realloc` was able to enlarge the current block of memory in which case the address in “ba” is exactly the same as the address in “book\_array”. Our 238 books are intact and the 38 new ones follow on after and are random, or
2. `realloc` was unable to enlarge the current block of memory and had to find an entirely new block. The address in “ba” and the address in “book\_array” are completely different. Our original 238 books have been copied to the new block of memory. The 38 new ones follow on after the copied books and are random.

We do not need to be concerned which of these two scenarios took place. As far as we are concerned the pointer “ba” points to a block of memory able to contain 276 books and specifically points to the value of the first book we written before the `realloc`.

---

---

## Using Dynamic Arrays (continued)

### Maintain as Few Pointers as Possible

One consequence of the second scenario is that all other pointers into the array of books are now invalid. For example, if we had a special pointer:

```
struct Book *war_and_peace;
```

which was initialized with:

```
war_and_peace = book_array + 115;
```

this pointer would now be invalid because the whole array would have been moved to a new location in memory. We must NOT use the pointer “book\_array”, or the pointer “war\_and\_peace”. Both must be “recalculated” as follows:

```
book_array = ba;
war_and_peace = ba + 115;
```

In fact it would probably be more convenient to remember “war\_and\_peace” as an offset from the start of the array (i.e. 115). In this way it wouldn’t have to be “recalculated” every time `realloc` was called, just added to the *single* pointer “book\_array”.

### Requests Potentially Ignored

As an aside, it is possible that the request:

```
da = realloc(double_array, 90 * sizeof(double));
```

might be completely ignored. Finding a new block in memory only slightly smaller than the existing block might be so time consuming that it would be easier just to return a pointer to the existing block and change nothing. The entire block would be guaranteed to be reclaimed when `free` was called.

### Releasing the Storage

Finally when all books, `short ints` and `doubles` have been manipulated, the storage must be released.

```
free(double_array);
free(book_array);
free(short_array);
```

Strictly speaking this doesn’t need to happen since the heap is part of the process. When the process terminates all memory associated with it will be reclaimed by the operating system. However, it is good practice to release memory in case program is altered so that a “one off” routine is called repeatedly.

Repeatedly calling a routine which fails to deallocate memory would guarantee that the program would eventually fail, even if the amount of memory concerned was small. Such errors are known as “memory leaks”.

---

## calloc/malloc Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 unsigned i, s;
 double *p;

 printf("How many doubles? ");
 scanf("%u", &s);

 if((p = calloc(s, sizeof(double))) == NULL) {
 fprintf(stderr, "Cannot allocate %u bytes "
 "for %u doubles\n", s * sizeof(double), s);
 return 1;
 }
 for(i = 0; i < s; i++)
 p[i] = i;
 free(p);
 return 0;
}
```

here we access the "s" doubles from 0..s-1

all of the allocated memory is freed

```
if((p = malloc(s * sizeof(double))) == NULL) {
```

## calloc/malloc Example

The previous page of notes mentioned rather fixed numbers, "238 books", "276 books", "90 doubles". If these numbers could be reliably predicted at compile time, "ordinary" fixed sized C arrays could be used.

The program above shows how, with simple modification, the program can start to manipulate numbers of **doubles** which cannot be predicted at compile time. There is no way of predicting at compile time what value the user will type when prompted. Note the use of **unsigned** integers in an attempt to prevent the user from entering a negative number. In fact **scanf** is not too bright here and changes any negative number entered into a large positive one.

Notice the straightforward way C allows us to access the elements of the array:

```
p[i] = i;
```

is all it takes. The pointer "p" points to the start of the array, "i" serves as an offset to access a particular element. The **\*(p+i)** notation, although practically identical, would not be as readable.

## realloc Example

```
double *p;
double *p2;

if((p = calloc(s, sizeof(double))) == NULL) {
 fprintf(stderr, "Cannot allocate %u bytes "
 "for %u doubles\n", s * sizeof(double), s);
 return 1;
}

printf("%u doubles currently, how many now? ", s);
scanf("%u", &s);

p2 = realloc(p, s * sizeof(double)); ← calculate new array
 size and allocate
 storage
if(p2 == NULL) {
 fprintf(stderr, "Could not increase/decrease array "
 "to contain %u doubles\n", s);
 free(p); ← pointer "p" is still
 return 1; valid at this point
}
p = p2; ← pointer "p" is invalid at this point, so
free(p); a new value is assigned to it
```

## realloc Example

The program shows the use of realloc. As previously discussed the assignment:

