Chapter

8

Library Mechanics

Functions are named "chunks" of code that **calculate a value** or that carry out an action. In C++, a function associates a computation—specified by a block of code that forms the body of the function—with a particular name. If a function calculates a value, it may be used in an expression; if it carries out an action (called a `void` function in C++ or procedure in other languages), it cannot.

Using functions reduces bugs and make maintenance more effective by allowing you to reuse proven code, instead of duplicating it. Placing related functions into a library, allows you to reuse them in many different contexts.

Separate Compilation

Programs using functions can be organized in several different ways. The question to ask is "Where do definitions and declarations go?"

1. You may define **your functions before calling them**. If you have only one or two functions in a "throw-away" program, this is fine. Because your functions need to appear in a particular order, though, your code is often harder to understand and maintain. In the future you will not to do this.

2. You can prototype **(or** declare**) your functions at the top of your file** (usually under the library `#include` statements) and then **define** the functions later in the file, after the `main` function. This makes it easier to read and understand your program, because the primary logic appears before the function details.

   It also makes your code much easier to maintain, because you can then define your functions in any order you like. **Do this for small programs** and for functions that are unique to a particular program.

3. If you have functions which you want to **reuse** in different programs, you should place those function in a library, (a collection of similar functions) and **place the prototype into a header file**. That is what you'll generally do in this class for all of your functions from now on.

In this chapter, you will start using separate compilation in our assignments. To **define a library** in C++, you'll supply four parts:

- the client or test program, which **uses** the functions in the library.
- the interface, which provides information needed to use the library. The interface consists of **declarations** or **prototypes**.
- the implementation, which provides the details. The implementation consists of **function definitions**.
- the make file which puts the parts together to produce the executable.

Start with a library containing three functions:

- `int firstDigit(int n)`, returning the **first** digit of its argument
- `int lastDigit(int n)`, returning the **last** digit of its argument
- `int numDigits(int n)`, returning the **number of digits** of its argument

Calling `firstDigit(1729)` will return `1`, calling `lastDigit(1729)` will return `9`, while calling `numDigits(1729)` returns `4`.

Click the little "running-man" to your left. Leave it open for the next few pages. (You must be logged into your GitHub account)

The Client File or Test Program

In your IDE, create a folder. Add a file named `client.cpp`. Add the usual `#include` statements and an empty `main()`. In the client file you need to:

1. Call each of your functions with some known input.
2. Compare the value returned (this is called the actual value), with the value that **should have been returned** (this is called the expected value).
3. Print a message indicating whether you got it right or wrong.

Here's some simple code for an example:

```
cout << "Testing digits library" << endl;
cout << (firstDigit(1729) == 1 ? "PASS" : "FAIL") << endl;
cout << (lastDigit(1729) == 9 ? "PASS" : "FAIL") << endl;
cout << (numDigits(1729) == 4 ? "PASS" : "FAIL") << endl;
```

When you compile with `make client`, you'll get the error message:

```
client.cpp:7:14: error: ‘firstDigit’ was not declared in this scope
```

An undeclared error message is a compiler syntax error. It means you are missing a prototype or you are calling a function incorrectly. Which leads us to the interface.

The Interface (Header) File

A library may contain several definitions: **functions**, **types**, and **constants**. In C++, the interface and implementation are in two files: a **header** (or interface) **file** and an **implementation** file. The **interface** file usually ends with the extension **`.h`**.

Add **`#include "digits.h"`** in your client file right after the **`using namespace std`** line. Then open **`digits.h`** and let's look at header guards.

Preprocessor Header Guards

It is possible for one header file to include another. You must do **something** to **make sure that the compiler doesn't include the same interface twice**. You do that by adding three lines to every header file that are known as the **interface boilerplate**, or **header guards**. They look like this:

```
#ifndef FILE_IDENTIFIER
#define FILE_IDENTIFIER
  // Entire contents of the header file
#endif
```

Header guards.

This pattern appears **in every interface**. These are **instructions to the preprocessor**, a program that examines your code before it is sent to the compiler. The boilerplate consists of the **`#ifndef`** and **`#define`** at the beginning and the **`#endif`** at the end.

1. The **`#ifndef`** preprocessor directive checks whether the **`FILE_IDENTIFIER`** symbol has been defined in the **current translation unit**. When the preprocessor reads this interface file **for the first time**, the answer is no.
2. The next line **defines that symbol**. Thus, if asked later to **`#include`** the same interface, the **`FILE_IDENTIFIER`** **will already be defined**, and the preprocessor will **skip over the contents** of the interface this time around.

A common convention is to simply **capitalize the name of the file itself**, replacing the dot with an underscore. You may use another convention if you like, but make sure that the name will be unique when you build your project.

