Chapter 4
Implementing Free Functions

3rd Edition

Computing Fundamentals with C++

Rick Mercer
Franklin, Beedle & Associates

Goals

- Implement free functions
- Pass values to your functions as input
- Return values from your functions as output
- Test your new functions
- Begin to understand the scope of objects and functions

Implementing Free functions

- The general form of a fully defined function

  return-type identifier (*parameters*)

  block

- The *block* groups together a collection of statements between { and }
- General form of a block

```
{ // begin block
    statements
} // end block
```

The `return` Statement

- Many functions are designed to return a single value through the return statement

  `return` *expression*`;`

- When a return statement is encountered
  - the function terminates
  - the function call is replaced with *expression*

$f(x) = 2x^2-1$ in C++

```
#include <cmath>
#include <iostream>
using namespace std;

double f(double x) {
  double result;
  result = 2 * pow(x, 2.0) - 1.0;
  return result;
}

int main() {
  cout << f(2) << endl;   // 7
  cout << f(4.0) << endl; // 31
  return 0;
}
```

Another Example

```
#include <iostream>
using namespace std;
const double MONTHLY_FEE = 5.00;

double serviceCharge(int checks, int ATMs) {
  // pre:  checks >= 0 and ATMs >= 0
  // post: return banking fee based on local rules
  double result;
  result = (0.25*checks) + (0.10*ATMs) + MONTHLY_FEE;
  return result;
}

int main() {
  double fee = serviceCharge(10, 7);
  cout << fee << endl; // 8.2
  return 0;
}
```

Test Drivers

- The preceding `main` function was meant to show how to call a new function
- A *test driver* is a program with the sole purpose of testing a new function
- On the next slide, `main` is a *test driver*
  - It is designed to test a new function with different arguments representing different test cases
  - The `==` means equals
    - If the function call's actual return values matches the expected values, all lines of output should be 1 (true)

A test driver

```
int main() {
  cout << (serviceCharge(0, 0) == 5) << endl;     // 1
  cout << (serviceCharge(1, 0) == 5.25) << endl; // 1
  cout << (serviceCharge(0, 1) == 5.1) << endl;  // 1
  cout << (serviceCharge(1, 1) == 5.35) << endl; // 1
  cout << (serviceCharge(2, 1) == 5.6) << endl;  // 1
  cout << (serviceCharge(2, 2) == 5.7) << endl;  // 1
  return 0;
}
```

Output

```
1
1
1
1
1
1
```

Scope of Identifiers

- The scope of an identifier is the part of a program from which an identifier can be referenced
- An identifier's scope starts at its declaration and ends at the end of the file (global identifiers)
- Variables declared inside a function are only known in that function (local identifiers)

Scope of Function Names

- The scope of a function name
  - begins at the point of declaration and continues to the end of the file *unless the function name is re-declared*
    - So a function can not be called until after it has been declared
  - the scope extends into any file that includes that function
    - Therefore, math.h functions are known after

```
#include <cmath>
```

Scope of Identifiers

```
// Makes cout, cin, endl global; known everywhere
#include <iostream>
using namespace std; // Reduces the need to use std::
// Scope of MAX extends to end of this file
const int MAX = 1000;
// Scope of f goes to end of file (not above here)
double f(double x) {
  // Scope of x and temp is limited to function f
  double temp;
  // MAX can be referenced from f
  return x * MAX;
}

int main() {
// Scope of j is limited to main
  int j;
  cout << endl;
}
```

Reference parameters

- General Form of a reference parameter:

  type **&** *identifier*

- Examples

```
void swap(double & parm1, double & parm2)
// exchanges the values of two arguments

void moveToExit(Grid & g)
// move the mover until it as an exit

void decimals(ostream & os, int n)
// always show n decimal places for floats
```

Reference Parameters

- The special symbol `&` is called the reference symbol
- When placed before the parameter name, that parameter becomes an alias, another name for, the argument
- A change to a reference parameter therefore changes the associated argument
- This is called pass by reference, the argument's address, not the value, is copied to the function

void Functions

- All functions shown so far have a return type
- Sometimes functions do not need to return anything
- We use the return type of `void`

```
void printDetails(int n, string str) {
   cout << . . .
}
```

