Language Elements

C++ is one of a number of well-known computer *languages*, about which we may say:

1. They genuinely are languages.
2. The good news is they have a very small, closed vocabulary.
3. The bad news is they are much fussier about grammar than a grade eight english teacher!

The purpose of computer languages is for human programmers to write out detailed instructions for computers to carry out.

Our objective is to get you familiar enough with C++ to write meaningful sets of iinstructions for computers.

There are two aspects to this:

Syntax
: The computer engineering term for *grammar*. You have to form your instructions correctly, or they will be rejected by the compiler.

Semantics
: The computer engineering term for *meaning*. *My dog flies books* is a well formed English sentence, whose meaning is at least suspect. The compiler is a grammar cop only. It is incapable of checking for meaning.

Fermez la porte means the same in French as its English counterpart *Close the door*. There is no equivalent in C++.

Computer languages are very specialized. They are designed to tell computers what to do. To use them effectively (to make them meaningful) you will have to learn something about how computers work.

So, throughout the course, there will be twin threads, woven together: **Syntax**—get the grammar right. **Semantics**—make it meaningful.

Initially, we will focus on syntax.

Some Language Elements

Alphabet

Uppercase & lowercase roman letters, A-Z and a-z, the digits 0-9, a well defined set of symbols available on every standard keyboard, e.g.


```
< > { } ( ) : ; , . ? / * & + - ! ^ ' " = | _
```

Some non-printing characters, the most common of which are the newline (`\n`) and tab (`\t`) characters

Any continuous combination of newline, tab and/or spaces are collectively known as **whitespace**

Words

The computer Engineering term is actually *token*.

Simply put

1. Tokens contain no whitespace
2. Operators are tokens
3. Words with no whitespace are tokens

So the following are all C++ tokens: `while window { x George +`

The following *expression* contains five tokens: `(x1+b)`

They are `(` `x1` `+` `b` `)`

You can see why the more specialised term token was adopted. `while` `window` `George` and `x` look like words but `{` and `+` don't

There are effectively three kinds of words

keywords
: words defined for the language. Basically, its vocabulary

identifiers
: words created by the programmer as *names* for things.

symbols
: Sometimes single as `{` `+` `*` and sometimes in pairs as `!=` or `/*`

Keywords are sometimes called *reserved words* as programmers may not use them as identifiers. They are *reserved* for the language.

Sentences

The equivalent of a sentence is a statement. Statements are always terminated by a semicolon.

```
x = y + z - 4;
cout << "Hello world!\n";
```

Paragraphs

Again we have a different technical term. The C++ equivalent of a paragraph is a block—a set of statements enclosed in a pair of curly brackets { }

We can turn the above statements into a block as follows:

```
{
   x = y + z - 4;
   cout << "Hello world!\n";
}
```

It is considered *good style* to indent statements inside a block.

Some Specifics

commented_hello.cpp

```
/*************************************************************
 * Memorial University of Newfoundland
 * Engineering 2420 Structured Programming
 *
 * hello.cpp -- The simplest C++ program.
 *
 * Author: Dennis Peters
 *   Date: 2000.01.11
 *
 *************************************************************/
#include <iostream>
using namespace std;

/*************************************************************
 * main
 *
 * Parameters: none
 * Modifies: cout -- outputs a message
 *
 * Returns: 0
 *************************************************************/
int main() {
    cout << "hello world!" << endl;
    return 0;
}

/*************************************************************
 *                     REVISION HISTORY
 *
 * REV 01
 * date: 2003.12.19
 * by: mpbl
 * description: some minor style editing
 *
 *
 *************************************************************/
```


Comments

Comments are for people. The precompiler actually throws them away

They are a major mechanism for documenting programs.

They come in two flavours:

```
//              A single line comment. Ends at the end of
   /*
   * Coments enclosed in slash-asterix ... asterix-slash ca
   * extend over several lines
   */
```

Includes

Pre-processor directive—tells the compiler to insert code from another file (typically declarations of other functions).

```
  #include <iostream>
```

Code from the iostream file is inserted exactly as if you had typed it in yourself.

using namespace

tells the compiler where to look for definitions that aren't defined in our program.

```
using namespace std;
```

main function definition

Every program must have a `main` function. It is the starting point of program execution. This is what it looks like

```
int main() {
     // the function body goes here
  }
```

When execution reaches the end of `main`, the program terminates.