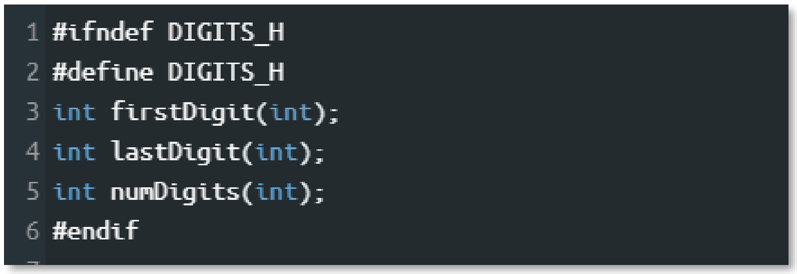
Add the header guards to **`digits.h`**.

The Interface or Declaration

When the compiler **encounters a function call** in your program, it needs information in order to generate the correct code; the compiler doesn't need to know **how** the function is implemented, but it **does** need to know:

- what types each of the **arguments** to the function are (and how many)
- what type of value the function **returns**

That information is provided by a **prototype**, or **function declaration**:



```
#ifndef DIGITS_H
#define DIGITS_H
int firstDigit(int);
int lastDigit(int);
int numDigits(int);
#endif
```

These prototypes associate the names `firstDigit`, `lastDigit`, **and** `numDigits` each with a function that **takes** a single `int` as an argument and **returns** an **int** as its result. These are **function declarations**. **Go ahead and complete the prototypes now.**

In a prototype, **parameter names are optional**. The compiler doesn't care about the names, but they help **you remember** which parameter matches which argument.

```
double focalLength(double d, double r1, double r2, double n);
```

Supplying names in a prototype often helps the reader. The parameter names in a prototype are in **prototype scope**; they have no meaning after the prototype ends.

Library Types in Interfaces

If the prototype **includes any types from the standard library** (such as `string` or `vector`), then you must `#include` the correct header, and **fully qualify** the name of the type. Here's an example:

```
// Header file
std::string zipZap(const std::string& str);
// Implementation file
string zipZap(const string& str) { . . . }
```

Header files **should never** use identifiers from the standard library without explicitly including the `std::` qualifier. In the implementation file, you may use the name as is, because your implementation file will contain a `using` declaration or directive.

Here are **three rules to remember**.

1. **Never** add `using namespace std` to a header file. Header files are `#included` in other files; doing so changes the environment of that file.
2. **Always** add `std::` in front of **every** library type, such as `std::string`, but **never** in front of primitive types like `double`.
3. For all library types, `#include` the appropriate header file inside your header file. If you use the `std::string` type, you must `#include <string>`

Linker Errors

Once you have finished commenting **all** three functions, build your project by typing `make client`. When you do, you **won't** see any compiler error messages; the client program compiles. However, you will see some **linker error messages**. Your function was **declared** correctly, but the **definition** could not be found at linking time.

```
$ make client
g++     client.cpp  -lcrypt -lcs50 -lm -o client
/tmp/cchLKb6X.o: In function `main':
client.cpp:(.text+0x32): undefined reference to `firstDigit(int)'
```

*If you still see **undeclared** (instead of **undefined**), make sure you have added the line `#include "digit.h"` to the top of the client program.*

Two words to note in your compiler's error messages: **undefined** and **undeclared**. Recognizing these will help you locate and fix the problem.

- An **undeclared** error message is a compiler syntax error. It means you are missing a prototype or you are calling a function incorrectly.
- An **undefined** error message comes from the linker and means that you are missing the definition for a function.

The Implementation (cpp) File

Place your **definitions** in an **implementation file**. The **implementation** file has the same root or **base name** as the header file, but a different extension. You'll use `.cpp` in this class but other conventions include `.cxx`, `.cc`, and `.C` (a capital **'c'**).

1. Create an **implementation file** using the extension `.cpp`.
2. Add a file comment to the top of the file.
3. `#include` the interface file for the library you are implementing.
4. If you use other libraries, such as `string`, you should include them as well. This example does not.

5. Add a **using directive** if you are using any library types. This library does not.
6. **Copy the prototypes** from each of the functions in the header file into the implementation file. You don't need to bring across the documentation. Make sure, also, that you **don't copy the header guards**.
7. **Stub out each of the functions.**

This is **purely mechanical**. You want to practice it until it becomes second nature. You should **memorize** this part, so you don't have to expend any brain cells to complete it.

Warning. *Make sure your stubs always include a* ***`return`*** *statement of the correct type, or your function may crash at runtime.*

Write the Stub or Skeleton

Always start by writing a "skeleton" or **stub** for your function. Make sure your code starts out syntactically correct, and then stays that way.

