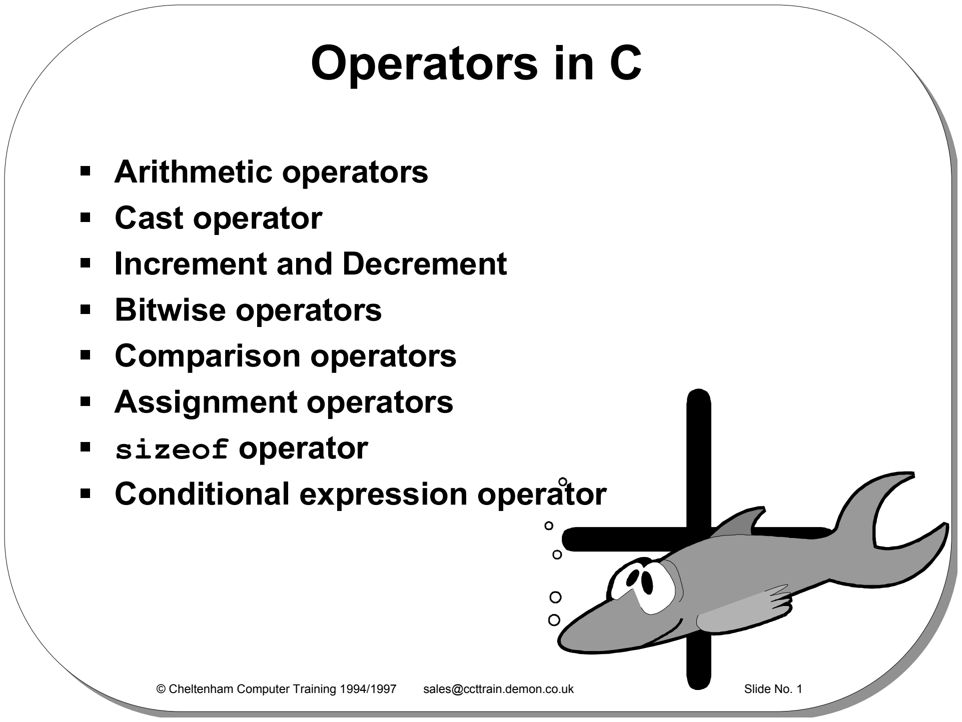
C Programming

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/

CONTENTS

Operators in C

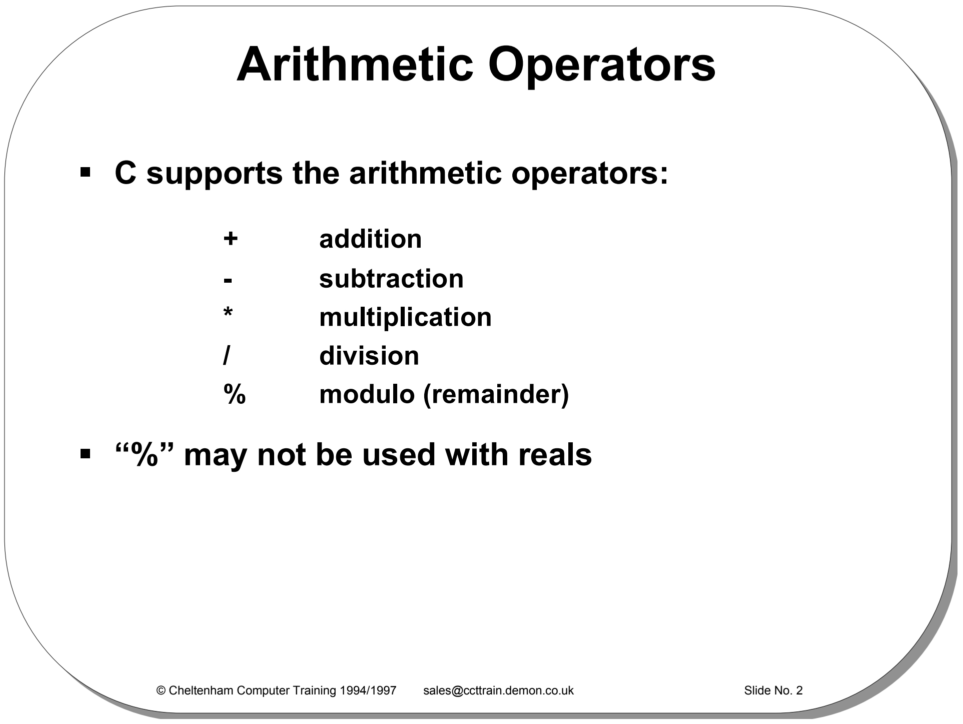


Operators in C

- **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**

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 2

Arithmetic Operators

`+, -, *, /`

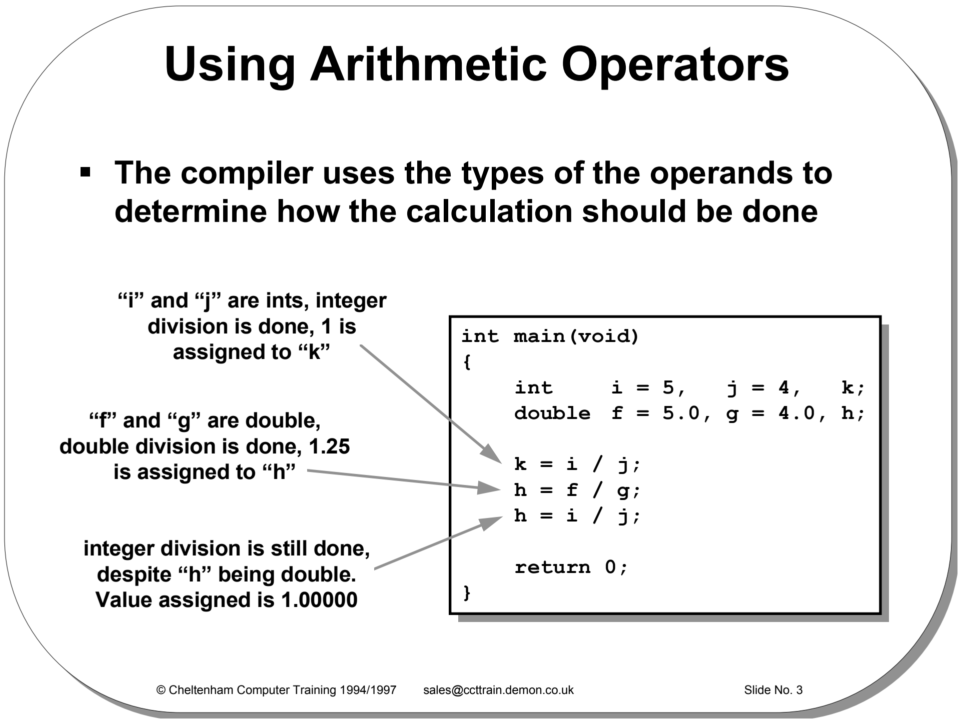
C provides the expected mathematical operators. There are no nasty surprises. As might be expected, “+” and “-” may be used in a unary sense as follows:

```
x = +y;
```
or
```
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**

"i" and "j" are ints, integer division is done, 1 is assigned to "k"

"f" and "g" are double, double division is done, 1.25 is assigned to "h"

integer division is still done, despite "h" being double. Value assigned is 1.00000