```
p2 = realloc(p, s * sizeof(double));
```

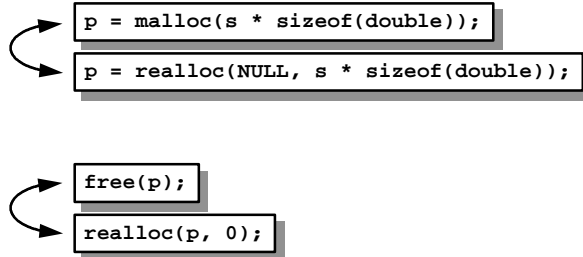
is more helpful than:

```
p = realloc(p, s * sizeof(double));
```

because if the re-allocation fails, assigning back to "p" will cause the existing array of **doubles** to be lost. At least if the block cannot be enlarged the program could continue processing the data it had.

## realloc can do it all

- § The routines `malloc` and `free` are almost redundant since `realloc` can do it all
- § There is some merit in `calloc` since the memory it allocates is cleared to zero



## realloc can do it all

With the right parameters, `realloc` can take the place of `malloc` and `free`. It can't quite take the place of `calloc`, since although it can allocate memory it does not clear it to zeros.

### realloc can Replace malloc

A `NULL` pointer passed in as a first parameter causes `realloc` to behave just like `malloc`. Here it realizes it is not enlarging an existing piece of memory (because there is no existing piece of memory) and just allocates a new piece.

### realloc can Replace free

A size of zero passed in as the second parameter causes `realloc` to deallocate an existing piece of memory. This is consistent with setting its allocated size to zero.

There is a case to be made for clarity. When seeing `malloc`, it is obvious a memory allocation is being made. When seeing `free`, it is obvious a deallocation is being made. Use of the `realloc` function tends to imply an alteration in size of a block of memory.

## Allocating Arrays of Arrays

§ Care must be taken over the type of the pointer used when dealing with arrays of arrays

```
float *p;
p = calloc(s, sizeof(float));
```



```
float **rain;
rain = calloc(s, 365 * sizeof(float));
```



```
float (*rainfall)[365];
rainfall = calloc(s, 365 * sizeof(float));
rainfall[s-1][18] = 4.3F;
```



## Allocating Arrays of Arrays

Thus far we have seen how to allocate an array. This is simply done by allocating the address of a block of memory to a pointer. It might seem logical that if an array is handled this way, an array of arrays may be handled by assigning to a pointer to a pointer. This is not the case.

**Pointers Access  
Fine with  
Dynamic Arrays**

In the example above an array is allocated with:

```
float *p;

p = calloc(s, sizeof(float));
```

The elements of the array are accessed with, for example, `p[3]`, which would access the fourth element of the array (providing “s” were greater than or equal to 4).

At the end of the Arrays In C chapter there was a “rainfall” example where an arrays of arrays were used. The rainfall for 12 locations around the country was to be recorded for each of the 365 days per year. The declaration:

```
float rainfall[12][365];
```

was used.

Say now that one of the “rainfall” arrays must be allocated dynamically with the number of countrywide locations being chosen at run time. Clearly for each location, 365 `float`s will be needed. For 10 locations an array able to contain 3650 `float`s would be needed.

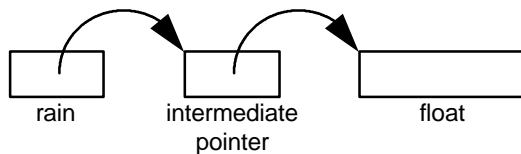
## Allocating Arrays of Arrays (continued)

Pointers to  
Pointers are not  
Good with  
Arrays of Arrays

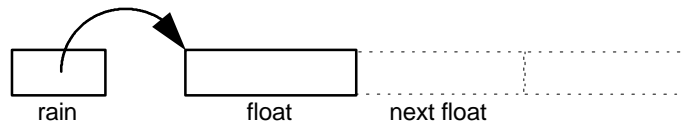
The way to do this would SEEM to be:

