C++ Elementary Data Types

Computers use memory to store all sorts of information

- integers
- reals
- characters

These are called elementary data types.

If the value stored in a particular location can be changed, it is called a variable.

If it can never change, it is a constant.

High level programming languages name these locations using identifiers.

Let's see what sort of names C++ allows.

Identifiers

- a name used to represent a memory location used to represent constants, variables (and functions)

- C++ allows a sequence of
  letters – upper & lower case
  digits – 0 to 9
  and
  underscore (_)

- it should start with a letter (_ can be used but avoid – it's for system use)
- avoid names with two consecutive underscores

and these reserved words:

```
and         const       false       not         signed      typeid
and_eq      const_cast  float       not_eq      sizeof      typename
asm         continue    for         operator    static      union
auto        default     friend      or          static_cast unsigned
bitand      delete      goto        or_eq       struct      using
bitor       do          if          private     switch      virtual
bool        double      inline      protected   template    void
break       dynamic_cast int        public      this        volatile
case        else        long        register    throw       wchar_t
catch       enum        mutable     reinterpret_cast        true while
char        explicit    namespace   return      try         xor
class       export      new         short       typedef     xor_eq
compl       extern
```

Most importantly
select names that mean something!!!

```
age
lvalue
miles_per_quart
Fred's height
Celsius
max-so-far
float
LitresPerKm
π
FlOzperkm
```

- consider that identifiers can be any length (sometimes limited to 256)
  – don't dare use a name anywhere near that long

- C++ names are case-sensitive

```
dollar
Dollar
dOlLaR
```

  are all different identifiers

Suppose we want an identifier to represent a student's age.

We can't use

```
student's age
```

or

```
students age
```

but can use

```
StudentAge
```

or

```
student_age
```

Integers

C++ has three forms of integer storage:

`short`

`int`

`long`

These vary in the number of bytes used for storage – implementation-dependent.

ANSI standard has conditions.

- `short` integers must be at least 16 bits
- `long` integers must be at least 32 bits
- `short` integers cannot be longer than the type `int`
- `long` integer cannot be shorter than the type `int`

g++ :

`short` is 16 bits

`long` is 32 bits

`int` is `long`

What should you use?

`int`

unless

particular sized integers are to be stored

use `long` if numbers exceed 32767
use `short` if space is at a premium
or ?

C++ integer constants

a string of digits with optional sign

```
12        -345        +5678
```

If leading digit is 0 –> octal constant

If led by 0x (or 0X) –> hex constant

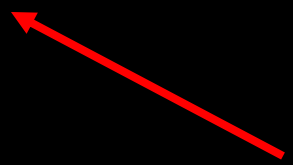
```
077   0124   044   0xff   0xAe4
```

If we want to name a memory location for use as the storage position of an integer variable declare the identifier as

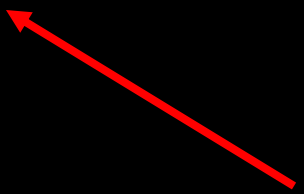
an integer variable

in a variable declaration

```
type    variable-identifier-list;
```



a basic type



a list of identifiers

```
int       distance, height1, employee_no;
short     age, IQ;
long      phone_no, Salary;
```

The keyword `int` can be included after `short` or `long` (but is usually left out).

```
short int street_no, calories;
long int TaxFileNo, serialnumber;
```

Note the use of tabs or spaces to separate the type from the list – be consistent.

Reals

C++ provides 3 sizes of real storage:

```
float
double
long double
```

Again, number of bytes used is implementation-dependent.

Again, the standard specifies some rules.

- `double` is no less precise than `float`
- `long double` is no less precise than `double`

Most implementations use
32 bits for a `float`
64 bits for a `double`

g++ uses the above and
96 bits for a `long double`.

Precision:

number of digits of accuracy

- size of the mantissa

Magnitude:

how big/small a number

- range of the exponent

Real variables are declared in a similar way to integer ones.

```
float Temperature, height_in_feet;
double theta, RadPerDegree;
```

As with integers, choose the type of declaration of any variables depending on the precision required (or the magnitude).

