C++ Programming

C++ code is essentially a collection of statements terminated by a semicolon, such as (spaces not needed):

```
a = b + c;
```

Most C++ code is organized into "header files" and "cpp files," i.e., C++ files. A header file, such as *mycode.h*, essentially gives a brief overview of the code that is written in the cpp file *mycode.cpp*. At the very top of any file, regardless of whether it is a header file or cpp file, are compiler directives such as "include-statements." Compiler directives are identified by `#`, such as

```
#include <stdio.h>
#include "mycode.h"
```

where the referenced files contain statements that are included by the compiler thanks to these include-statements. Angle brackets cause the compiler to search for the file in folders specified by include-type environment variables. Conversely, double quotation marks mean that the compiler first searches the directory where the file with the include statement is located. The double quotation marks can also be used to specify the exact path to the included file.

Methods

In computer programming, the concept of "subroutines" is a basic one. Most beginners understand that it is smart to organize the code into subroutines to avoid keeping all the code in one place. In C++ these subroutines are called "functions" or "methods" and they will later be referred to as "member functions" in the context of object-oriented programming. In C++, a method is declared in the header file as follows:

```
int methodName(double b, double &c, double *d);
```

The first word can be `void`, `int`, `double`, etc. It states the type of the output of the method. If void, the method does not return any value and the implementation of the function simply ends with `return;`. Otherwise, the implementation of the method is terminated by `return value;`. Often, the int output type is employed as a flag that reveals how the computations in the method progressed. For example, a negative result is returned when the computations were unsuccessful. Next, the parenthesis of the declaration contains the argument list, which can also be void. The variables in the argument list are utilized to input and output values to/from the method. The sum total of return type, method name, and argument list is called the signature of the method. Methods with the same name but different overall signature, called overloading, are not uncommon.

Arguments are either passed-by-value or passed-by-reference. The former is the default and means that a copy of the variable is passed and the original parameter remains untouched. The latter is indicated by the ampersand, `&`, see above, and means that the variable's memory address is passed, which implies that the original value is easily

changed within the function. A third option is to pass pointers, which also represent memory addresses, as described shortly. Passing memory addresses instead of data is usually more efficient, but more dangerous because the passed data can be changed within the method, unless it is prefaced by `const` to make it read-only.

A method can be called in various ways, depending on the location of the call relative to where the method is implemented. If the method is implemented in the same cpp file, then it can be called like this:

```
a = methodName(b,c,d);
```

If a method is implemented in a class (see separate document on object-oriented programming) for which an instance is available, the method can be called like this:

```
a = theClassInstance.methodName(b,c,d);
```

If a method is implemented in a class for which a pointer to an instance is available, then the method is called like this:

```
a = theClassInstance->methodName(b,c,d);
```

Every C++ program automatically starts with a call to the "main method," which must always be implemented in a file named *main.cpp*. The main method needs no declaration and usually no *main.h* file exists. A basic implementation if the *main.cpp* file can be like this:

```
#include <iostream>
int main()
{
    std::cout << "Hello World!";
}
```

Variable Declarations

In "interpreted" languages, such as Matlab, it is usually unnecessary to declare variables before they are used. In other words, a statement like `a=2` is sufficient to allocate, name, and initialize a piece of the computer's memory. Compiled languages like C++ require more. In addition to the include-statements mentioned above, each variable that is used in the code must be declared. The declarations are of the form

```
int a=2;
double b=3.0;
```

Specifying an initial value is optional. The word `const` before the declaration means the variable is read-only. Variable types include int, double, bool, char, string, short, long, and float of which the first few are most common.

Pointers

A pointer contains a memory address and is declared by a the asterisk:

```
double *a=0;
```

Initializing the pointer to zero, as above, is optional but common; it facilitates a check of whether it has been given an actual address: `if (a==0) {...}`. The pointer declaration above does not actually assign any memory; that is done with a "new" statement:

```
a = new double;
```

The two statements above can be combined into one:

```
double *a = new double;
```

The variable value that is stored at the pointer's memory address is set by employing the asterisk symbol to "dereference" the pointer:

```
*a = 5.0;
```

Dereferencing is necessary because the statement

```
a = 5.0;
```

would actually increase the memory address by 5, with unpredictable and bad result. Dereferencing is also used to carry out mathematical operations with the variables that are stored in pointers:

```
double x = (*a)+2.0;
```

where, without the asterisk it would be the memory address that would increase by two. Pointers can lead to "memory leaks" if they are not deleted. Every "new" statement must be accompanied by a "delete" statement later in the code, to release the memory that was occupied by the pointer:

```
delete a;
```

Reference variables are similar to pointers; they contain the memory address of another parameter:

```
double &b = c;
```

where *b* is a reference variable that now contain the memory address of *c*.