```
float **rain;
rain = calloc(s, 365 * sizeof(float));
```

(where “s” presumably contains the 10). However, there is not enough “information” in the pointer “rain” to move correctly. Consider, for example, accessing element `rain[2][100]`. The 2 is required to jump into the third block of 356 `float`s, i.e. over 2 entire blocks of 356 `float` ( $2 * 365 * 4 = 2920$  bytes) and then on by another  $100 * 4 = 400$  bytes. That’s a total move of 3320 bytes. However the compiler *cannot* determine this from the pointer. “rain” could be drawn as:



However, the memory has been allocated as:



Where the `float` pointed to is followed by several thousand others. Any attempt to use the pointer “rain” will cause the compiler to interpret the first float as the intermediate pointer drawn above. Clearly it is incorrect to interpret an IEEE value as an address.

Use Pointers to  
Arrays

The solution is to declare the pointer as:

```
float (*rainfall)[365];
```

(“rainfall” is a pointer to an array of 365 float). The compiler knows that with access of `rainfall[2][100]` the 2 must be multiplied by the size of 365 `float`s (because if “rainfall” is a pointer to an array of 365 `float`, 2 must step over two of these arrays). It also knows the 100 must be scaled by the size of a `float`. The compiler may thus calculate the correct movement.



## Dynamic Data Structures

§ It is possible to allocate structures in dynamic memory too

```
struct Node {
 int data;
 struct Node *next_in_line;
};

struct Node* new_node(int value)
{
 struct Node* p;

 if((p = malloc(sizeof(struct Node))) == NULL) {
 fprintf(stderr, "ran out of dynamic memory\n");
 exit(9);
 }
 p->data = value; p->next_in_line = NULL;
 return p;
}
```

---

## Dynamic Data Structures

It is not only arrays that may be allocated in dynamic memory, structures can be allocated too. This is ideal with “linked” data structures like linked lists, trees, directed graphs etc. where the number of nodes required cannot be predicted at compile time. The example above shows a routine which, when called, will allocate storage for a single node and return a pointer to it. Because of the “next\_in\_line” member, such nodes may be chained together.

Notice that the integer value to be placed in the node is passed in as a parameter. Also the routine carefully initializes the “next\_in\_line” member as NULL, this is important since by using **malloc**, the pointer “p” points to memory containing random values. The value in the “data” member will be random, as will the address in the “next\_in\_line” member. If such a node were chained into the list without these values being changed, disaster could result. This way, if this node is used, the presence of the NULL in the “next\_in\_line” member will not cause us to wander into random memory when walking down the list.

---

## Linking the List

```
struct Node *first_node, *second_node, *third_node, *current;

first_node = new_node(-100);

second_node = new_node(0);

first_node->next_in_line = second_node;

third_node = new_node(10);

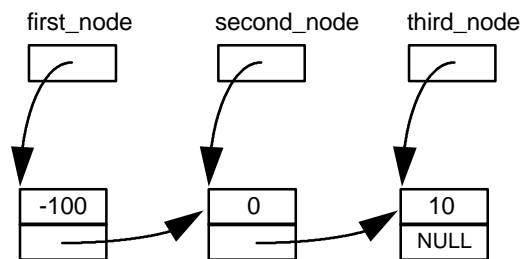
second_node->next_in_line = third_node;

current = first_node;
while(current != NULL) {
 printf("%i\n", current->data);
 current = current->next_in_line;
}
```

---

## Linking the List

Above is a simple example of how a linked list could be built. In reality, rather than wiring each node to point to the next, a function would be written to find the insertion point and wire up the relevant "next\_in\_line" members. The chain resulting from the above would be:



## Summary

- § The heap and stack grow towards one another
- § Potentially a large amount of heap storage is available given the right operating system
- § The routines `malloc`, `calloc`, `realloc` and `free` manipulate heap storage
- § Only `realloc` is really necessary
- § Allocating dynamic arrays
- § Allocating dynamic arrays of arrays
- § Allocating dynamic structures

---

## Summary

---



---

---

## C and the Heap Practical Exercises

---

---

**Directory:**      **HEAP**

1. Write a program "**MAX**" which allocates all available heap memory. The way to do this is to write a loop which allocates a block of memory, say 10 bytes, continually until malloc returns NULL. When this happens, print out the total number of bytes allocated.
2. Alter your "**MAX**" program such that the block size (which above was 10 bytes) is read from the command line (the function **atoi** will convert a string to an integer and return zero if the string is not in the correct format). Use your program to find out if the total amount of memory that can be allocated differs for 10 byte, 100 byte, 1000 byte and 5000 byte blocks. What issues would influence your results?
3. The program "**BINGEN.C**" is a reworking of an earlier **FILES** exercise. It reads a text file and writes structures to a binary file (the structure is defined in "**ELE.H**"). Compile and run the program, taking "**ELE.TXT**" as input and creating the binary file "**ELE.BIN**".

Write a program "**ELSHOW**" which opens the binary file "**ELE.BIN**". By moving to the end of the file with **fseek** and finding how many bytes there are in the file with **ftell**, it is possible to find out how many records are in the file by dividing by the total bytes by the size of an Element structure.

Allocate memory sufficient to hold all the structures. Reset the reading position back to the start of the file and read the structures using **fread**. Write a loop to read an integer which will be used to index into the array and print out the particular element chosen. Exit the loop when the user enters a negative number.

---

---

---

## C and the Heap Solutions

---

---

1. Write a program “MAX” which allocates all available heap memory.
2. Alter your “MAX” program such that the block size is read from the command line

*The printf within the malloc loop is advisable because otherwise the program appears to “hang”. Outputting “\n” ensures that pages of output are not produced. Each number neatly overwrites the previous one.*

