C Programming

Cheltenham Computer Training

Crescent House
24 Lansdown Crescent Lane
Cheltenham
Gloucestershire
GL50 2LD
United Kingdom

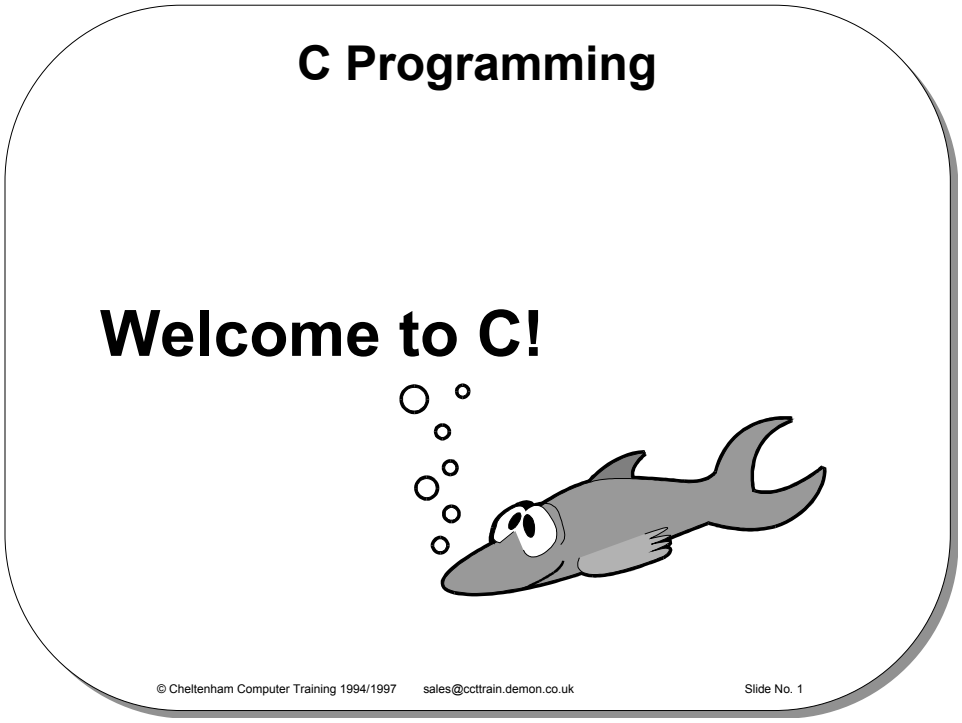
Tel: +44 (0)1242 227200
Fax: +44 (0)1242 253200
Email: sales@cctglobal.com
http://www.cctglobal.com/

© Cheltenham Computer Training 1995-2000

CONTENTS

 © 1995-2000 Cheltenham Computer Training

Introduction



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:

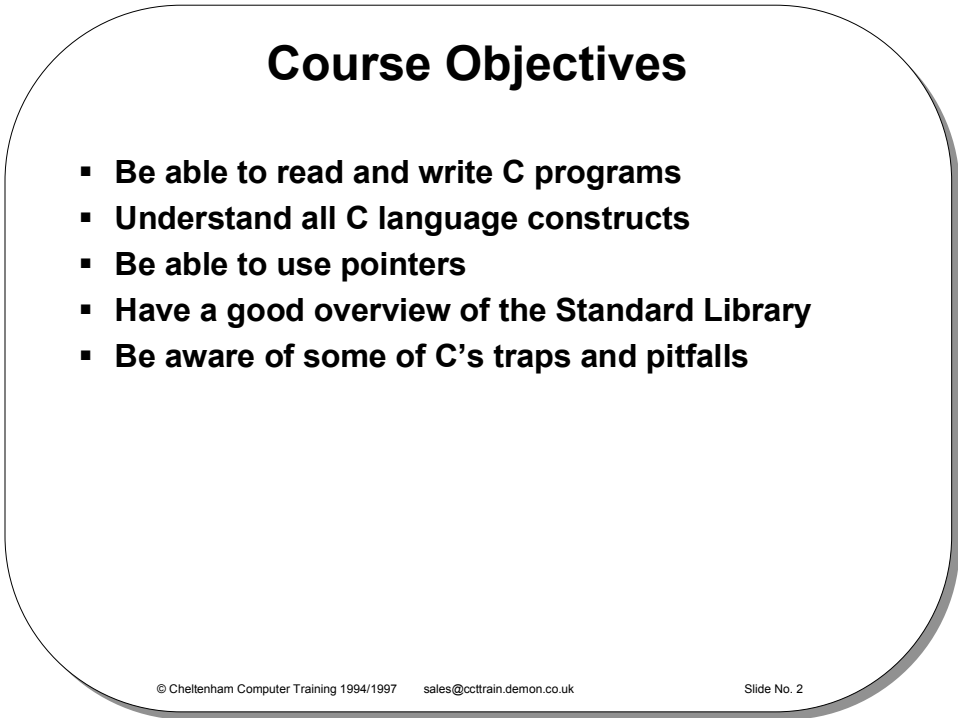
- What a variable is
- The difference between a variable and a constant
- The idea of a decision ("*if* it is raining, *then* I need an umbrella, *else* I need sunblock")
- The concept of a loop

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.



