An Introduction to Functions

Remember a function is the smallest programming module. It is the *squad* of the programming world. Each function should have

1. a small amount of code
2. a single well specified task

Thus, if we have multiple tasks to do in a program we will break it up into function modules, 1 task per function.

Since `main` is a function we've already been programming functions. Now we see how to add more functions.

Functions appear in programs in three ways:

1. Functions have to be *implemented* (or in C++ we say *defined*).
2. Functions have to be *invoked* or *called*.
3. Just like variables, functions have be declared before the can be used (called).

Implementation is just what we have been doing for `main` all along. Writing code for it.

`main` gets called automatically whenever we *run* our program. For other functions we will have to decide and specify when to call them.

Let's see an example.

complex_print.cpp

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

void printComplex(double re, double im);  // function DECLARATION


int main(){
    double re1 = 2.4;
    double im1 = 3.1;
    double re2 = -1.;
    double im2 = -2.9;

    cout << "The first number is ";
    printComplex(re1, im1);

    cout << endl << "The second number is ";
    printComplex(re2, im2);                    // function CALL

    cout << endl << "Their sum is ";
    printComplex(re1 + re2, im1 + im2);        // function CALL
```



```
    cout << endl << "Their difference is ";
    printComplex(re1 - re2, im1 - im2);        // function CALL

    cout << endl;
    return 0;
}
 void printComplex(double re, double im){    // function DEFINITION
    cout << '(' << re ;
    cout <<" + " << im;
    cout << "j)";
}
```

There's lots going on here

Let's focus now just on the *definition* or *implementation* of the function

Function Definition

complex_print.cpp

```
/*********************************************************************
 * printComplex
 *
 * Parameters: re: the real part
 *             im: the imaginary part
 * Modifies: cout -- outputs the complex no. whose
 *             real part is re and imaginary part is im
 *
 * Returns: nothing
 *********************************************************************/
void printComplex(double re, double im){    // function DEFINITION
    cout << '(' << re ;
    cout <<" + " << im;
    cout << "j)";
}
```

`printComplex` is a function that carries out an task for us without returning anything. Here is the syntax for its definition

Non-value returning function:

Form:
void *Identifier* **(** *ParameterList* **) {**
  Statement
}

Example:

```
void printInt(int i){
    cout << i;
}
```

Interpretation: whatever integer is passed in in place of i is printed to the output stream

printed to the output stream.

The `void` in front of the function name means that the function does not return any value at all.

Since it does not return anything it does not require a `return` statement.

The paramater list allows us to pass *inputs* (called *arguments*) into functions

In our `printComplex` example the parameters are `re` and `im` (which are both `doubles`)

The body of the function appears between the block operators { }

As with `main` it is a series of statements that are executed sequentially.

In general, if parameters have been passed *into* the function, the statement will do something *with* the data passed in.

In `printComplex`, whatever values for `re` and `im` are passed in when the function is called will be printed out in standard complex notation.

Function Calling

Let's look at the `main` function of our complex printing program again.

```
int main(){
    double re1 = 7.;                                    complex_print.cpp
    double im1 = 3.1;
    double re2 = -1.;
    double im2 = -2.9;

    cout << "The first number is ";
    printComplex(re1, im1);

    cout << endl << "The second number is ";
    printComplex(re2, im2);                    // function CALL

    cout << endl << "Their sum is ";
    printComplex(re1 + re2, im1 + im2);        // function CALL

    cout << endl << "Their difference is ";
    printComplex(re1 - re2, im1 - im2);        // function CALL

    cout << endl;
    return 0;
}
```

Here is the syntax of a function call:

function call:

Form:
Identifier **(** *ArgumentList* **) ;**

Example:

```
sqrt(2*x);
```

Interpretation: the square root of 2x is computed and the result returned.

where *ArgumentList* is zero or more expressions, separated by commas.

- Argument values are assigned to the parameters in the order that they appear.
- If the function is value-returning, then function call is an expression.
- If the function is `void`, then function call is a statement.

Notice that we call the `printComplex` function in three separate places

Since it returns `void`, this is equivalent to 3 statements.

This illustrates a fundamental principle of functions (and more generally program modules)

use often? implement once

Function Call Control Flow

Instructions are normally executed *sequentially*. We call this a *flow of control*.

Control flows from one instruction to the next in a step-like sequence.

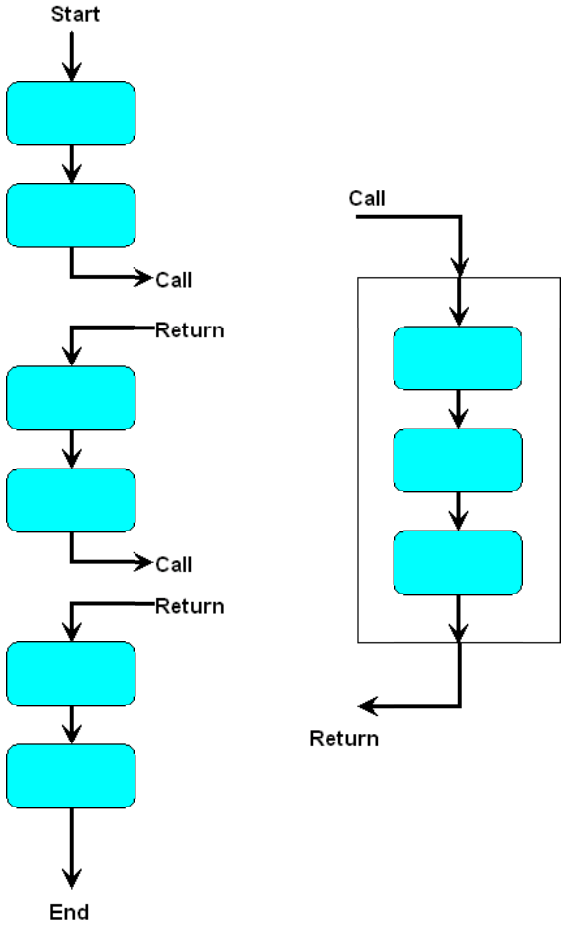
Function calls alter the normal flow of control.

When a call is executed we leave the sequence and flow over to the beginning of the function

Instructions are then executed inside the function using the normal sequential sequence.

When the instructions in the function are finished control returns to the original sequence

After the return control flow picks up where it left off.



Function as a Contract

In professional progams there are usually multiple programmers on a project (writing multiple files)

A function written by one programmer is used my many others.

When implementing a function think of it as providing a service.

Whoever calls your function is your client.

The function *prototype*, e.g.

```
void printComplex(double re, double im)
```

is the start of a contract between the service provider (programmer implementing the function) and the client (programmer using the function).

It is the client's responsibility to provide values for the parameters `re` and `im`

Here's something we see on exams.

complex_print_aargh.cpp

```
void printComplex(double re, double im){    // function DEFINITION

    // Please note, these 4 lines are wrong, Wrong, WRONG!
    cout << "Please input real part: " ;
    cin >> re;
    cout << "And imaginary part: ";
    cin >> im;
    // end of wrong, Wrong, WRONG

    cout << '(' << re ;
    cout << " + j" << im;
    cout << "j)";
```

Don't, don't, don't do this.

You are ignoring the values sent to you by the client and asking for new ones.

When you turn up at security at the airport, you are required to have a valid boarding pass.

You would be seriously annoyed if you had a valid boarding pass and security tried to sell you a new ticket.

The contract is you show up with a boarding pass and they will pass you into the boarding area.

Getting you a boarding pass is Air Canada's or CanJet's responsibility.

Getting you a boarding pass is a different function, handled in a different part of the airport!

This illustrates a second important principle

parameter values are provided by the caller

Function Declaration

Before a function can be used, it must be declared.

Just like variables, functions must be declared before they are used. There is a difference, though.

1. variables are declared *inside* functions (at the *internal level*)
2. functions are declared *outside* functions (at the *external level*)

In our complex printing example the declaration of `printComplex` appears before `main`.

complex_print.cpp

```
void printComplex(double re, double im);  // function DECLARATION
```

Note that the declaration is almost identical to the function prototype. We simply follow it with a `;` to make a declaration instead of `{ }` to create an implementation.

Why declare? It appears redundant.

Big programs are spread across multiple files.

1. Functions are implemented once only (in one file)
2. Functions are used (called) in many places (many files) so form of function must be declared before it is called.
3. Function declaration allows compiler to check that the grammar of the call is correct.

Clearly the following is in error.

complex_print_bad_grammar.cpp

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

void printComplex(double re, double im);

int main(){
    printComplex(3.2, "im part");
    return 0;
}
```


We don't actually need the implementation code to see the syntax is incorrect.

Functions Returning a Value

Consider a slightly different example.

function_1.cpp

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

/*************** functions ******************

  A program to find the minimum of a pair
  of integers.

*********************************************/

// function DECLARATION which forms an INTERFACE
int minimum(int a1, int a2);

void main(){
    int first, second;

    cout << "Input the first number: ";
    cin >> first;
    cout << "  & the second one: ";
    cin >> second;

    cout << "\nThe minimum of the two numbers is ";
    cout << minimum(first,second) << '\n';
}
```

This program inputs a pair of `ints` then tells you which is the lesser of the two

It uses a function called minimum which is declared as follows:

```
int minimum(int a1, int a2);
```

We don't see the implementation code here (normal for bigger programs) but the declaration tells us a lot:

1. The function takes two `int` arguments (syntax)
2. The function returns an `int` value (syntax)
3. The name suggests the returned value will be the minimum of the two args (style)

That's all we need to know to use the function!

The syntax is all the compiler needs to check we are using it correctly.

Functions that return values require a slightly ammended syntax definition:

Value returning function:

Form:
ReturnType Identifier **(** *ParameterList* **) {**
Statement
}

Example:

```
double sin(double x){
    //code to compute y=sin(x)
    return y;
}
```

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

The *returnType* is the type of the value being returned. e.g. `char, int, double, String`, etc.

Here's the implementation of the minimum `function`

function_1.cpp