C++ provides two ways of representing real constants:

- **fixed point**
  - **an integer part**
  - **a decimal point**
  - **a fractional part**

  (integer or fractional part can be absent but not both)

```
23.5    0.345    50.    .5
```

C++ provides two ways of representing real constants:

- floating point or scientific notation
  - an integer constant

  or
  - fixed point real constant

  followed by
  - `e` or `E`

  and
  - an integer exponent (+ or -)

```
0.12e5    .12E5    12e3    12.e3    12000e-1
```

Important fact:

C++ always considers real constants to be

```
double
```

(unless)

Often, it is easier (and more explanatory) to use a name to represent a constant value.

Universal constants like π and e are examples.

C++ allows the programmer to specify a name for a constant – a memory location whose contents never change.

Use keyword `const`

So we have

```
const type varname = value;
```

The `value` must be given when the constant is declared.

```
// pi - ratio of circumference to diameter
    const double pi = 3.14159265;
// e - the natural logarithm base
    const double e = 2.71828182864;
```

In fact any value which cannot change can be specified as a named constant.

```
const int Year=1996;  // current year
const short days_in_week = 7,
      minutes_in_hour = 60;
```

Any identifier specified as a named constant will be watched by the compiler.

If it appears in an instruction likely to change its value, the compiler will report an error.

Named constants are often used to enable easy modification of programs.

```
/*
This program is designed to calculate
the duration of a home loan.
If the interest rate changes please
modify the next line.
*/
     const double InterestRate = 0.105;
```

Initialisation

Any variable can be given a starting value when declared – not just constants.

Just specify the initial value when declared.

```
int i = 0;
double Altitude = 100.3;
```

Without the specifier `const` the identifier's value can be altered.

Logical Values

These now form part of the C++ standard.

Only two possible values:

`true` and `false`

These are the constants.

The variable type is called `bool`.

```
bool IsMale;
const bool use_debug = false;
```

So how is such a value stored?

'usually' in one byte:

- 0 representing `false`
- 1 representing `true`

Characters

a character constant consists of a character placed between a pair of single quotes

```
'a'    'F'    '+'    '3'
```

What about ' – the character?

