Pointers to functions

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

```
1 void one() { cout << "One\n"; }
2 void two() { cout << "Two\n"; }
3
4
5 int main()
6 {
7         void (*fptr)(); //Declare a function pointer to voids with no params
```

```
8
9           fptr = &one; //fptr -> one
10          *fptr(); //=> one()
11
12          fptr = &two; //fptr -> two
13          *fptr(); //=> two()
14
15          return 0;
16  }
```

The above would work as expected, outputting "One" and then "Two". When setting and accessing the value of function pointers, note that the dereference and "address of" operators aren't actually necessary - usually they aren't used for the sake of simplicity, as shown in the modified snippet below:

```
1  void one() { cout << "One\n"; }
2  void two() { cout << "Two\n"; }
3
4
5  int main()
6  {
7           void (*fptr)(); //Declare a function pointer to voids with no params
8
9           fptr = one; //fptr -> one
10          fptr(); //=> one()
11
12          fptr = two; //fptr -> two
13          fptr(); //=> two()
14
15          return 0;
16  }
```

If a function pointer needs to point to functions that take parameters, the data-types of these parameters can simply be separated by commas in the function pointer declaration. Take for example the following:

```
1  void one(int a, int b) { cout << a+b << "\n"; }
2  void two(int a, int b) { cout << a*b << "\n"; }
3
4
```

```
int main()
{
        void (*fptr)(int, int); //Declare a function pointer to voids with two
int params

        fptr = one; //fptr -> one
        fptr(12, 3); //=> one(12, 3)

        fptr = two; //fptr -> two
        fptr(5, 4); //=> two(5, 3)

        return 0;
}
```

Just to make sure you understand the syntax of function pointers, I recommend trying to create a few of these very basic examples - a function pointer to functions which take in, and return, doubles for example.
We can also create arrays of function pointers. These aren't as scary as they sound, and they're simply created much like you would with an array of "normal" pointers, by specifying a number of elements in square brackets next to the pointer name. If we wanted a function pointer array which contained pointers to both functions "one" and "two" in elements of index 0 and 1, we could use the following:

```
void one(int a, int b) { cout << a+b << "\n"; }
void two(int a, int b) { cout << a*b << "\n"; }


int main()
{
        void (*fptr[2])(int, int);
        fptr[0] = one;
        fptr[1] = two;

        fptr[0](12, 3); //one(12, 3)
        fptr[1](5, 4); //two(5, 3)

        return 0;
}


Function Pointers Summary

Syntax
```

Declaring

Declare a function pointer as though you were declaring a function, except with a name like *foo instead of just foo:

```
void (*foo)(int);
```

Initializing
You can get the address of a function simply by naming it:

```
void foo();
func_pointer = foo;
```

or by prefixing the name of the function with an ampersand:

```
void foo();
func_pointer = &foo;
```

Invoking
Invoke the function pointed to just as if you were calling a function.

```
func_pointer( arg1, arg2 );
```

or you may optionally dereference the function pointer before calling the function it points to:

```
(*func_pointer)( arg1, arg2 );
```

Benefits of Function Pointers

- Function pointers provide a way of passing around instructions for how to do something
- You can write flexible functions and libraries that allow the programmer to choose behavior by passing function pointers as arguments
- This flexibility can also be achieved by using classes with virtual functions