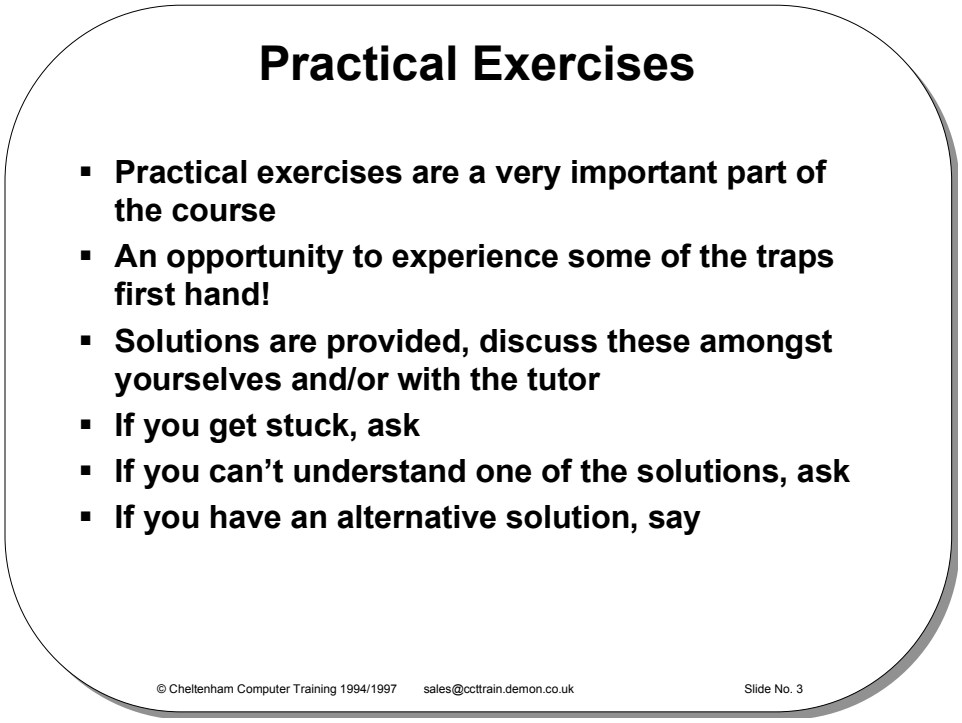
Course Objectives

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

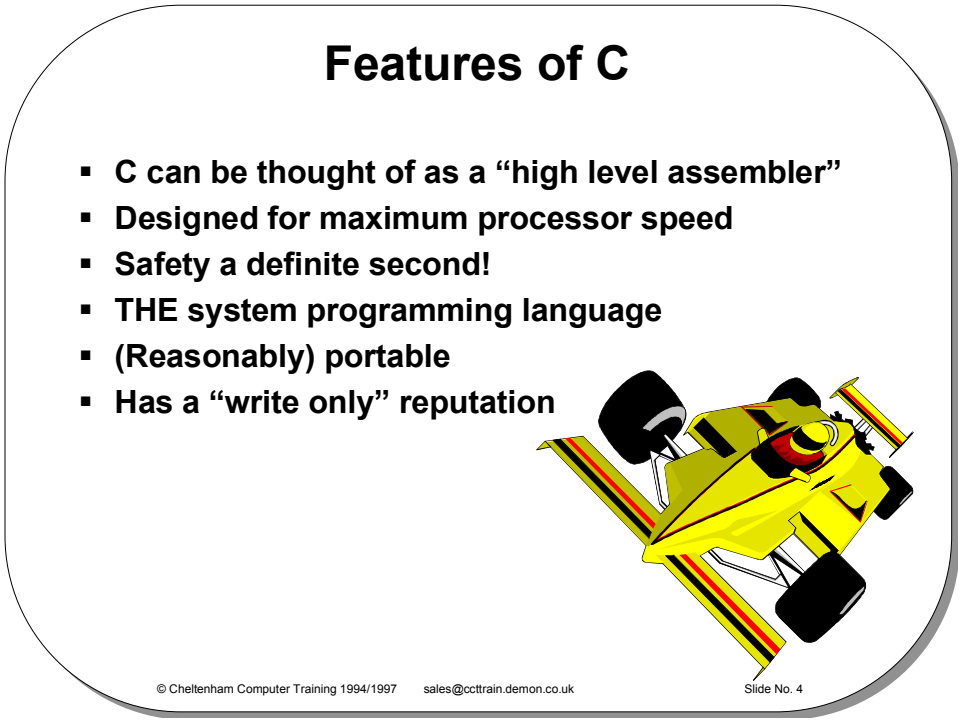
Writing C is Important!

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

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"**

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 5

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.

 © 1995-2000 Cheltenham Computer Training

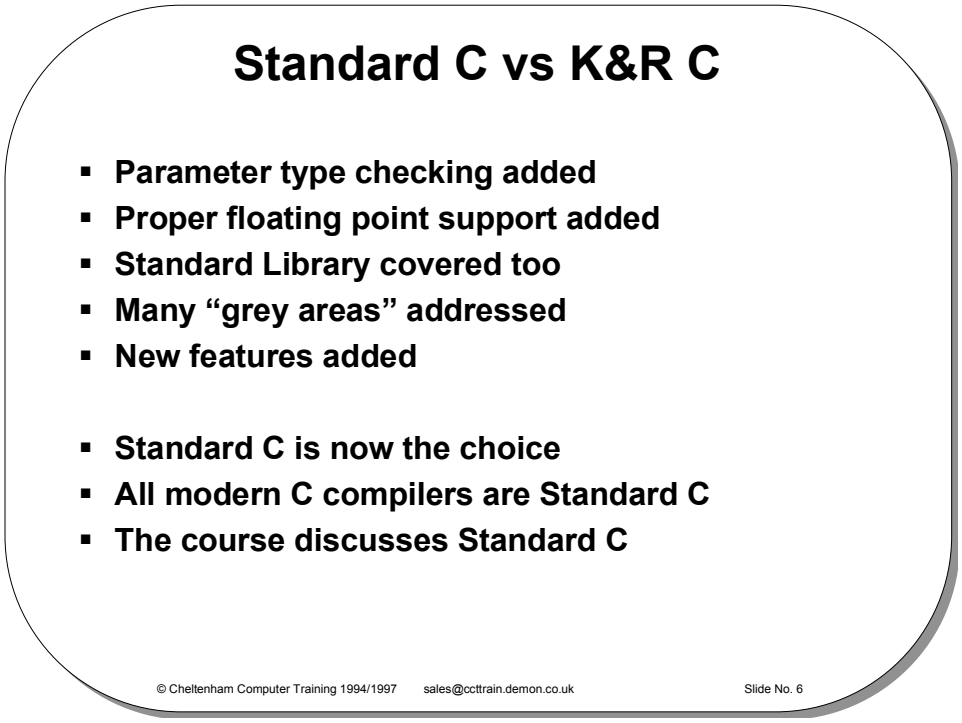
The History of C

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”.