1. **Copy the prototype or declaration** into the implementation file. You don't need to bring the documentation with the prototype, but you may.
2. Remove the semicolon at the end of the prototype and **add some braces** to supply a **body** for the function.
3. Unless your function is a procedure (**`void`** function), you must create a return variable to hold the result. Look at the function return type to decide what type to make this variable. Initialize it to the "empty" value.
4. Add a **`return`** statement at the very end of your function.

Here's a **stubbed-out** version of one function:

```
#include "digits.h"

int firstDigit(int n)
{
    int result{};
    return result;
}
```

Once you've stubbed out the two other functions, you'd expect your program to compile and link, but it does not. You get the same linker errors that appeared in Section 4. Why? Since you now have two, separately compiled portions of object code, you have to tell the compiler **how to link them together**. You do that with a makefile.

The Make File

For the homework your instructor provides a project or make file. For this lesson, though, you'll get to build one on your own. Open the file named `makefile`. It has no extension. Then, add the following code:

```
EXE=digit-tester
OBJS=client.o digits.o

$(EXE): $(OBJS)
	$(CXX) $(CXXFLAGS) $(OBJS) -o $(EXE)

run: $(EXE)
	./$(EXE)
```

Here are what these three sections mean:

1. Macros: these are like variables that can be **expanded** later. `EXE` is the name of our executable, and `OBJS` is the names of the object files. Since you have two `.cpp` files in our project, you'll have two object files.
2. Both of the next two sections contain targets, dependencies and actions.
   - → Line 4 **expands** the macros `EXE` and `OBJS`, producing a line that says: `digit-tester : client.o digits.o`
   - → Interpret this line as: "to build `digit-tester`, make sure that `client.o` and `digits.o` are up to date."
   - → `digit-tester` is a **target**; `client.o` and `digits.o` are **dependencies**
   - → Line 5 is the action to perform to produce `digit-tester`. Each action line must start with a tab character, not spaces. Line 5 expands a few other macros meaning "run the compiler with these object files and produce this executable".
3. Line 7 is called a **pseudo target**. When you type `make run`, the action line is executed. If you just type `make`, only the first target is built.

Once you've saved your make file, type `make` and the linker errors should disappear. If they don't then go over the previous steps. Type `make run`, and the client program should run (even if you have some warnings about unused parameters.)

Even though all of your tests fail, that's OK. The purpose of the stub is to get the mechanical details out of the way, so that you can use all of your brainpower to concentrate on solving the problems.

Document & Implement

A documentation tool named Doxygen is used to generate HTML and other kinds of documentation. Use the following tags.

The File Comment

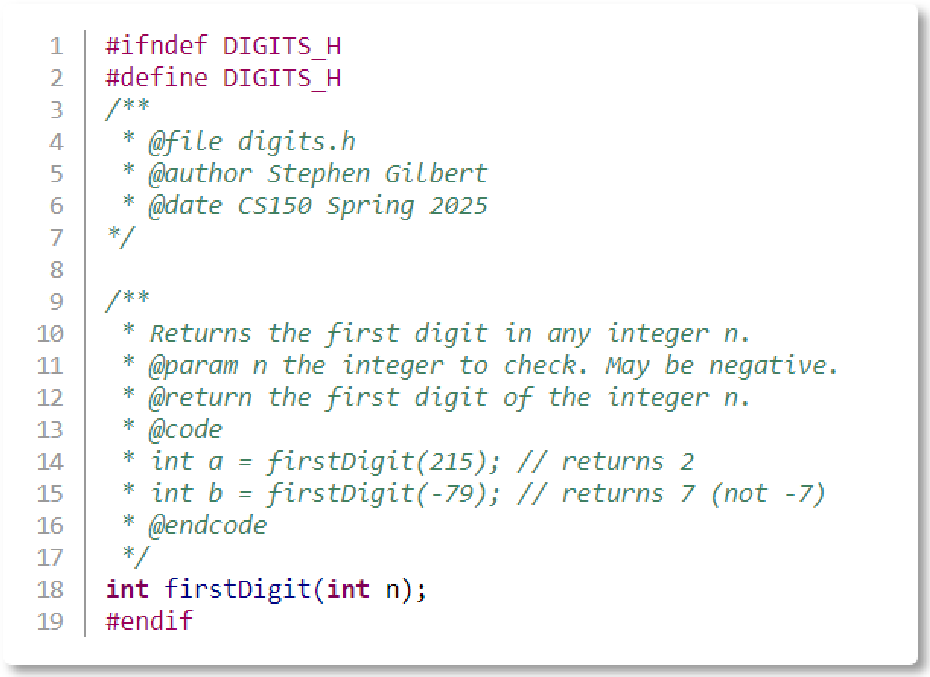
The header file should have a **file comment** that contains these common elements.

