

Chapter 5

Sending Messages

3rd Edition

Computing Fundamentals with C++

Rick Mercer

Franklin, Beedle & Associates

Goals

- Send messages to objects
- Learn some string and ostream messages and understand their effects
- Problem solve with string, Grid and BankAccount objects
- Appreciate why programmers partition software into classes, which are collections of member functions combined with their related data members

Find the objects in this specification

Problem Implement a bank teller application to allow bank customers to access bank accounts through an identification number. The customer, with the help of the teller, may complete any of the following transactions: withdraw money, deposit money, query account balances, and view any and all transactions between any two given dates. The system must maintain the correct balances for all accounts and produce monthly statements.

Nouns are potential classes

- Candidate Objects to model this new system
 - bank teller transaction
 - customers recent 10 transactions
 - bank account balance
- We will focus on one new `class BankAccount`

Operations and State

- Design is the choice of functions and values
- Member function names
 - deposit
 - withdraw
 - getBalance
 - getName
- State (values)
 - name
 - balance

Some objects need 2 or more arguments in the constructor

- `int`, `double`, and `string` objects are initialized with only one argument and optional use ()

```
int n = 0;
```

```
int n(0);
```

```
double x = 0.001;
```

```
double x(0.001);
```

```
string s = "Kim";
```

```
string s("kim");
```

- `BankAccount` objects require two arguments, initialization with `=` is not an option
- The special function is named `BankAccount`

```
BankAccount anAccount("Kim", 100.00);
```

Class Diagram: Name, State, Functions

BankAccount

string name

double balance

BankAccount(string initName, double
initBalance)

void deposit(double depositAmount)

void withdraw(double withdrawalAmount)

double getBalance() const

string getName() const

Messages

- *General Form*: sending a message to an object
object-name . function-name (arguments)

- Examples

```
anAccount.deposit(100.00);  
cout.width(6);  
cout << anAccount.balance() << endl;  
cout << aString.length() << endl;  
aGrid.move(3);
```


Messages

- Some messages return the object's state. Other messages tell an object to do something.
- Here is a *message* that asks the `BankAccount` object to return a value

```
cout << anAccount.balance() << endl;
```

- Here is a *message* that tells the object to do something:

```
anAccount.withdraw(25.00);
```

Example Program Needs BankAccount.h and BankAccount.cpp

```
#include <iostream>
using namespace std;
#include "BankAccount.h" // class BankAccount
int main() {
    BankAccount acct("Chris", 0.00);
    acct.deposit(222.22);
    acct.withdraw(20.00);

    cout << "Name: " << acct.getName() << endl;
    cout << "Balance: " << acct.getBalance() << endl;

    return 0;
}
```

Output

Name: Chris

Balance: 202.22

Object Diagram

- Every object has
 - a name: a variable that references the entire object
 - state: the values that the object currently has
- This object diagram shows the state of the object from the previous program

```
BankAccount anAcct
```

```
name = "Chris"
```

```
balance = 202.00
```

class string

- C++ has a type named `string` that is also implemented as a C++ class
- `string` objects store a collection of characters
- `string` objects are initialized with "string literals"
- The `string` class has many functions and operators
`length` `at` `find` `substr` `front` `back` `insert`
`[]` `+` `<<` `>>`

string member functions

length at find

```
#include <iostream>
#include <string> // class string
using namespace std;

int main() {
    // Initialize a string:
    string aString("Chris Boatright");
    // How many characters are in aString:
    cout << aString.length() << endl;           // 15
    // Show the first character at index 0
    cout << aString.at(0) << endl;             // 'C'
    // Return the index where "Boat" is found
    cout << aString.find("Boat") << endl;      // 6
    return 0;
}
```

string member functions

- `substr` returns a newly constructed [string](#) object
`string substr (int pos, int len)`
- The substring is the portion of the object that starts at index `pos` and spans `len` characters (or until the end of the string, whichever comes first)

```
string aString("Kim Thatcher");  
cout << aString.substr(0, 1) << endl; // "K"  
cout << aString.substr(0, 2) << endl; // "Ki"  
cout << aString.substr(0, 3) << endl; // "Kim"  
cout << aString.substr(5, 7) << endl; // "hatcher"  
// Go to the end with 99, which is 7 characters  
cout << aString.substr(5, 99) << endl; // "hatcher"
```