```
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_BLOCK_SIZE 10

int main(int argc, char* argv[])
{
 unsigned long total = 0;
 unsigned block = DEFAULT_BLOCK_SIZE;

 if(argc > 1) {
 block = atoi(argv[1]);
 if(block == 0)
 block = DEFAULT_BLOCK_SIZE;
 }

 while(malloc(block) != NULL) {
 printf("\r%lu", total);
 total += block;
 }

 printf("\rblock size %u gives total %lu bytes allocated\n",
 block, total);

 return 0;
}
```

*The issues regarding block sizes vs total memory allocated are that each allocation carries an overhead. If a large block size is used, the ratio of this overhead to the block is small and so many allocations may be done. If a small block size is used (perhaps 2 bytes) the ratio of overhead to block is very large. Available memory is filled with control information rather than data. Making the block size too large means that the last allocation fails because it cannot be completely satisfied.*

3. Compile and run “BINGEN.C” taking “ELE.TXT” as input and creating the binary file “ELE.BIN”. Write a program “ELSHOW” which opens the binary file “ELE.BIN”. Allocate memory sufficient to hold all the structures and read the structures using `fread`. Write a loop to read an integer which will be used to index into the array and print out the particular element chosen. Exit the loop when the user enters a negative number.

*Displaying an element structure must be done carefully. This is because the two character array “name” is not necessarily null terminated (two characters plus a null won’t fit into a two character array). Always printing two characters would be incorrect when an element with a single character name (like Nitrogen, “N” for instance) were met.*



```
#include <stdio.h>
#include <stdlib.h>
#include "ele.h"

int get_int(void);
int show(char*);
void display(struct Element * p);
struct Element* processFile(FILE* in, unsigned * ptotal);

int main(int argc, char* argv[])
{
 char* in;
 char in_name[100+1];

 if(argc == 1) {
 printf("File to show ");
 scanf("%100s", in_name);
 getchar();
 in = in_name;
 } else
 in = argv[1];

 return show(in);
}

int show(char* in)
{
 int which;
 unsigned total;
 FILE* in_stream;
 struct Element* elems;

 if((in_stream = fopen(in, "rb")) == NULL) {
 fprintf(stderr, "Cannot open input file %s, ", in);
 perror("");
 return 1;
 }
 if((elems = processFile(in_stream, &total)) == NULL)
 return 1;

 fclose(in_stream);

 while((which = get_int()) >= 0)
 if(which >= total || which == 0)
 printf("%i is out of range (min 1, max %i)\n",
 which, total);
 else
 display(&elems[which - 1]);

 return 0;
}
```

---

```
struct Element* processFile(FILE* in, unsigned * ptotal)
{
 unsigned long total_size;
 unsigned int elements;
 unsigned int el_read;
 struct Element* p;

 fseek(in, 0L, SEEK_END);
 total_size = ftell(in);
 fseek(in, 0L, SEEK_SET);

 elements = total_size / sizeof(struct Element);

 p = calloc(elements, sizeof(struct Element));

 el_read = fread(p, sizeof(struct Element), elements, in);

 if(el_read != elements) {
 fprintf(stderr, "Failed to read %u elements (only read %u)\n",
 elements, el_read);
 free(p);
 return NULL;
 }
 *ptotal = elements;

 return p;
}

int get_int(void)
{
 int status;
 int result;

 do {
 printf("enter an integer (negative will exit) ");
 status = scanf("%i", &result);
 while(getchar() != '\n')
 ;
 } while(status != 1);

