MA400: Financial Mathematics

Introductory Course

Lecture 2: Introduction

What's in a name?

Our program will have to work with variables (and constants).
More advanced programs may also require functions.

- What makes a valid name?
- What names are invalid?
- What makes a good name?

The “Alphabet” of the C++ language

Only a subset of the ASCII characters are available to you to use in a C++ program.

```
0 1 2 3 4 5 6 7 8 9

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

(   )   [   ]   {   }

+   -   *   %   <   =   >

!   ?   .   ,   :   ;   "   '

#   &   /   |   \   ^   ~   _
```

Identifiers: What makes a valid name?

Valid identifiers can only be constructed from:

- the letters a-z and A-Z,
- the digits 0-9, and
- the underscore symbol, _.

That is: no other symbols, or white space, are allowed.

Identifiers can only start with:

- a character: a - z, A - Z, or
- an underscore: _ .

That is: they cannot start with a digit.

Identifiers: What makes a valid name?

There are no constraints on the size of your identifiers, so carefully chosen identifiers can help your code be **self-documenting**. E.g.

`S0` versus `initial_price`.

Identifiers are **case sensitive**, so that the following are distinct:

strike_price, strike_Price, strikePrice, sTrIkEpRiCe.

Try to pick a consistent method for your names.

While the following identifiers are valid:

_Gamma , standard__deviation , K_ ,

you are strongly advised to avoid following such naming conventions using the underscore.

Identifiers: What names are invalid?

However, some perfectly formed names are still invalid - these are **keywords** reserved by the language for particular things. E.g.:

- data types
- built-in functions

See Capper, p. 12, for a complete list of such keywords.

Identifiers: Excluded keywords

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_case	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

http://tinyurl.com/cpp-keywords

Identifiers: Additional keywords

The following alternatives for logical operators have also been introduced for readability

and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

Fundamental types

C++ is a **statically typed** language.

- All variables and constants must be of a pre-declared type.
- These are checked at compilation time (static).

Each type determines:

- what these variables can store, and
- what storage is required for them.

The following types exist in C++:

- 4 fundamental types, defined internally,
- various *derived* types, and
- user-defined types.

Fundamental Types

Also known as **built-in** or **standard types**.

These include the **integral data types**

- Integer: **short**, **int** and **long**;
- Boolean: (**true** or **false**);
- Character: **char**, **signed char** and **unsigned char**.

The floating point data types:

- Floating-point: **float**, **double** and **long double**.

These 4 types are referred to as the **arithmetic types**.

The last fundamental type is:

- **void**: an empty set of values.

Derived types

In addition to the fundamental types, there are:

- Arrays;
- Pointers;
- References.

There are also the **user-defined types**:

- Enumerations
- Data structures and classes

Limits on the fundamental data types

Each data type comes with restrictions:

- how much storage they require,
- their possible range of values

These vary between systems and compilers.

Limits for the fundamental types can be found in:

- **limits.h**, for the integral data types;
- **float.h**, for the floating point data types.

These are found in `C:\Dev-Cpp\include`.

These values can be access from within a program by adding:

```
#include <climits>     // limits for integral types
#include <cfloat>      // limits for floating types
```

Defining variables

Before a variable can be used, it must be declared. In doing so:

- the variable's type is specified, and
- an appropriate amount of memory is reserved.

```
int i;
double temp;
bool exam_pass;
char response;

int j, row, floor;
double x, gamma;

int n, m, double u;    // WRONG: this is meaningless
double y, z, y;        // WRONG: y already declared
```

The assignment operator: =

We assign values to variables using the assignment operator =.

```
temp = 100.00;
```

This is distinct from an equality operator (which is, in fact, ==).

We can perform multiple assignments (of different data types) on one line:

```
x = 5.0, gamma = 3.425, i = 1;
```

However, this may make your code less readable, so you may want to avoid this.

Variables can be declared and initialized at the same time;

```
int n = 1, m = 2;
```

The **char** data types

There are actually 3 character data types:

- **char**
- **unsigned char**
- **signed char**

A single character of the compiler's character set is stored in 1 byte.

Single quotes are used to assign characters:

```
char c1 = 'C';
char c2 = c3 = '+';
cout << c1 << c2 << c3 << '\n';
```

They are distinct from **strings**, which use double quotes:

```
"Hello, world!\n"
```

Escape characters

name	C++ name
newline	`\n`
horizontal tab	`\t`
vertical tab	`\v`
backspace	`\b`
carriage return	`\r`
form feed	`\f`
alert	`\a`
backslash	`\\`
question mark	`\?`
single quote	`\'`
double quote	`\"`

The **bool** data type

A Boolean (variable), with data type **bool**, can have one of two values: **true** or **false**.

```
bool a = true;
bool b = false;
```

Booleans only take up 1 byte of storage.