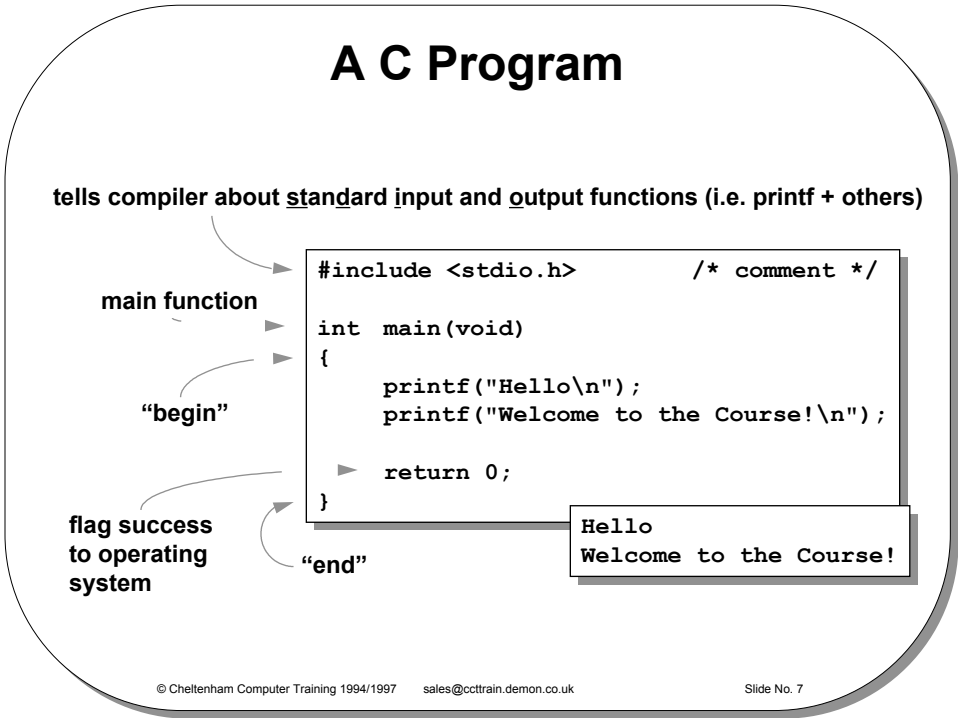
Standard C vs. K&R C

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.

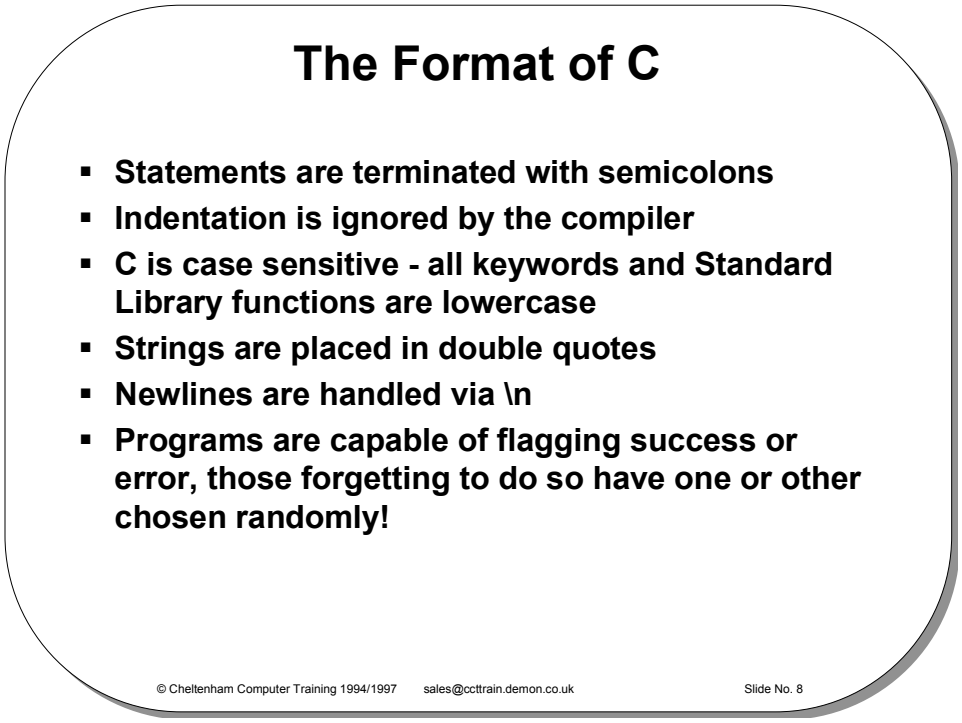
`main` The `main` function is most important. This defines the point at which your program starts to execute. If you do not write a `main` function your program will not run (it will have no starting point). In fact, it won't even compile.

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 "\" 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.



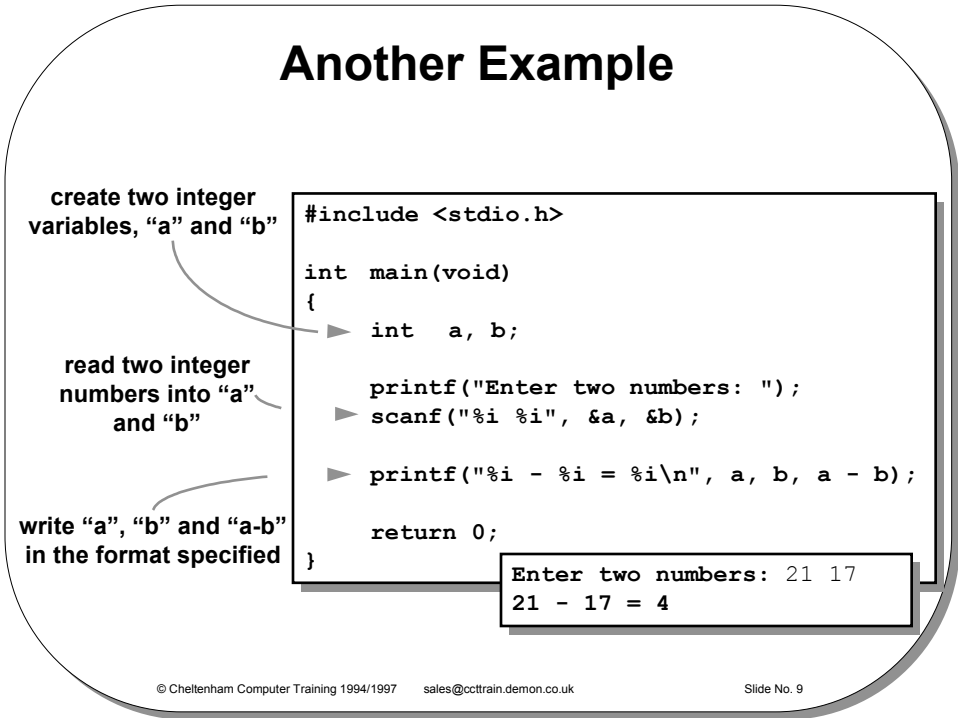
The Format of C

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 `int` 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 `main` is to return an integer to the operating system, forgetting to do so (either by saying `return` only or by omitting the `return` 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

