Functions

Class 19

Introduction

- so far, the programs we have written consist of one monolithic block of statements
- they are just a single big chunk of code, in main()

Introduction

- so far, the programs we have written consist of one monolithic block of statements
- they are just a single big chunk of code, in main()

- this is not ideal
- there are several problems with this approach, especially
  - a large block of code is harder to understand than a small block of code
  - it is harder to locate problems in a large 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
  - in a large block of code it is more likely that different statements will act at different levels of abstraction, again making it harder to understand

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
- the return type is void if the function does not return a value
- 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
- the return type is void if the function does not return a value
- 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 must include a parameter list
- the parameter list may be empty, but the parentheses are still 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 21–24
- Gaddis uses a less strict compiler system that does not require prototypes
- but our compiler system absolutely requires them

Functions

- the function call is on line 15
- a function is called by invoking its name with a parameter list
- when called, the body of the called function executes
- 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

Details

- more specifically, when a function is called:
    1. execution of the calling scope is suspended
    2. parameters, if any, are copied from the calling scope to the function (see below)
    3. the function body is executed
    4. the return value, if any, is copied from the function to the calling scope
    5. execution of the calling scope is resumed

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
- other functions can call 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
- all of these are supplied by a function prototype
- like a function definition without the body

Order

- from the style guide: organize the material in each file as follows:
  - a comment explaining the purpose of this file, along with your full name
  - includes, if any
  - a namespace statement, if used
  - typedefs, defines, and global constants, if any
  - function prototypes (with documentation)
  - the main function definition, if present
  - other function definitions, if present

Sending Data Into a Function

- when a function is called, the calling scope may send values into the function
- you are already quite familiar with this
- in the function call `setw(WIDTH)` the value of WIDTH is being sent into the function `setw`
- in the function call `setprecision(PRECISION)` the value of PRECISION is being sent into the function `setprecision`

Sending Data Into a Function

- when a function is called, the calling scope may send values into the function
- you are already quite familiar with this
- in the function call `setw(WIDTH)` the value of WIDTH is being sent into the function `setw`
- in the function call `setprecision(PRECISION)` the value of PRECISION is being sent into the function `setprecision`

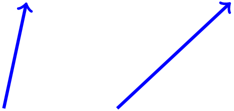
- a value sent into a function is called an argument
- by using parameters, you can create a function that accepts values when called

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;
```

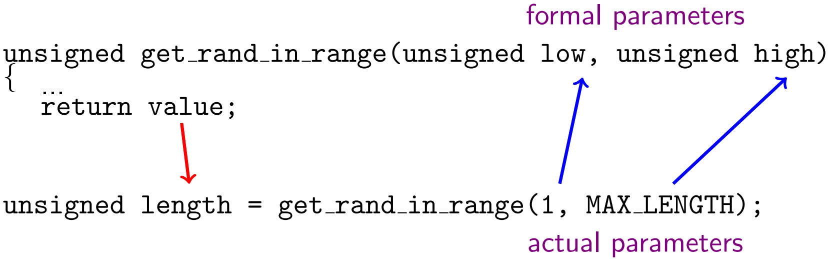


```
unsigned length = get_rand_in_range(1, MAX_LENGTH);
```

actual parameters

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

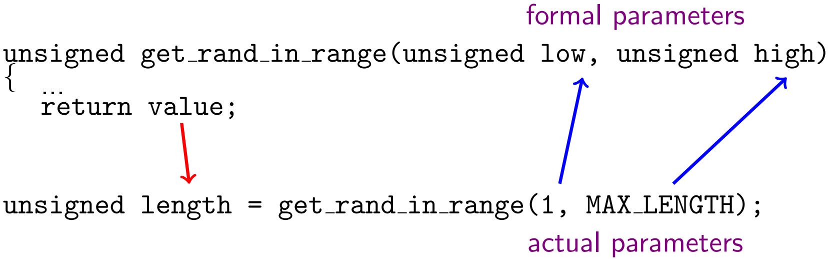
```
unsigned get_rand_in_range(unsigned low, unsigned high)
{  ...
   return value;


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

actual parameters

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

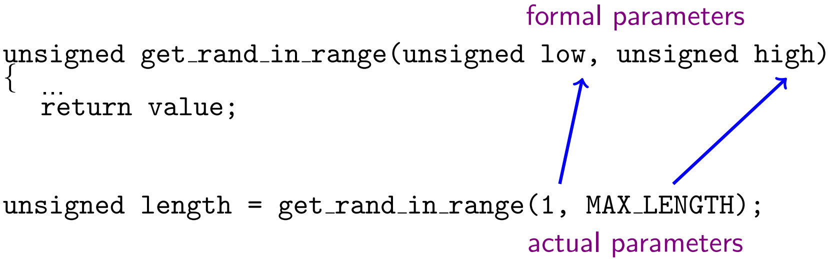
Javadoc

- every function, except main, must have a header comment
- we use the Javadoc format, because
  - C++ does not have a native documentation system of its own
  - all programmers learn Javadoc, so you might as well also
  - all programmers understand Javadoc, so it's a good way to communicate

```
/**
  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
*/
unsigned get_rectangle_area(unsigned length, unsigned 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

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, so it is not called in the context of an assignment statement:

  ```
  display_message();
  ```

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, so it is not called in the context of an assignment statement:

  ```
  display_message();
  ```

- a function may
  - have zero parameters, one parameter, or many parameters
  - return no value (void) or return one value (the return type)
- parameters and return values are independent

Formal Parameters and Variables

```
1  unsigned get_rand_in_range(unsigned low, unsigned high)
2  {
3    unsigned value =
4        static_cast<unsigned>(rand()) % (high - low + 1) + low;
5    return value;
6  }
```

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

Formal Parameters and Variables

```
1  unsigned get_rand_in_range(unsigned low, unsigned high)
2  {
3    unsigned value =
4        static_cast<unsigned>(rand()) % (high - low + 1) + low;
5    return value;
6  }
```

- variables may be declared in a function
- on line 3, value is declared and initialized
- this is a local variable in scope from lines 3 to 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 in scope from lines 1 to 6
- within the function body, they can be used, modified, in every way treated as normal variables
- they are pre-initialized with the values passed into them from the calling scope

Function Names

- a function name is a programmer-defined identifier
- functions DO something
- their names should contain VERBS
  - get_rand_in_range()
  - display_value()
  - show_menu()
  - setprecision()
- all functions you write should follow this rule
- unfortunately, many library functions do not : e.g., pow(), main()
- Gaddis is sometimes sloppy, e.g., sum() should be get_sum() or compute_sum()

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;
}
```

then the data validation loop in lab 6 could be:

```
while (is_negative(gigs_data))
{
  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
- if the compiler can't tell, it will complain

```
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;
}
```

Return Structure

- returning out of the middle of a loop is not allowed
- it breaks the loop's logic structure just like break

```
int foo()
{
  ...
  while (!done)
  {
    ...
    return 10;
    ...
  }
}
```

Return From Void Function

- you cannot return a value from a void function
- in a void function, `return 0;` will not compile
- however, a return statement without a value is a legal construct in the language

  ```
  return;
  ```

Return From Void Function

- you cannot return a value from a void function
- in a void function, `return 0;` will not compile
- however, a return statement without a value is a legal construct in the language
  `return;`
- however, using an empty return in a void function almost always indicates poor logic and poor code organization
- return should never be the last statement of a void function (there's no point)
- return should never be used within a loop (it violates the loop structure)