 return result;
}

void display(struct Element * p)
{
 printf("element %c", p->name[0]);
 if(p->name[1])
 printf("%c ", p->name[1]);
 else
 printf(" ");

 printf("rmm %6.2f melt %7.2f boil %7.2f\n",
 p->rmm, p->melt, p->boil);
}
```

---

---

---

Appendices

---

---

## Precedence and Associativity of C Operators:

|                        |                                                                |               |
|------------------------|----------------------------------------------------------------|---------------|
| primary                | <code>() [] -&gt; .</code>                                     | left to right |
| unary                  | <code>! ~ ++ -- - + (cast) * &amp; sizeof</code>               | right to left |
| multiplicative         | <code>* / %</code>                                             | left to right |
| additive               | <code>+ -</code>                                               | left to right |
| shift                  | <code>&lt;&lt; &gt;&gt;</code>                                 | left to right |
| relational             | <code>&lt; &lt;= &gt;= &gt;</code>                             | left to right |
| equality               | <code>== !=</code>                                             | left to right |
| bitwise and            | <code>&amp;</code>                                             | left to right |
| bitwise or             | <code> </code>                                                 | left to right |
| bitwise xor            | <code>^</code>                                                 | left to right |
| logical and            | <code>&amp;&amp;</code>                                        | left to right |
| logical or             | <code>  </code>                                                | left to right |
| conditional expression | <code>? :</code>                                               | right to left |
| assignment             | <code>= += -= *= /= %= &lt;&lt;= &gt;&gt;= &amp;=  = ^=</code> | right to left |
| sequence               | <code>,</code>                                                 | left to right |

### Notes:

1. The `()` operator in “primary” is the function call operator, i.e. `f(24, 37)`
2. The `*` operator in “unary” is the “pointer to” operator, i.e. `*pointer`
3. The `&` operator in “unary” is the “address of” operator, i.e. `pointer = &variable`
4. The `+` and `-` in “unary” are the unary counterparts of plus and minus, i.e. `x = +4` and `y = -x`
5. The `,` operator is that normally found in the `for` loop and guarantees sequential processing of statements, i.e. `for(i = 0, j = i; i < 10; i++, j++)` guarantees “`i = 0`” is executed before “`j = i`”. It also guarantees “`i++`” is executed before “`j++`”.

## Summary of C Data Types

|                |                                                                                                                                                              |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| char           | one byte character value, may be signed or unsigned                                                                                                          |
| signed char    | one byte signed character, ASCII characters will test positive, extended ASCII characters will test negative                                                 |
| unsigned char  | one byte unsigned character, all values test positive                                                                                                        |
| int            | integer value, i.e. whole number, no fraction                                                                                                                |
| short [int]    | integer value with potentially reduced range (may have only half the storage available as for an int)                                                        |
| long [int]     | integer value with potentially increased range (may have twice the storage available as for an int)                                                          |
| signed [int]   | as for int                                                                                                                                                   |
| unsigned [int] | an integer value which may contain positive values only. Largest value of an unsigned integer will be twice that of the largest positive value of an integer |
| signed short   | as for short                                                                                                                                                 |
| unsigned short | a positive integer value with potentially reduced range                                                                                                      |
