Beginning C++ Game Programming
Chapter Discussion Question and Exercise Solutions

Chapter 1

Discussion Questions

1. How does having a widely adopted C++ standard help game programmers?

Solution:
Having a widely adopted standard helps game programmers in several ways. First, it helps to ensure consistency among compilers—this means that the same program, written to the standard, should compile successfully across multiple compilers that implement the standard. Second, the standard makes it easier to write cross-platform code—code written to the standard should compile and work correctly across different operating systems (again, given compilers that faithfully implement the standard). Third, it helps ensure that multiple programmers can work more seamlessly together—if both are writing to the standard their code should have fewer conflicts.

2. What are the advantages and disadvantages of employing the using directive?

Solution:
The major advantage of a employing the using directive is that it saves typing. If a programmer puts `using namespace std;` in his program, he saves having to prefix every element in the `namespace std` with `std::`. One could also argue that removing all of the `std::` references from a program makes it less cluttered and easier to read. A disadvantage of employing a using directive is that it may not be clear where different objects and functions originate—what namespace they're from. Another disadvantage with employing a using directive is that you run the risk of name conflicts. For example, if you employed the using directive for two namespaces that had elements with the same name, there would be a name conflict. This, of course, is the very thing that namespaces were created to prevent.

3. Why might you define a new name for an existing type?

Solution:
You might define a new name for an existing type if you simply wanted a shorter name for a type that you use often in a program. For example, you might do something like:

```
typedef unsigned short int ushort;
```

so that you can use the concise `ushort` instead of the much longer `unsigned short int`. But you could also argue that the name a programmer creates for an existing type might be clearer than the syntax for the existing type. For example, `ushort` might scan better than the longer `unsigned short int`.

4. Why are there two versions of the increment operator? What's the difference between them?

Solution:
Both versions of the increment operator increment a variable, but there's a subtle and important difference in the way the two operators work. The prefix increment operator is placed before the variable to be incremented, as in `++score`, while the postfix increment operator is placed after the variable to be incremented, as in `score++`. The prefix increment operator increments a variable before the evaluation of a larger expression involving the variable while the postfix increment operator increments a variable after the evaluation of a larger expression involving the variable.

5. How can you use constants to improve your code?

Solution:
Constants can provide two important benefits. First, they can make programs clearer. `MAX_HEALTH` more clearly conveys the intention of a value than some literal, like say `100`. Second, constants make changes easier. If you want to change the value of a constant, you only need to make a change in one place: where it was defined. If you used the same literal value throughout a program, you'd have to change that literal everywhere (while making sure not to change the literal value where it's not related to the constant value).

Exercises

1. Create a list of six legal variable names -- three good and three bad choices. Explain why each name falls into the good or bad category.

Solution:
Responses will vary, but the following is a set of possible answers:

Good Names

`health`	A clear, short name
`numEnemies`	Clear that variable represents a number; descriptive
`isGameOver`	Clear that variable represents a `bool`

Bad Names

`HeAlTh`	While it's legal to used a mixed-case name, it's unconventional and distracting
`TotalNumberofCurrentEnemies`	While it may be clear, the name is cumbersome; there must be a shorter, yet-still-clear name
`igo`	Short but not clear; a little more typing may be worthwhile for the sake of clarity

2. What's displayed by each line in the following code snippet? Explain each result.

```
cout << "Seven divided by three is " << 7 / 3 << endl;
cout << "Seven divided by three is " << 7.0 / 3 << endl;
cout << "Seven divided by three is " << 7.0 / 3.0 << endl;
```

Solution:
`cout << "Seven divided by three is " << 7 / 3 << endl;` displays `2`. That's because both numbers in the expression `7 / 3` are integers, making the operation integer division, which always results in an integer.

`cout << "Seven divided by three is " << 7.0 / 3 << endl;` displays `2.33333`. That's because at least one number in the expression `7.0 / 3` is a floating point number. A division operation with at least one floating point number yields a floating point result.

`cout << "Seven divided by three is " << 7.0 / 3.0 << endl;` displays `2.33333`. That's because at least one number in the expression `7.0 / 3.0` is a floating point number. A division operation with at least one floating point number yields a floating point result.

3. Write a program that gets three game scores from the user and displays the average.

Solution:
The source code is in the file `bcppgp_solutions\source\chap01\chap01_exercise03.cpp`.

Chapter 2

Exercises

1. Rewrite the Menu Chooser program from this chapter using an enumeration to represent difficulty levels.

Solution:

The source code is in the file `bcppgp_solutions\source\chap02\chap02_exercise01.cpp`.

2. What's wrong with the following loop?

```
int x = 0;
while (x)
{
    ++x;
    cout << x << endl;
```

```
}
```

Solution:
The problem is that since `x` is set to `0` just before the loop, the loop will never be entered—it may as well not exist in the program. A student might say that the loop, if entered, would be infinite, but that's not necessarily true. If `x` were initialized with a negative value, it would eventually be incremented to 0, at which point the loop would end.

3. Write a new version of the Guess My Number program in which the player and the computer switch roles. That is, the player picks a number and the computer must guess what it is.

Solution:

The source code is in the file `bcppgp_solutions\source\chap02\chap02_exercise03.cpp`.

Chapter 3

Discussion Questions

1. What are some of the things from your favorite game that you could represent as objects? What might their data members and member functions be?

Solution
Responses will vary, but the following is a set of possible answers:

Ogre
Data Members

`position`

`speed`

`health`

Methods

`Move()`

`Attack()`

`Die()`

Sword
Data Members

`damage`

`weight`

`price`

Methods

`Swing()`

2. What are the advantages of using an array over a group of individual variables?

Solution:
While you could represent a set of values with individual variables, using an array has advantages – especially if the values are related and will need to have the same operations applied to them. Using an array allows you to easily iterate through a set of values instead of being forced to access a group of individual variables. For example, if you wanted to represent a set of 10 high scores with individual variables, you'd have to send each variable to `cout` to display all of the values, as in:

```
cout << "High Scores";
cout << score1;
cout << score2;
cout << score3;
cout << score4;
cout << score5;
cout << score6;
cout << score7;
cout << score8;
cout << score9;
cout << score10;
```

But using an array allows a shorter and more elegant solution:

```
for (int i = 0; i < numScores; ++i)
    cout << score[i] << endl;
```

3. What are some limitations imposed by a fixed array size?

Solution:
The major limitation imposed by a fixed array is that it can't grow in size. Once you establish the number of elements for the array, you can't change it. So, as a game programmer, you're forced to know the largest possible number of elements you will need to store with an array. Another disadvantage is that a fixed array can't shrink in size. Once you declare an array of a certain size, the memory needed to store all of the values in the array is allocated, even if only a few values are actually ever stored in the array as elements.

4. What are the advantages and disadvantages of operator overloading?

Solution:
An advantage of operator overloading is that using operators we already know with new types can be clear and require no special explanation. For example, using the `+` operator with the `string` objects intuitively represents the concatenation (the adding) of two strings. A disadvantage of operator overloading is that a programmer may feel that the use of an operator is clear when it isn't to most other programmers.

5. What kinds of games could you create using string objects, arrays, and for loops as your main tools?

Solution:
Responses will vary, but the following is a set of possible answers:

Hangman	A player gets a certain number of tries to guess the letters in a word, if he fails to guess in time, he loses and is hanged
Antonyms	A player is given a word and must supply its opposite
Cryptograms	A player is given an encrypted message where each letter in the message is substituted with a different letter from the alphabet

Exercises

1. Improve the Word Jumble game by adding a scoring system. Make the point value for a word based on its length. Deduct points if the player asks for a hint.