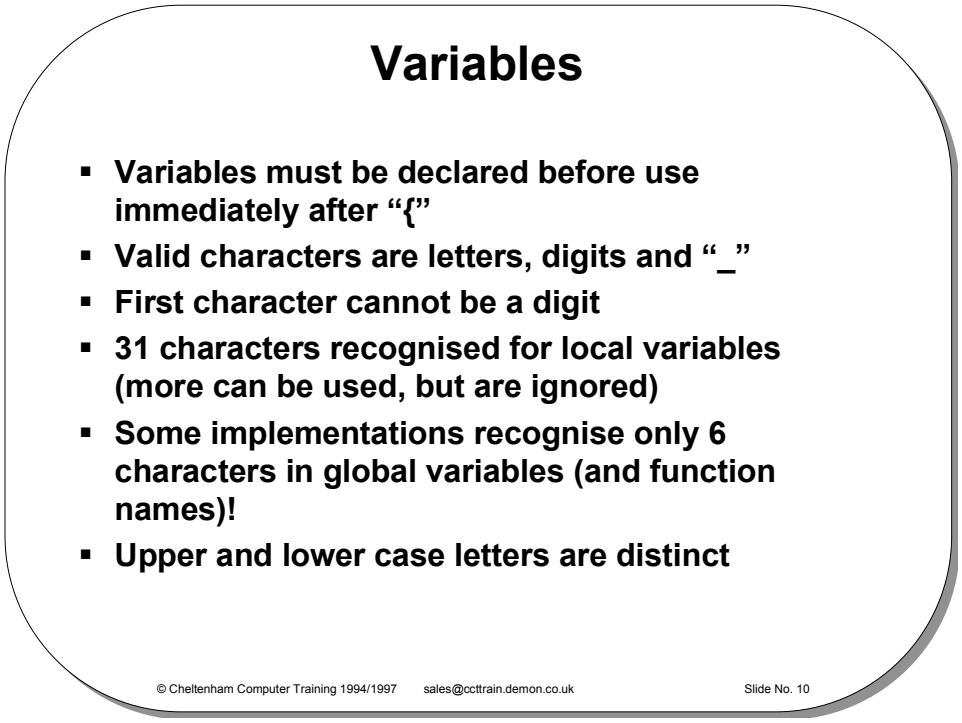
int The **int** keyword, seen before when defining the return type for **main**, is used to create integer variables. Here two are created, the first “a”, the second called “b”.

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

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

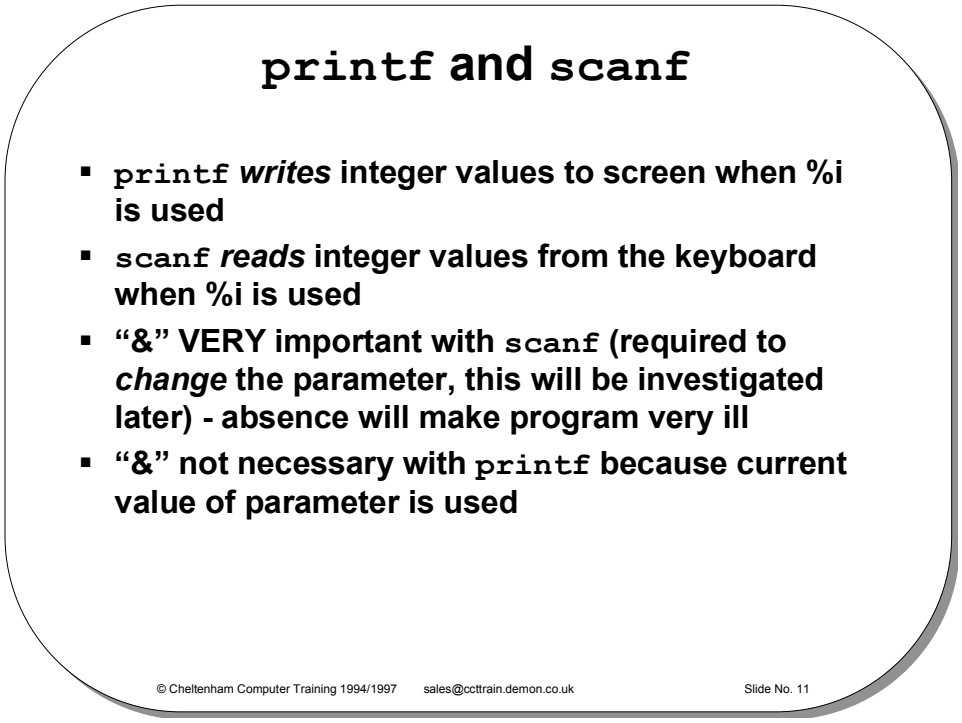
Only letters, digits and the underscore character may be validly used in variable names. The first character of a variable may be a letter or an underscore, although The Standard says to avoid the use of underscores as the first letter. Thus the variable names "temp_in_celsius", "index32" and "sine_value" 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 "tempInCelsius". This naming style has become quite popular in recent years and the underscore has fallen into disuse.



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:

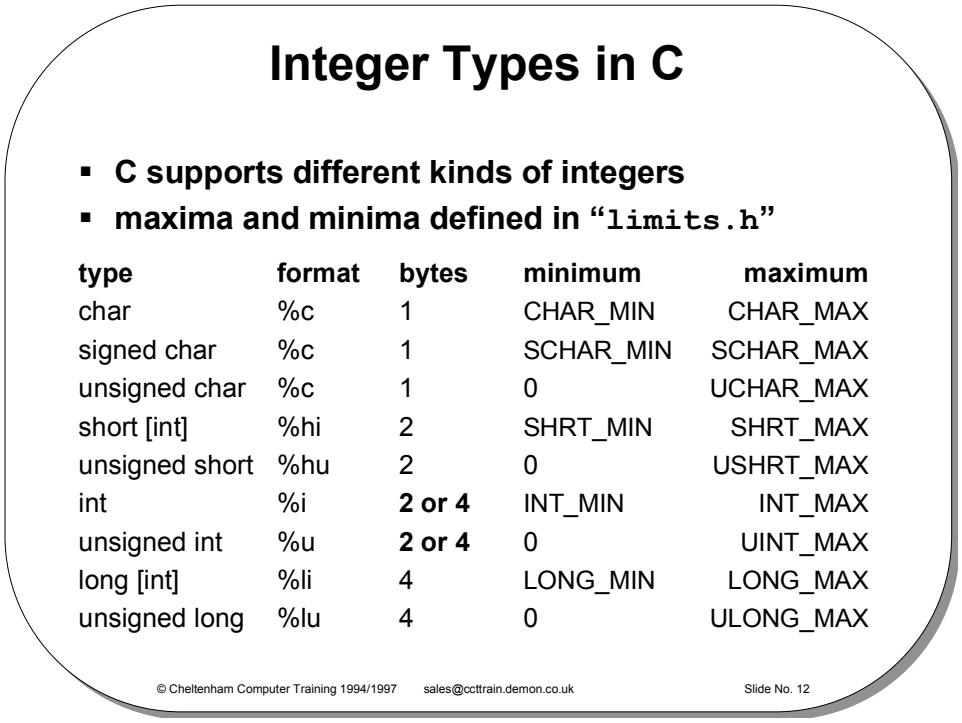
```
scanf("%i", &j)
```

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

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.

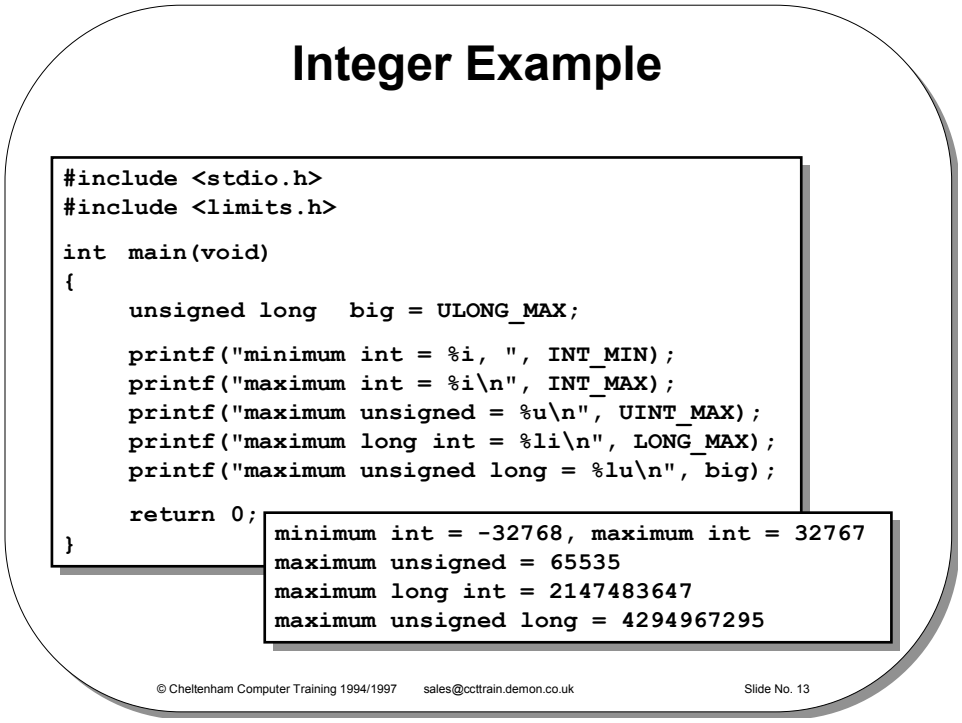
The most important thing to notice is that the size of **int** is either 2 or 4 bytes. Thus we cannot say, for a particular implementation, whether the largest value an integer may hold will be 32 thousand or 2 thousand million. For this reason, truly portable programs never use **int**, only **short**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