- `@file` – Name of the file. Required if you are going to document functions, global variables and constants. **Always use this.**
- `@author` – your name. or ID (e.g. sgilbert)
- `@date` – date it was created (can be general, like: CS 150 S'25, MWAM)
- `@version` – version information about the library (optional)

Function Comments

Each function should also be documented. Here are the tags to use.

- Start your comment block with a single line ending in a period.
- `@param` – the name and description of every parameter to your function.
- `@return` – if the function returns a value, you should add this tag
- `@code` (optional) – a **block** of code that will be syntax highlighted in the generated documentation. This block must end with a `@endcode` tag.



```
#ifndef DIGITS_H
#define DIGITS_H
/**
 * @file digits.h
 * @author Stephen Gilbert
 * @date CS150 Spring 2025
*/

/**
 * Returns the first digit in any integer n.
 * @param n the integer to check. May be negative.
 * @return the first digit of the integer n.
 * @code
 * int a = firstDigit(215); // returns 2
 * int b = firstDigit(-79); // returns 7 (not -7)
 * @endcode
 */
int firstDigit(int n);
#endif
```

Sample Header File.

Planning & Implementation

The documentation in the header file is for the **client**-what us necessary to **use** the function. In the implementation file, add **implementation comments**, in the form of a function plan, to help write the function. These comments **are intended for programmers**, not for the clients of the function. Don't use **`DOXYGEN`** tags, but describe the algorithms and important implementation details.

Here is my plan for ***`lastDigit()`***:

```
// result <- |n| % 10
```

Single-line comments are simplest for this, since editors will comment and un-comment a portion of code, using only a single keystroke. In the CS50 IDE, the keystroke is **`Shift+/`**.

Implementing *`lastDigit`*

You should write your comments first, and then implement the function. Of course, with small functions like this, you often will plan it out, and then simply write the code directly from the comment.

```
int result = n < 0? -n : n;
return result % 10;
```

This uses the **conditional operator** to set the **`result`** to convert **`n`** to a positive number. You could use the **`abs()`** function in `<cstdlib>`. However, a new C++ programmer might not realize that there are separate versions in the function `<cmath>` (for floating-point numbers), and in `<cstdlib>` for (integers).

Parameters vs. Local Variables

Note that instead of changing **`n`** itself, this solution creates a separate variable named **`result`**. This is better style because it doesn't **confuse the concept** of a parameter or input variable, with that of **a local variable** used for the function output. It won't make any difference to the answer, but it may keep you from getting confused.

The *`while`* Loop

The other two functions in our library are more difficult. Both of them require you to learn about a new kind of loop bounds, called limit bounds. Loops that do some processing and then check the results against a boundary condition are limit loops.

To write a limit loop, use the **`while`** loop, which executes a statement repeatedly while its condition remains **`true`**. The general form of the **`while`** loop looks like this:

```
while (condition)
{
    statements
}
```

A `while` loop first evaluates the condition. If `false`, the loop terminates and the program continues with the next statement after the loop body. If `true`, the actions in the body are run, after which control **returns to the loop condition**. One pass through the body constitutes a **cycle or iteration of the loop**.

1. The test is performed **before every cycle of the loop**, including the first. If the test is `false` initially, the body of the loop is **not executed at all**.
2. The test is performed **only** at the beginning of a loop cycle. If the condition **becomes** `false` at some point during the loop, the **program won't notice** that fact until it completes the entire cycle. When the program evaluates the test condition again, if it is still `false`, the loop terminates.

Limit Bounds

Here is a limit loop which computes the sum of the digits in an integer, `n`:

```
int sum = 0;
while (n > 0)
{
    sum += n % 10;
    n /= 10;
}
```

- The expression `n % 10` always returns the last digit in a positive integer `n`.
- The expression `n / 10` returns a number without its final digit.

In this case, the condition that is being tested is the value of `n`, which is changed each time through the loop. Once `n` reach its **limit** (`0`), the loop terminates.

Loop Plans

Here are the plans I created for each of the remaining two functions. Using limit loops, you should be able to complete these on your own.

First Digit

```
// result <- |n| // abs(n)
// while result >= 10
//     result <- result / 10
// return result
```

Number of Digits

```
// Number of digits in n
// Assume that 0 is a digit
// result <- |n|
// counter <- 1
// while result >= 10
//     result <- result / 10
//     counter <- counter + 1
// return counter
```

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using `make submit`.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.