| signed long    | as for long                                                                                                                                                  |
| unsigned long  | a positive integer value with potentially increased range                                                                                                    |
| float          | a floating point value (a number with a fraction)                                                                                                            |
| double         | a floating point value with potentially increased range and accuracy                                                                                         |
| long double    | a floating point value with potentially very great range and accuracy                                                                                        |
| void           | specifies the absence of a type                                                                                                                              |

C guarantees that:

$$\text{sizeof(char)} < \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$$

and

$$\text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}$$

## Maxima and Minima for C Types

| type           | usual size | minimum value | maximum value | defined in |
|----------------|------------|---------------|---------------|------------|
| char           | 1 byte     | CHAR_MIN      | CHAR_MAX      | limits.h   |
| signed char    | 1 byte     | SCHAR_MIN     | SCHAR_MAX     | limits.h   |
| unsigned char  | 1 byte     | -             | UCHAR_MAX     | limits.h   |
| short          | 2 bytes    | SHRT_MIN      | SHRT_MAX      | limits.h   |
| unsigned short | 2 bytes    | -             | USHRT_MAX     | limits.h   |
| int            | ?          | INT_MIN       | INT_MAX       | limits.h   |
| unsigned int   | ?          | -             | UINT_MAX      | limits.h   |
| long           | 4 bytes    | LONG_MIN      | LONG_MAX      | limits.h   |
| unsigned long  | 4 bytes    | -             | ULONG_MAX     | limits.h   |
| float          | 4 bytes    | FLT_MIN       | FLT_MAX       | float.h    |
| double         | 8 bytes    | DBL_MIN       | DBL_MAX       | float.h    |
| long double    | 10 bytes   | LDBL_MIN      | LDBL_MAX      | float.h    |

## Printf Format Specifiers

| type           | format specifier | decimal | octal | hexadecimal |
|----------------|------------------|---------|-------|-------------|
| char           | %c               | %d      | %o    | %x          |
| signed char    | %c               | %d      | %o    | %x          |
| unsigned char  | %c               | %u      | %o    | %x          |
| short          | %hi              | %hd     | %ho   | %hx         |
| unsigned short | %hu              |         | %ho   | %hx         |
| int            | %i               | %d      | %o    | %x          |
| unsigned int   | %u               |         | %o    | %x          |
| long           | %li              | %ld     | %lo   | %lx         |
| unsigned long  | %lu              |         | %lo   | %lx         |

| type        | format specifier | alternate | alternate |
|-------------|------------------|-----------|-----------|
| float       | %f               | %g        | %e        |
| double      | %lf              | %lg       | %le       |
| long double | %Lf              | %Lg       | %Le       |

| type    | format specifier |
|---------|------------------|
| pointer | %p               |

When characters are passed to `printf`, they are promoted to type `int`. Thus any format specifier used with `int` may also be used with `char`. The only difference is in the output format. Thus a `char` variable containing 97 will print 97 when `%d` or `%i` is used, but 'a' when `%c` is used.

When using floating point types, `%f` prints 6 decimal places in "standard" notation, `%e` prints six decimal places in exponential notation. `%g` chooses the most concise of `%f` and `%e` and uses that output format.

A pointer may be printed with `%p`, regardless of its type. The output format is machine specific, but usually hexadecimal. There is no clean way to output a pointer in decimal notation.

## Table of Escape Sequences

|     | What                      | ASCII    | Causes:                                                                                                                                                             |
|-----|---------------------------|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \a  | alert character           | 7        | the speaker to sound                                                                                                                                                |
| \b  | backspace                 | 8        | cursor to move one space backwards. Useful with printers with print heads for emboldening characters by printing them, backspacing and printing the character again |
