Building Programs with Functions

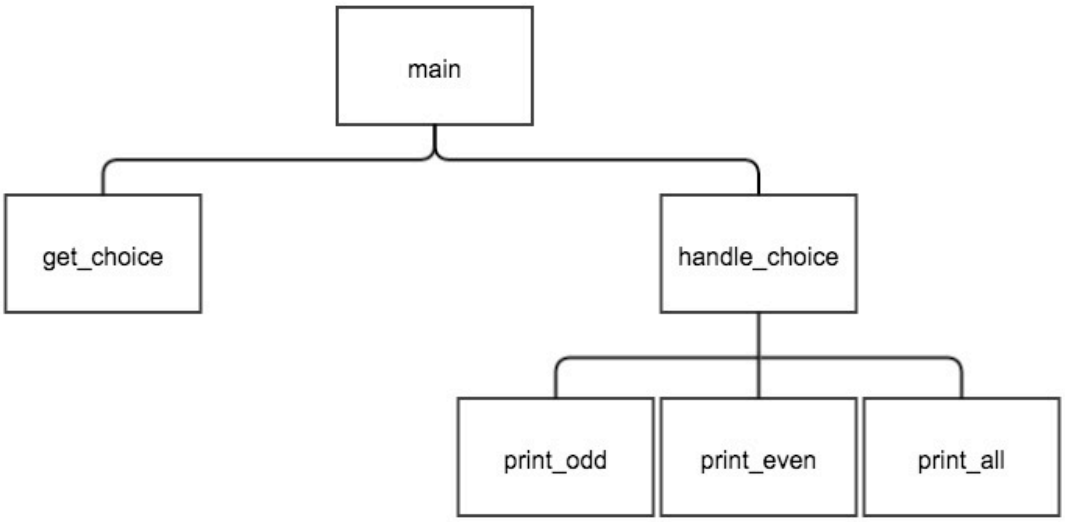
Lets take a look at a small program that is similar to our Numbers.cpp program.

```
int main()
{
     int choice;
     int i;

     do
     {
          cout << "Which sequence do you wish to display?"
          << endl;
          cout << "1 -- Odd numbers from 1 to 30" << endl;
          cout << "2 -- Even numbers from 1 to 30" << endl;
          cout << "3 -- All numbers from 1 to 30" << endl;
          cin >> choice;

          if ((choice < 1 || choice > 3))
          {
               cout << "Choice must be 1, 2, or 3" << endl;
          }
     }

     while ((choice < 1) || (choice > 3));

     switch (choice)
     {
          case 1:
               for (i = 1; i <= 30; i = i + 2)
               {cout << i << " ";}
               cout << endl;
               break;
          case 2:
               for (i = 2; i <= 30; i = i + 2)
               {cout << i << " ";}
               cout << endl;
               break;
          case 3:
               for (i = 1; i <= 30; i++)
               {cout << i << " ";}
               cout << endl;
               break;
     }
return 0;
}
```

The program works but it certainly isn't elegant to jam all the code into the main function.

This and many programs would be better if they are written with functions that not only organize the code better but also allow snippets of code to be reused again and again.

Consider the following flow chart that shows the different modules or functions that could be called in a "better" version of this program.



In this case the main function "calls" the get_choice function to ask the user what sequence to display. Next, the handle_choice function is called to direct the program to one of the three functions under it.

Guidelines for Building Programs with Functions

Using functions helps the programmer develop programs that can be easily coded, debugged, and maintained. Keep the following guidelines in mind when building programs of more than one function.

1) **Organization** – A large program is easier to read and modify if its logically organized into functions. It is easier to work with a program in parts, rather than in one large chunk. A well-organized program, consisting of multiple parts, is easier to read and debug. Once a single

function is tested and performs properly, you can set it aside and concentrate on problem areas.

2) **Autonomy** – Programs should be designed so that they consist of mainly standalone functions or modules. Each function is autonomous, meaning the function does not depend on data or code outside of the function any more than necessary.

3) **Encapsulation** – The term encapsulation refers to enclosing the details of a function within the function itself, so that those details do not need to be known in order to use the function.

4) **Reusability** – Because functions typically perform a single well-defined task, they may be reused in the same program or even in other programs.

A function can be written for any purpose and can also be a go-between for other parts of the program. Meaning one function can call other functions in the flow of the program.