```
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;
}
```

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 3

Using Arithmetic Operators

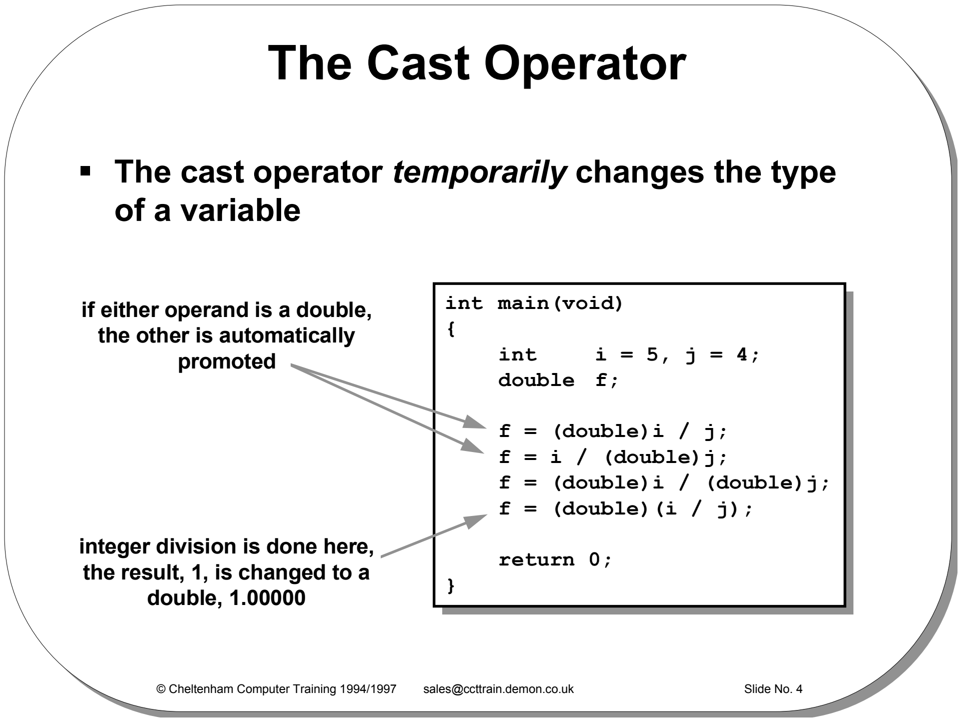
One operator "+" must add integers together and add reals together. It might almost have been easier to provide two, then the programmer could carefully choose whenever addition was performed. But why stop with two versions? There are, after all, different kinds of integer and different kinds of real. 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.

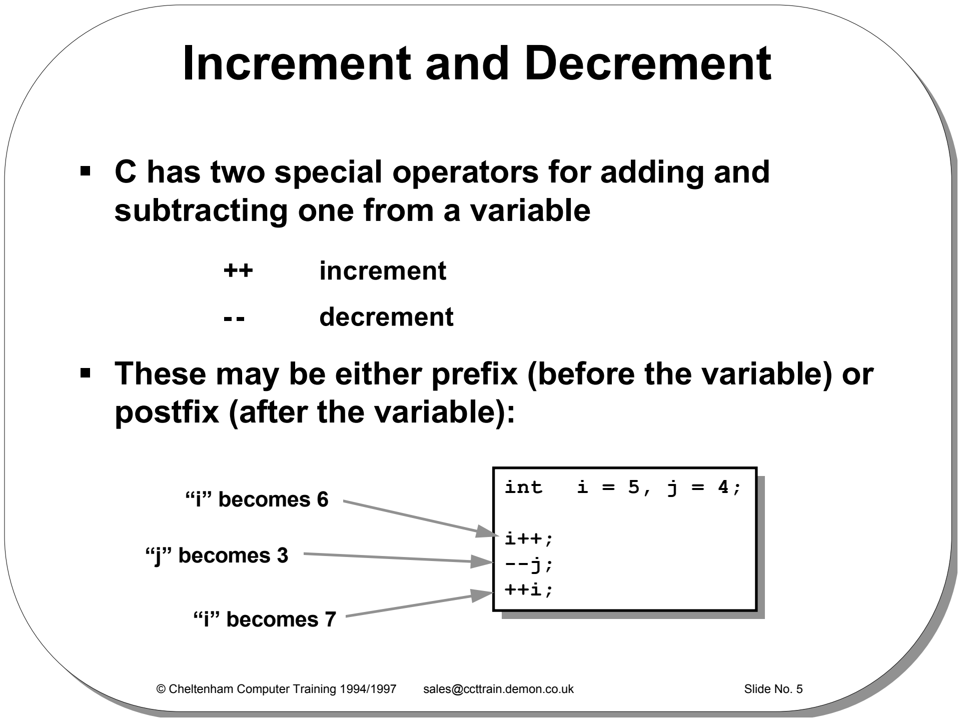
However, the compiler allows us to "change our mind" about the type of a variable or expression. This is done with the cast operator. The cast operator *temporarily* changes the type of the variable/expression it is applied to. Thus in:

```
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):**

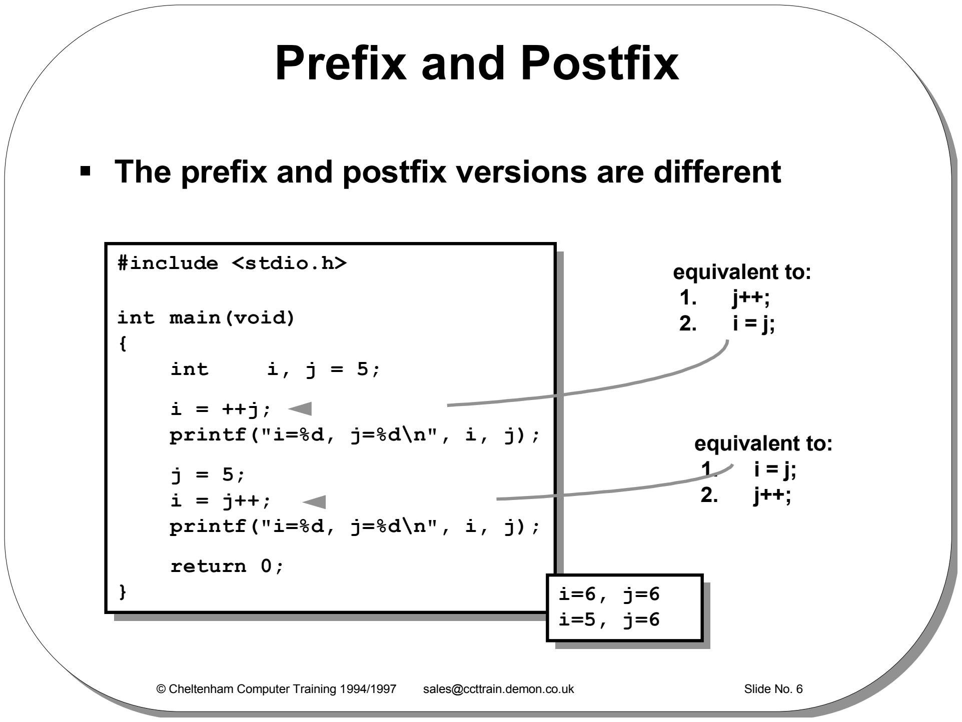
```
int   i = 5, j = 4;

i++;
--j;
++i;
```

“i” becomes 6
“j” becomes 3
“i” becomes 7

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

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
```

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

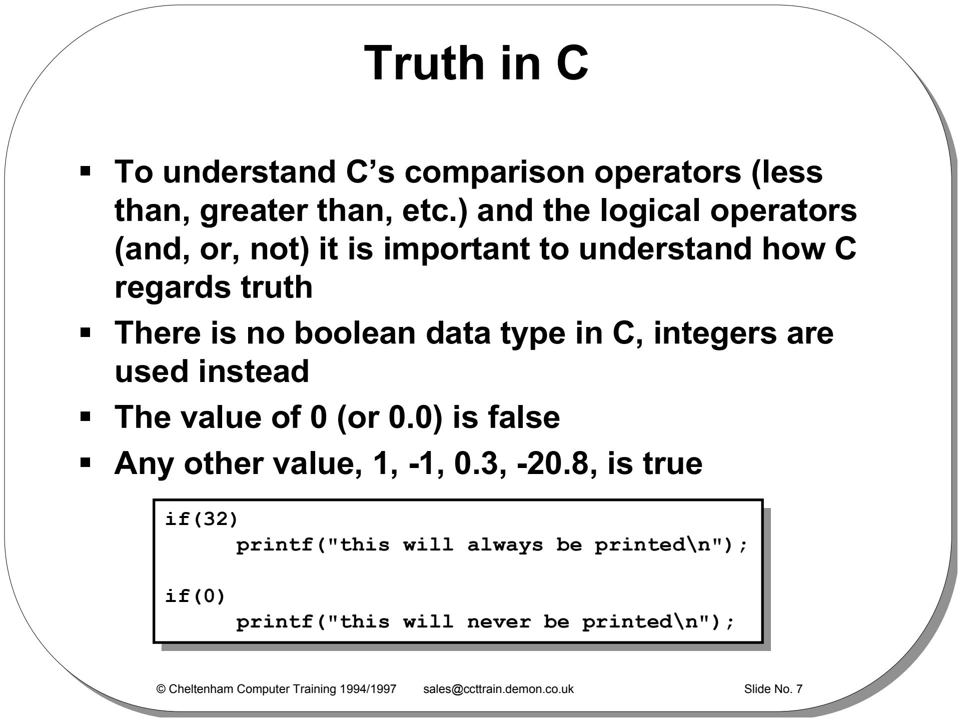
With the postfix operators, the increment or decrement happens **second**. The **unchanged** value is assigned, then the value changed. Thus with:

```
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");
```

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 7

Truth in C

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

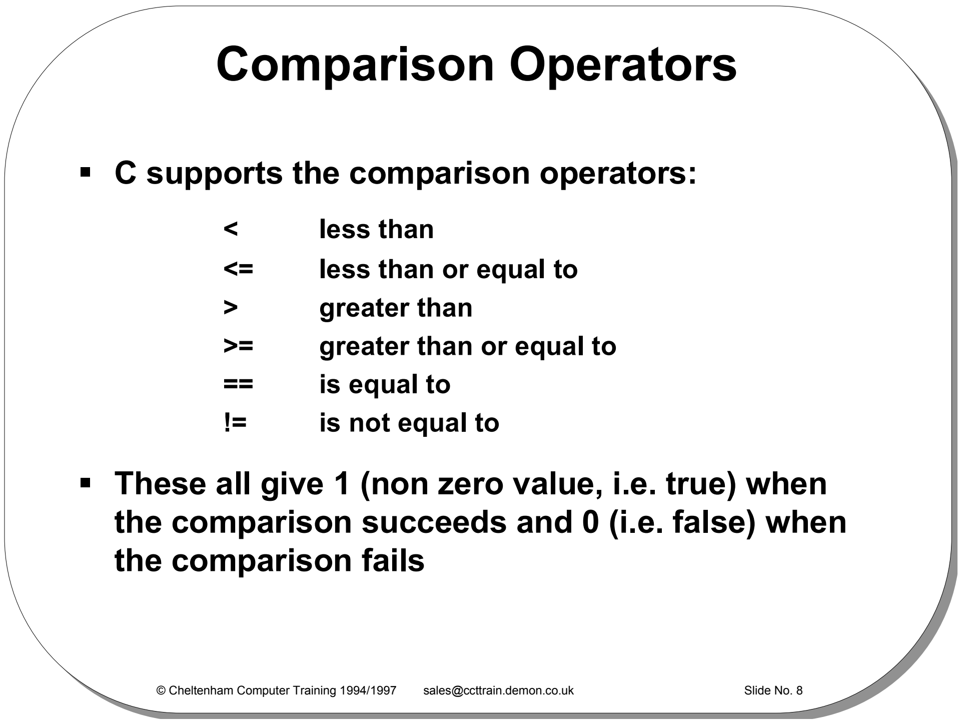
True Any non zero value is true. Thus, 1 and -5 are both true, because both are non zero. Similarly 0.01 is true because it, too, is non zero.

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**

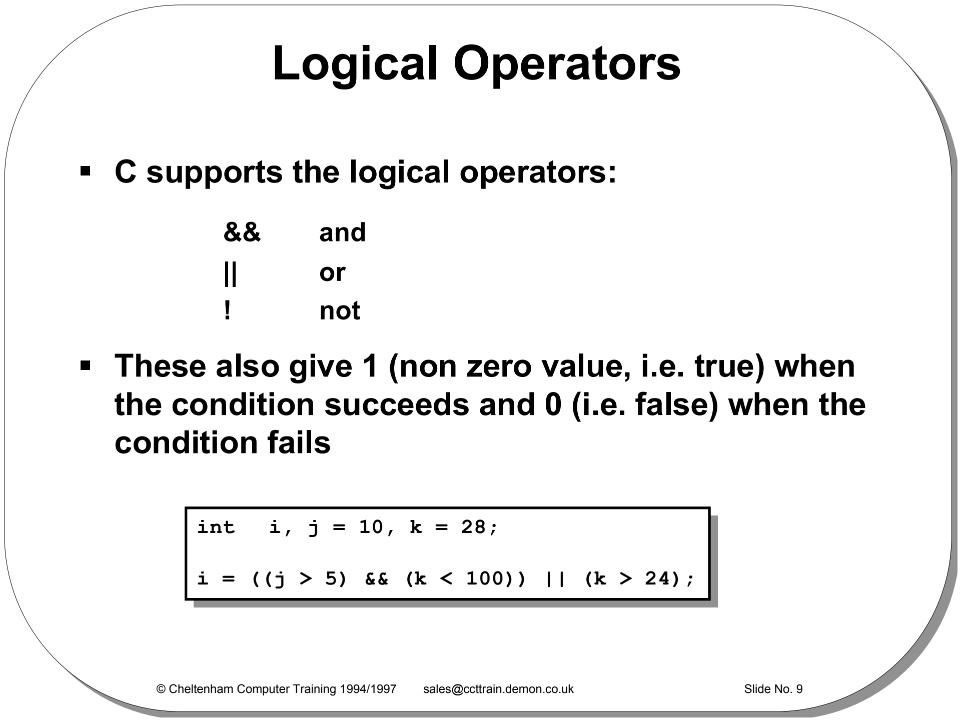
Comparison Operators

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**

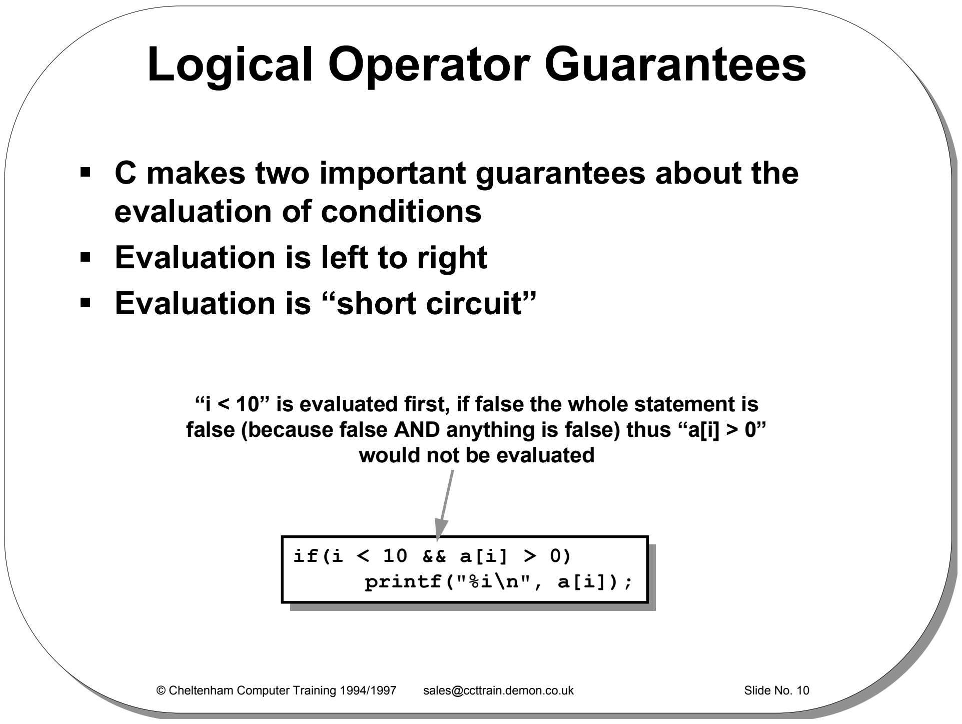
```
int   i, j = 10, k = 28;

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. It 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".

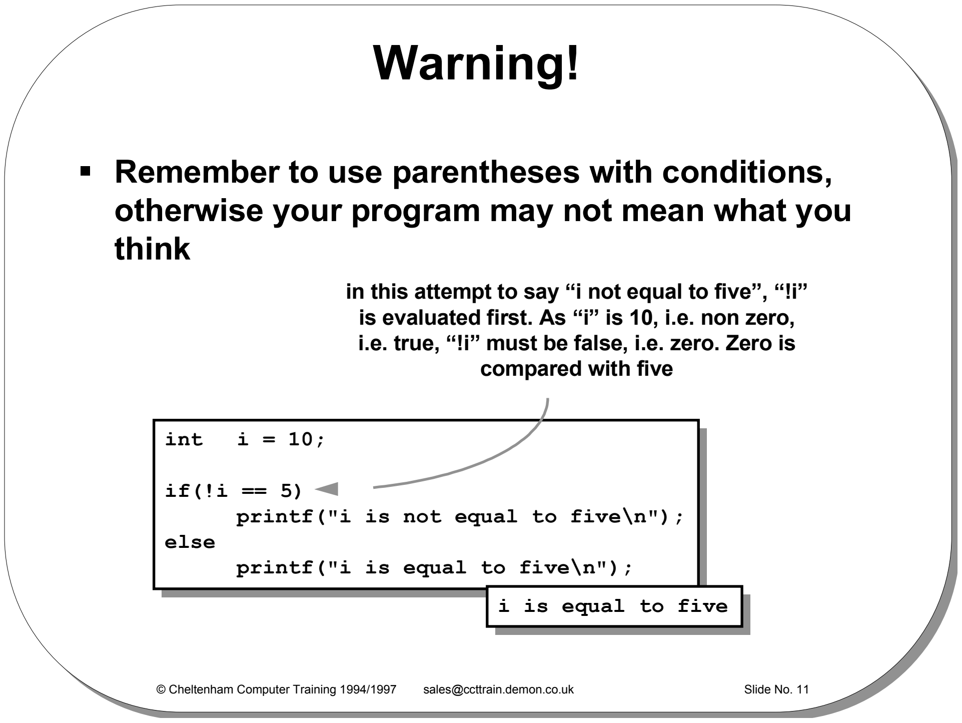
This is probably a good time to remind you about truth tables:

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**

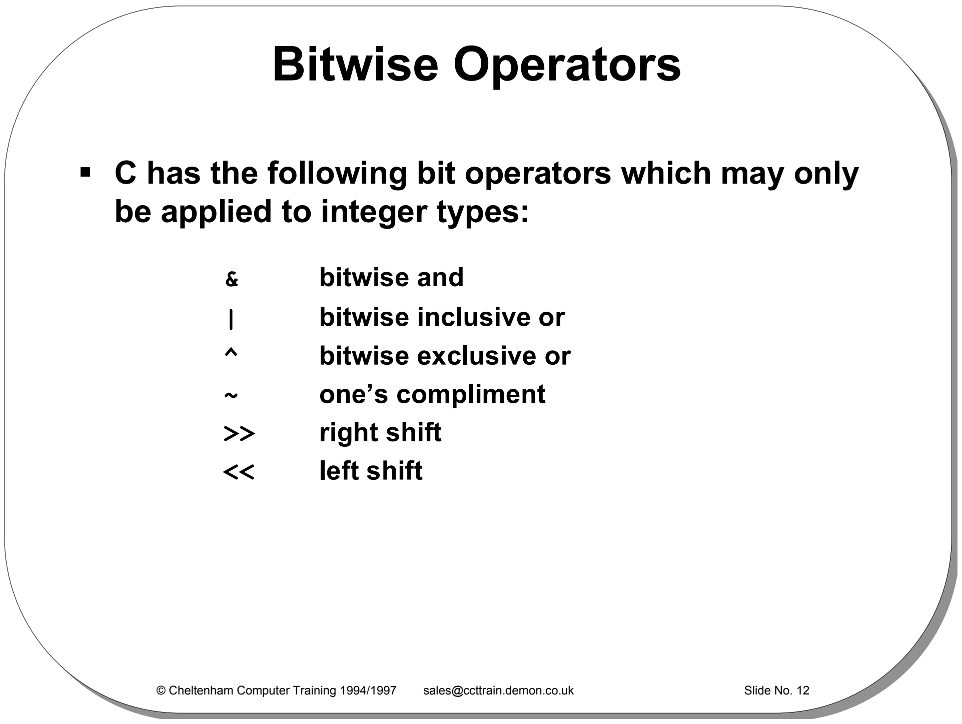
in this attempt to say “i not equal to five”, “!i” is evaluated first. As “i” is 10, i.e. non zero, i.e. true, “!i” must be false, i.e. zero. Zero is compared with five

```
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

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 12

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.

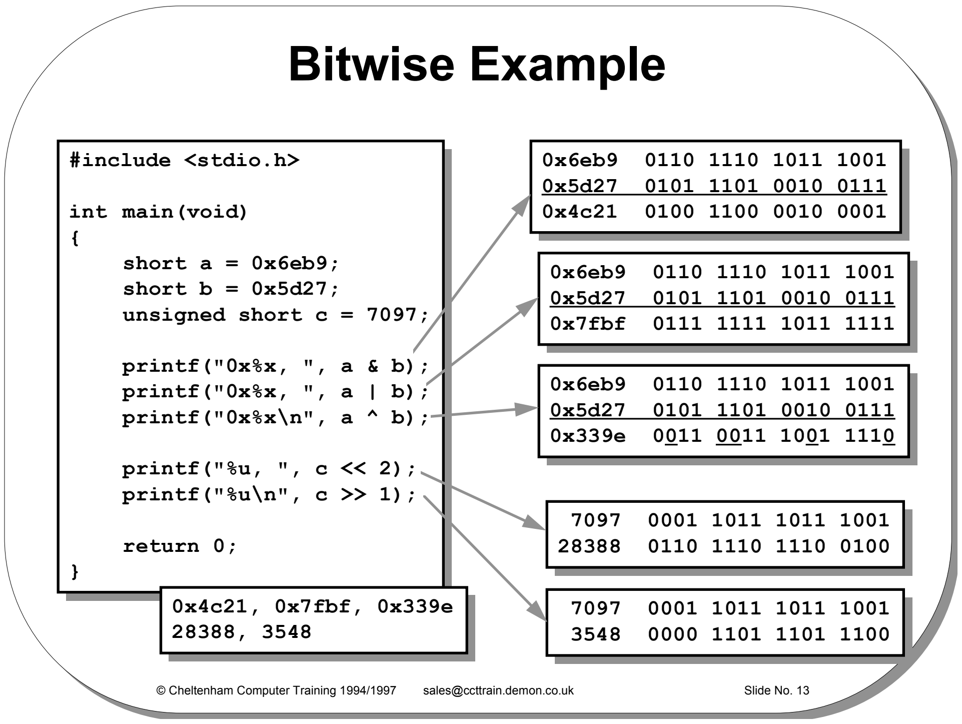
Truth Tables For Bitwise Operators

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
```

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 13

Bitwise Example

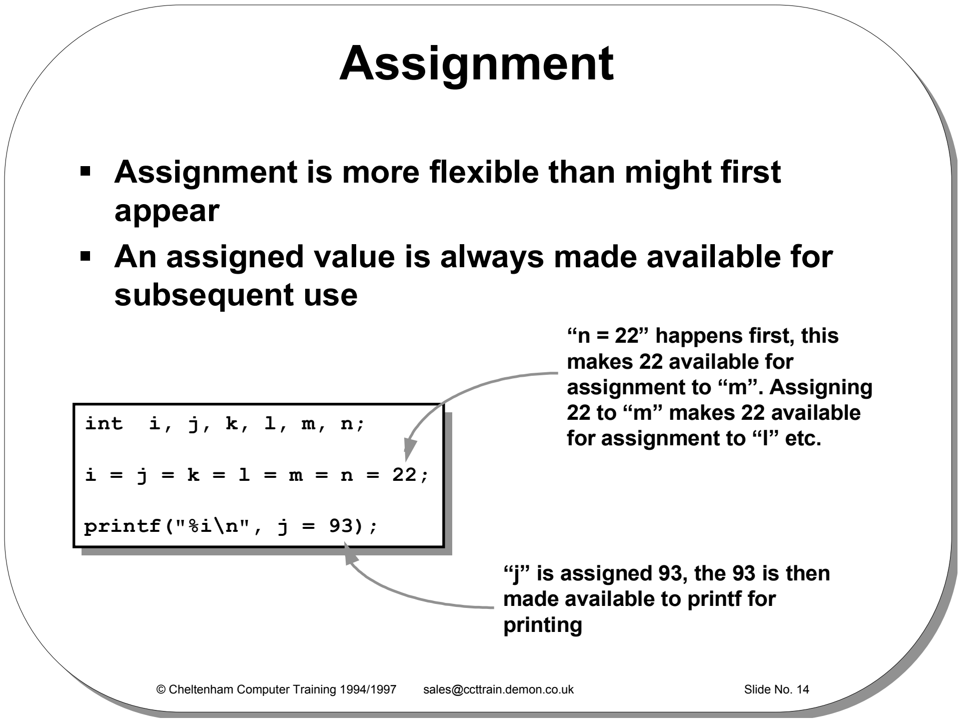
This example shows bitwise manipulation. `short` integers are used, because these can be relied upon to be 16 bits in length. If `int` had been used, we may have been manipulating 16 or 32 bits depending on machine and compiler.