Solution:

The source code is in the file `bcppgp_solutions\source\chap03\chap03_exercise01.cpp`.

2. What's wrong with the following code?

```
for (int i = 0; i <= phrase.size(); ++i)
    cout << "Character at position " << i << " is: " << phrase[i] << endl;
```

Solution:
Given that `phrase` is a `string` variable, the final iteration of the loop attempts to access an element beyond the last character of `phrase`. This could lead to disastrous results, including a program crash. The loop attempts to access this nonexistent character because the test of the loop, `i <= phrase.size()`, is true when `i` is equal to `phrase.size()`. The problem is that the valid element positions of `phrase` range from 0 to `phrase.size() - 1`.

3. What's wrong with the following code?

```
const int ROWS = 2;
const int COLUMNS = 3;
char board[COLUMNS][ROWS] = { {'O', 'X', 'O'},
                              {' ', 'X', 'X'} };
```

Solution:
The code will fail to compile with an message such as, "error: too many intializers for 'char[2]'." This happens because the code declares an array 2x3 and tries to initialize its elements with data in a 3x2 grid. The initialization attempts to set values for array elements that don't exist. The corrected code is:

```
const int ROWS = 2;
const int COLUMNS = 3;
char board[ROWS][COLUMNS] = { {'O', 'X', 'O'},
                              {' ', 'X', 'X'} };
```

Chapter 4

Discussion Questions

1. Why should a game programmer use the STL?

Solution:
The STL gives a game programmer a tested and honed set of useful containers, iterators and algorithms. This saves a game programmer the time and effort of creating his own data structures and algorithms. Even better, it saves a game programmer from spending time and effort writing less efficient data structures and algorithms. Now, there are occasions when a game programmer might not use the STL. For example, if there are unique requirements for working with collections of objects that are best met by a custom data structure. In addition, the STL imposes some extra memory requirements that might make it less attractive for some game consoles. It's also possible that on unique platforms, such as some game consoles, the STL might not be as mature as it is on the PC. But overall, the STL should be the first place a game programmer looks to meet all of his data structure needs.

2. What are the advantages of a vector over an array?

Solution:
The main advantage of a vector is that it is dynamic: it can grow or shrink as needed. This can result in a huge memory savings. With an array, a game programmer has to acquire all the memory necessary for the maximum number of elements he expects the array to ever hold during any run of the program -- and that memory is captured by the program regardless of how little the program actually needs for the elements. Conversely, the vector can use only the memory required by the number of elements that needs to be stored in it. With a vector, a game programmer doesn't have to know the

maximum number of elements he'll need to store ahead of time. Another advantage is that a vector can be manipulated with many of the STL algorithms. So it's like the game programmer gets commonly used algorithms for free. Vectors do require a little more memory than arrays, but this tradeoff is almost always worth the benefits.

3. What types of game objects might you store with a vector?

Solution:
You can store any collection of game objects with a vector, but the best use of this container is for game objects that make up a sequence. Of course, the vector can be used for random access based on a position number as well. A vector could be used to store all kinds of game objects. Here are just a few examples: a player's inventory, a list of weapons a player can cycle through, a list of high scores and a list of all the players in a multiplayer game.

4. How do performance characteristics of a container type affect the decision to use it?

Solution:
Understanding the performance of a container along with how you plan to use the container are critical in deciding which container from the STL best suits your needs. Different container types have different performance properties. They're like tools in a toolbox; each tool is best suited for a different job.

As an example, let's look at vectors and lists. Lists and vectors are two different types of sequence containers, each with their own performance characteristics. With lists, inserting and erasing elements takes the same amount of time no matter where in the sequence you insert or delete. This is not the case with vectors because inserting or deleting an element in the middle of a vector requires all of the elements after the insertion or deletion point to be moved. However, vectors are faster when it comes to adding or deleting an element at the end of a sequence. Vectors are also faster to traverse than lists. So, if a game programmer needs to represent a sequence of objects and he knows that he will rarely insert into or delete from the middle of the sequence, a vector is probably his best bet. But, if the game programmer knows that he will have a large sequence and plans to do a lot of insertion and deletion in the middle of that sequence, a list may be his best choice.

5. Why is program planning important?

Solution:
Planning your programs will almost always result in time and frustration saved. Programming is a lot like construction. If a contractor were to build a house for you without a blueprint, you might end up with a house that has 12 bathrooms, no windows, and a front door on the second floor. Plus, it probably would cost you 10 times the

estimated price. Programming is the same way. Without a plan, you'll likely struggle through the process and waste time. You might even end up with a program that doesn't quite work or a program that is difficult to modify.

Exercises

1. Write a program using vectors and iterators that allows a user to maintain a list of his or her favorite games. The program should allow the user to list all game titles, add a game title, and remove a game title.

Solution:

The source code is in the file `bcppgp_solutions\source\chap04\chap04_exercise01.cpp`.

2. Assuming that `inventory` is a vector that holds `string` object elements, what's wrong with the following code snippet?

```
vector<int>::const_iterator iter;
cout << "Your items:\n";
for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    cout << iter << endl;
```

Solution:
The code in the loop body sends the iterator `iter` to `cout` instead of the `string` object that `iter` references. To fix the code, change the line:

```
    cout << iter << endl;
```

to:

```
    cout << *iter << endl;
```

3. Write pseudocode for the Word Jumble game from Chapter 3.

Solution:
Create a collection of secret words and hints
Randomly, pick a secret word and its corresponding hint
Create a jumble from the secret word
Welcome the player to the Word Jumble Game
Display the jumble
Get the player's guess

While the player hasn't guessed the word or hasn't asked to quit
  If the player asked for a hint
    Show the hint
  Otherwise
    Tell the player that he guessed incorrectly

Get the player's next guess

If the player has guessed the secret word
Congratulate the player

Thank the player for playing

Chapter 5

Discussion Questions

1. How does function encapsulation help you write better programs?

Solution:
Function encapsulation helps you write better programs by sectioning off the code in functions from the rest of your program. First, this sectioning off allows you to focus on just one problem at a time. Second, it keeps all of a function's local variables separate from variables in the rest of the program. This way, you don't have to worry about potential name collision. For example, if you write function and declares a variable `count`, you don't have to worry about some other function also declaring a variable named `count`. The two functions are completely separate and the two `count` variables live independently from each other.

2. How can global variables make code confusing?

Solution:
Global variables can make code confusing because it can be difficult to tell where and under what circumstances the global variables change. The problem arises from the fact that a global variable can be accessed from any part of a program. This is unlike working with function parameters where it's clear which variables are being passed to which functions. In general, it's a good idea to try to limit the number of global variables in a program.

3. How can global constants make code clearer?

Solution:
Global constants can make a program clearer by providing an identifier instead of a literal value. For example, imagine that a game programmer is writing an action game in which he wants to limit the total number of enemies that can blast the poor player at once to ten. Instead of using a numeric literal everywhere, such as 10, the game programmer could define a global constant `MAX_ENEMIES` that's equal to 10. This is an example of self-

documenting code in action. The identifier name itself, `MAX_ENEMIES`, helps programmers reading the code to understand its meaning. The literal 10 does not.

4. What are the pros and cons of optimizing code?

Solution:
At first blush, it may seem like there is no downside to optimizing code. The benefit of optimizing is of course clear: code executes more quickly. This can translate into a more responsive game or a higher frame rate for games with graphics. And what game programmer wouldn't want his code to execute faster? But there are of course cons to code optimization. The biggest is that the amount of work that goes into optimizing code may not justify the speed gains. For example, a game programmer could spend dozens of hours eking out only a small performance gain. Also, not all code is performance critical. For example, the menu code of a game probably doesn't need to be highly tuned. The best optimization strategy is to wait until the game is stable and closer to completion. Code can then be profiled and any bottlenecks can be examined for potential optimization.

