Methods and Pointers in C++

Methods

It is smart to organize code into subroutines to avoid keeping all the code in one long sequence. Hence, in computer programming, the concept of "subroutines" is a basic one. In Python they are called functions; in C++ they are called methods or functions, 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!";
}
```

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*.