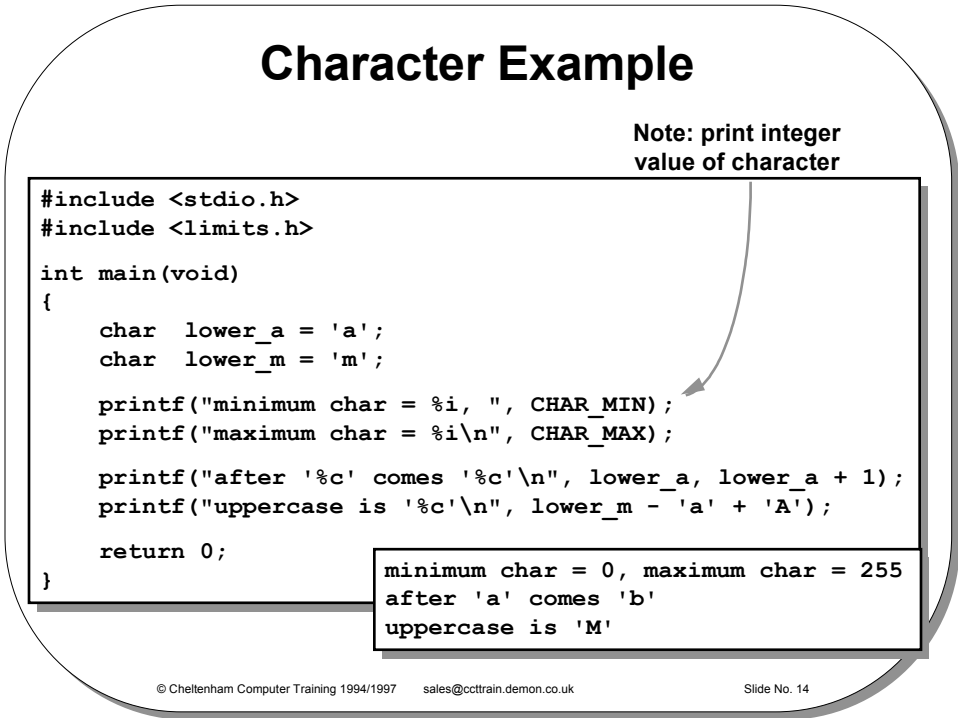


Integer Example

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

`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
```

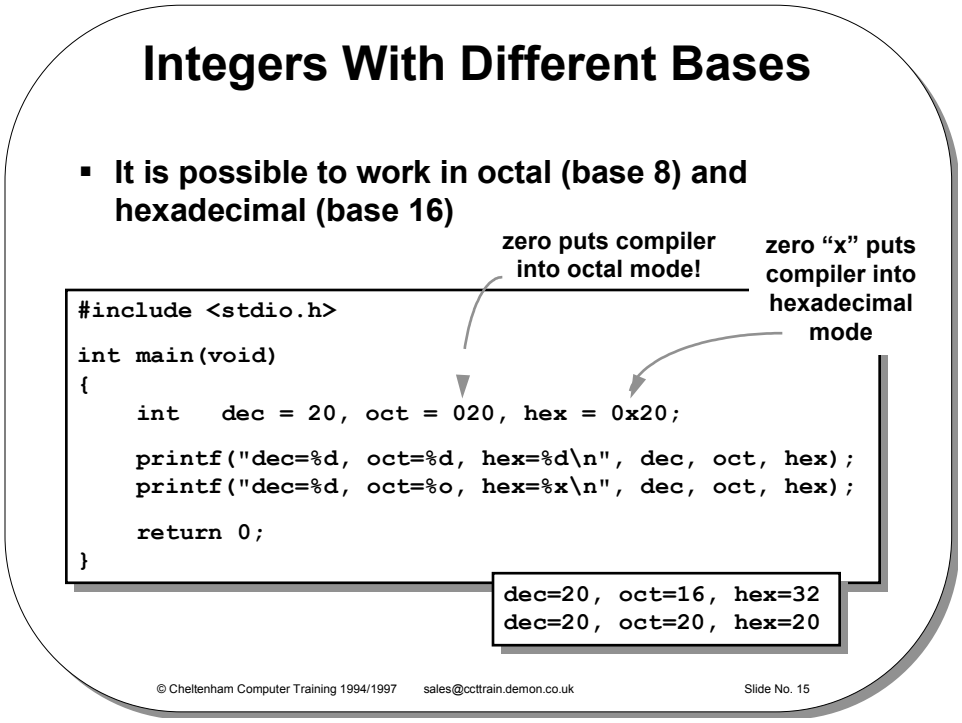
which yields 97 + 1, i.e. 98. This prints out as the value of lowercase “b” (one character immediately beyond lowercase “a”). The calculation:

```
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

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.

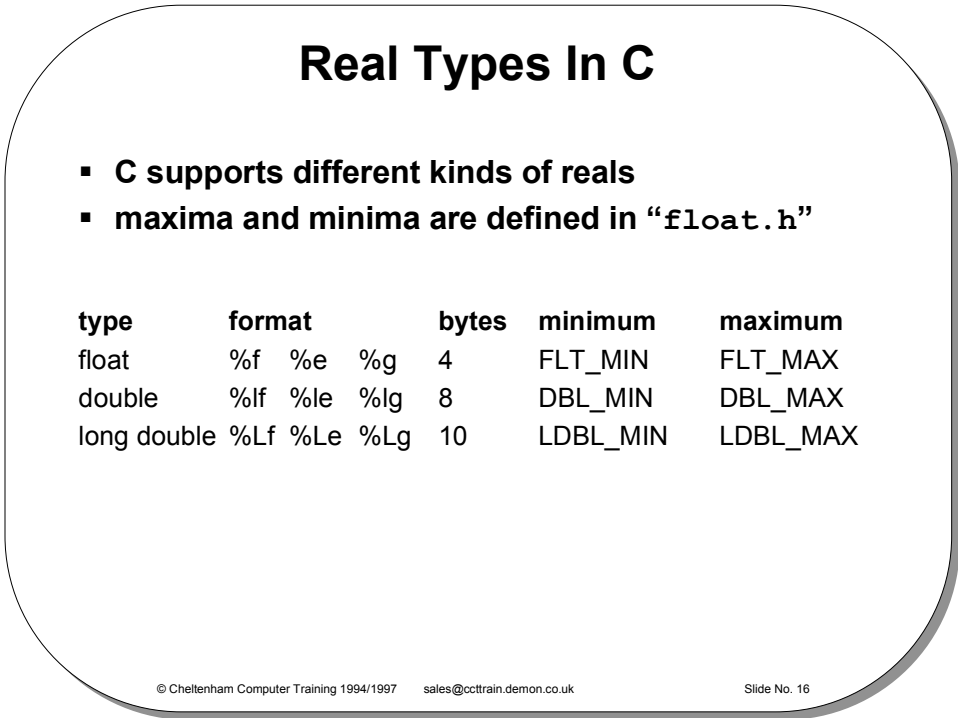
Specifying zero followed by "x" places the compiler into hexadecimal mode. Now the letters "a", "b", "c", "d", "e" and "f" may be used to represent the numbers 10 though 15. The case is unimportant, so 0x15AE, 0x15aE and 0x15ae represent the same number as does 0X15AE.

`%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.



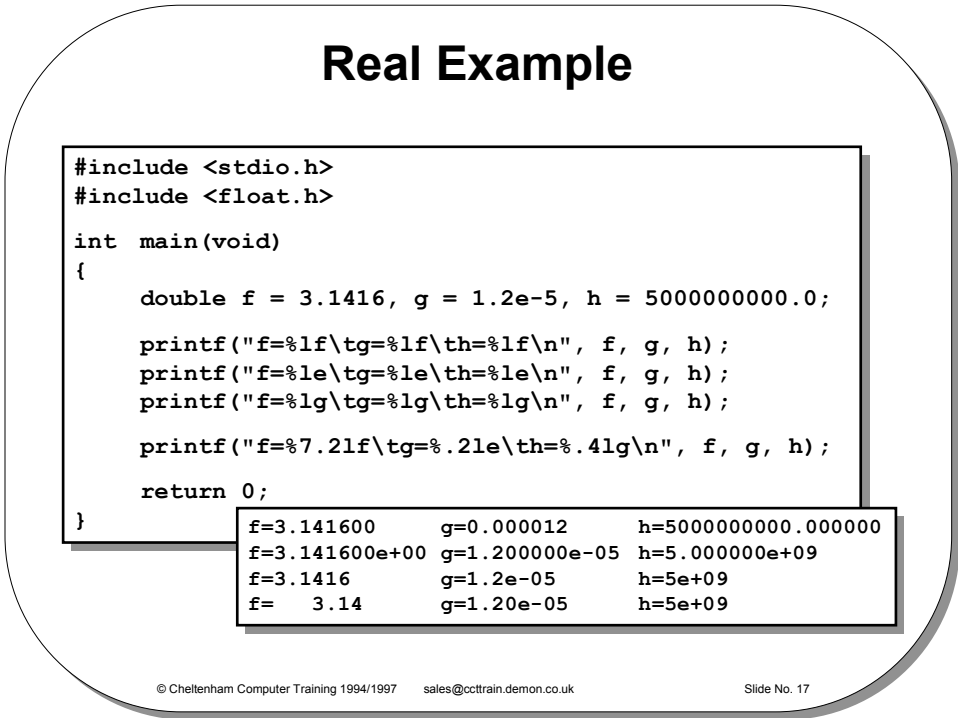
Real Types In C

`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

%lf This format specifier causes **printf** to display 6 decimal places, regardless of the magnitude of the number.