string operators [] and +

- The [] operator is like the at () function
- The [] operator returns a single character at the given index
- The + operator concatenates two strings into one

```
string aString("Kim Thatcher");
```

```
cout << aString[0] << endl; // K
```

```
cout << aString[1] << endl; // i
```

```
cout << aString[2] << endl; // m
```

```
// Output: Mr. or Mrs. Kim Thatcher
```

```
cout << "Mr. or Mrs. " + aString << endl;
```

string operators >> and <<

- `string` also overloads the input and output operators `>>` `<<` to allow use with `cin` and `cout`

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string name;
    cout << "Enter your name: ";
    cin >> name;
    cout << "Hello " + name << endl;
}
```

Dialog

```
Enter your name: Chris
Hello Chris
```


classes ostream and istream

- class ostream allows for program output
- class istream allows for program input
- Not many useful istream functions yet

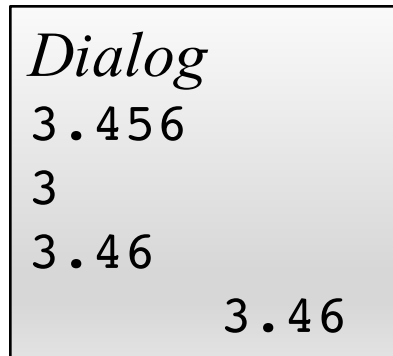
```
#include <iostream>
#include <string>
using namespace std;
int main() {
    int anInt;
    cout << cin.good() << endl;
    cin >> anInt; // Enter a bad integer
    cout << cin.good();
    return 0;
}
```

<i>Dialog</i>
1
abc
0

classes ostream and istream

- class ostream has a few functions useful for formatting output
 - precision rounds floats to a specific number of digits
 - width specifies the number of columns

```
double x = 3.456;
cout << x << endl; // 3.456
cout.precision(1); // Modify cout
cout << x << endl; // 3
cout.precision(3); // Modify cout
cout << x << endl; // 3.46
cout.width(9);
cout << x << endl; // 3.46
```



Dialog
3.456
3
3.46
3.46

Class Member Function Headings

- Member function names are distinguished from non-member functions by qualifying the operation with the class-name and `::` (the scope resolution operator)
 - Example function names as you might see them referred to in the text
 - `ostream::width`
 - `istream::good`
 - `string::length`
 - `string::substr`
 - `BankAccount::withdraw`

Class member function headings

- Member function headings are also qualified
- Here are the member functions used so far:

```
int string::length()
```

```
int string::find(string subString)
```

```
string string::substr(int pos, int n)
```

```
int ostream::width(int nCols)
```

```
int ostream::precision(int nDigits)
```

```
int istream::good()
```

```
double BankAccount::withdraw(double amount)
```

```
void BankAccount::deposit(double amount)
```

Why qualify member function headings?

- Free functions do not belong to a class
- The previous class member function headings indicate the class to which the function belongs with *class-*

name:::

```
string string::substr(int pos, int n)
// post: return n characters of this string
// beginning at position pos
```

- You will have to do this when implementing C++ classes later

Another nonstandard class `Grid`

- A *Grid* object
 - stores a rectangular map made up of rows and columns
 - has an object to move around.
 - is initialized with five arguments

```
Grid Grid_name(int rows, int cols,  
               int mover_row, int mover_col,  
               Direction direction);
```

- `Direction` is either `north` `south` `east` or `west`

```
Grid aGrid(7, 14, 5, 8, east);  
// Column 8 is the ninth column
```

Example

```
#include "Grid.h" // for class Grid
int main() {
    Grid aGrid(5, 10, 0, 0, east);
    aGrid.display();
}
```

Program Output:

The Grid:

```
> . . . . .
. . . . .
. . . . .
. . . . .
. . . . .
```

Access the state of a Grid object

- We observe state of Grid objects with `Grid::display` (*const means a display message does not modify the Grid object*)

```
void Grid::display() const
// post: The current state of the Grid
// is displayed on the computer screen
```

- Also access the state of Grid objects with

```
Grid::row        // the row the mover is in
Grid::column     // the column the mover is in
Grid::nRows      // the maximum number of rows
Grid::nColumns   // the maximum number of rows
```


Member Functions that Modify Grid objects

```
void Grid::move(int nSpaces)
// pre:  The mover has no obstructions in the next nSpaces
// post: the mover has moved nSpaces forward

void Grid::putDown(int putDownRow, int putDownCol)
// pre:  The intersection (putDownRow, putDownCol) has
// nothing on it expect the mover
// post: There is one thing at the intersection

void Grid::pickUp()
// pre:  There is something to pickup at the movers location
// post: There is nothing to pick up

void Grid::turnLeft()
// post: The mover is facing 90 degrees counter-clockwise

void Grid::block(int blockRow, int blockCol)
// pre:  There is nothing at all at the intersection
// post: The intersection can no longer be used
```

Failing to Meet the Preconditions

- There are many "illegal" messages you can send to a `Grid` object
 - send a message to move through a block ('#')
 - send a message telling the mover to move off the edge of the world
 - send a `pickUp` message when there is nothing to pickup

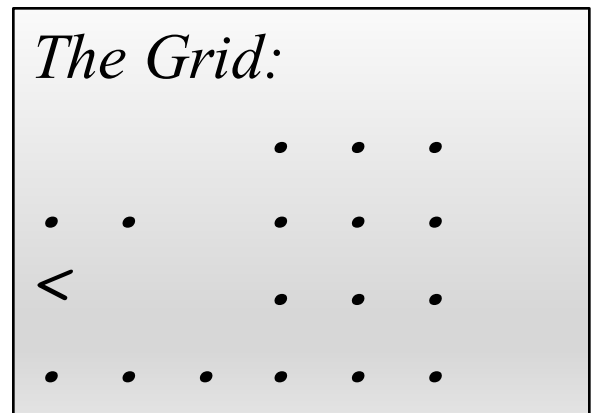
So what are we to do?

- A precondition is a statement the client must ensure is true before sending a message
- If the client ignores it, the resulting behavior is undefined--tough luck
- You can experiment and see what happens when you run programs with Grid objects

One small Grid program

```
#include "Grid.h"    // for class Grid

int main() {
    Grid aGrid(4, 6, 0, 0, east);
    aGrid.move();
    aGrid.move();
    aGrid.turnRight();
    aGrid.move();
    aGrid.move();
    aGrid.turnRight();
    aGrid.move();
    aGrid.move();
    aGrid.display();
}
```



Why Functions and Classes?

- Abstraction
 - has many meanings
 - is the process of pulling out and highlighting the relevant features of a complex system
 - allows us to use existing functions and classes more easily
 - allows us to use existing software without knowing all the implementation details *how it works*

Functions hide a lot of detail

- One function call can represent many statements

Operation	The object-oriented way	Statements
Construct one Grid object	<code>Grid g(15,15,9,4,east);</code>	35
Move in current direction	<code>g.move(2);</code>	112
Output the Grid	<code>g.display();</code>	6
Change direction	<code>g.turnRight();</code>	15

Reasons for functions

- Can reuse existing well-tested code rather than write it and test it from scratch
- To concentrate on the bigger issues at hand
- To reduce errors by writing the function only once and testing it thoroughly
 - Programs that once had 1,000 statements in main might now have 100 functions that are 10 lines long
 - With object-orientation, it could be 10 classes with 10 member functions, that have 10 statements each

Structured Programming

Object-Oriented Programming

- Structured Programming
 - Partition programs by functions
 - The data is passed around from one non-member function to another
- Object-Oriented Programming
 - Partition programs by classes
 - The data is safely encapsulated functions with the functions that make up the type