Use an escape sequence

```
'\''
```

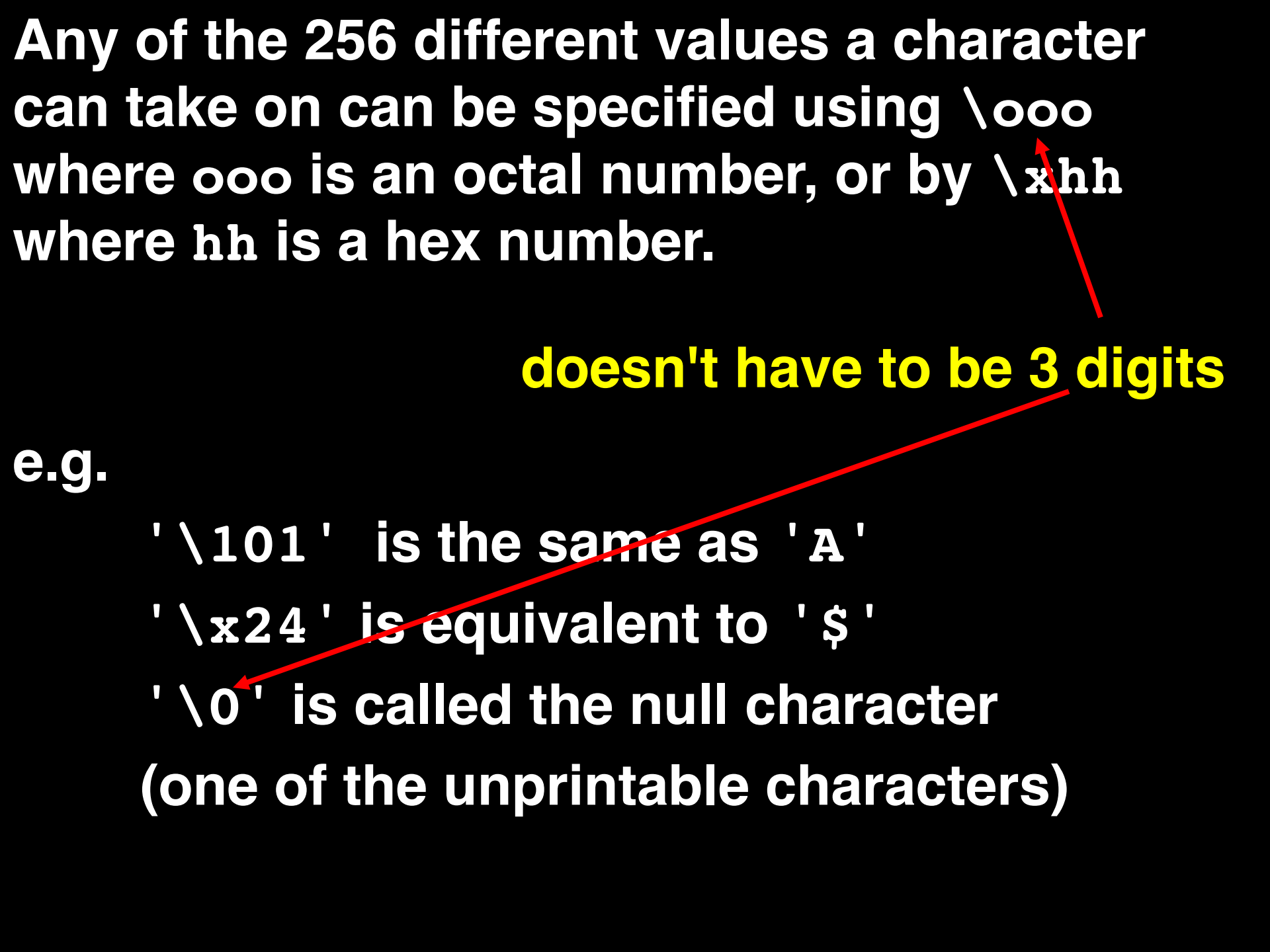
What about \ ?

```
'\\'
```

Other escape sequences

new-line	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
double quote	\"
backslash	\\
single quote	\'
question mark	\?

Any of the 256 different values a character can take on can be specified using `\ooo` where `ooo` is an octal number, or by `\xhh` where `hh` is a hex number.



e.g.

`'\101'` is the same as `'A'`

`'\x24'` is equivalent to `'$'`

`'\0'` is called the null character

(one of the unprintable characters)

Character variables and named constants are declared using

```
const char dollar_sign = '$',
          backspace = '\b';
char FirstInitial;
```

Note that, in fact, a character variable is just a one-byte integer – and can be used as such.

If the top bit is treated as a sign bit, then a character variable can store values from -128 to 127.

If not, the values range from 0 to 255.

Whether the top bit is a sign bit is implementation-dependent.

Linux considers it a sign bit.

Thus, we can assume a character variable is a

```
signed char
```

Thus there must be an

```
unsigned char
```

In fact all integer types can be qualified by the keyword `unsigned` to make them non-negative.

```
unsigned short I;
```

allows values in the range 0 to 65535 to be stored in the variable `I`.

This is one way of ensuring that variables which should never be negative cannot become negative without causing an error.

Another form of character storage is the

character string

This consists of one or more characters (including escape sequences) between pairs of double quotes.

```
"The answer is "
"University of Wollongong"
"Go to the next line\n"
```

Each character occupies a byte.

So that the computer knows where the string ends, the memory used by the string is terminated by a byte containing the value 0, (or `'\0'`) called a null character.

Thus `"The String"` occupies 11 bytes.

How do we get a " in a string? `\"`

The string is not a basic C++ data type.

We cannot declare string variables or named constants (not just yet).

We will see later how to do it.

We'll even see how C++ can create a string type, so we better call this a C-string.

We can certainly use C-string constants like

```
"My name is"
"This is the END"
```

What's the use of being able to store numbers and characters in a computer?

To manipulate them to produce further values.

We need operators.

Let's start with numerical operators.

C++ provides five basic arithmetic operators:

+	addition, unary plus
–	subtraction, unary minus
*	multiplication
/	division
%	modulus (remainder of int division)

When we combine one or more of these operators with variables and constants, we get arithmetic expressions.

Let's look at some simple expressions.

unary +

does nothing

+ ***expr***

has the same value as *expr*

unary –
reverses the sign of the expression following the symbol

– *expr*

has the magnitude of *expr* but the opposing sign.

+ and – and the other 3 operators usually appear as binary operators between two operands

What's a unary minus?

If we write
-23
then the computer can store the value
-23 in memory.

If we say
`-x`
we mean "*fetch the value in the variable* `x` *and reverse its sign*"

When used in a binary expression, the five operators perform the usual mathematical processes on the two operands.

Care must be taken concerning the type of the two operands.

If both operands are of the same type, the operation is generally easy to explain by the usual mathematical result – of the same type.