| \t  | tab character             | 9        | cursor to move forward to next tab stop                                                                                                                             |
| \n  | newline character         | 10       | cursor to move to start of next line                                                                                                                                |
| \v  | vertical tab              | 11       | interesting effects with a small class of terminals and printers                                                                                                    |
| \f  | formfeed character        | 12       | a page feed when sent to a printer                                                                                                                                  |
| \r  | carriage return character | 13       | cursor to move to first column of screen                                                                                                                            |
| \0n |                           | <i>n</i> | <i>n</i> is interpreted as octal and the corresponding ASCII character is generated                                                                                 |
| \xn |                           | <i>n</i> | <i>n</i> is interpreted as hexadecimal and the corresponding ASCII character is generated                                                                           |



## Ascii Table

|          |     |    |      |             |    |          |     |     |      |            |
|----------|-----|----|------|-------------|----|----------|-----|-----|------|------------|
| 00000000 | 000 | 0  | 0x0  | nul         | \0 | 01000000 | 100 | 64  | 0x40 | @          |
| 00000001 | 001 | 1  | 0x1  | ^A          |    | 01000001 | 101 | 65  | 0x41 | A          |
| 00000010 | 002 | 2  | 0x2  | ^B          |    | 01000010 | 102 | 66  | 0x42 | B          |
| 00000011 | 003 | 3  | 0x3  | ^C          |    | 01000011 | 103 | 67  | 0x43 | C          |
| 00000100 | 004 | 4  | 0x4  | ^D          |    | 01000100 | 104 | 68  | 0x44 | D          |
| 00000101 | 005 | 5  | 0x5  | ^E          |    | 01000101 | 105 | 69  | 0x45 | E          |
| 00000110 | 006 | 6  | 0x6  | ^F          |    | 01000110 | 106 | 70  | 0x46 | F          |
| 00000111 | 007 | 7  | 0x7  | alert       | \a | 01000111 | 107 | 71  | 0x47 | G          |
| 00001000 | 010 | 8  | 0x8  | backspace   | \b | 01001000 | 110 | 72  | 0x48 | H          |
| 00001001 | 011 | 9  | 0x9  | tab         | \t | 01001001 | 111 | 73  | 0x49 | I          |
| 00001010 | 012 | 10 | 0xA  | newline     | \n | 01001010 | 112 | 74  | 0x4A | J          |
| 00001011 | 013 | 11 | 0xB  | vt. tab     | \v | 01001011 | 113 | 75  | 0x4B | K          |
| 00001100 | 014 | 12 | 0xC  | formfeed    | \f | 01001100 | 114 | 76  | 0x4C | L          |
| 00001101 | 015 | 13 | 0xD  | return      | \r | 01001101 | 115 | 77  | 0x4D | M          |
| 00001110 | 016 | 14 | 0xE  | ^N          |    | 01001110 | 116 | 78  | 0x4E | N          |
| 00001111 | 017 | 15 | 0xF  | ^O          |    | 01001111 | 117 | 79  | 0x4F | O          |
| 00010000 | 020 | 16 | 0x10 | ^P          |    | 01010000 | 120 | 80  | 0x50 | P          |
| 00010001 | 021 | 17 | 0x11 | ^Q          |    | 01010001 | 121 | 81  | 0x51 | Q          |
| 00010010 | 022 | 18 | 0x12 | ^R          |    | 01010010 | 122 | 82  | 0x52 | R          |
| 00010011 | 023 | 19 | 0x13 | ^S          |    | 01010011 | 123 | 83  | 0x53 | S          |
| 00010100 | 024 | 20 | 0x14 | ^T          |    | 01010100 | 124 | 84  | 0x54 | T          |
| 00010101 | 025 | 21 | 0x15 | ^U          |    | 01010101 | 125 | 85  | 0x55 | U          |
| 00010110 | 026 | 22 | 0x16 | ^V          |    | 01010110 | 126 | 86  | 0x56 | V          |
| 00010111 | 027 | 23 | 0x17 | ^W          |    | 01010111 | 127 | 87  | 0x57 | W          |
| 00011000 | 030 | 24 | 0x18 | ^X          |    | 01011000 | 130 | 88  | 0x58 | X          |
| 00011001 | 031 | 25 | 0x19 | ^Y          |    | 01011001 | 131 | 89  | 0x59 | Y          |
| 00011010 | 032 | 26 | 0x1A | ^Z          |    | 01011010 | 132 | 90  | 0x5A | Z          |
| 00011011 | 033 | 27 | 0x1B | esc         |    | 01011011 | 133 | 91  | 0x5B | [          |
| 00011100 | 034 | 28 | 0x1C | ^\<br>[     |    | 01011100 | 134 | 92  | 0x5C | \          |
| 00011101 | 035 | 29 | 0x1D | ^]<br>]     |    | 01011101 | 135 | 93  | 0x5D | ]          |
| 00011110 | 036 | 30 | 0x1E | ^^<br>^     |    | 01011110 | 136 | 94  | 0x5E | ^          |
| 00011111 | 037 | 31 | 0x1F | ^_<br>_     |    | 01011111 | 137 | 95  | 0x5F | _          |
| 00100000 | 040 | 32 | 0x20 | space       |    | 01100000 | 140 | 96  | 0x60 | open quote |
| 00100001 | 041 | 33 | 0x21 | !           |    | 01100001 | 141 | 97  | 0x61 | a          |
