C++ Functions

Introduction

Program organization begins to be needed when programs reach a size of about fifty lines and becomes crucial if a program is longer than a few hundred lines. Clearly a program that is thousands of lines long and has no organization will be extremely difficult for anyone but the original programmer to understand, and eventually over time will be hard even for this programmer.

The organizational method adopted by languages such as FORTRAN, BASIC, Pascal and C++ is to divide a program into subprograms, each of which performs some clearly defined task. Many languages have two kinds of subprograms, which are usually called **subroutines** (or procedures) and **functions**. In C and C++, both kinds of subprograms are called **functions**, but there are important differences between value returning functions and void (or non-value) returning functions. The former are "true functions" while the later are similar to the subroutines or procedures of other languages.

Library Functions

Technically, C++ has no **built-in functions** such as *abs()*, the absolute value function of Pascal. However, a C++ programmer has access to numerous **library functions** that serve the same purpose as the built-in functions of other languages. Below is a brief description of some of the commonly available C++ libraries.

iostream
Contains various "stream" I/O functions. See handouts C++ Stream Input and Stream Output.

math
Contains various mathematical functions, including trig functions, exponential functions and log functions.

stdlib
Contains a variety of useful functions, including searching, sorting, random number generator, conversion routines and operating system access routines.

string
Contains routines to support C/C++ strings, which are null terminated character arrays.

ctype
Routines that involve characters, such as changing case, detecting specific types of characters.

iomanip
Contains functions for altering the way data is outputted beyond the standard.

time
Functions for accessing time and date.

Designing and writing functions is an important activity that is central to the activities of a C++ programmer. Below are some simple C++ function examples.

Example 1

```
void ClearScreen(void)
{
  const NUMBER_OF_ROWS = 25;  //number of rows shown on monitor

  for ( int R = 1; R <= NUMBER_OF_ROWS; ++R)
    cout << endl;
}
```

The word "`void`" to the left of the function name tells the compiler that `ClearScreen` does not return a value. The void in the parentheses tells the compiler that `ClearScreen` has no formal parameters, i.e. no information is exchanged by `ClearScreen` and the caller of `ClearScreen`.

A typical call to this function would therefore be:

```
ClearScreen();
```

Function definitions, such as the one given above, can be physically located in various places in your source file. However, it is generally advantageous to arrange a source program with main first and other functions below. When this is done, it is necessary to **prototype** the function. For the function `ClearScreen`, this simply means putting the following line above `main()`.

```
void ClearScreen(void);   // or "void ClearScreen();"
```

Having seen a prototype, the compiler is able to check the syntax of a function call it encounters in the source file. The semicolon after the prototype is required!

Communicating with Functions

Functions such as `ClearScreen` are useful, but are limited in power. To allow functions to be more powerful, we need to be able to:

1) Send information to a function.

2) Get information back from a function.

We'll look at some more examples to see how this is done in C++.

Example 2

A program containing a function that returns the leftmost digit of an unsigned int.

In order for the function to return the leftmost digit of X, the value of X must be passed to the function. The example below shows how this is done. (example written for standard compiler)

```
#include <iostream.h>

unsigned LeftMostDigit(unsigned N);    // the function prototype

void main()
{
  unsigned X = 4321,  LeftDigit;

  LeftDigit = LeftMostDigit(X);

  cout << "The leftmost digit of " << X << " is " << LeftDigit;
}

/*******************  LeftMostDigit  ************************
  Function that returns the leftmost digit of the unsigned int N
-------------------------------------------------------------*/
unsigned LeftMostDigit(unsigned N)
{
  while ( N >= 10 )
    N = N / 10;    //divide by 10 until only leftmost digit remains

  return N;
}
```

The identifier N in the function above is called a **formal parameter**, because it is used to define the function. The variable X in `main()` is called the **actual parameter**, because it is the parameter used when `LeftMostDigit` is actually called. Note that the return value of the function is returned to the caller via the function name.

The actual parameter X is **passed by value**, which means that a copy of X is made when the function is entered. Thus, even though the algorithm "destroys" the formal parameter N, the actual parameter X remains unchanged.

Parameters can also be **passed by reference** in C++. The example below has a reference parameter `Leftmost`.

```
void FindLeftMostDigit(unsigned N, unsigned & Leftmost)
{
  while ( N >= 10)
    N = N / 10;

  Leftmost = N;
}
```

This is almost identical to the function above.

Value parameters are sometimes called **in** or **input parameters**, because they are used to send information in to a function.

Reference parameters are also called **out** or **output parameters**, because they are used to send information out of a function. A reference parameter could also be used as input to the function as well as output from the function.

Note that output parameters have an & that precede the name, while in parameters have no suffix. The default parameter passing method in C++ is to pass by value.

The C programming language follows the rule that all parameters are passed by value! This creates some inconveniences for the programmer, as the address of a variable must be passed if one wishes a function to change the value of an actual parameter.

Proper Commenting of Functions

Functions are often written in such a way that they can be reused in other programs. In this case, it is especially important to comment them properly. Below is a listing of the function `FindLeftMostDigit`, this time properly commented. **All functions written for this class should be commented in this fashion.**

```
/***************  FindLeftMostDigit  ******************************
Action: Computes and returns the leftmost digit of an unsigned int.

Parameters:

  IN:    N, integer to find leftmost digit of
  OUT:   Leftmost, holds value of leftmost digit of integer N

Returns:   Nothing
Precondition: Input N needs to be non-negative.
-------------------------------------------------------------------*/
void FindLeftMostDigit(unsigned N, unsigned & Leftmost)
{
  while ( N >= 10 )
    N = N / 10;       // get rid of rightmost digit of N

  Leftmost = N;       // Leftmost holds leftmost digit of N
}
```