5. How can software reuse benefit the game industry?

Solution:
The benefits of software reuse include:

- Increased company productivity. By reusing code and other elements that already exist, such as a graphics engine, game companies can get their projects done with less effort.
- Improved software quality. If a game company already has a tested piece of code, such as a networking module, then the company can reuse the code with the knowledge that it's bug-free.
- Improved software performance. Once a game company has a high-performance piece of code, using it again not only saves the company the trouble of reinventing the wheel, it saves them from reinventing a less efficient one.

Exercises

1. What's wrong with the following prototype?

```
int askNumber(int low = 1, int high);
```

Solution:
The prototype is illegal because a default value is specified for the first parameter (`low`) but not the second parameter (`high`). Once you specify a default argument in a list of parameters, you must specify default arguments for all remaining parameters.

2. Rewrite the Hangman game from Chapter 4 using functions. Include a function to get the player's guess and another function to determine whether the player's guess is in the secret word.

Solution:

The source code is in the file `bcppgp_solutions\source\chap05\chap05_exercise02.cpp`.

3. Using default arguments, write a function that asks the user for a number and returns that number. The function should accept a string prompt from the calling code. If the caller doesn't supply a string for the prompt, the function should use a generic prompt. Next, using function overloading, write a function that achieves the same results.

Solution:

The source code is in the file `bcppgp_solutions\source\chap05\chap05_exercise03.cpp`.

Chapter 6

Discussion Questions

1. What are the advantages and disadvantages of passing an argument by value?

Solution:
The main advantage of passing an argument by value is that the object being passed can't accidentally be changed by the function to which it was passed. The main disadvantage of passing an argument by value is that the program must make a copy of the object for the receiving function. If a large object is being passed by value, this can result in unnecessary overhead in terms of memory and processor time.

2. What are the advantages and disadvantages of passing a reference?

Solution:
The main advantage of passing a reference is efficiency. When you pass a reference to an object, you essentially pass a small handle to that object, rather than the entire object itself. For larger objects, this can result in processor time and memory savings. For simple, built-in types, there is no real time or memory savings. The disadvantage of passing a reference is that the receiving function can alter the object being passed to it. This is something to avoid and can be solved by passing a constant reference.

3. What are the advantages and disadvantages of passing a constant reference?

Solution:

The advantage of passing a constant reference to a function is that you get the efficiency of passing a reference without the possibility of the receiving function altering the original object. This is important because it can become unclear how objects are changed if objects used as argument can be changed by the functions to which they're passed. In general, you should avoid changing the value of an object passed to a function. The disadvantage of passing a constant reference is that it does take extra syntax. It's easy to leave out the necessary `const`, but the extra typing is worth the added safety.

4. What are the advantages and disadvantages of returning a reference?

Solution:
The advantage of returning a reference is the same as passing a reference: efficiency. When you return a reference to an object, you essentially return a small handle to that object, rather than the entire object itself. For larger objects, this can result in processor time and memory savings. For simple, built-in types, there is no real time or memory savings. One disadvantage of returning a reference is that it's possible to return a reference to an out of scope object, which is illegal. A common way for this to occur is through returning a reference to a local variable.

5. Should game AI cheat in order to create a more worthy opponent?

Solution:
This is a question that can generate passion from both sides of the argument – and there is no "right" answer. Some game programmers will say that the goal of game AI is simply to provide a fun challenge to gamers and the way in which that challenge is provided doesn't matter. So, for example, if a computer opponent in a poker game "cheats" by examining the player's facedown cards, it doesn't matter – so long as the computer opponent creates an entertaining challenge. Others will say AI should play by the same rules as the human player, otherwise the player will feel shortchanged – especially if the player can tell that the computer is "cheating."

Exercises

1. Improve the Mad Lib game from Chapter 5 by using references to make the program more efficient.

Solution:

The source code is in the file `bcppgp_solutions\source\chap06\chap06_exercise01.cpp`.

2. What's wrong with the following program?

```
int main()
{
```

```
    int score;
    score = 1000;
    float& rScore = score;
    return 0;
}
```

Solution:
`rScore` is a reference for a `float`, but the program attempts to use it to refer to an `int` value, which is illegal.

3. What's wrong with the following function?

```
int& plusThree(int number)
{
    int threeMore = number + 3;
    return threeMore;
}
```

Solution:
The function attempts to return a reference to a local variable, which can lead to disastrous results.

Chapter 7

Discussion Questions

1. What are the advantages and disadvantages of passing a pointer?

Solution:
The main advantage of passing a pointer is efficiency. When you pass a pointer to an object, you only pass the memory address of the object rather than the entire object itself. For larger objects, this can result in processor time and memory savings. For simple, built-in types, there is no real time or memory savings. One disadvantage of passing a pointer is that the receiving function can alter the object being passed to it. This is something to avoid when possible. Another disadvantage of passing a pointer is the extra syntax that's involved in dereferencing the pointer to get to the object to which the pointer points.

2. What kinds of situations call for a constant pointer?

Solution:
Any situation where you need the pointer to point only to the object it was initialized to point to. This works a lot like a reference; however, you should use a reference over a

constant pointer whenever possible. There may be times where you have to work with a constant pointer – for example, if you're working with a game engine that returns a constant pointer.

3. What kinds of situations call for a pointer to a constant?

Solution:
Any situation where you need the pointer to point only to an object that can't be changed. There may be times where you have to work with a pointer to a constant – for example, if you're working with a game engine that returns a pointer to a constant.

4. What kinds of situations call for a constant pointer to a constant?

Solution:
Any situation where you need the pointer to point only to the object it was initialized to point to and where that object itself can't be changed. This works a lot like a constant reference; however, you should use a constant reference over a constant pointer to a constant whenever possible. There may be times where you have to work with a constant pointer to a constant – for example, if you're working with a game engine that returns a constant pointer to a constant.

5. What kinds of situations call for a non-constant pointer to a non-constant object?

Solution:
Any situation where you need the pointer and the object to be able to be changed. There may be times where you have to work with a non-constant pointer to a non-constant object, but you should opt for using references over pointers whenever possible.

Exercises

1. Write a program with a pointer to a pointer to a string object. Use the pointer to the pointer to call the `size()` member function of the string object.

Solution:

The source code is in the file `bcppgp_solutions\source\chap07\chap07_exercise01.cpp`.

2. Rewrite the final project from Chapter 5, the Mad Lib Game, so that no string objects are passed to the function that tells the story. Instead, the function should accept pointers to string objects.

Solution:

The source code is in the file `bcppgp_solutions\source\chap07\chap07_exercise02.cpp`.

3. Will the three memory addresses displayed by the following program all be the same? Explain what's going on in the code.

