Functions

Class 19

Introduction

- so far, the programs we have written consist of one function that contains of a group of statements
- they have nonlinear structure using branches and loops within that block
- they are just a single big chunk of code

Introduction

- so far, the programs we have written consist of one monolithic block of statements
- they have nonlinear structure using branches and loops within that block
- they are just a single big chunk of code

- this is not ideal
- there are several problems with this approach, especially
  - a large block of code is harder to understand and harder to locate problems in than a small block of code
  - in a large block of code it is more likely that the same set of statements need to be written in several places, called code duplication, than in a small block

Modularization

- a central concept in programming is to create small modules of code rather than large monolithic blocks
- one form of module is the function

- all executable statements in C++ are contained within a function
- so far, we have had only one function, named main
- we will now learn how to create other functions and use them, allowing us to make main smaller and more understandable

Function Definition

- a function must be defined
- the structure of a function definition is

```
return_type function_name(parameter list)
{
  statement;
  statement;
  ...
}
```

for example:

```
int main()
{
  cout << "Hello, world!" << endl;
  return 0;
}
```

Functions

- the function definition must include a return type
- if the function does not return a value, the return type is void
- if the function's return type is not void, then the function must return a value (this is enforced by the compiler in every case except, unfortunately, main)

Functions

- the function definition must include a return type
- if the function does not return a value, the return type is void
- if the function's return type is not void, then the function must return a value (this is enforced by the compiler in every case except, unfortunately, main)

- the function definition may include a parameter list
- the parameter list can be empty; the parentheses are always required

```
// A program to illustrate a function call, from Gaddis 6-2
#include <iostream>
using namespace std;

/**
  display a simple message on the screen
*/
void display_message();

int main()
{
  cout << "Hello from main" << endl;
  for (unsigned count = 0; count < 5; count++)
  {
    display_message();
  }
  cout << "Back in main again" << endl;
  return 0;
}

void display_message()
{
  cout << "Hello from display_message" << endl;
}
```

Functions

- several things to note
- line 8 is a function declaration, or function prototype, while the function definition is on lines 21–23
- Please use function prototypes, before defining the function in your programs.

Functions

- the function call is on line 15
- a function is called by invoking its name with a parameter list
- when called, program executes the body of the called function
- after the function terminates, execution resumes in the calling function at point of call
- the code where the function is called is termed the calling scope

Functions

- a function can be called many times
- it can be called anywhere in main that a statement is allowed

Calling Functions

- main can call any number of other functions
- the compiler must know the following about a function before it is called
  - the name
  - the return type
  - the number of parameters
  - the data type of each parameter

Order

- When writing a function , please follow these guidelines
  - add comment explaining the purpose of the program
  - includes
  - a namespace statement, if used
  - global constants
  - function prototypes (with documentation) if used
  - the main function definition
  - other function definitions, if used

Sending Data Into a Function

- when a function is called, the calling scope may send values into the function
- We have seen some examples of such previously,
- For example, setw is a function to which we have supplied data as the following:

```
setw(10);
```

- a value sent into a program is called an argument
- by using parameters, you can create a function that accepts values when it is called

Returning Values

- a function may return a value to the calling scope
- typically this is because the function does a computation with the values supplied as arguments and now has the results

formal parameters

```
unsigned get_rand_in_range(unsigned low, unsigned high)
{
   ...
   return value;
}

unsigned length = get_rand_in_range(1, MAX_LENGTH);
```

actual parameters

see program rectangle_area.cpp

Function Comments

- every function, except main, must have a header comment
- Please use the following format

```
/**
  compute the area of a rectangle
  @param length the length of the rectangle
  @param width the width of the rectangle
  @return the area of a length by width rectangle
*/
int get_rectangle_area(int length, int width);
```

- a comment explaining the purpose of the function
- every parameter documented with @param
- the return value documented with @return, unless it is a void function

Parameter Terminology

formal parameters

```
unsigned get_rand_in_range(unsigned low, unsigned high)
{
   ...
   return value;
}

unsigned length = get_rand_in_range(1, MAX_LENGTH);
```

actual parameters

- formal parameters are also called parameters
- actual parameters are also called arguments
- you need to know both names for each, as both are used interchangeably

Formal Parameters and Variables

```
1  int get_rand_in_range(int low, int high) {
2
3    int value = rand() % (high - low + 1) + l;
4     return value;
5  }
```

- variables may be declared in a function
- on line 3, value is declared and initialized
- this is a local variable visible only from lines 3 through 5
- not in existence before or after the function is executing

Formal Parameters and Variables

```
1  int get_rand_in_range(int low, int high) {
2
3    int value = rand() % (high - low + 1) + l ;
4    return value;
5  }
```

- variables may be declared in a function
- on line 3, value is declared and initialized
- this is a local variable visible only from lines 3 through 6
- not in existence before or after the function is executing

- the formal parameters high and low are also variables within the function
- they are visible from lines 1 through 5
- within the function body, they can be used as normal variables

- they are pre-initialized with the values passed into them from the calling function

Function Names

- a function name is a programmer-defined identifier
- functions DO something
- their names should contain VERBS whenever possible
  - get_rand_in_range()
  - display_value()
  - show_menu()
  - setprecision()

Returning a Boolean Value

- an extremely useful concept is a function that returns a Boolean value
- this is similar in concept to a Boolean flag variable, but is a function rather than a variable

```
bool is_negative(double amount)
{
  bool result = amount < 0.0;
  return result;
}
```

You can use it in our data validation loop as the following:

```
while (is_negative(gigs_data)==true)
{
  cout << "Data usage cannot be negative. Please re-enter: ";
  cin >> gigs_data;
}
```

Return Structure

- there are limitations on where in a function a return statement can appear
- when the return is executed, the function stops instantly and returns its value and control to the calling scope
- any code after the return executes will not be reached

```
int foo()
{
  statement_A;
  statement_B;
  return 10;
  statement_C;
  statement_D;
}
```

- statements C and D are dead code i.e., unreachable
- the compiler will not allow this

Return Structure

- a non-void function must return a value

```
int foo()
{
  ...
  if (x)
  {
    return 10;
  }
  else if(!x)
  {
    return 20;
  }
}
```

- a human can tell that a return will execute, but the compiler cannot, and will not allow this

Return Structure

- but the following is fine

```
int foo()
{
  ...
  if (x)
  {
    return 10;
  }
  else
  {
    return 20;
  }
}
```

- or even simpler, and more understandable

```
int foo()
{
  ...
  if (x)
  {
    return 10;
  }
  return 20;
}
```

Void Functions

- not all functions return values
- these are termed void functions

```
void display_message()
{
}
```

- when the function is called, it does not return a value

  It can be called as the following:

```
display_message();
```

Return From Void Function

- a void function does not return any value
- If you prefer, you can end a function with a void return as the following

```
return;
```