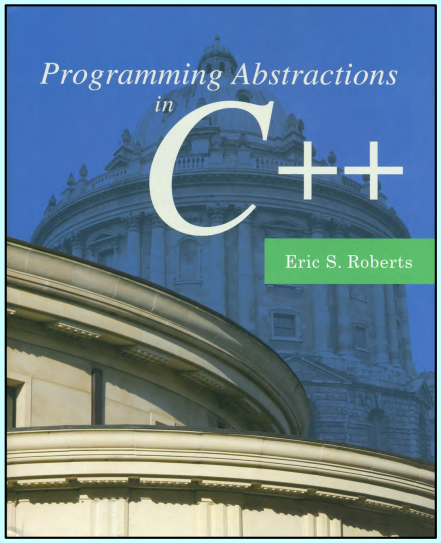
CHAPTER 2

Functions and Libraries

Your library is your paradise.
—Desiderius Erasmus, *Fisher's Study at Rotterdam,* 1524



Programming Paradigms

- A ***programming paradigm*** is a "style" or "way" of programming.
- One of the characteristics of a programming language is its support for particular paradigms. Some languages make it easy to write in some paradigms but not others.
- A language purposely designed to allow programming in many paradigms is called a multi-paradigm programming language, e.g., C++, Python. You can write programs or libraries that are largely *procedural*, *object-oriented*, or *functional* (i.e., some typical paradigms) in these languages.
- In a large program, even different sections can be written in different paradigms.
- C++ supports the *procedural* and *object-oriented* paradigms naturally, supports the *functional* paradigm through the **`<functional>`** interface, and supports many other paradigms through various external libraries.

The Procedural Programming Paradigm

- **Imperative** programming paradigm: an explicit sequence of statements that change a program's state, specifying how to achieve the result.
  - **Structured**: Programs have clean, `goto`-free, nested control structures.
  - **Procedural**: Imperative programming with procedures operating on data.
- Procedural programming is a programming paradigm, derived from structured programming, based on the concept of the procedure call.
- Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out.
- In a computer program, a ***function*** is a named section of code that performs a specific set of statements.

The Idea of a Function

- ***Functions*** are a familiar concept from algebra, where they specify a mathematical calculation based on one or more unknown values, which are called ***arguments***.
- As an example, the mathematical function

$$f(x) = x^2 + 1$$

  specifies that you can calculate the value of the function f by squaring the argument x and then adding 1 to the result. Thus, $f(0)$ is 1, $f(1)$ is 2, $f(2)$ is 5, and so on.
- In C++, you can implement a mathematical function by encoding the calculation inside a function definition, like this:

```
double f(double x) {
   return x * x + 1;
}
```

Function in Programming

- In programming languages, a ***function*** is a block of code that has been organized into a separate unit and given a name.
- Once a function is defined, the act of using the name to invoke that code is known as ***calling*** that function.
- The calling program can pass information to the function using ***arguments***.
- Once called, the function takes the data supplied as arguments, does its work, and then ***returns*** to the calling program at the point in the code from which the call was made. As it returns, a function often passes a value back to its caller.
- Essentially you are copying the code inside a function definition to where it is called, with the variables (i.e., ***parameters***) in the function replaced by the ***arguments*** from the caller.

Parameters vs. Arguments

- A parameter is a placeholder for one of the arguments supplied in the function call and acts in most respects just like a local variable.
  - A parameter is a variable in a function declaration and definition, and a placeholder for the argument.
  - An argument is an expression used when calling the function, and an actualization of the parameter.
  - When a function is called, the arguments are the data (actual values) you pass into the function's parameters (local variables).

The Advantages of Using Functions

- Functions allow you to shorten a program by allowing you to include the code for a particular operation once and then ***reuse*** it as often as you want in a variety of different contexts, typically with different arguments.
- Functions make programs easier to read by allowing you to invoke an entire sequence of operations using a single name (***abstraction***) that corresponds to a higher-level understanding of the purpose of the function.
- Functions simplify program maintenance by allowing you to divide a large program into smaller, more manageable pieces. This process is called ***decomposition***. You can update the implementation of a function without affecting the programs that call the function.
- Top-down design/stepwise refinement.

Defining Functions in C++

- The general form of a function definition in C++ looks much the same as it does in other languages derived from C, such as Java, but slightly more complicated than in Python:

```
type  name (parameter list) {
      statements in the function body
}
```

  where *type* indicates what type the function returns, *name* is the name of the function, and ***parameter*** *list* is a list of variable declarations used to hold the values of each argument.