We are able to convert from boolean form to integer form:

- **false** takes the value 0;
- **true** takes the value 1.

Similary, integers can be converted into booleans:

- 0 takes the value **false**;
- Any non-zero integer takes the value **true**.

As **bool** types are integral data types, they obey the rules of integer arithmetic.

The integer data types

The integer types have:

- 3 forms: (plain) **int**, **short int** and **long int**;
- 2 parities: **signed** and **unsigned**.

parity	name	size	min	max
(signed)	short	2	-32768	32767
unsigned			0	65535
(signed)	int	4	-2147483648	2147483647
unsigned			0	4294967295
(signed)	long	4	-2147483648	2147483647
unsigned			0	4294967295

By default, the integer types are signed: thus **signed int**, **signed long** and **signed short** are redundant.

Limits on the integers

Where do these ranges come from?

$$1 \text{ byte} = 8 \text{ bits}$$

So for **short** integers

$$2 \text{ bytes} = 16 \text{ bits}$$

$$2^{16} = 65526 \text{ ways of choosing 0's and 1's.}$$

While for **int** and **long** integers

$$4 \text{ bytes} = 32 \text{ bits}$$

$$2^{32} = 4294967296 \text{ ways of choosing 0's and 1's.}$$

Integer overflow and underflow

Immediately we should notice that C++ can only handle integers within a finite (though apparently large) range.

What happens at the edges of this range?

For the case of a (**signed**) **short** integer, we can see results such as:

$$32767 + 1 = -32768.$$

This is known as **integer overflow**.

Similarly, for an (**signed**) **long** integer we might get

$$-2147483648 - 1 = 2147483647.$$

This is known as **integer underflow**.

Unsigned integers and modulo arithmetic

For unsigned integers, we do not get underflow and overflow.

However, these integers obey arithmetic modulo 2^n, where n represents the number of bits in the representation:

- $2^{16} = 65526$ for **unsigned short**;
- $2^{32} = 4294967296$ for **unsigned int** and **unsigned long**.

These may seem quite large, but note that,

$$8! < 2^{16} = 65526 < 9!$$

$$12! < 2^{32} = 4294967296 < 13!$$

Using integer data types

Due to their limitations, it is probably best to avoid using the **short** and **unsigned** integral types unless you have a good reason to do so.

Declaring and initializing integers

As with all variables, integer variables need to be declared before they are used.

```
int a = 1, b, c;
b = 2;
```

Note that white spaces are not allowed in integer declarations.

```
m = 1000;       // Correct
m = 1 000;      // WRONG: no white spaces allowed
```

Commas are valid, but can give strange results

```
m = 1,000,000;  // Assigns m the value 1
```

Initializing variables

Be careful that you initialize all variables before you use them!

For example:

```
int a = 1, b, c;
b = 2;
cout << a << '\t' << b << '\t' << c << '\n';
```

produces:

```
1         2         -1881141193
```

What is going on here?

Uninitialized variables

When we make a declaration of the form:

```
int a = 1 , b , c;
```

The compiler:

- assigns three addresses in memory to a, b and c of 4 bytes;
- places the value 1 in the memory assigned to variable a.

Unless told to, it does nothing to the values already stored at that address.

```
b = 2;
```

tells the compiler to store 2 at the address assigned to b.

```
cout << a << '\t' << b << '\t' << c << '\n';
```

tells the compiler to output whatever is stored at the addresses reserved for a, b and c. For the latter, this could be anything.

Declarations inside a program

Note, C++ does not require all definitions to be at the start of a program.

It simply requires that identifiers be defined before they are used.

Therefore, it is probably best to only define identifiers when they can be initialized.

Basic arithmetic operators

As well as the assignment operator, =, C++ comes provides familiar arithmetic operators for the integral data types:

- addition: denoted by the token +
- subtraction: denoted by the token -
- multiplication: denoted by the token *

These can be used as you expect: binary operators that act upon two integer operands, and result in an integer.

```
a = 7 + 3;      // Assigns a the value 10
b = 8 - 10;     // Assigns (signed) b the value -2
c = 3 * 6;      // Assigns c the value 18
```

There are also two binary operators relating to division:

- integer division: denoted by the token /
- integer modulo, or remainder: denoted by the token %

Division operator: /

Integer division is a binary operator that requires two integer operands and whose result is an integer.

But what happens for non-trivial rationals, such as 3/2?

If the integers `i` and `j`:

- are both positive integers, then `i / j` returns a truncated integer.
  E.g. `3 / 2` returns 1 (truncated from 1.5).
- are both negative integers, then `i/j` returns a truncated integer.
  E.g. `-5 / -2` returns 2 (truncated from 2.5).
- are of opposite sign, then the result is dependent on the compiler.
  E.g. `-3 / 2` may result in either -2 or -1.