Suppose we have

```
int x = 3, y = 4;
float z = 2.5;
```

Then

```
x + y
```

is of integer type and has the value 7

```
x + z
```

is of real type and has the value 5.5

```
x/y
```

is of integer type and has value 0

In particular, consider the last two operators, / and %, when acting on two integer operands.

Integer division yields integer results.

5/2

has the value 2 not 2.5.

That is, the result is truncated to the type of the operands.

The remainder from that division is given by the modulus operator `%` so that

```
5%2
```

has the value 1.

It is an error to use `%` with real operands, and the compiler will report such an expression as illegal syntax.

C++ declares that

```
(x/y)*y + x%y
```

is always equal to the value of `x`.

This may seem obvious, but is not so obvious if one of the operands is negative.

Suppose `x` and `y` have the values -5 and 2 respectively.

What are the values of

```
x/y
```

and

```
x%y
```

One oft-used definition requires the remainder of a division to always be non-negative.

Thus `x%y` i.e. `(-5)%2` must be 1, and `x/y` has to be -3

An alternative approach is that division ignores the sign, finds the magnitude, and then puts the sign back on the result.

Thus `x/y` is `(-5)/2` or -2, so that `x%y` must be -1

In both cases `(x/y)*y + x%y` is the same as `x`

The C++ standard does not prescribe which of the two alternatives (or any other) is correct.

It specifies the expression's equivalence.

The g++ compiler uses the latter of the two alternatives:
if one of the two operands of `%` is negative, the result is negative.

If the right operand of `/` or `%` is zero, the result is an exception called

division by zero

This causes the program to fail when it is executed.
It is not detected on compilation.

There are also problems of a similar nature if the result of an operation is not a valid value for the type of the expression.

Consider the expression

```
x + y
```

where `x` and `y` are both of type (`signed`) `short` with values 32767 and 2.

The arithmetic result is 32769
– not a valid `short`

This is an exception called overflow.

This is not an exception which causes the program to terminate. Integer overflows merely drift into the sign bit.

Thus the result is interpreted as
–32767
which shares its bit pattern with 32769.

If the answer to an expression exceeds 65535, then only the remainder modulo 65536 is stored, and interpreted as a negative value if the sign bit is then 1.

Of course, expressions involving unsigned integers always produces results modulo 2^n where n is the number of bits in the representation.

There is no overflow of unsigned integer types.

The same sorts of problems occur when expressions involving reals exceed their range.

Consider the expression

```
u * v
```

when `u` and `v` are both of type `float` with the values `1e30`.

The arithmetic result `1e60` is not storable as a float.

An overflow occurs. The IEEE standard says the answer is INF.

If both values are 1e-30, with an arithmetic result of 1e-60, we get an underflow, but the result is just set to 0.

If the denominator is 0 in a real division, and the numerator is non-zero, then the result is also INF with the sign of the numerator.

If both operands are zero, the result is indeterminate so we get NAN.

These special values ±INF and NAN do not terminate the execution of a program.

They can continue to be used in expressions although NAN will always yield NAN results.

Mixed Mode Expressions

What if the operands of a binary arithmetic operator are of different types?

Such **mixed mode** expressions are prohibited in some languages such as Modula-2, but not in C++.

When encountered, the operands' values are converted to one type
– called **propagation**.

For mixed integer expressions
  if not `long`, both converted to `int`
  unless one is `unsigned` –> `unsigned int`
  unless it would fit in an `int` –> `int`

An expression involving a `float` and a `double` –> all `double` and a `double` result

If a real and an integer type are involved, the integer is converted to the real type with a possible loss of precision.

You're probably confused.

Then avoid mixed mode expressions.

Later we'll see how to specifically cause type conversions so that expressions are not mixed.

One situation to be prepared for is the mixture of variables and constants in expressions.

Constants have an assumed type.

You can force a constant to a specific type by appending

`f` (or `F`) to force a real to be `float`

`l` (or `L`) to force an integer to be `long` or a real to be `long double`

This can often avoid under/overflows or save space.

So far expressions have involved one operator and one or two operands.

But most expressions are more complex.

```
5.*(F-32.)/9.
```

involves 4 operands, 3 operators and some other symbols.

How is this expression evaluated?

expr1 *op1* *expr2* *op2* *expr3*

Only two possible orders

op1 **first then** ***op2***

or

op2 **then** ***op1***

The order is determined by precedence.