| 00100010 | 042 | 34 | 0x22 | "           |    | 01100010 | 142 | 98  | 0x62 | b          |
| 00100011 | 043 | 35 | 0x23 | #           |    | 01100011 | 143 | 99  | 0x63 | c          |
| 00100100 | 044 | 36 | 0x24 | \$          |    | 01100100 | 144 | 100 | 0x64 | d          |
| 00100101 | 045 | 37 | 0x25 | %           |    | 01100101 | 145 | 101 | 0x65 | e          |
| 00100110 | 046 | 38 | 0x26 | &           |    | 01100110 | 146 | 102 | 0x66 | f          |
| 00100111 | 047 | 39 | 0x27 | close quote |    | 01100111 | 147 | 103 | 0x67 | g          |
| 00101000 | 050 | 40 | 0x28 | (           |    | 01101000 | 150 | 104 | 0x68 | h          |
| 00101001 | 051 | 41 | 0x29 | )           |    | 01101001 | 151 | 105 | 0x69 | i          |
| 00101010 | 052 | 42 | 0x2A | *           |    | 01101010 | 152 | 106 | 0x6A | j          |
| 00101011 | 053 | 43 | 0x2B | +           |    | 01101011 | 153 | 107 | 0x6B | k          |
| 00101100 | 054 | 44 | 0x2C | comma       |    | 01101100 | 154 | 108 | 0x6C | l          |
| 00101101 | 055 | 45 | 0x2D | -           |    | 01101101 | 155 | 109 | 0x6D | m          |
| 00101110 | 056 | 46 | 0x2E | .           |    | 01101110 | 156 | 110 | 0x6E | n          |
| 00101111 | 057 | 47 | 0x2F | /           |    | 01101111 | 157 | 111 | 0x6F | o          |
| 00110000 | 060 | 48 | 0x30 | 0           |    | 01110000 | 160 | 112 | 0x70 | p          |
| 00110001 | 061 | 49 | 0x31 | 1           |    | 01110001 | 161 | 113 | 0x71 | q          |
| 00110010 | 062 | 50 | 0x32 | 2           |    | 01110010 | 162 | 114 | 0x72 | r          |
| 00110011 | 063 | 51 | 0x33 | 3           |    | 01110011 | 163 | 115 | 0x73 | s          |
| 00110100 | 064 | 52 | 0x34 | 4           |    | 01110100 | 164 | 116 | 0x74 | t          |
| 00110101 | 065 | 53 | 0x35 | 5           |    | 01110101 | 165 | 117 | 0x75 | u          |
| 00110110 | 066 | 54 | 0x36 | 6           |    | 01110110 | 166 | 118 | 0x76 | v          |
| 00110111 | 067 | 55 | 0x37 | 7           |    | 01110111 | 167 | 119 | 0x77 | w          |
| 00111000 | 070 | 56 | 0x38 | 8           |    | 01111000 | 170 | 120 | 0x78 | x          |
| 00111001 | 071 | 57 | 0x39 | 9           |    | 01111001 | 171 | 121 | 0x79 | y          |
| 00111010 | 072 | 58 | 0x3A | :           |    | 01111010 | 172 | 122 | 0x7A | z          |
| 00111011 | 073 | 59 | 0x3B | ;           |    | 01111011 | 173 | 123 | 0x7B | {          |
| 00111100 | 074 | 60 | 0x3C | <           |    | 01111100 | 174 | 124 | 0x7C |            |
| 00111101 | 075 | 61 | 0x3D | =           |    | 01111101 | 175 | 125 | 0x7D | }          |
| 00111110 | 076 | 62 | 0x3E | >           |    | 01111110 | 176 | 126 | 0x7E | ~          |
| 00111111 | 077 | 63 | 0x3F | ?           |    | 01111111 | 177 | 127 | 0x7F | del        |



---

---

## Bibliography

---

---

**The C Puzzle Book****Alan R Feuer****Prentice Hall****ISBN 0-13-115502-4**

around £32

This is a book of “what will the following program print” questions. The reader is expected to work through successive programs. There are answers and comprehensive explanations in case you haven’t got the answer right. An excellent book for learning C, although not how to write programs in it (puzzling programs are necessarily written in a puzzling style). “Something for the new and experienced C programmer”.

**The C Programming Language 2nd edition****B. W. Kernighan and D. M. Ritchie****Prentice Hall****ISBN 0-13-110362-8**

around £32

Although this book is written by the two creators of C it is not a tutorial introduction to the language. It is directed more toward very experienced programmers who require the essential elements of C in the most concise manner possible. An ideal book if you are planning to write a C compiler. Be sure to buy the 2nd edition which describes Standard C as opposed to the 1st edition which describes K&R C.

**The C Standard Library****P. J. Plauger****Prentice Hall****ISBN 0-13-131509-9**

around £30

The definitive guide to the why and how of C’s Standard Library. Not only is source code provided for each and every function (the source code in disk form may be purchased separately), but there are explanations of why the committees decided on the behavior of each function. The sort of book that describes why `fgetpos` was invented when `fseek` was already available.

**C Traps and Pitfalls****Andrew Koenig****Addison Wesley****ISBN 0-20-117928-8**

around £18

“Even C Experts come across problems that require days of debugging to fix. This book helps to prevent such problems by showing how C programmers get themselves into trouble. Each of the book’s many examples have trapped a professional programmer”.

Initially a rather depressing book. Horrible errors are catalogued, leaving the reader asking “how can I possibly ever write a C program that works”. The second part of the book addresses the problems and contains many useful tips for improving code.

---