Type Conversion

It is clear from above that C++ requires each variable to be declared before it is used in computations. However, if it makes sense the type can be converted to facilitate a computation. For example, an integer can be cast as a double to facilitate a computation:

```
double a = 2.0;
int    b = 2;
double c = a+(double)b;
```

Standard Libraries

In C++, the basic algebraic operations (+, -, *, /) are always available. Additional mathematical functionality, such as sqrt and the trigonometric functions, is added by

```
#include <math.h>
```

This is one example of a standard library, namely a set of functions that are available in most compiler environments without the need for compiling and adding external libraries.

External Libraries

Many programmers spend time on developing libraries of code that can be used by other developers. One example is the GNU Scientific Library (GSL), which provides a host of powerful mathematical operations once the library is included. The inclusion of such external libraries is a great way to increase the capabilities of a program, but it also has a downside. The inclusion of external libraries in various developer environments, such as Qt Creator and Xcode on Mac, or Qt Creator and Visual Studio on Windows, is error prone and requires expertise that is sometimes beyond the beginners level. For that reason it is usually best to include such libraries in optional parts of the code that may or may not be used by the beginner developer.

Structured Programming

Regardless of whether a computer code is organized by "procedural programming" or "object-oriented programming" the concepts of structured programming are employed. Structured programming utilizes "control structures" such as if-statements, for-loops, and while-blocks to carry out the right operations in the right order. An important distinction of structured programming is that it abolishes the "go-to" statements of earlier languages. Academic papers have been written to prove that every possible eventuality is covered by the constructs of structured programming, without the need for go-to statements. Go-to statements have been the source of many and tricky bugs.

If

```
if (a<b) {
   c=1;
}
else if (a>b)
   c=2;
}
else {
   c=3;
}
```

For

```
for (int i=0; i<n; i++) {
   a++;
}
```

While

```
while (a<b) {
   a++;
}
```

Switch

```
switch (a) {
   case 5  : b=1;
             c=1;
             break;
   case 10 : b=2;
             c=3;
             break;
   default : b=0;
             c=0;
}
```

Logical Operators

The control structures depend on logical expression to check if an operation should proceed. A logical expression employs the logical operators listed below to return either true or false. For example, $a>b$ is true if a is greater than b.

>	greater than
<	less than
==	equal to
!=	not equal to
>=	greater than or equal to
<=	less than or equal to
&&	and
||	or

Vectors and Matrices

In engineering computations, vectors and matrices are omnipresent. However, vectors and matrices are not a core part of C++. The closest is "arrays" and they can indeed be used to store vectors and matrices. However, while arrays are computationally efficient, they do not even deliver basic functionality such as giving the size upon request. For that reason it may be better to make use of the "vector" class that is part of the standard C++ library. Both arrays and vectors are described briefly in the following.

Arrays

An array of integers is declared like this:

```
int a[3]
```

It can be initialized in the same line as it is declared:

```
int a[3] = {1,2,3};
```

When utilizing this vector it is important to note that in C++ the indexing starts at zero:

```
b=a[0]+a[1]+a[2];
```

A matrix of double-precision real numbers is declared as follows:

```
double c[3][3];
```

Std::Vector

To use vectors, the following include statement is necessary:

```
#include <vector>
```

It is also possible to say "include namespace std" immediately thereafter, but this approach is not adopted here to avoid the possibility of conflict between constructs in the standard library and the developer's convention. Instead, "std::" is used, so that a vector (either a pointer or not) is declared like this:

```
std::vector<double> *vp = new std::vector<double>(10);
std::vector<double> v(10);
```

Similarly, a pointer to a matrix is declared as follows (notice the space between > and >):

```
std::vector< std::vector<int> > *mp = new . . .
   std::vector< std::vector<int> >(10, std::vector<int>(12));
std::vector< std::vector<int> > m(10, std::vector<int>(12));
```

The elements of the vectors and matrices are automatically initialized to zero. Other values can be inserted like this:

```
(*vp)[3] = 19.0;
v[3] = 18.0;
(*mp)[2][3] = 19.0;
m[2][3] = 18.0;
```

The elements are read like this:

```
double a = (*vp)[3];
double b = v[3];
double c = (*mp)[2][3];
double d = m[2][3];
```