There are two popular methods for designing programs using functions. The first method, called **top-down design** begins with functions at the top of the flow chart and works towards the functions at the bottom.

Bottom-up design begins at the bottom and works to the top.

The Syntax of Functions

So far every program we have created contains a **main** function. The structure of the main program always looked like:

```
int main()
{
 //body of program
return 0;
}
```

When the program reaches the return 0; statement the value of 0 is returned to the operating system. This value tells that the program ended normally.

We return is an integer because we specified an int data type when the main function was declared.

There are times when the function has no reason to return a value. To prevent a value from being returned, the void keyword is used in place of the data type.

For example, consider a function whose only purpose is to display information:

```
void menu()
{
     cout << "--------------------------------" << endl;
     cout << "     Billy Bob's Food Truck" << endl;
     cout << "--------------------------------" << endl;
     cout << "   Please make your selection:" << endl;
     cout << "--------------------------------" << endl;
     cout << " S - Sandwich----------$3.00" << endl;
     cout << " C - Chips-------------$1.75" << endl;
     cout << " B - Brownie-----------$1.00" << endl;
     cout << " R - Regualr Drink-----$1.50" << endl;
     cout << " L - Large Drink-------$1.75" << endl;
     cout << "--------------------------------" << endl;
     cout << "--------------------------------" << endl;
     cout << " X - Cancel sale and restart" << endl;
     cout << " T - Total the sale" << endl;
     cout << "--------------------------------" << endl;
     cout << "--------------------------------" << endl;
     cout << setprecision(2);
     cout << "Your current total is: " << total << endl;
     cout << "--------------------------------" << endl;
     cout << "--------------------------------" << endl;
     cout << "Selection: ";
}
```

The main of the function is menu.

The void keyword indicates that no value is returned.

The parenthesis after the name lets the compiler know that menu is a function.

The statements between the braces are executed when menu is "called."

```
int main()
{
     menu();   //call to print the menu

return 0;
}
```

Function Prototypes

There is one more thing you have to do to make your functions work.

At the top of your program, you must tell the compiler that the function exists. You do this by creating a **prototype**.

Basically, a prototype defines the function for the compiler.

Example

```
#include<iostream.h>

void print_title();    // prototype for print_title
void print_goodbye();  // prototype for print_goodbye

int main()
{
     print_title();      // call to print_title

     // insert the rest of the program here

     print_goodbye();

 return 0;

} // end of main function

// Function to print program title to screen.
void print_title()
{
     cout << "Tennis Tournament Scheduler Program\n";
     cout << "By Jennifer Baker\n";
}
```

```
// Function to print closing message to screen.
void print_goodbye()
{
     cout << "Thank you for using the Tennis Tournament
     Scheduler.\n";
}
```

The function prototype is identical to the first line of the function itself.

There is, however, a semi-colon at the end of the prototype.

Functions and Program Flow

To understand the program flow with functions you need to consider that the program will now "jump around" to execute function calls instead of just executing line by line as our programs have done so far.

Consider the order of execution shown in the program below:

```
#include<iostream.h>

void print_title();    // prototype for print_title function
void print_goodbye();  // prototype for print_goodbye function


int main()
 {
 print_title();      // call to print_title

  // insert the rest of the program here

 print_goodbye();

  return 0;
} // end of main function


// Function to print program title to screen.
void print_title()
 {
 cout << "Tennis Tournament Scheduler Program\n";
 cout << "By Jennifer Baker\n";
 }

// Function to print closing message to screen.
void print_goodbye()
 {
 cout << "Thank you for using the Tennis Tournament Scheduler.\n";
 }
```

1 2 3 4 5

Scope of Variables

When building a program that consists of functions, you must be concerned with how data is made available to the functions.

As programs get larger, it is important to keep control over variables to prevent errors in your programs.

One way to do this is to make the data in the variables only accessible in the areas that it's needed.

When we only used main() our variables were accessible in that function only. They wouldn't be available to any other function.

The "availability" of a variable is known as its **scope**.

Variables in C++ can either be local or global. A **local variable** is declared within a function and is only accessible in that function.

A **global variable** is declared BEFORE the main function and is accessible by any function.

As you will see in the next section global variables should be used vary sparingly and is considered sloppy coding.