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

int main()
{
    int a = 10;
    int& b = a;
    int* c = &b;

    cout << &a << endl;
    cout << &b << endl;
    cout << &(*c) << endl;

    return 0;
}
```

Solution:
Each line will display the memory address where the literal 10 is stored, so they will all display the same value. The line `cout << &a << endl;` simply displays the memory address of the variable a, which stores the literal 10. The line `cout << &b << endl;` displays the memory address of `b`. But since `b` is a reference to `a`, the line simply displays the memory address of the variable `a`, which stores the literal 10. The line `cout << &(*c) << endl;` displays the memory address of the variable pointed to by `c`. Since `c` points to `b` and `b` is a reference to `a`, the line displays the memory address of the variable `a`, which stores the literal 10.

Chapter 8

Discussion Questions

1. What are the advantages and disadvantages of procedural programming?

Solution:
The main advantage of procedural programming is that it allows you to break up long programs into parts, like C++ functions. This helps game programmers focus on programming one task for each function. Large programs can be divided and conquered. In addition, functions are encapsulated, which saves game programmers the headaches of *name collision* where an identifier is defined more than once in the same scope. The

main disadvantage of procedural programming is that it separates functions from data, which seem to naturally go together, especially in game programming.

2. What are the advantages and disadvantages of object-oriented programming?

Solution:

The major advantage of object-oriented programming is that it combines data and functions into one unit: the object. This is helpful to game programmers because so many things that we want to represent in games are objects – everything from swords and shields to ships and docking stations. Object-oriented programming can also be used to define relationships among objects in a game. For example, a healing potion could be defined to cause a creature's health value to increase. Object-oriented programming also helps with code reuse as new classes of objects can inherit and be based on existing classes. A disadvantage of object-oriented programming is that it might introduce unnecessary complexity for simple programs.

3. Are accessor member functions a sign of poor class design? Explain.

Solution:
This is a topic that can be argued from either side. Some game programmers say you should never provide access to data members through accessor member functions because even though this kind of access is indirect, it goes against the idea of encapsulation. Instead, they say you should write classes with member functions that provide the client with all of the functionality the client could require, eliminating the client's need to access a specific data member. Other game programmers would argue that by providing an accessor member function you are providing indirect access and therefore a class can protect its object's data members (for example, by making them read-only). These game programmers would say that accessor member functions provide an important and necessary convenience.

4. How are constant member functions helpful to a game programmer?

Solution:
Using constant member functions makes your intentions clear to other programmers – that a member function doesn't modify an object's data members. It can also help catch errors at compile time. For example, if you inadvertently change a data member in a constant member function, your compiler will complain, alerting you to the problem.

5. When is it a good idea to calculate an object's attribute on the fly rather than storing it as a data member?

Solution:
It's a good idea to calculate an object's "attribute" on the fly rather than storing it as a data member when that "attribute" depends on other attributes. For example, in the Critter Caretaker program in this chapter, the critter's mood "attribute" is not stored as a data member, but rather, is calculated based on the critter's current hunger and boredom levels. But to client code, getting the critter's mood looks like indirectly accessing any object attribute since the critter's mood can be retrieved through a call to the object's `GetMood()` method. If you didn't calculate a critter's mood on the fly, then every time the critter's hunger or boredom level changed, you would also have to potentially make a change to an attribute that stored its mood.

Exercises

1. Improve the Critter Caretaker program so that you can enter an unlisted menu choice that reveals the exact values of the critter's hunger and boredom levels.

Solution:

The source code is in the file `bcppgp_solutions\source\chap08\chap08_exercise01.cpp`.

2. Change the Critter Caretaker program so that the critter is more expressive about its needs by hinting at how hungry and bored it is.

Solution:

The source code is in the file `bcppgp_solutions\source\chap08\chap08_exercise02.cpp`.

3. What design problem does the following program have?

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

class Critter
{
public:
    int GetHunger() const {return m_Hunger;}
private:
    int m_Hunger;
};

int main()
{
```

```
    Critter crit;
    cout << crit.GetHunger() << endl;
    return 0;
}
```

Solution:
The problem is that there is no way to set the value of a `Critter` object's `m_Hunger` data member. In fact, the preceding program creates a `Critter` object with an undefined value for the `m_Hunger` data member – it could hold any value when the object is created.

Chapter 9

Discussion Questions

1. What types of game entities could you create with aggregation?

Solution:
Using aggregation you could create game entities that are collections of other objects or that are composed of other objects. For example, a `Squadron` object could be a collection of `Soldier` objects; a `Battalion` object could be a collection of `Squadron` objects; a `Regiment` object could be a collection of `Battalion` objects and so on. Another example is a `Tank` class that has exactly one `Gun` object and one `Tread` object as attributes.

2. Do friend functions undermine encapsulation in OOP?

Solution:
Although friend functions may seem to violate encapsulation because they can access any member of a class to which they are friends, they don't violate encapsulation. Friend functions are just part of the interface to a class. Using a friend function is preferable to making class data members public.

3. What advantages does dynamic memory offer to game programs?

Solution:
Dynamic memory allows game programmers to create new objects on the fly. This frees game programmers from having to know, before a program is run, exactly how many objects of a given type will be in memory at any one time. This can help to make game programs more flexible and efficient. Using dynamic memory also provides a way for objects created in one function to be used in another without making a copy of the object – instead, the function can return a pointer to the object on the heap.

4. Why are memory leaks difficult errors to track down?

Solution:
A memory leak is difficult to track down because the error is not always obvious. Memory leaks can be small and their existence may not be apparent. But small memory leaks can become big ones. And when a memory leak is big enough, it can affect system performance or even cause a program to crash. A memory debugger can be used to help find memory leaks.

5. Should objects that allocate memory on the heap always be required to free it?

Solution:
In general, it's a good idea to have objects that allocate memory on the heap be responsible for freeing that memory. This can be accomplished in the object's destructor. Although garbage collection can automatically reclaim memory that's no longer in use, there is no single garbage collection solution that's part of Standard C++.

Exercises

1. Improve the `Lobby` class from the Game Lobby program by writing a friend function of the `Player` class that allows a `Player` object to be sent to `cout`. Next, update the function that allows a `Lobby` object to be sent to `cout` so that it uses your new function for sending a `Player` object to `cout`.

Solution:

The source code is in the file `bcppgp_solutions\source\chap09\chap09_exercise01.cpp`.

2. The `Lobby::AddPlayer()` member function from the Game Lobby program is inefficient because it iterates through all of the player nodes to add a new player to the end of the line. Add an `m_pTail` pointer data member to the Lobby class that always points to the last player node in the line and use it to more efficiently add a player.

Solution:

The source code is in the file `bcppgp_solutions\source\chap09\chap09_exercise02.cpp`.

3. What's wrong with the following code?

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

int main()
{
```

```
    int* pScore = new int;
    *pScore = 500;
    pScore = new int(1000);
    delete pScore;
    pScore = 0;