```
ction DEFINITION (it's IMPLEMENTATION)
himum(int a1, int a2){
    if (a1 < a2)
        return a1;
    else
        return a2;
}
```

This includes an `if` statement which we haven't introduced yet, but its meaning should be clear. (We wanted to include the video for later study).

Notice we have two different `return` statements. This underlines that the `return` statement does two things.

1. It returns a value
2. It exits from the program

In fact, the value returning is optional

return statement

Forms:
`return;`
`return` *value*`;`

Examples:

```
return;
return y;
```

Interpretation: the first example is a return from a function that returns nothing (void). It simple forces an exit from the function. The second forces an exit and returns the value of y

Some programming shops decry the use of multiple returns from a function as it can make code difficult to follow and therefore maintain.

Here's an alternate version with only a single return.

function_2.cpp

```
ction DEFINITION (it's IMPLEMENTATION)
himum(int a1, int a2){
    int temp = a2;
    if (a1 < a2)
        temp = a1;
    return temp;
```

Notice, we've been able to change the *implementation* of the function without changing its *declaration*.

That is we don't change *what* the `minimum` function does only *how* it does it.

For big programs, this is a major advantage of modularization:

We can change a module without affecting the rest of our program.

Library Functions

There are thousands of functions available in dozens of *libraries*

- Don't re-invent the wheel!
- Many common functions are included in *libraries*.
- Need to `#include` the appropriate library header file for each.

The header files mostly contain the declarations needed to use the libraries

For example, the math library has declarations for

- Trigonometric functions e.g. `double sin(double x), double tan (double x)`
- Hyperbolic functions. e.g. `double atan(double x)`
- The power function which has to be used since C++ has no exponentiation operator: `double pow(double x, double exp)`
- the special function `double atan2(double x, double y)` which returns the arc tangent of `y/x`, a function which is defined for `x = 0`.

To use any of these functions you would add to the top of your file the line

```
#include <math>
```

which would instruct the pre-compiler (our administrative assistant) to insert the declarations for *all* the math functions into our file.

libfunc.cpp

```
/*************************************************************
 * Memorial University of Newfoundland
 * Engineering 2420 Structured Programming
 * libfunc.cpp -- Demonstrate using library functions.
 *
 *
 * Author: Dennis Peters
 * Date: 2001.01.27
 *
 *************************************************************/
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;


/*************************************************************
 * main
 *
 * Parameters: none
 *
 * Returns: 0
 *************************************************************/
int
main()
{
  float x; // First number input by user
  float y; // Second number input by user

  cout << "Please enter two  floating-point numbers: ";
  cin >> x >> y;
  cout << "\nTable of library functions : " << endl;
  cout << "Function\tValue" << endl;
  cout << "--------\t------" << endl;

  cout << "abs(" << int(x) << ") =\t" << abs(int(x)) << endl;
  cout << "abs(" << int(y) << ") =\t" << abs(int(y)) << endl;
```



```
  cout << "ceil(" << x << ") =\t" << ceil(x) << endl;
  cout << "ceil(" << y << ") =\t" << ceil(y) << endl;
  cout << "cos(" << x << ") =\t" << cos(x) << endl;
  cout << "cos(" << y << ") =\t" << cos(y) << endl;
  cout << "exp(" << x << ") =\t" << exp(x) << endl;
  cout << "exp(" << y << ") =\t" << exp(y) << endl;
  cout << "fabs(" << x << ") =\t" << fabs(x) << endl;
  cout << "fabs(" << y << ") =\t" << fabs(y) << endl;
  cout << "floor(" << x << ") =\t" << floor(x) << endl;
  cout << "floor(" << y << ") =\t" << floor(y) << endl;
  cout << "log(" << x << ") =\t" << log(x) << endl;
  cout << "log(" << y << ") =\t" << log(y) << endl;
  cout << "log10(" << x << ") =\t" << log10(x) << endl;
  cout << "log10(" << y << ") =\t" << log10(y) << endl;
  cout << "pow(" << x << ", " << y << ") =\t" << pow(x, y) << endl;
  cout << "pow(" << y << ", " << x << ") =\t" << pow(y, x) << endl;
  cout << "sin(" << x << ") =\t" << sin(x) << endl;
  cout << "sin(" << y << ") =\t" << sin(y) << endl;
  cout << "sqrt(" << x << ") =\t" << sqrt(x) << endl;
  cout << "sqrt(" << y << ") =\t" << sqrt(y) << endl;
  cout << "tan(" << x << ") =\t" << tan(x) << endl;
  cout << "tan(" << y << ") =\t" << tan(y) << endl;


  return 0;
}

/*************************************************************
 * Revision: 1.1
 * Date: 2001-01-27 14:49:38-03:30
 *
 *
 *                     REVISION HISTORY
 *
 *
 * By:  dpeters
 * Description: Initial revision
 *
 *
 *************************************************************/
```

This page last updated on Monday, January 19, 2004