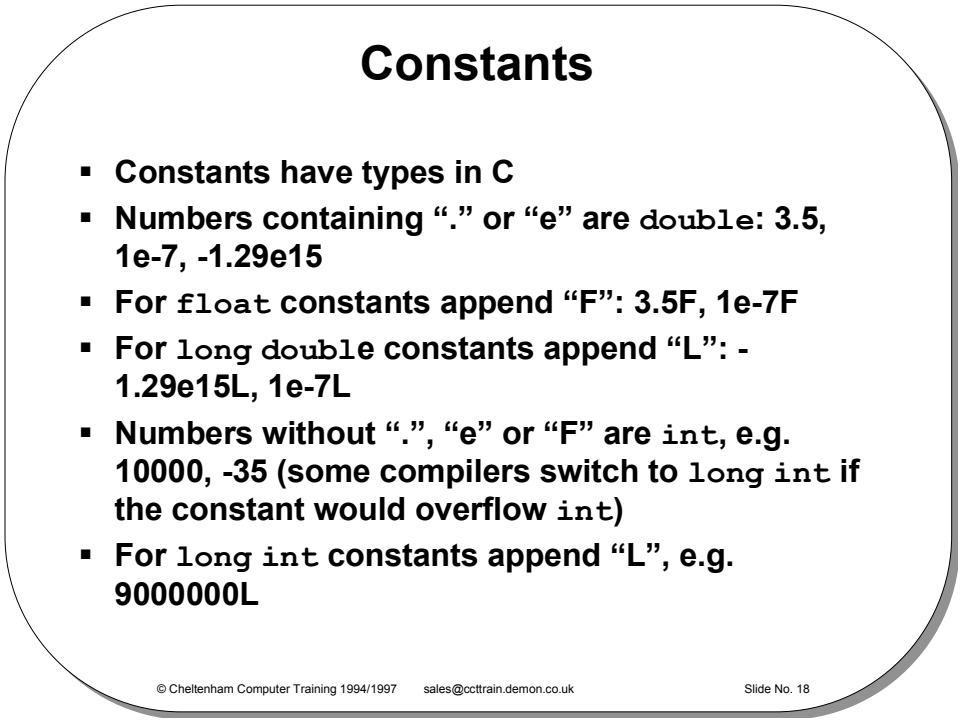
%le This format specifier still causes **printf** to display 6 decimal places, however, the number is displayed in “exponential” notation. For instance 1.200000e-05 indicates that 1.2 must be multiplied by 10^{-5}.

%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.



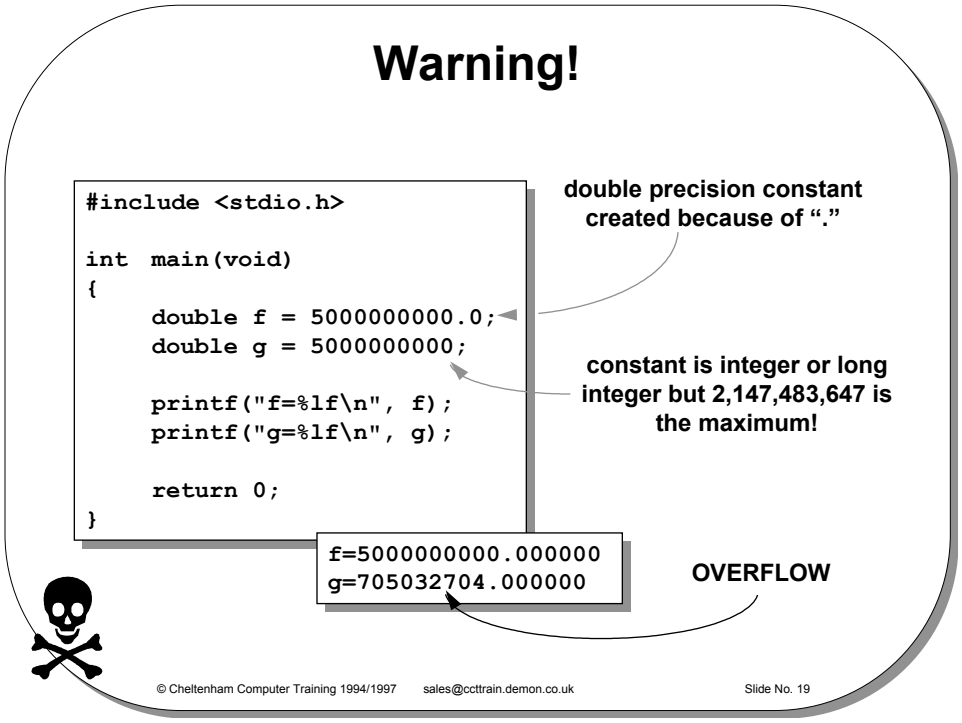
Constants

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`.