    return 0;
}
```

Solution:
The program creates a memory leak. When `pScore` is reassigned to point to the new memory location that stores1000, it no longer points to the memory location that stores 500. That memory location (with 500 stored in it) can no longer be accessed and is therefore a memory leak.

Chapter 10

Discussion Questions

1. What benefits does inheritance bring to game programming?

Solution:
Inheritance let's you reuse classes that you've already written. This reusability produces benefits that include:

- Less work. There's no need to redefine functionality you already have. Once you have a class that provides the base functionality for other classes, you don't have to write that code again.
- Fewer errors. Once you've got a bug-free class, you can reuse it without errors cropping up in it.
- Cleaner code. Because the functionality of base classes exist only once in a program, you don't have to wade through the same code repeatedly, which makes programs easier to understand and modify.

Most related game entities cry out for inheritance. Whether it's the series of enemies that a player faces, squadrons of military vehicles that a player commands, or an inventory of weapons that a player wields, you can use inheritance to define these groups of game entities in terms of each other, which results in faster and easier programming.

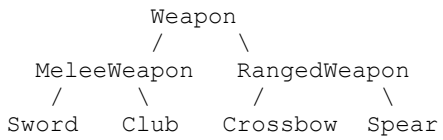
2. How does polymorphism expand the power of inheritance?

Solution:
Polymorphism expands the power of inheritance by allowing a game programmer to work with an object without having to know its exact type. For example, suppose you

had a group of enemies that the player is facing: minions and bosses. Lets also say that the two classes for minions and bosses are both derived from the same base class. Through the magic of polymorphism, you could call the same member function for each enemy in the group, like one for attacking the player, and the type of object would determine the result. The call to a minion object could produce one result, such as a weak attack, while the call to a boss object could produce a different result, such as a powerful attack.

3. What kinds of game entities might it make sense to model through inheritance?

Solution:
Any hierarchy of game entities that are related can be modeled through inheritance. For example:



4. What kinds of game-related classes would be best implemented as abstract?

Solution:
Any class that represents a generic kind of thing would be best implemented as abstract. Said another way, if it doesn't make sense to have an actual object of some type, then you are probably best off using an abstract class to represent it. Examples of abstract classes for games could be `Weapon`, M`eleeWeapon` and `RangedWeapon`. It doesn't make sense to have an object from any of these classes in a game program, but they could certainly be useful as base classes from which to derive non-abstract classes such as `Sword`, `Club`, `Crossbow` and `Spear`.

5. Why is it advantageous to be able to point to a derived class object with a base class pointer?

Solution:
It's advantageous to be able to point to a derived class object with a base class pointer to take advantage of polymorphism. For example, imagine that you implemented the classes shown in the solution to Discussion Question 3. If you had a collection of objects from the `Sword`, `Club`, `Crossbow` and `Spear` classes, you could use a single pointer of the `Weapon` class to access the member functions defined in `Weapon` for any of the objects.

Exercises

1. Improve the Simple Boss 2.0 program by adding a new class, `FinalBoss`, that is derived from the `Boss` class. The `FinalBoss` class should define a new method, `MegaAttack()`, that inflicts 10 times the amount of damage as the `SpecialAttack()` method does.

Solution:
The source code is in the file `bcppgp_solutions\source\chap10\chap10_exercise01.cpp`.

2. Improve the Blackjack game program by forcing the deck to repopulate before a round if the number of cards is running low.

Solution:

The source code is in the file `bcppgp_solutions\source\chap10\chap10_exercise02.cpp`.

3. Improve the Abstract Creature program by adding a new class, `OrcBoss`, that is derived from `Orc`. An `OrcBoss` object should start off with 180 for its `health` data member. You should also override the virtual `Greet()` member function so that it displays: `The orc boss growls hello.`

Solution:

The source code is in the file `bcppgp_solutions\source\chap10\chap10_exercise03.cpp`.

The following is the source code solutions to the exercises located at the end of the chapters.

Chapter 2.

```
// Menu Chooser with Enumeration - Chapter 2, Exercise 1
// Displays menu and gets choice - enumeration for difficulty levels

#include <iostream>

using namespace std;

int main()
{
   cout << "Difficulty Levels\n\n";
   cout << "1 - Easy\n";
   cout << "2 - Normal\n";
   cout << "3 - Hard\n\n";

   int choice;
   cout << "Choice: ";
   cin >> choice;

   // enum that represents difficulty levels
   enum difficulty {EASY=1, NORMAL, HARD};

   switch (choice)
   {
   case EASY:
         cout << "You picked Easy.\n";
         break;
   case NORMAL:
         cout << "You picked Normal.\n";
         break;
   case HARD:
         cout << "You picked Hard.\n";
         break;
   default:
         cout << "You made an illegal choice.\n";
   }

   return 0;
}
```

```
// Guess Your Number - Chapter 2, Exercise 3
// Player picks a number, the computer guesses

#include <iostream>

using namespace std;

int main()
{
   cout << "\t\tWelcome to 'Guess Your Number'!\n\n";
   cout << "Think of a number between 1 and 100 and I'll try to guess it.\n";
   cout << "After I take a guess, please respond with 'h' for higher, \n";
   cout << "'l' for lower or 'c' for correct'.\n";

   int tries = 0;
   int low = 0;
   int high = 101;
   char response = 'h';    // set to some initital value that's not 'c'

   do
   {
      int guess = (low + high) / 2;
      cout << "\nMy guess is " << guess << endl;
      tries += 1;

      cout << "Is your number (h)igher, (l)ower or am I (c)orrect?: ";
      cin >> response;

      switch (response)
      {
      case 'h':
         low = guess;
         break;
      case 'l':
          high = guess;
          break;
      case 'c':
          cout << "I guessed your number in only " << tries << " tries.";
          break;
      default:
         cout << "I'm sorry, I don't understand " << response << endl;
      }
   } while (response != 'c');

   return 0;
}
```

Chapter 3.

```
// Word Jumble with Scoring - Chapter 3, Exercise 1
// Player earns 10 points per character, loses half if asks for hint

#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
   enum fields {WORD, HINT, NUM_FIELDS};
   const int NUM_WORDS = 5;
   const string WORDS[NUM_WORDS][NUM_FIELDS] =
   {
      {"wall", "Do you feel you're banging your head against something?"},
      {"glasses", "These might help you see the answer."},
      {"labored", "Going slowly, is it?"},
      {"persistent", "Keep at it."},
      {"jumble", "It's what the game is all about."}
   };

         srand(time(0));
         int choice = (rand() % NUM_WORDS);
   string theWord = WORDS[choice][WORD];
   string theHint = WORDS[choice][HINT];

   string jumble = theWord;
   int length = jumble.size();
   for (int i=0; i<length; ++i)
   {
      int index1 = (rand() % length);
      int index2 = (rand() % length);
      char temp = jumble[index1];
      jumble[index1] = jumble[index2];
      jumble[index2] = temp;
   }

   cout << "\t\t\tWelcome to Word Jumble!\n\n";
   cout << "Unscramble the letters to make a word.\n";
   cout << "Enter 'hint' for a hint.\n";
   cout << "Enter 'quit' to quit the game.\n\n";
   cout << "The jumble is: " << jumble;
```

```
    bool askedHint = false;    // keep track of whether player asked for hint

    string guess;
    cout << "\n\nYour guess: ";
    cin >> guess;

    while ((guess != theWord) && (guess != "quit"))
    {
       if (guess == "hint")
       {
          cout << theHint;
          askedHint = true;    // set to reflect player asked for hint
       }
       else
       {
          cout << "Sorry, that's not it.";
       }

       cout <<"\n\nYour guess: ";
       cin >> guess;
    }

    if (guess == theWord)
    {
       cout << "\nThat's it!  You guessed it!\n";
       int score = 10 * theWord.size();    // score is 10 points per character
       if (askedHint)
          score /= 2;    // player loses half of points if asked for hint
       cout << "You earned a score of " << score << endl;
    }

    cout << "\nThanks for playing.\n";

    return 0;
}
```

Chapter 4.