Operations are ordered by the operator:

`*` and `/` and `%`

before

`+` and `-`

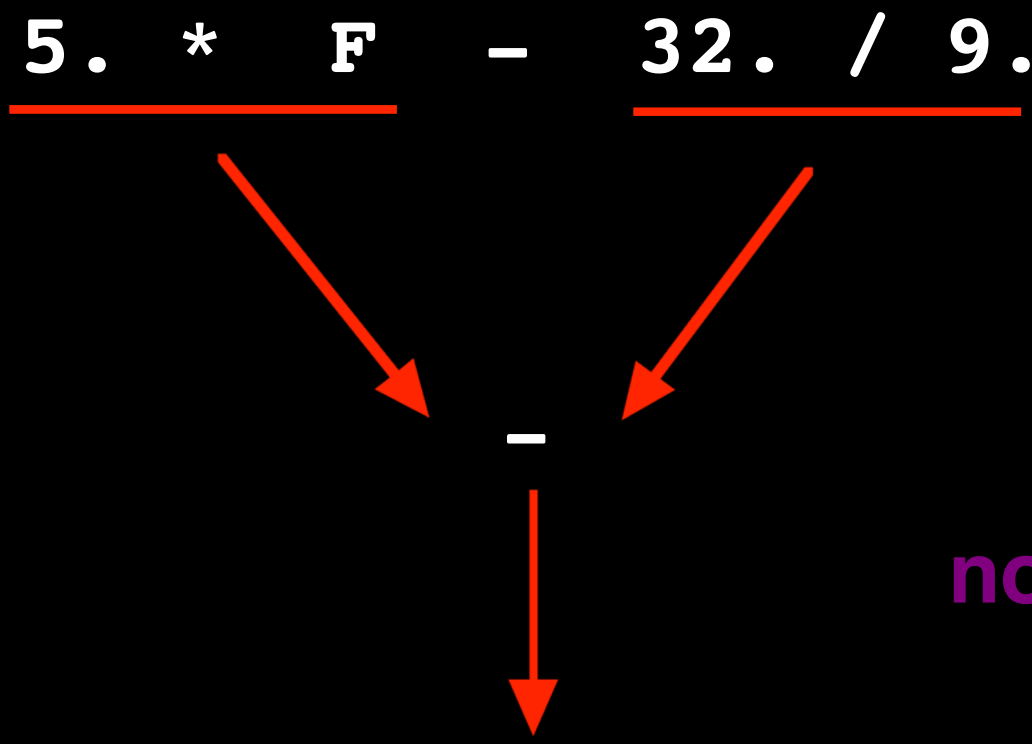
Two operands of the same precedence are performed left to right.

Unary operators take precedence.

So

```
5. * F - 32. / 9.
```

is performed in the following order:



not quite

If this is not the order required, use (and) around the operation we want done first.

```
5. * (F - 32.) / 9.
```

means the subtraction is performed first.

Let's have a look at some traps for the unwary.

```
5/9*a_double
```

is always zero.

```
a_double*5/9
```

or

```
5./9*a_double
```

```
x / y * z
```

is the same as

```
x * z / y
```

and not

```
x / (y * z)
```

which would produce that result.

Note that in an expression

expr1 *op* *expr2*

there is no specification in C++ as to the order of calculation of the two expressions *expr1* and *expr2*.

We will return to this later.

One traditional arithmetic operator should be conspicuous by its absence:

exponentiation
taking one value to a power

C++ has no such operator.

Low integer powers –> multiplication

More general –> later.

One of the most important of C++'s operators is

the assignment operator =

This allows the programmer to provide an identifier, whether constant or variable, with a value merely by saying

```
identifier = expression
```

What does this mean?

identifier = expression

- calculate the value of ***expression***
- store the value in the memory location given the name `identifier`

This may involve a type conversion if the two operands are of different types.

For example

```
Area_of_circle = pi*radius*radius
```

- fetch the value of `pi` (possibly a named constant)
- multiply that value by the value found at the memory location `radius`
- multiply the result by that same value of `radius`
- store the result in the memory location called `Area_of_circle`

Thus

do not interpret = to mean

"is equal to"

rather say

"is assigned the value of"

or

"becomes"

```
a = b
```

`a` and `b` may not be equal to each other at the time of the operation, nor after it.

But `a` will have a value 'appropriately near' the value of `b`.