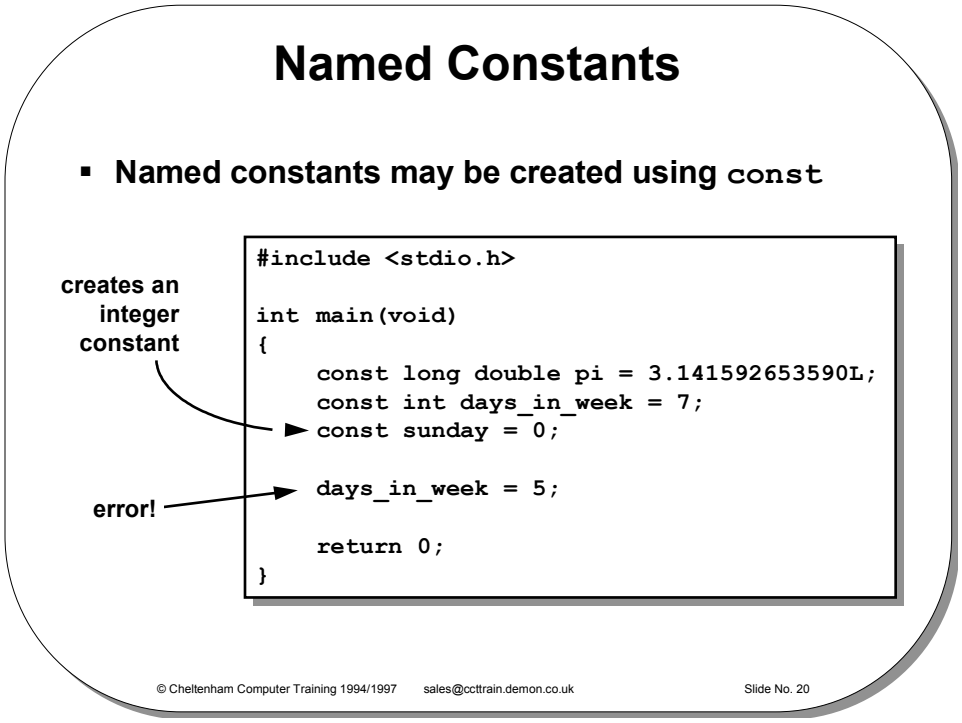
Although "12.L" has type `long double`, "12L" has type `long int`.



Warning!

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

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.

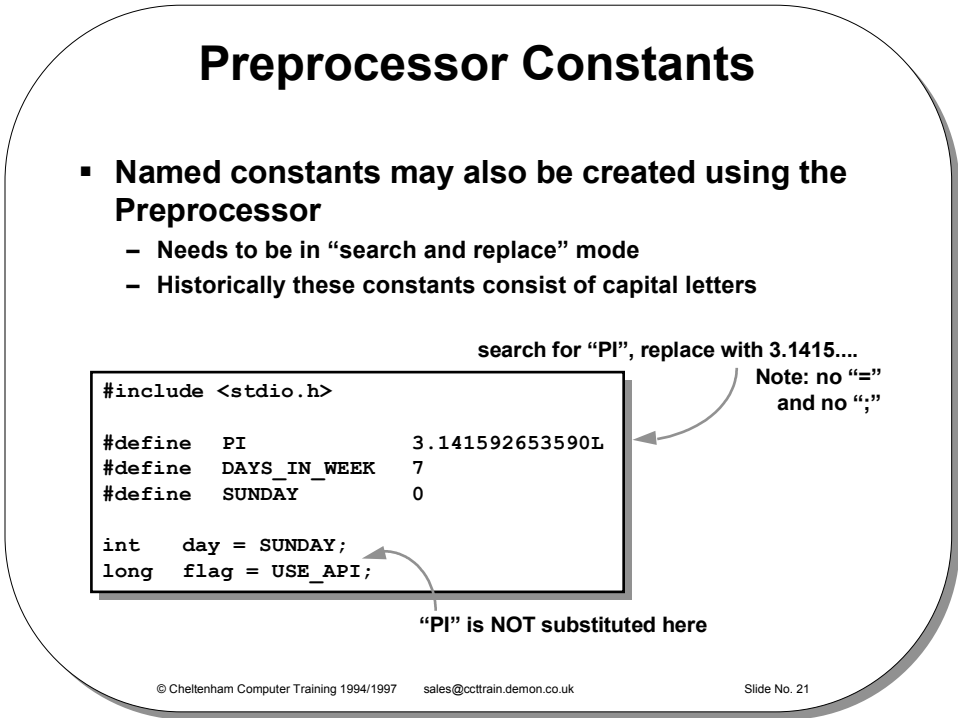
Using **const** the type is explicitly stated, except with **const sunday** where the integer type is the default. This is consistent with existing rules, for instance **short** really means **short int**, **long** really means **long int**.

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.



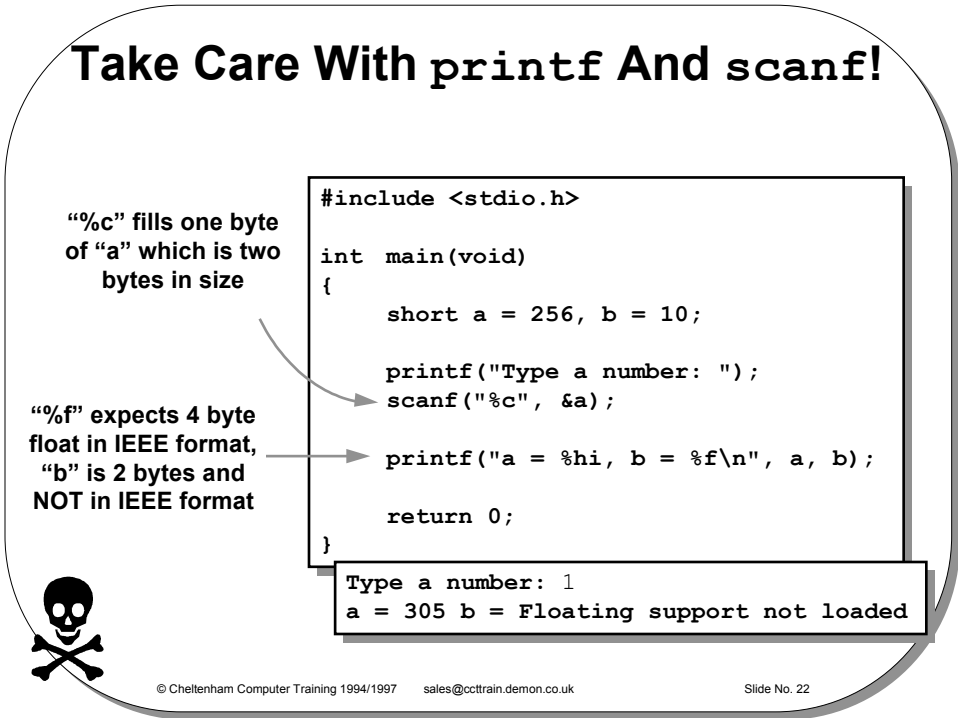
Preprocessor Constants

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`!

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 `char`s).

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`**

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 23

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).

 © 1995-2000 Cheltenham Computer Training

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: area = πr^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: 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.

 © 1995-2000 Cheltenham Computer Training

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: area = πr^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: circum = 2πr

   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;
}
```