```
// Game Title - Chapter 4, Exercise 1
// Maintain a list of favorite game titles

#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main()
{
   //setup
   vector<string> games;
   vector<string>::iterator iter;

   int selection;
   enum menu {LIST_TITLES = 1, ADD_TITLE, REMOVE_TITLE, EXIT};
   string title;

   cout << "Welcome to the personal game title list.\n";

   //menu loop
   do
   {
      cout << "\n1 - List all game titles.";
      cout << "\n2 - Add a game title.";
      cout << "\n3 - Remove a game title.";
      cout << "\n4 - Exit.\n\n";

      cout << "\nSelection: ";
      cin >> selection;

      switch (selection)
      {

      case LIST_TITLES:
         {
            if (games.empty())
               cout << "\n\nCurrently there aren't any game titles in your list.\n";
            else
            {
               cout << "\n\nYour game titles: \n";
               for (iter = games.begin(); iter != games.end(); ++iter)
                  cout << *iter << endl;
```

```
            }
         }
         break;

      case ADD_TITLE:
         {
            cout << "\n\nPlease enter game title to add: ";
            cin >> title;
            games.push_back(title);
         }
         break;

      case REMOVE_TITLE:
         {
            cout << "\n\nPlease enter game title to remove: ";
            cin >> title;

             // Solution 1
//           bool titleFound = false;
//           for (iter = games.begin(); iter != games.end(); ++iter)
//           {
//              if (*iter == title)
//                 {
//                    titleFound = true;
//                    games.erase(iter);
//                    break;
//                 }
//           }
//           if (!titleFound)
//              cout << "\n\nGame title not found!\n";

            // Solution 2
            iter = find(games.begin(), games.end(), title);
            if (iter != games.end())
               games.erase(iter);
            else
               cout << "\n\nGame title not found!\n";

         }
         break;

      case EXIT:
         {
            cout << "\nThank you for using the personal game title list.\n";
         }
         break;
```

```
        default:
            {
                cout << "\nYou have entered an invalid selection.";
            }
        }

    } while (selection != EXIT);

    //shut down
    return 0;
}
```

Chapter 5.

```
// Hangman with functions - Chapter 5, Exercise 2
// Added two functions: getGuess() and isInWord()

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cctype>

using namespace std;

char getGuess(string used_letters);        // returns guess from player
                                           // continues to ask if guess already made

bool isInWord(char letter, string word);   // returns true or false
                                           // tests if letter is in word

int main()
{
   const int MAX_WRONG = 8;

   vector<string> words;
   words.push_back("GUESS");
   words.push_back("HANGMAN");
   words.push_back("DIFFICULT");

         srand(time(0));
   random_shuffle(words.begin(), words.end());
   const string THE_WORD = words[0];
   int wrong = 0;
   string soFar(THE_WORD.size(), '-');
   string used = "";

   cout << "Welcome to Hangman.  Good luck!\n";

   while ((wrong < MAX_WRONG) && (soFar != THE_WORD))
   {
      cout << "\n\nYou have " << (MAX_WRONG - wrong) << " incorrect guesses
left.\n";
      cout << "\nYou've used the following letters:\n" << used << endl;
      cout << "\nSo far, the word is:\n" << soFar << endl;

      char guess = getGuess(used);
```

```
        used += guess;

        if (isInWord(guess, THE_WORD))
        {
            cout << "That's right! " << guess << " is in the word.\n";

            for (int i = 0; i < THE_WORD.length(); ++i)
                if (THE_WORD[i] == guess)
                    soFar[i] = guess;
        }
        else
        {
            cout << "Sorry, " << guess << " isn't in the word.\n";
            ++wrong;
        }
    }

    if (wrong == MAX_WRONG)
        cout << "\nYou've been hanged!";
    else
        cout << "\nYou guessed it!";

    cout << "\nThe word was " << THE_WORD << endl;

    int x; cin >> x;

    return 0;
}

// returns guess from player; continues to ask if guess already made
char getGuess(string used_letters)
{
    char response;
    cout << "\n\nEnter your guess: ";
    cin >> response;
    response = toupper(response);
    while (used_letters.find(response) != string::npos)
    {
        cout << "\nYou've already guessed " << response << endl;
        cout << "Enter your guess: ";
        cin >> response;
        response = toupper(response);
    }

    return response;
}
```

```
// tests if letter is in word, returns true or false
inline bool isInWord(char letter, string word)
{
    return (word.find(letter) != string::npos);

}
```

```
// Getting input with functions - Chapter 5, Exercise 3
// Using default parameter values and function overloading

#include <iostream>
#include <string>

using namespace std;

// default parameter value
int askNumber1(string prompt = "Enter a number: ");

// function overloading
int askNumber2();
int askNumber2(string prompt);

int main()
{
   int number = askNumber1();
   cout << "Thanks for entering: " << number << "\n\n";

   int score = askNumber1("What was your score?: ");
   cout << "Your score is " << score << "\n\n";

   number = askNumber2();
   cout << "Thanks for entering: " << number << "\n\n";

   score = askNumber2("What was your score?: ");
   cout << "Your score is " << score << "\n\n";


   int x; cin >> x;

   return 0;
}

int askNumber1(string prompt)
{
   int number;
   cout << prompt;
   cin >> number;

   return number;
}

int askNumber2()
{
```

```
   int number;
   cout << "Enter a number: ";
   cin >> number;

   return number;
}

int askNumber2(string prompt)
{
   int number;
   cout << prompt;
   cin >> number;

   return number;
}
```

Chapter 6.

```
// Mad Lib with references - Chapter 6, Exercise 1
// Use references where appropriate
// EDIT KMH 10/23/06

#include <iostream>
#include <string>

using namespace std;

// pass constant reference instead of copy of string object
string askText(const string& prompt);
int askNumber(const string& prompt);
void tellStory(const string& name,
               const string& noun,
               int number,
               const string& bodyPart,
               const string& verb);

int main()
{
   cout << "Welcome to Mad Lib.\n\n";
   cout << "Answer the following questions to help create a new story.\n";

   string name = askText("Please enter a name: ");
   string noun = askText("Please enter a plural noun: ");
   int number = askNumber("Please enter a number: ");
   string bodyPart = askText("Please enter a body part: ");
   string verb = askText("Please enter a verb: ");
```

```
   tellStory(name, noun, number, bodyPart, verb);

   return 0;
}

string askText(const string& prompt)
{
   string text;
   cout << prompt;
   cin >> text;
   return text;
}

int askNumber(const string& prompt)
{
   int num;
   cout << prompt;
   cin >> num;
   return num;
}

void tellStory(const string& name,
               const string& noun,
               int number,
               const string& bodyPart,
               const string& verb)
{
   cout << "\nHere's your story:\n";
   cout << "The famous explorer ";
   cout << name;
   cout << " had nearly given up a life-long quest to find\n";
   cout << "The Lost City of ";
   cout << noun;
   cout << " when one day, the ";
   cout << noun;
   cout << " found the explorer.\n";
   cout << "Surrounded by ";
   cout << number;
   cout << " " << noun;
   cout << ", a tear came to ";
   cout << name << "'s ";
   cout << bodyPart << ".\n";
   cout << "After all this time, the quest was finally over. ";
   cout << "And then, the ";
   cout << noun << "\n";
```

```
    cout << "promptly devoured ";
    cout << name << ". ";
    cout << "The moral of the story? Be careful what you ";
    cout << verb;
    cout << " for.";
}
```

Chapter 7.