Because = is an operator, the term

```
identifier = expr
```

has a value!

It is the value (and type) of the left operand after the operation.

This means we can say

```
ident1 = ident2 = expr
```

Note that, for the operator =

the right operand has to be evaluated first, and

the left operand has to be an identifier, not an expression.

Initialisation is a form of assignment. As such, it can contain arithmetic expressions as well as constants.

e.g.

```
const double cm_in_inch = 2.54;
const double inch_in_cm = 1./cm_in_inch;
```

Many advocates of C++ advise that any declaration of a variable should involve initialisation (just like a named constant).

This may force declarations well into the body of a program, mixed in with calculations and other control structures.

This is contrary to traditional C, which requires all declarations prior to any other operations.

It is a matter of style to require C++ programs to adhere to this concept.

I'm a fan of the C convention, and expect you to follow it.

Note:

A constant expression is defined as involving only numeric constants or expressions, which themselves involve only constants and named constants initialised with constant expressions.

– a circular definition!!!

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
xCoord = (R1 + R2) * R2
```

`xcoord` becomes 15.0
expression has the value 15.0

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
xCoord = (R2 + R1 / R3) * 2
```

`xCoord` becomes `7`
expression has the value 7

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
xCoord = I1 / I2 + 5
```

`xCoord` becomes 6.0
expression has the value 6.0

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
I2  = I1 / I2 + 5
```

`I2` becomes `6`
expression has the value 6

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
Symbol = 4
```

`Symbol` becomes '\4'
expression has the value '\4'

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
Symbol = Numeral
```

`Symbol` becomes '2'
expression has the value '2'

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
Symbol = R3
```

`Symbol` becomes '\4'
expression has the value '\4'

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
R1 = Numeral
```

`R1` becomes 50.0
expression has the value 50.0

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
I3 = 1 + Numeral
```

`I3` becomes 51
expression has the value 51

Suppose `R1`, `R2`, `R3` and `xCoord` are reals with `R1` = 2.0, `R2` = 3.0 and `R3` = 4.0; `I1` and `I2` are integer variables with `I1` = 8 and `I2` = 5; `Numeral` and `Symbol` are character variables with `Numeral` = '2'.

What are the results of this expression?

```
Symbol = R3 = Numeral*2
```

`R3` becomes 100.0
`Symbol` becomes 'd'
expression has the value 'd'

Other forms of assignment

Often, the variable on the left of an assignment is merely being added to on the right-hand-side.

e.g.

```
sum = sum + x

counter = counter + 1

divisor = divisor * 2
```

These expressions also confirm the difference between assignment and equality.

```
sum = sum + x
```

does not imply that `x` is zero.

C++ provides a shorthand for performing these assignments.

variable = variable op expression

can be written as

```
variable op= expression
```

So the previous examples become

```
sum += x

counter += 1

divisor *= 2
```

no space between op and =

C++ provides two special unary operators for the express purpose of incrementing and decrementing a variable by 1.

++ for incrementing
-- for decrementing

These operators also have a special property that the two unary operators + and - do not have
they can appear before
or after the variable!!

The expression

```
++variable
```

increments the value stored at *`variable`* by 1. Thus it is essentially the same as

```
variable += 1
```

and in fact has the same expression value (the resulting value of *`variable`*).

But it saves on a `=`.

```
sum += (++x)
```

increments `x`, then adds the new value to `sum` producing a new value of `sum`.

Thus it is the same as

```
sum = sum + (x = x + 1)
```

As a post operator, the incrementing is still performed on the variable, but the expression has the old value of the variable.

```
sum = 0.;
x = 5.;
sum += (x++);
```

would result in `sum` being 5 but `x` being 6.

Decrement (--) works analogously.

Danger...danger...danger !!!

When is the increment done?

Standard C++ does not specify the order of calculation of individual operands within a single operator expression.

Side-effects (such as increment and decrement) which occur separately from the calculation of the operands can only be guaranteed to have been done upon . .

```
(x++) - (--x)
```

has an indeterminate value.

Cannot determine which expression is evaluated first and when the post-increment is performed.

Avoid using a variable in an expression with the same variable in sub-expressions with side-effects.

```
float x = 23.5;
int i = 1;

x++;          // x is now 24.5
++x;          // x is now 25.5
i = x++;      // x is 26.5, i is 25
x = ++i;      // i is 26, x is 26.0
```