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

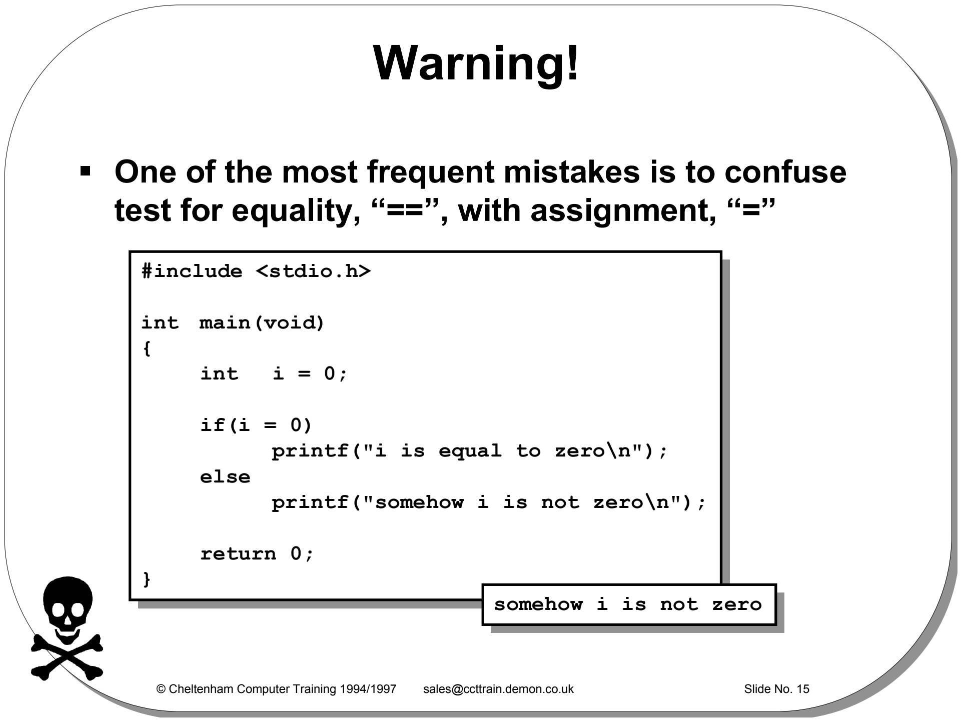
Assignment Uses Registers

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;
}
```

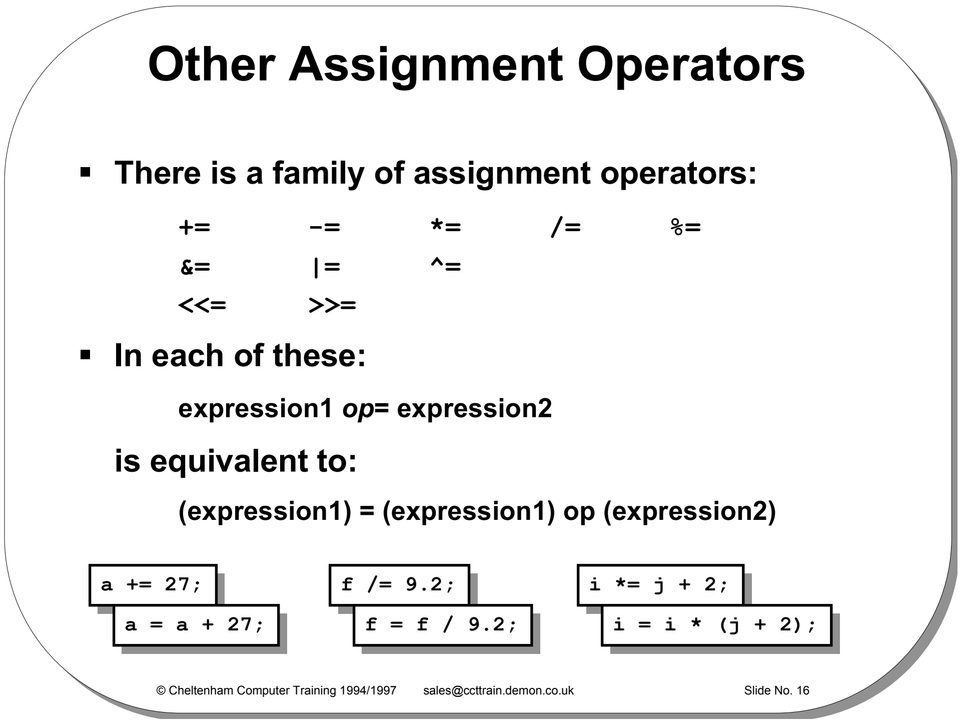
```
somehow i is not zero
```

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);
```