We're going to introduce our first syntax definition here:

Function returning value:

Form:
`returnType` *Identifier* **`(`** *ParameterList* **`) {`**
Statement
`}`
Example:

```
int square(int num){
    return num * num;
}
```

Interpretation: the square of x is computed and the result returned.

`main` is actually a function returning value, which means it fits this form.

Both `main` and `square` return an integer value (`int`). In the case of `square` the value returned (the output) represents the square of `num`, the value input.

In the case of `main`, the return value is actually passed to the operating system after the program is run. Normally it will be zero, meaning your program ran successfully. Other numbers represent codes for various kinds of errors. In this course we will generally return just 0;

The *identifier* represents the name of the function (`main` or `square`)

The *parameter list* (which goes between parentheses) represents the arguments passed into the function. Although it is possible to pass arguments into `main` we won't be doing so in this course, so our version of `main` has an *empty parameter list*, `()`.

The Return Statement

The example in our **function returning value** syntax definition includes one type of return statement, so we need to define its syntax as well.

Return statement:

Form:
`return` *Expression ;*

Example:


```
return num * num;
```

Interpretation: the *Expression* (num x num) is computed then return causes execution to jump back to the point of the function call, substituting the value of *Expression* for the function call..

There's a lot going on here. For example we haven't really defined what an *Expression* is yet. We're kind of at a chicken-and-egg stage. Don't worry, we'll get it all filled in over the next few lectures.

Here's our program again

commented_hello.cpp

```
/*************************************************************************
 * Memorial University of Newfoundland
 * Engineering 2420 Structured Programming
 *
 * hello.cpp -- The simplest C++ program.
 *
 * Author: Dennis Peters
 *   Date: 2000.01.11
 *
 *************************************************************************/
#include <iostream>
using namespace std;

/*************************************************************************
 * main
 *
 * Parameters: none
 * Modifies: cout -- outputs a message
 *
 * Returns: 0
 *************************************************************************/
int main() {
    cout << "hello world!" << endl;
    return 0;
}

/*************************************************************************
 *                      REVISION HISTORY
 *
 * REV 01
 * date: 2003.12.19
 * by: mpbl
 * description: some minor style editing
 *
 *
 *************************************************************************/
```

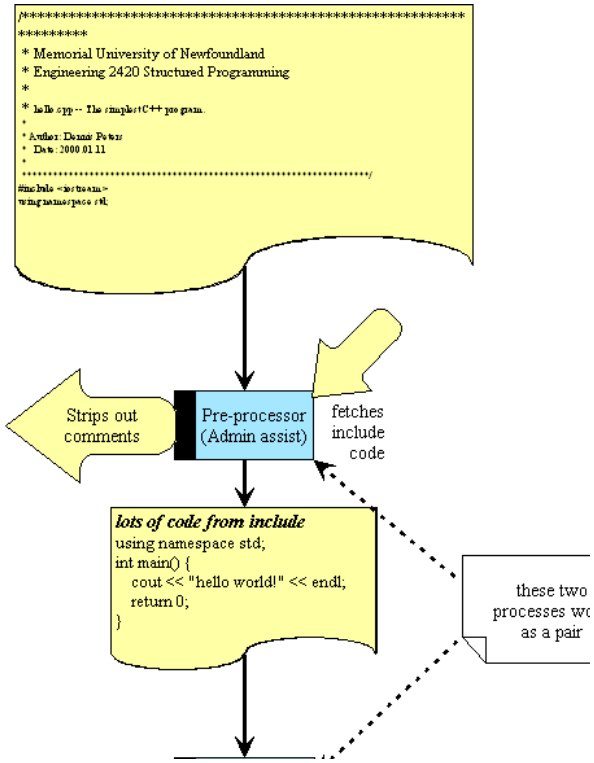
This drawing looks at just the pre-compile / compile process.

The two work invisibly as a pair. It looks like one process to the programmer.

Any instruction that begins with a hash mark (#) is actually an instruction to the pre-compiler, the programmer's automated administrative assistant.