Integer Modulus, or Remainder, Operator: %

Integer modulus is a binary operator which requires two integer operands and whose result is the integer remainder of dividing the first operand by the second.

If the integers `i` and `j`:

- are both positive integers, then `i % j` returns the remainder from dividing `i` by `j`.
  E.g. `7 % 4` returns 3.
- are both negative integers, then the result is negative.
  E.g. `-7 % -4` returns -3.
- are of opposite sign then the result is dependent on the compiler, but consistent with:

```
i = (i / j) * j + i % j
```

Addition and subtraction as unary operators

$+$ and - can also be **unary operators** that require one integer operand and whose result is one integer.

```
a = 2;     // Assigns a the value 2
b = -a;    // Assigns b the value -2

c = -b;    // Assigns c the value 2
d = +b;    // Assigns d the value -2, NOT 2!!
```

In particular, the unary operator + does not make a negative number positive, it is simply equivalent to multiplying by +1. Thus, the unary $+$ operator, while valid, is largely redundant.

Increment and decrement operators: ++ and −−

Consider the statement:

```
i = i + 1;     // Increments the value of i by 1
```

This is used so often, C++ provides the unary increment operators, ++. This requires a single integer operand and its result is an integer.

However, the single operand can occur on either side of the operator, with slightly different results:

```
++i;           // Prefix increment operator
i++;           // Postfix increment operator
```

Both have the same final result of advancing the integer i by one, but they apply these at different times:

- either before i is used in the calling statement (prefix),
- or after it is used (postfix).

Increment and decrement operators: $++$ and $--$

Similarly, we have the unary decrement operator, $--$.

```
j = j - 1;    // The following are equivalent to this
--j;          // Prefix decrement operator
j++;          // Postfix decrement operator
```

Increment and decrement operators: ++ and −−

Consider the following code fragment:

```
i = 1;
j = 2;
k = 3;

i = ++i;            // Assigns 2 to i
                    // (2, 2, 3)
j = j++;            // Assigns 3 to j
                    // (2, 3, 3)
k = k + ++i;        // Assigns 3 to i THEN 6 to k
                    // (3, 3, 6)
k = k + j++;        // Assigns 9 to k THEN 4 to j
                    // (3, 4, 9)
```

Ambiguity and maximal munch

The previous 'program' contained the statement:

```
k = k + ++i;
```

What if we had written the (quite valid) statement:

```
k = k+++i;
```

How does the compiler resolve this?

It adopts a **maximal munch** strategy: when parsing it will take the largest sequence of characters that form a valid token. Thus, the above statement is in fact equal to:

```
k = k++ + i;
```

This is clearly not the same as the first statement.

Exercise: How would this affect the final values of the previous 'program'?

Additional assignment operators: += and -=

It is quite common to see variants of the following:

```
i = i + k;
j = j - k;
a = a * b;
p = p / q;
r = r % s;
```

C++ provides additional operators for this:

```
i += k;
j -= k;
a *= b;
p /= q;
r %= s;
```

Associativity and precedence of operators

Consider the following expression:

```
k = i + j / m * n - k;
```

How is this interpreted by the compiler?

The compiler interprets this according to two rules:

- operator **precedence**: which determines which operators in an expression are applied first;
- operator **associativity**: which determines in what order operators of equal precedence are applied.

Associativity and precedence of operators

++	postfix increment	right to left
-- --	postfix decrement	right to left
++	prefix increment	right to left
-- --	prefix decrement	right to left
−	unary minus	right to left
*	multiplication	left to right
/	division	left to right
%	modulo	left to right
+	addition	left to right
−	subtraction	left to right
=	assignment	right to left
+=	add and assign	right to left
−=	subtract and assign	right to left
*=	multiply and assign	right to left
/=	divide and assign	right to left
%=	modulo and assign	right to left

Associativity and precedence of operators

So, for example, associativity allows us to make sense of the following code fragment:

```
i = j = k = 1;
```

Since associativity of the assignment operator is right to left, the compiler:

- assigns `k` the value 1;
- assigns `j` the value of `k`, which is 1;
- assigns `i` the value of `j`, which is 1.

Associativity and precedence of operators

```
k = i + j / m * n - k;
```

Precedence tells us that the division and multiplication is applied first, followed by the addition and subtraction.
However:

- which of the division and multiplication is applied first?
- which of the addition and subtraction is applied first?

Associativity tells us that:

- $*$ and / are applied from left to right, and
- + and − are applied from left to right.