- Call void functions with no expectation of a return

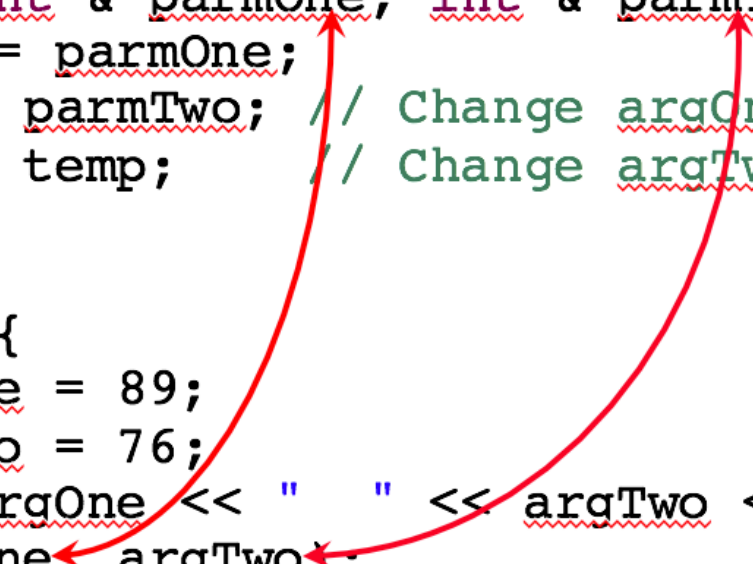
```
printDetails(99, "Adams University")
```

Need & to change two arguments

```
#include <iostream>
using namespace std;

void swap(int & parmOne, int & parmTwo) {
   int temp = parmOne;
   parmOne = parmTwo;  // Change argOne in main
   parmTwo = temp;     // Change argTwo in main
}

int main() {
   int argOne = 89;
   int argTwo = 76;
   cout << argOne << "   " << argTwo << endl; // 89 76
   swap(argOne, argTwo);
   cout << argOne << "   " << argTwo << endl; // 76 89
   return 0;
}
```



Reference Parameters

- `swap` has two reference parameters

  class-name `&` *identifier*

- The reference parameter `parmOne` acted as an alias for the argument `argOne`

- The change to `parmOne` also changed `argOne`
- The argument `argOne` was passed to the parameter `parmOne` by *reference*
- `parmOne` and `argOne` refer to the same object

What is the Output?

```
void add2(int & n1) {
  n1 = n1 + 2;
}

int main() {
  int anInt = 0;

  add2(anInt);
  add2(anInt);
  cout << anInt;
  return 0;
}
```

```
void add3(int n1) {
  n1 = n1 + 3;
}

int main() {
  int anInt = 0;

  add3(anInt);
  add3(anInt);
  cout << anInt;
  return 0;
}
```

`const` Reference Parameters

- C++ has four type of parameters
  1. value parameters—for passing the values of small objects such as `int`
  2. reference parameters—to allow a function to modify the state of one or more arguments
  3. `const` reference parameters—use `&` for safety and efficiency use `const` reference when inputting "large" objects to a function and you also do not want to alter the object
  4. pointer type parameter—allows a pointer to be passed to a function (not covered in this book)

`const` Reference Parameters

- A "big" object is one that consumes a lot of memory
- Here are the sizes of some classes

Type	Bytes
int	4
double	8
ostream	72

- Use `const` reference parameters when a large object is passed to a function and there should be no changes to that object
  - Often used when passing collections (Chapter 9)

`const` Reference Parameters

- When an object is passed by value
  - the function must allocate enough memory to hold a copy of that object *the argument*
  - all bytes of the object must be copied to the function
  - passing "big objects, slows down the program
- Solution: pass by reference--only 4 bytes needed
  - The argument can ***not*** be accidentally modified

Summary of Parameters

Value Parameter	Reference Parameter	const Reference
`int f1(int j)` Grab enough memory to store the entire object and copy the value to the function. An attempt to change the parameter is not an error--it simply has no effect on the argument.	`int f2(Vector<int> & b)` Grab about four bytes of memory to store the address of the object and copy that address to the function. Use this when you need to modify the argument. It's efficient too.	`void f(const Grid & g)` Like pass by reference, except the `const` means the argument can not be changed. To attempt a change, results in a compiletime error. This is the efficient and safe parameter passing mode.