C++ Functions

In Java, we refer to the fields and methods of a class. In C++, we use the terms data members and member functions. Furthermore, in Java, every method must be inside some class. In contrast, a C++ program can also include free functions - functions that are not inside any class.

Every C++ program must include one free function named main. This is the function that executes when you run the program.

Here's a simple example C++ program:

```
# include <iostream>

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

Things to note:

1. Function main should have return type int; you should return zero to indicate normal termination and any non-zero value to indicate that an error occurred (e.g., bad data was read, an attempt to open a non-existent file was made).
2. Function main is not required to have any parameters. However, if you intend your program to be run with command-line arguments, you should declare main as follows:

```
int main( int numargs, char *args[])
```

The first parameter is the number of command-line arguments, including the name of the executable itself (more about executable files below). The second parameter is an array of C-style strings (a C-style string is a sequence of characters, terminated with a special null character). Note that, like all formal parameters, you can use any names you want in place of "numargs" and "args". For historical reasons, some programmers use "argc" and "argv", but those are not very informative names.

Parameter Passing

There are different ways in which parameter data can be passed into and out of methods and functions. It is beyond the scope of these notes to describe all such schemes, so we will consider only the two most common methods used in C++ and Java: "pass by value" and "pass by reference".

First some important terminology:

Definition: Formal Parameter

A variable and its type as they appear in the prototype of the function or method.

Definition: Actual Parameter

The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Pass by Value

Using "pass by value", the data associated with the actual parameter is copied into a separate storage location assigned to the formal parameter. Any modifications to the formal parameter variable inside the called function or method affect only this separate storage location and will therefore *not* be reflected in the actual parameter in the calling environment. For example (this could be either C++ or Java):

```
void cribellum(int x) // Formal parameter is "x" and is passed by value.
{
        x = 27;
}

void someCaller()
{
        int y = 33;
        cribellum(y); // Actual parameter is "y"
        // Actual parameter "y" still has 33 at this point.
        ...
}
```

Pass by Reference

Using "pass by reference", the formal parameter receives a reference (or pointer) to the actual data in the calling environment, hence any changes to the formal parameter *are* reflected in the actual parameter in the calling environment. Here is a modified version of this example (now strictly a C++ example since Java does not allow primitive types to be passed by reference):

```
void cribellum(int& x) // Formal parameter "x" is now passed by reference.
{
        x = 27;
}

void someCaller()
{
        int y = 33;
        cribellum(y);
        // Actual parameter "y" now has 27 at this point.
        ...
}
```

Using expressions as actual parameters

When the formal parameter is passed by value, the actual parameter can be an expression. However, when the formal parameter is passed by reference, the actual parameter must refer to one specific instance of the formal parameter type stored in programmer-accessible memory. The following table illustrates valid and invalid examples.

Assume the following variables are defined in the calling environment:

```
int i = 3, y = 33;
```

Assume further that `values` is an array of length 10 of integers.

***Language**:* *prototype* → *call* ↓	***Java**:* *void cribellum(int x)*	***C++**:* *void cribellum(int x)*	***C++**:* *void cribellum(int& x)*
cribellum(2*y);	OK	OK	compilation error
cribellum(14*y + i - 10);	OK	OK	compilation error
cribellum(3);	OK	OK	compilation error
cribellum(y);	OK	OK	OK
cribellum(values[2]);	OK	OK	OK
cribellum(values[3*i - 2]);	OK	OK	OK

How do Java and C++ Compare?

Java only supports "pass by value" for primitive types, and it only supports "pass by reference" for object types (including arrays since they are objects in Java). This is why you never see type modifiers like "&" on formal parameters in Java – it only supports one way to pass parameters.

C++ supports both "pass by value" and "pass by reference" for both primitive types and object types. Hence it requires use of "&" so that the programmer can specify which of the two is desired for a given parameter.

What about arrays? In C++, conventional arrays are *not* objects. (Recall the comments in the earlier **Arrays** section.) Conventional C++ arrays can only be passed by reference, in large part due to the fact that their size is unknowable to the C++ runtime system, hence it would be impossible to make a local copy for a "pass by value" scheme. From a syntactical point of view, the "&" modifier is **not** used for formal parameter arrays in C++ since they can only be passed by reference.

Another example program (multiple functions and forward declarations)

Usually, when you write a C++ program you will write more than just the main function. Here's an example of a program with two functions:

```
# include <iostream>
using namespace std;

void print() {
    cout << "Hello world!" << endl;
}

int main() {
    print();
    return 0;
}
```

In this example, function main calls function print. Since print is a free function (not a method of some class), it is called just by using its name (not xxx.print()). It is important that the definition of print comes before the definition of main; otherwise, the compiler would be confused when it saw the call to print in main. If you do want to define main first, you must include a **forward declaration** of the print function (just the function header, followed by a semi-colon), like this:

```
# include <iostream>
using namespace std;

void print();

int main() {
    print();
    return 0;
}

void print() {
    cout << "Hello world!" << endl;
```