```
// Pointer to Pointer - Chapter 7, Exercise 1
// Create a pointer to pointer to string and call string object's size() method

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str = "I'm a string";    // string
    string* pStr = &str;            // pointer to string
    string** ppStr = &pStr;         // pointer to pointer to string

    // access the string object through two levels of indirection
    cout << "(*(*ppStr)).size() is: " << (*(*ppStr)).size() << endl;

    return 0;
}
```

```
// Mad Lib with pointers - Chapter 7, Exercise 2
// Use pointers and constants where appropriate

#include <iostream>
#include <string>

using namespace std;

// use constant pointers to constants
string askText(const string* const prompt);
int askNumber(const string* const prompt);
void tellStory(const string* const name,
               const string* const noun,
               int number,
               const string* const bodyPart,
               const string* const verb);

int main()
{
   cout << "Welcome to Mad Lib.\n\n";
   cout << "Answer the following questions to help create a new story.\n";

   // create string object so can pass address to pointer
   string prompt = "Please enter a name: ";
   string name = askText(&prompt);

   prompt = "Please enter a plural noun: ";
   string noun = askText(&prompt);

   prompt = "Please enter a number: ";
   int number = askNumber(&prompt);

   prompt = "Please enter a body part: ";
   string bodyPart = askText(&prompt);

   prompt = "Please enter a verb: ";
   string verb = askText(&prompt);

   tellStory(&name, &noun, number, &bodyPart, &verb);

   int x; cin >> x;

   return 0;
}

// receive memory address
```

```
string askText(const string* const prompt)
{
   string text;
   // dereference pointer
   cout << *prompt;
   cin >> text;
   return text;
}

int askNumber(const string* const prompt)
{
   int num;
   cout << *prompt;
   cin >> num;
   return num;
}

void tellStory(const string* const name,
               const string* const noun,
               int number,
               const string* const bodyPart,
               const string* const verb)
{
   cout << "\nHere's your story:\n";
   cout << "The famous explorer ";
   cout << *name;
   cout << " had nearly given up a life-long quest to find\n";
   cout << "The Lost City of ";
   cout << *noun;
   cout << " when one day, the ";
   cout << *noun;
   cout << " found the explorer.\n";
   cout << "Surrounded by ";
   cout << number;
   cout << " " << *noun;
   cout << ", a tear came to ";
   cout << *name << "'s ";
   cout << *bodyPart << ".\n";
   cout << "After all this time, the quest was finally over. ";
   cout << "And then, the ";
   cout << *noun << "\n";
   cout << "promptly devoured ";
   cout << *name << ". ";
   cout << "The moral of the story? Be careful what you ";
   cout << *verb;
   cout << " for.";    }
```

Chapter 8.

```
//Critter Caretaker with Status() - Chapter 8, Exercise 1
//Unlisted menu choice reveals values of critter's hunger and boredom levels

#include <iostream>

using namespace std;

class Critter
{
public:
   Critter(int hunger = 0, int boredom = 0);
   void Talk();
   void Eat(int food = 4);
   void Play(int fun = 4);
   void Status();

private:
   int m_Hunger;
   int m_Boredom;

   int GetMood() const;
   void PassTime(int time = 1);

};

Critter::Critter(int hunger, int boredom):
   m_Hunger(hunger),
   m_Boredom(boredom)
{}

inline int Critter::GetMood() const
{
   return (m_Hunger + m_Boredom);
}

void Critter::PassTime(int time)
{
   m_Hunger += time;
   m_Boredom += time;
}

void Critter::Talk()
{
```

```
    cout << "I'm a critter and I feel ";
    int mood = GetMood();
    if (mood > 15)
        cout << "mad.\n";
    else if (mood > 10)
        cout << "frustrated.\n";
    else if (mood > 5)
        cout << "okay.\n";
    else
        cout << "happy.\n";
    PassTime();
}

void Critter::Eat(int food)
{
    cout << "Brruppp.\n";
    m_Hunger -= food;
    if (m_Hunger < 0)
        m_Hunger = 0;
    PassTime();
}

void Critter::Play(int fun)
{
    cout << "Wheee!\n";
    m_Boredom -= fun;
    if (m_Boredom < 0)
        m_Boredom = 0;
    PassTime();
}

void Critter::Status()
{
    cout << "Critter Status\n";
    cout << "m_Hunger: " << m_Hunger << endl;
    cout << "m_Boredom: " << m_Boredom << endl;
}


int main()
{
    Critter crit;

    int choice = 1;
    while (choice != 0)
    {
```

```
        cout << "\nCritter Caretaker\n\n";
        cout << "0 - Quit\n";
        cout << "1 - Listen to your critter\n";
        cout << "2 - Feed your critter\n";
        cout << "3 - Play with your critter\n\n";

        cout << "Choice: ";
        cin >> choice;

        switch (choice)
        {
        case 0:
           cout << "Good-bye.\n";
                                break;
        case 1:
           crit.Talk();
                                break;
        case 2:
           crit.Eat();
                                break;
        case 3:
           crit.Play();
                                break;
        case 4:    // undocumented menu choice for critter status
           crit.Status();
           break;
        default:
           cout << "\nSorry, but " << choice << " isn't a valid choice.\n";
        }
    }

    return 0;
}
```

```
//More Expressive Critter - Chapter 8, Exercise 2
//Critter now more expressive about how hungry and bored it is

#include <iostream>
#include <string>

using namespace std;

class Critter
{
public:
   Critter(int hunger = 0, int boredom = 0);
   void Talk();
   void Eat(int food = 4);
   void Play(int fun = 4);

private:
   int m_Hunger;
   int m_Boredom;

   void PassTime(int time = 1);
   // returns adjective based on how large a value passed to it
   // will be passed either m_Hunger or m_Boredom
   string GetModifier(int) const;

};

Critter::Critter(int hunger, int boredom):
   m_Hunger(hunger),
   m_Boredom(boredom)
{}

void Critter::PassTime(int time)
{
   m_Hunger += time;
   m_Boredom += time;
}

// returns adjective based on how large a value passed to it
// will be passed either m_Hunger or m_Boredom
string Critter::GetModifier(int level) const
{
   string modifier;

   if (level >= 8)
      modifier = "very";
```

```
    else if (level <= 7 && level >= 5)
        modifier = "pretty";
    else if (level <= 5 && level >= 3)
        modifier = "kind of";
    else
        modifier = "not";

    return modifier;
}

void Critter::Talk()
{
    // get adjective based on m_Hunger and m_Boredom
    cout << "I'm " << GetModifier(m_Hunger) << " hungry and ";
    cout << "I'm " << GetModifier(m_Boredom) << " bored.\n";
    PassTime();
}

void Critter::Eat(int food)
{
    cout << "Brruppp.\n";
    m_Hunger -= food;
    if (m_Hunger < 0)
        m_Hunger = 0;
    PassTime();
}

void Critter::Play(int fun)
{
    cout << "Wheee!\n";
    m_Boredom -= fun;
    if (m_Boredom < 0)
        m_Boredom = 0;
    PassTime();
}

int main()
{
    Critter crit;

    int choice = 1;
    while (choice != 0)
    {
        cout << "\nCritter Caretaker\n\n";
        cout << "0 - Quit\n";
        cout << "1 - Listen to your critter\n";
```

```
        cout << "2 - Feed your critter\n";
        cout << "3 - Play with your critter\n\n";

        cout << "Choice: ";
        cin >> choice;

        switch (choice)
        {
        case 0:
            cout << "Good-bye.\n";
                        break;
        case 1:
            crit.Talk();
                        break;
        case 2:
            crit.Eat();
                        break;
        case 3:
            crit.Play();
                        break;
        default:
            cout << "\nSorry, but " << choice << " isn't a valid choice.\n";
        }
    }

    return 0;
}
```

Chapter 9.