Here the assistant strips out all the comments and fetches the code from the `iostream` *header file*

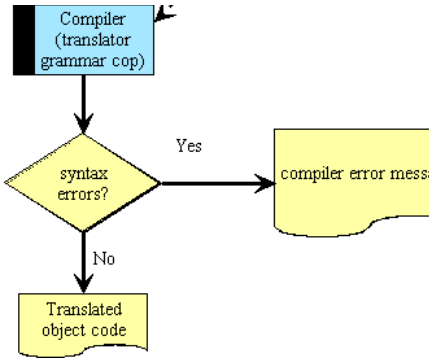
The resulting code the



compiler gets is much reduced *except that*all code from the header file has been added. We don't show it because it's too complex.

The compiler checks the reultant code for syntax errors. If it finds any it will generate a report

Only if the code is grammatically correct will an output file be generated.



Literals

The program above includes the following line:

```
cout << "hello world!"<< endl;
```

`"hello world!"` is known as a literal. Although there are many kinds of literals in C++ we will focus on four for now:

integer literals
: standard integers such as **`1`**, **`2`**, **`-17`** or **`2056`**

double literals
: real number literals using decimal notation (**`3.`**, **`3.0`**, **`3.14159`** or **`-17.65`**) or floating point (exponential) notation (**`1.0e11`** or **`-23.6e-3`**)

character literals
: single characters such as **`'a'`**, **`'x'`**, **`'H'`**, **`'!'`** or **`'\n'`**

string literals
a sequence of characters such as **`"hello world!\n"`**, **`"Michael"`** or **`"The quick fox jumped over the lazy dog."`**

Note that single characters are always enclosed by a single quote while strings are always enclosed in double quotes.

Thus

```
cout << 'hello world!'<< endl;
```

would result in a *syntax error*.

Executable Statements

Output: Stream Insertion Operator

`cout << expression;` —output the value of expression to the *standard output* (screen) stream.

```
cout << "Hello world!" << endl;
```

`<<` is a left associative operator -- expressions are ouput left-to-right.

`endl` causes an newline (`'\n'`) character to be output.

Assignment Statement

Store a value in a variable:

```
x = expression;
```

Compute the value of expression and store it in x.

expression may contain x —the 'old' value is used:

`x = x + 1;` —Increase the value in x by 1.

Always think of the = as a replacement operator

```
x <- x+1;
```

Input: Stream Extraction Operator

`cin >> x;` —Read a value from the *standard input* stream (usually the keyboard) and store it in the variable named x.

`cin` is an *identifier* for the *standard input* stream (keyboard) .

1. Assigns to variables left to right order.
2. What can be input depends on the data type of the variable.
3. whitespace (tab, space, newline) is skipped.
4. The reading marker keeps track of the next character to be read.

String Expressions

Objects of class string store sequences of characters:

```
string bookTitle;
bookTitle = "Programming in C++";
```

Strings can be joined with `'+'`.

printname.cpp

```
/*************************************************************
 * Memorial University of Newfoundland
 * Engineering 2420 Structured Programming
 * printname.cpp -- Demonstrate string expressions.
 *
 *
 * Author: Dennis Peters
 * Date: 2000.01.15
 *
 *************************************************************/
#include <iostream>
#include <string>
using namespace std;

const string FIRST = "Dennis";      // My first name
const string LAST = "Peters";       // My last name
const char INITIAL = 'K';           // My initial

/*************************************************************
 * main
 *
 * Parameters: none
 * Modifies: cout -- outputs the name in various forms.
 *
 * Returns: 0
 *************************************************************/
int
main()
{
  string firstLast;                 // Name in First Last format

  firstLast = FIRST + " " + LAST;
  cout << "My name is: " << firstLast << endl;

  string lastFirst;                 // Name in Last, First format
  lastFirst = LAST + ", " + FIRST;
  cout << "in last, first format: " << lastFirst;
  cout << ", and with my initial: " << lastFirst << " "
```

```
      << INITIAL << "." << endl;

  return 0;
}
/*************************************************************
 *
 *
 *
 *
 *                     REVISION HISTORY
 *
 *
 *
 *************************************************************/
```

Arithmetic Expressions

Numeric operators: `+`, `-`, `*`, `/`, `%`

Integer division

If the operands are integers `/` give the integer part:

`13/5` is 2.

`%` gives the remainder:

`13%5` is 3.

This page last updated on Thursday, January 8, 2004