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 16

Other Assignment Operators

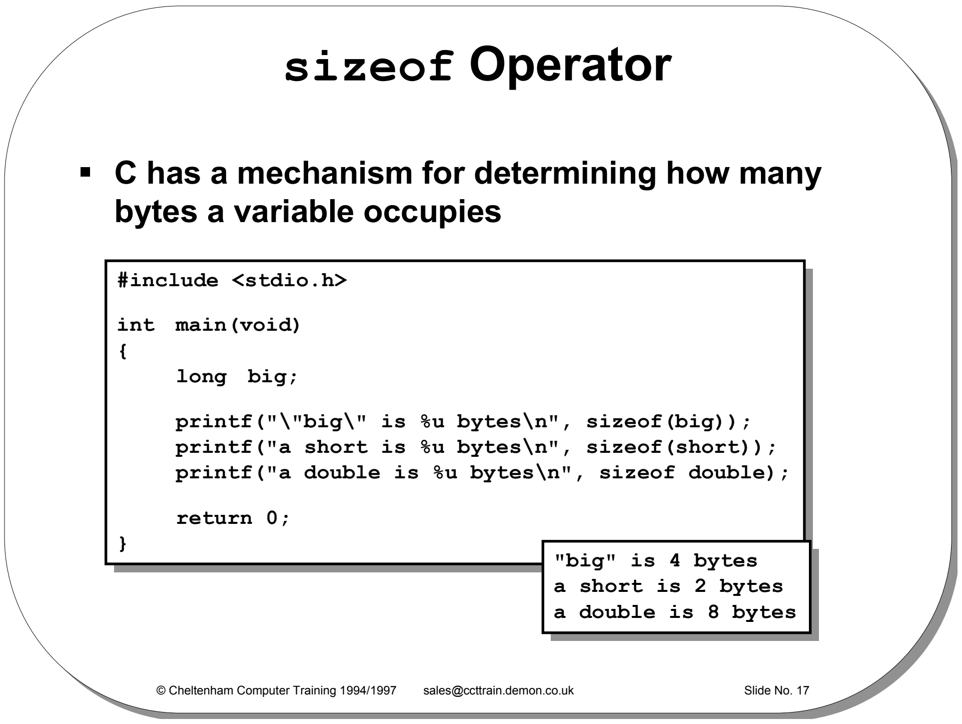
`+=`, `-=`, `*=`, `/=`, `%=` etc.

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 `int`s and another using 32 bit `int`s. 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:

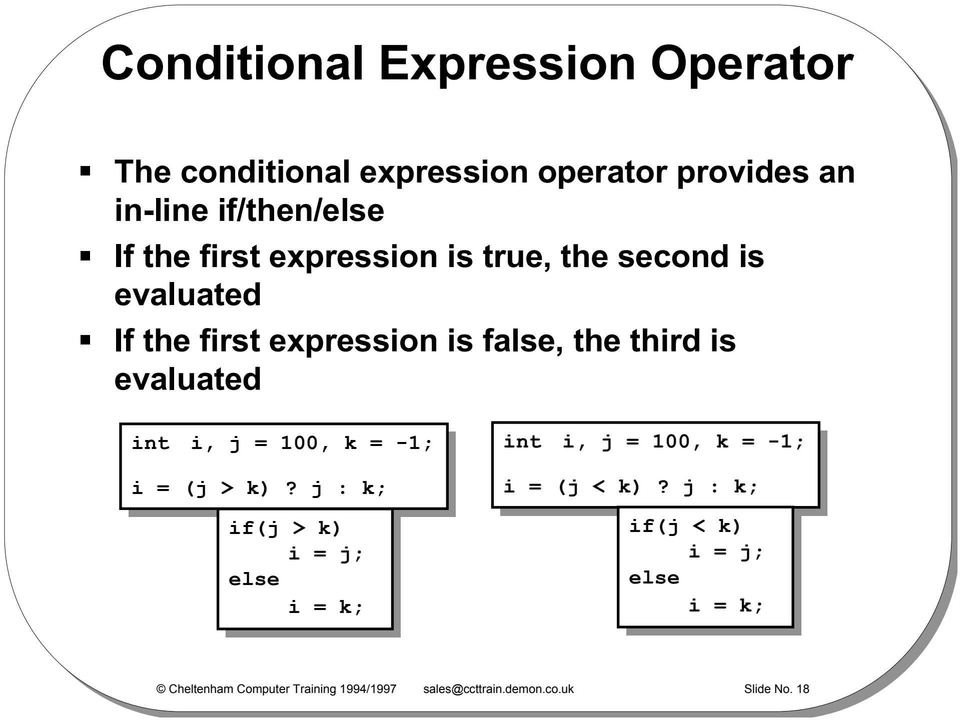
```
sizeof(double)
```

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;
```

© Cheltenham Computer Training 1994/1997 sales@ccttrain.demon.co.uk Slide No. 18

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).

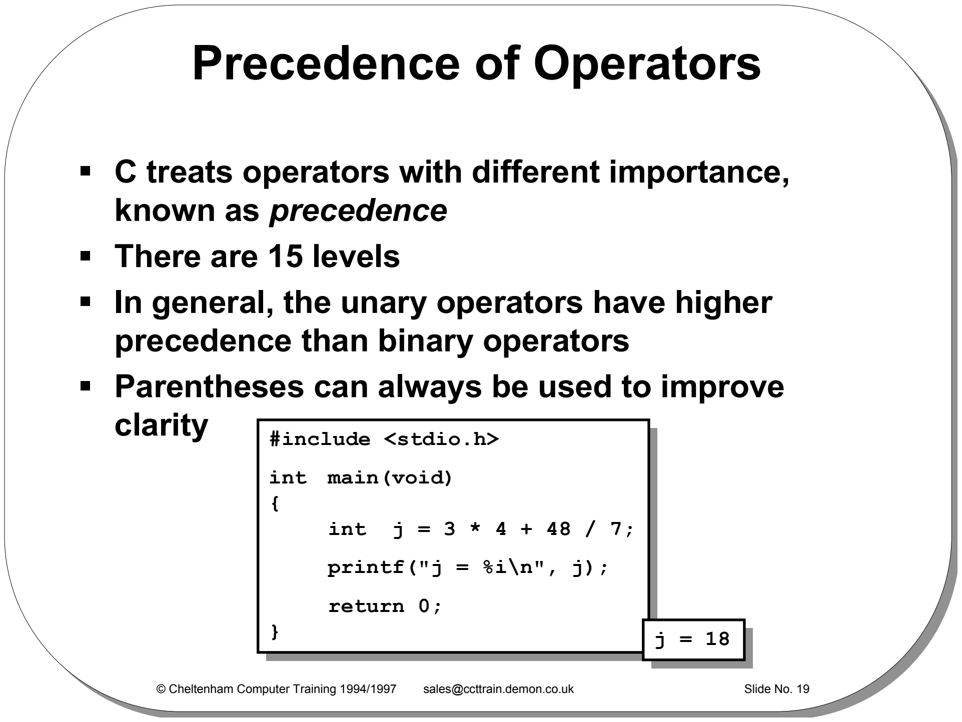
Conditional expression vs. if/then/else
This is another of those C operators that you must take at face value and decide whether to ever use it. If you feel *if* then *else* is clearer and more maintainable, use it. One place where this operator is useful is with pluralisation, for example:

```
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

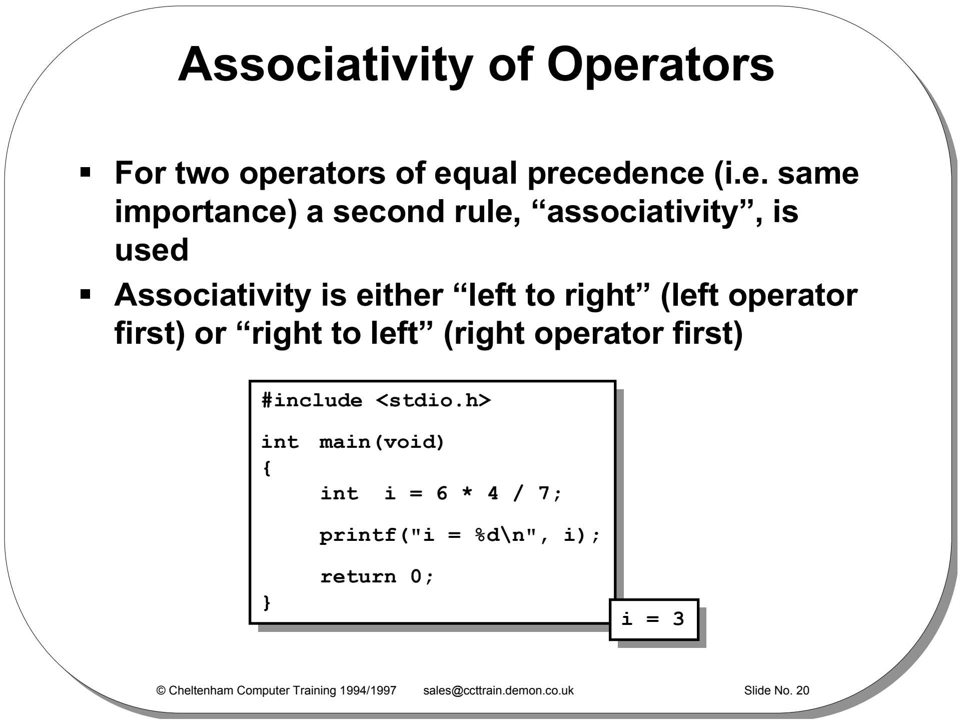
`3 * 4 + 48 / 7`

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

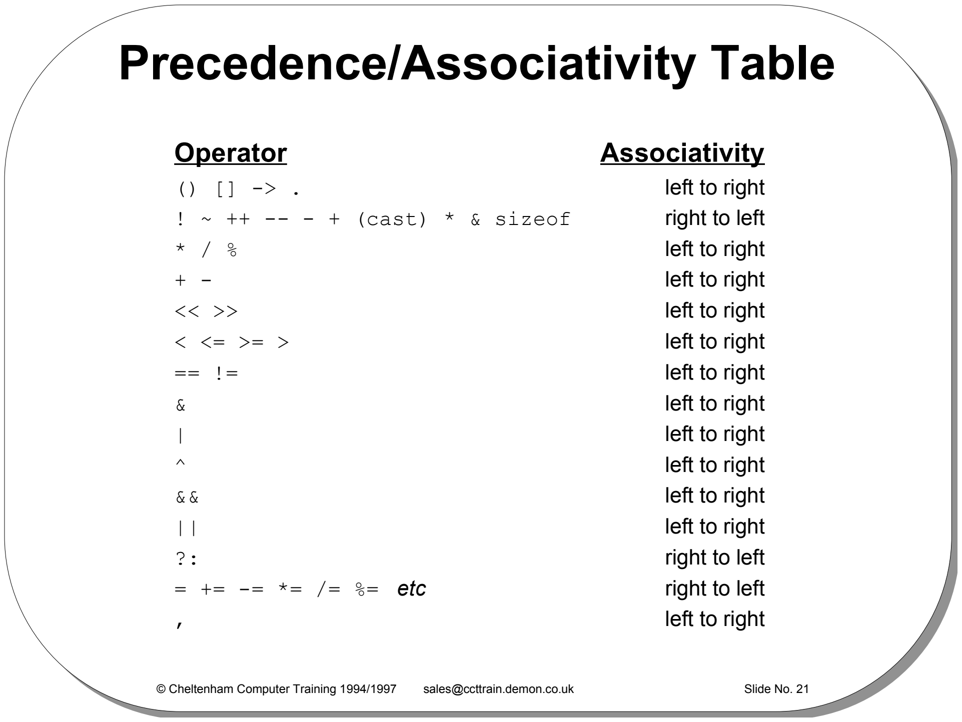
The right most operator is done first, then the left.

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".

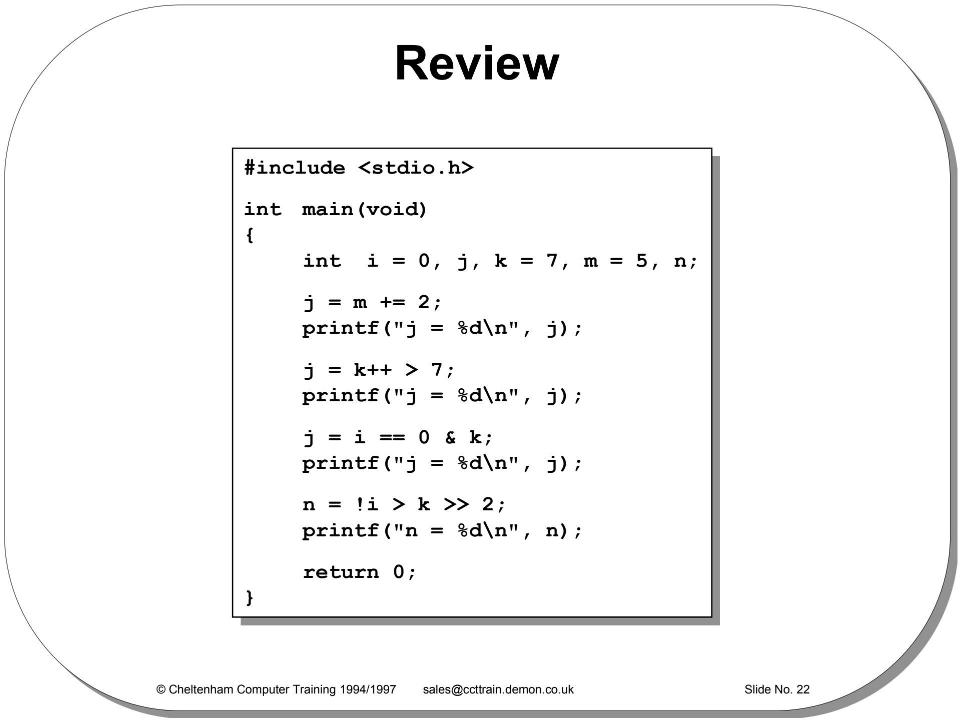


Precedence/Associativity Table

Operator	**Associativity**
`() [] -> .`	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;
}
```