The above statement is then interpreted as:

```
k = ( ( i + ( ( j / m ) * n ) ) - k );
```

Using white spaces and parentheses

While associativity and precedence allow one to write very compact code, you may prefer to use parentheses and spacing to make your code more readable.

That is you may find

```
k = ( ( i + ( ( j / m ) * n ) ) - k );
```

more helpful than:

```
k = i + j / m * n - k;
```

The floating point data types

The final group of arithmetic data types is that containing the floating point types:

- **float**
- **double**
- **long double**

name	size	min	max	accuracy
float	4	10^{-38}	10^{38}	7
double	8	10^{-308}	10^{308}	16
long double	12	10^{-4932}	10^{4932}	19

```
float pi = 3.142F;
double pi = 3.1415926535897932;
long double pi = 3.14159265358979323855L;
```

Note that `f` and `l` could be used instead of `F` and `L`.
Long doubles **require** the suffix.

Defining a floating point number

A general floating point number consists of three parts:

- the integer part: 2
- the fractional part: 997295
- the exponent: 8

Thus for the speed of light

$$2.997295 \times 10^8$$

```
double c;
c = 2.997925e8
```

Here `E` could be used instead of `e`.
Note that `f` and `l` still need to be used where appropriate.

Defining a floating point number

Note that every floating point requires either a decimal point or an exponent.

```
x = 1.0;
y = 0.1;
z = 1e10;
u = 1.1e10;
v = 11e9;
w = .11e11;
```

Invalid floating point constants include:

```
.e9        // WRONG: no integer or fractional part
1,000.0    // WRONG: an embedded comma is not permitted
1 000.0    // WRONG: an embedded space is not permitted
1000       // WRONG: this is an integer constant
e10        // WRONG: no integer part
```

Arithmetic operators for floating point data types

Except for the modulus operator, all the integer arithmetic operators are valid for the floating point data types.

```
double x, y, z;      // Declaration/definition
x = 3.1416;          // Assignment
y = -x;              // Unary minus
z = -3.1416;         // Unary minus
++x;                 // Increment (prefix version)
y--;                 // Decrement (postfix version)
x = x - 1.0;         // Subtraction
y = y + 1.0;         // Addition
z = x * y;           // Multiplication
z = z / 3.1416;      // Division
```

Exercise: What is the final result for z in the above code fragment?

Division by zero

Be careful that you do not implicitly (or explicitly) divide by zero in your program.

IEEE standard C++ compilers will flag the results of such an operation as NaNs (Not A Number), which then propagate through your program.

Your compiler may come back with warnings, but it is not sufficient to break compilation.

Defining constants

C++ allows variables to be defined as constants, using the **const** specifier.
Once they have been defined, they cannot be changed.

```
const double speed_of_light = 2.9979e8;
```

Note that because they cannot be changed once defined, they **must** be initialized at declaration.

```
const double c;      // c cannot be uninitialized
c = 2.9979e8;        // cannot change const
```

In fact, the compiler should prevent you from defining an uninitialized constant.

Changing types

Note that the C++ compiler can automatically change types.

```
double x;
float y;
int i;
long j;
i = 2;
x = 1 + 3.7;  // + has integer and double operands
y = 3.7;      // = has float and double operands
j = i;        // = has long and int operands
```

The C++ compiler has a set of rules for conversion between types, see Capper, p.35.
However, you can avoid reliance on this by careful programming.

Break and exercise

If `x` has type **double**, why do

```
x = ( 1.0 + 1 ) / 2;
```

and

```
x = 0.5 + 1 / 2;
```

give different results?
What are they?

Exercise: Solving a quadratic equation

Write a program that will:

- allow the user to specify an arbitrary quadratic equation, and

- return the roots of this quadratic equation.

Exercise: Solving a quadratic equation

$$ax^2 + bx + c = 0 \tag{1}$$

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

$$x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{3}$$

Exercise: Solving a quadratic equation

```
// Solves the quadratic equation: ax^2 + bx + c = 0
#include <iostream>    // For input/output streams
#include <cmath>       // For sqrt() function
using namespace std;   // Make all std names global
int main()
{
  double a, b, c;
  cout << "Enter the coefficients a, b, c: ";
  cin >> a >> b >> c;
  double root_delta = sqrt(b * b - 4.0 * a * c);
  double x_1 = 0.5 * ( -b - root_delta ) / a;
  double x_2 = 0.5 * ( -b + root_delta ) / a;
  cout << "The solutions are " << x_1;
  cout << " and " << x_2 << "\n";
  system("PAUSE");
  return(0);
}
```

Exercise: Solving a quadratic equation

```
Enter the coefficients a, b, c: _1 3 2_
The solutions are -2 and -1
Press any key to continue . . .
```