- *statements in the function body* implement the function (e.g., the algorithm), at least including a **`return`** statement.

```
return expression;
```

Question

- What if I want to return more than one value?

Defining Functions in C++

- Functions that return Boolean results are called ***predicate functions***.

```
bool isEven(int n) {
   return n % 2 == 0;
}
. . .
if (isEven(i)) . . .
```

- Functions that return no value at all are usually called ***procedure***, using `void` as the result type.
- All functions need to be declared before they are called by specifying a ***prototype*** consisting of the header line followed by a semicolon.

Function Prototypes

- A ***function prototype*** is simply the header line of the function followed by a semicolon, which you put in front of the main function so that the compiler knows what arguments the function requires and what type of value it returns, before you actually define the function.
- If you always define functions before you call them, prototypes are not required.
  - E.g., the low-level functions come at the beginning, followed by the intermediate-level functions that call them, with the main program coming at the very bottom.
- While this strategy can save a few prototype lines, it is counterintuitive in the sense of ***top-down design***, since the most general functions appear at the end.
- And there might be situations where you cannot always define functions before you call them (e.g., mutual recursion).

Functions and Algorithms

- Functions are critical to programming because they provide a structure in which to express ***algorithms***.
- Algorithms for solving a particular problem can vary widely in their efficiency (or complexity). It makes sense to think carefully when you are choosing an algorithm because making a bad choice can be extremely costly.
- The next few slides illustrate this principle by implementing two algorithms for computing the ***greatest common divisor*** of the integers *x* and *y,* which is defined to be the largest integer that divides evenly into both.

The Brute-Force Approach

- One strategy for computing the greatest common divisor is to count backwards from the smaller value until you find one that divides evenly into both. The code looks like this:

```
int gcd(int x, int y) {
   int guess = (x < y) ? x : y;
   while (x % guess != 0 || y % guess != 0) {
      guess--;
   }
   return guess;
}
```

- This algorithm must terminate for positive values of **x** and **y** because the value of **guess** will eventually reach 1. At that point, **guess** must be the greatest common divisor because the **while** loop will have already tested all larger ones.
- Trying every possibility is called a brute-force strategy.

Euclid's Algorithm

- If you use the brute-force approach to compute the greatest common divisor of 1000005 and 1000000, the program will take almost a million steps to tell you the answer is 5.
- You can get the answer much more quickly if you use a better algorithm. The Greek mathematician Euclid of Alexandria described a more efficient algorithm 23 centuries ago, which looks like this:

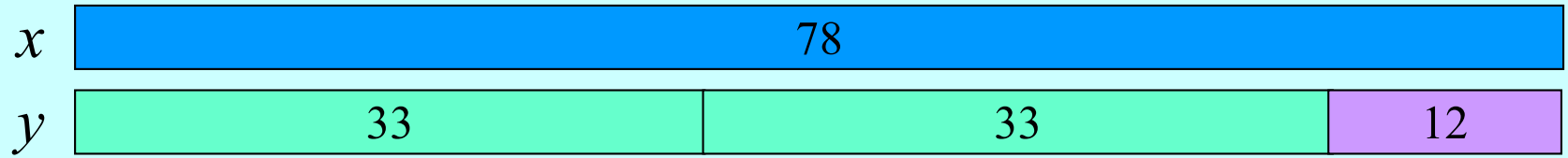
```
int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

How Euclid's Algorithm Works

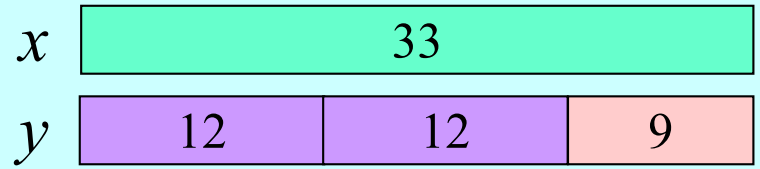
- Euclid's great insight was that the greatest common divisor of x and y must also be the greatest common divisor of y and the remainder of x divided by y. He was, moreover, able to prove this proposition in Book VII of his *Elements*.
- If you use Euclid's algorithm on 1000005 and 1000000, you get the correct answer in just 2 steps, which is much better than the million steps required by brute force.
- It is easy to see how Euclid's algorithm works if you think about the problem geometrically, as Euclid did. The next slide works through the steps in the calculation when x is 78 and y is 33.

An Illustration of Euclid's Algorithm

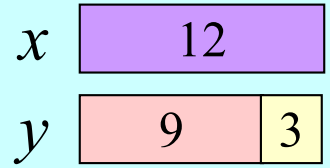
Step 1: Compute the remainder of 78 divided by 33:



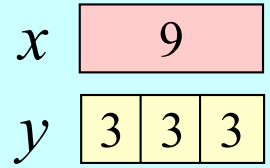
Step 2: Compute the remainder of 33 divided by 12:



Step 3: Compute the remainder of 12 divided by 9:



Step 4: Compute the remainder of 9 divided by 3:



Because there is no remainder, the answer is 3.

C++ Enhancements to Functions

- Functions can be ***overloaded***, which means that you can define several different functions with the same name as long as the correct version can be determined by looking at the number and types of the arguments.
- The pattern of arguments (i.e., the number and types of the arguments but not the parameter names) required for a particular function is called its ***signature***.
- The primary advantage of using overloading is that doing so makes it easier for you as a programmer to keep track of different function names for the same operation when it is applied in slightly different contexts.
- Prototype vs. Signature

Exercise: overloading and signature

- If we input 1 or 1.0 in each case, what is the output?

```
#include <iostream>

int add1(int x);
double add1(double x);

int main() {
    double x;
    std::cin >> x;
    std::cout << add1(x)/4;
    return 0;
}
double add1(int x) { // ambiguous signature
    return x + 1;
}
int add1(int y) {
    return y + 1;
}
double add1(double x) {
    return x + 1;
}
```

C++ Enhancements to Functions

- Functions can specify ***optional parameters*** by including an initializer after the variable name in the function prototype. For example, the function prototype

```
void setMargin(int margin = 72);
```

  indicates that **`setMargin`** takes an optional argument that defaults to 72.

- Define a procedure **`setInitialLocation`** that takes an **`x`** and a **`y`** coordinate as arguments:

```
void setInitialLocation(double x, double y);
```

- Giving ***default values*** to the parameters makes them ***optional***:

```
void setInitialLocation(double x = 0, double y = 0);
```

C++ Enhancements to Functions

- When you use default values for optional parameters, it helps to keep the following rules in mind:
  - The specification of the default values (***default arguments***) appears only in the function prototype and not in the function definition.
  - Any optional parameters must appear at the end of the parameter list. No required parameters may appear after an optional one.

Exercise

```
void setInitialLocation(double x = 0, double y = 0);
```

- In the above example, if the user only provides one argument, it will be assigned to the first parameter. What shall we do if we want to keep the users from calling **`setInitialLocation`** with only one argument (i.e., the user can provide either all arguments or no arguments at all)?
- Using ***overloading*** to implement default arguments:

```
void setInitialLocation(double x, double y);

void setInitialLocation() {
   setInitialLocation(0, 0);
}
```

The Mechanics of Calling a Function

When you invoke a function, the following actions occur:

1. The calling function evaluates the argument expressions in its own context.
2. C++ then copies each argument value into the corresponding parameter variable, which is allocated in a newly assigned region of memory called a ***stack frame***. This assignment follows the order in which the arguments appear: the first argument is copied into the first parameter variable, and so on.
3. C++ then evaluates the statements in the function body, using the new stack frame to look up the values of local variables.
4. When C++ encounters a **`return`** statement, it computes the return value and substitutes that value in place of the call.
5. C++ then discards the stack frame for the called function and returns to the caller, continuing from where it left off.

The Combinations Function

- To illustrate function calls, the text uses a function $C(n, k)$ that computes the combinations function, which is the number of ways one can select k elements from a set of n objects.
- Suppose, for example, that you have a set of five coins: a penny, a nickel, a dime, a quarter, and a dollar:

How many ways are there to select two coins?

penny + nickel	nickel + dime	dime + quarter	quarter + dollar
penny + dime	nickel + quarter	dime + dollar	
penny + quarter	nickel + dollar		
penny + dollar			

for a total of 10 ways.

Combinations and Factorials

- Fortunately, mathematics provides an easier way to compute the combinations function than by counting all the ways. The value of the combinations function is given by the formula:

$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

- Decompose combinations into factorials. From the top level, the combination function needs a factorial function. Given that you already have a **`fact`** function, it is easy to turn this formula directly into a C++ function, as follows:

```
int combinations(int n, int k) {
   return fact(n) / (fact(k) * fact(n - k));
}
```

Computing Factorials

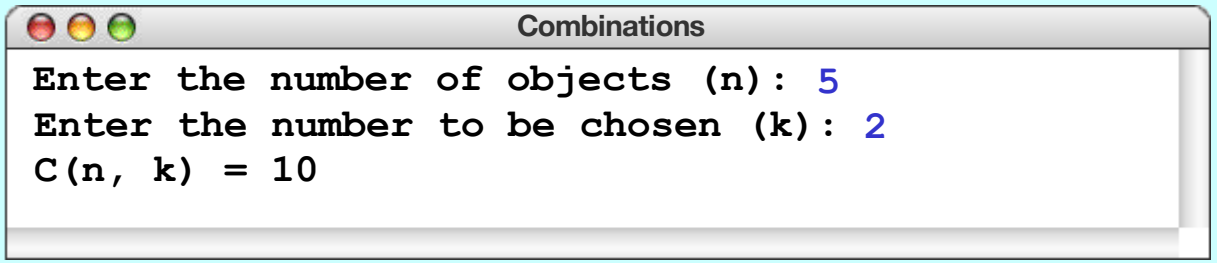
- The factorial of a number *n* (which is usually written as *n*! in mathematics) is defined to be the product of the integers from 1 up to *n*. Thus, 5! is equal to 120, which is 1×2×3×4×5.
- The following function definition uses a **`for`** loop to compute the factorial function:

```
int fact(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

- The next slide simulates the operation of **`combinations`** and **`fact`** in the context of a simple **`main`** function.

The Combinations Program

```
int main() {
   int n, k;
   cout << "Enter the number of objects (n): ";
   cin >> n;
   cout << "Enter the number to be chosen (k): ";
   cin >> k;
   cout << "C(n, k) = " << combinations(n, k) << endl;
   return 0;
}
```



Question

- What does this animation remind you of?

Exercise

- Write a function to swap the values of two integer variables.
- Given the following function/procedure:

```
void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

what is the result of the following calling to the function:

```
int n1 = 1, n2 = 2;
swap(n1, n2);
```

and why?

- The reason this function does not work as expected lies in the mechanism of calling a function discussed before.

Reference variables in C++

- A reference variable is an alias, that is, another name for an already existing variable.

```
int n1 = 1, n2 = 2;
int & x = n1, & y = n2; // x = ?, y = ?
n1 = 2, n2 = 1; // x = ?, y = ?
x = 1, y = 2; // n1 = ?, n2 = ?
```

- The `&` characters in these declarations indicate reference variables.
- A reference must be initialized when it is declared. Once a reference is initialized with a variable, it cannot be reassigned to refer to a different variable.
- Now both the variable name and the reference name may be used to refer to the same variable.

Call by Reference Example

- The following function swaps the values of two integers:

```
void swap(int & x, int & y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

- The arguments to **`swap`** must be assignable objects, which for the moment means variables, but cannot be constants.
- If you left out the **`&`** characters in the parameter declarations, calling this function would have no effect on the calling arguments because the function would exchange local copies.

Question

- What if I want to return more than one value?

  Use the parameters.

Reference Parameters

- C++ allows callers and functions to share information using a technique known as ***call by reference***.
- C++ indicates call by reference by adding an ampersand (**`&`**) before the parameter name. A single function often has both ***value parameters*** and ***reference parameters***, as illustrated by the **`solveQuadratic`** function:

```
int main() {
   double a, b, c, r1, r2;
   getCoefficients(a, b, c);
   solveQuadratic(a, b, c, r1, r2);
   printRoots(r1, r2);
   return 0;
}

void solveQuadratic(double a, double b, double c,
                    double & r1, double & r2) {
   if (a == 0) error("The coefficient a must be nonzero.");
   double disc = b * b - 4 * a * c;
   if (disc < 0) error("This equation has no real roots.");
   double sqrtDisc = sqrt(disc);
   r1 = (-b + sqrtDisc) / (2 * a);
   r2 = (-b - sqrtDisc) / (2 * a);
}
```

Reference Parameters

- Call by reference has two primary purposes:
  - It creates a sharing relationship that makes it possible to pass information in both directions through the parameter list.
  - It increases efficiency by eliminating the need to copy an argument. This consideration becomes more important when the argument is a large object.

Reference in C++

- In C++, a reference is a simple reference datatype that is less powerful but safer than the pointer type inherited from C.
- It can be considered as a new name for an existing object, but not a copy of the object it refers to.
- From a C++ programmer's point of view, a reference does not occupy any memory space; from a compiler implementer's point of view, a reference may be implemented just like a pointer (i.e., an address), therefore it may occupy the memory space of an address.
- We will delay the discussion of references until when we have a deeper understanding of pointers. For now, just remember a reference is a new name of an existing object, and use call by reference to pass information in both directions through the parameter list.

The End