```
//Game Lobby improved - Chapter 9, Exercise 1
//Player object can be sent to cout

#include <iostream>
#include <string>

using namespace std;

class Player
{
   // make friend so function can access m_Name
   friend ostream& operator<<(ostream& os, const Player& aPlayer);

public:
   Player(const string& name = ""): m_Name(name), m_pNext(0) {}
   string GetName() const { return m_Name; }
   Player* GetNext() const { return m_pNext; }
   void SetNext(Player* next) { m_pNext = next; }
private:
   string m_Name;
   Player* m_pNext;
};

class Lobby
{
   friend ostream& operator<<(ostream& os, const Lobby& aLobby);
public:
   Lobby(): m_pHead(0) {}
   ~Lobby() { Clear(); }
   void AddPlayer();
   void RemovePlayer();
   void Clear();
private:
   Player* m_pHead;
};

void Lobby::AddPlayer()
{
   cout << "Please enter the name of the new player: ";
   string name;
   cin >> name;
   Player* pNewPlayer = new Player(name);

   if (m_pHead == 0)
```

```
    {
        m_pHead = pNewPlayer;
    }
    else
    {
        Player* pIter = m_pHead;
        while (pIter->GetNext() != 0)
        {
            pIter = pIter->GetNext();
        }
        pIter->SetNext(pNewPlayer);
    }
}

void Lobby::RemovePlayer()
{
    if (m_pHead == 0)
    {
        cout << "The game lobby is empty.  No one to remove!\n";
    }
    else
    {
        Player* pTemp = m_pHead;
        m_pHead = m_pHead->GetNext();
        delete pTemp;
    }
}

void Lobby::Clear()
{
    while (m_pHead != 0)
    {
        RemovePlayer();
    }
}

//overload the << operator so can send a Player object to cout
ostream& operator<<(ostream& os, const Player& aPlayer)
{
    os << aPlayer.m_Name;
    return os;
}


ostream& operator<<(ostream& os, const Lobby& aLobby)
{
```

```
    Player* pIter = aLobby.m_pHead;

    os << "\nHere's who's in the game lobby:\n";
    if (pIter == 0)
    {
       os << "The lobby is empty.\n";
    }
    else
    {
       while (pIter != 0)
       {
          // send Player object directly to cout
          os << *pIter << endl;
                pIter = pIter->GetNext();
       }
    }

    return os;
}

int main()
{
    Lobby myLobby;
    int choice;

    do
          {
             cout << myLobby;
       cout << "\nGAME LOBBY\n";
       cout << "0 - Exit the program.\n";
       cout << "1 - Add a player to the lobby.\n";
       cout << "2 - Remove a player from the lobby.\n";
       cout << "3 - Clear the lobby.\n";
       cout << endl << "Enter choice: ";
       cin >> choice;

       switch (choice)
       {
          case 0: cout << "Good-bye.\n"; break;
                case 1: myLobby.AddPlayer(); break;
          case 2: myLobby.RemovePlayer(); break;
          case 3: myLobby.Clear(); break;
          default: cout << "That was not a valid choice.\n";
       }
          }
    while (choice != 0);
```

```
    return 0;
}
```

```
//Game Lobby with pointer to last player - Chapter 9, Exercise 2
//m_pTail points to last player in the list

#include <iostream>
#include <string>

using namespace std;

class Player
{
public:
   Player(const string& name = ""): m_Name(name), m_pNext(0) {}
   string GetName() const { return m_Name; }
   Player* GetNext() const { return m_pNext; }
   void SetNext(Player* next) { m_pNext = next; }
private:
   string m_Name;
   Player* m_pNext;
};

class Lobby
{
   friend ostream& operator<<(ostream& os, const Lobby& aLobby);
public:
   Lobby(): m_pHead(0), m_pTail(0){}
   ~Lobby() { Clear(); }
   void AddPlayer();
   void RemovePlayer();
   void Clear();
private:
   Player* m_pHead;
   Player* m_pTail;  // pointer to last player in the list
};

void Lobby::AddPlayer()
{
   cout << "Please enter the name of the new player: ";
   string name;
   cin >> name;
   Player* pNewPlayer = new Player(name);

   if (m_pHead == 0)
   {
      m_pHead = pNewPlayer;
      m_pTail = m_pHead;
   }
```

```
    // add the player to the end of the list using m_pTail
    else
    {
        m_pTail->SetNext(pNewPlayer);
        m_pTail = m_pTail->GetNext();
    }
}

void Lobby::RemovePlayer()
{
    if (m_pHead == 0)
    {
        cout << "The game lobby is empty.  No one to remove!\n";
    }
    else
    {
        Player* pTemp = m_pHead;
        m_pHead = m_pHead->GetNext();
        delete pTemp;
    }
}

void Lobby::Clear()
{
    while (m_pHead != 0)
    {
        RemovePlayer();
    }
}

ostream& operator<<(ostream& os, const Lobby& aLobby)
{
    Player* pIter = aLobby.m_pHead;

    os << "\nHere's who's in the game lobby:\n";
    if (pIter == 0)
    {
        os << "The lobby is empty.\n";
    }
    else
    {
        while (pIter != 0)
        {
            os << pIter->GetName() << endl;
                pIter = pIter->GetNext();
        }
```

```
    }

    return os;
}

int main()
{
    Lobby myLobby;
    int choice;

    do
          {
             cout << myLobby;
       cout << "\nGAME LOBBY\n";
       cout << "0 - Exit the program.\n";
       cout << "1 - Add a player to the lobby.\n";
       cout << "2 - Remove a player from the lobby.\n";
       cout << "3 - Clear the lobby.\n";
       cout << endl << "Enter choice: ";
       cin >> choice;

       switch (choice)
       {
          case 0: cout << "Good-bye.\n"; break;
                case 1: myLobby.AddPlayer(); break;
          case 2: myLobby.RemovePlayer(); break;
          case 3: myLobby.Clear(); break;
          default: cout << "That was not a valid choice.\n";
       }
          }
    while (choice != 0);

    return 0;
}
```

Chapter 10.

```
//Simple Boss 2.0 improved - Chapter 10, Exercise 1
//Added FinalBoss class

#include <iostream>
using namespace std;

class Enemy
{
public:
   Enemy(): m_Damage(10) {}

   void Attack() const
   { cout << "Attack inflicts " << m_Damage << " damage points!\n"; }

protected:
   int m_Damage;
};

class Boss : public Enemy
{
public:
   Boss(): m_DamageMultiplier(3) {}

   void SpecialAttack() const
   { cout << "Special Attack inflicts " << (m_DamageMultiplier * m_Damage);
     cout << " damage points!\n"; }

// data member needs to be protected so FinalBoss class can access it
protected:
   int m_DamageMultiplier;
};

// Add FinalBoss class
class FinalBoss : public Boss
{
public:
   // Boss constructor automatically called, so don't need to explicitly call it
   FinalBoss() {}

   void MegaAttack() const
   { cout << "Mega Attack inflicts " << (10 * m_DamageMultiplier * m_Damage);
     cout << " damage points!\n"; }
};
```

```
int main()
{
    cout << "Creating an enemy.\n";
    Enemy enemy1;
    enemy1.Attack();

    cout << "\nCreating a boss.\n";
    Boss boss1;
    boss1.Attack();
    boss1.SpecialAttack();

    cout << "\nCreating a final boss.\n";
    FinalBoss finalBoss1;
    finalBoss1.Attack();
    finalBoss1.SpecialAttack();
    finalBoss1.MegaAttack();

    return 0;
}
```

```
//Blackjack improved - Chapter 10, Exercise 2
//Deck is repopulated if number of cards too low

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>

using namespace std;

class Card
{
public:
   enum rank {ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE,
TEN,
              JACK, QUEEN, KING};
   enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};

   friend ostream& operator<<(ostream& os, const Card& aCard);

   Card(rank r = ACE, suit s = SPADES, bool ifu = true);

   int GetValue() const;

   void Flip();

private:
   rank m_Rank;
   suit m_Suit;
   bool m_IsFaceUp;
};

Card::Card(rank r, suit s, bool ifu):  m_Rank(r), m_Suit(s), m_IsFaceUp(ifu)
{}

int Card::GetValue() const
{
   int value = 0;
   if (m_IsFaceUp)
   {
      value = m_Rank;
      if (value > 10)
         value = 10;
   }
   return value;
```

```
}

void Card::Flip()
{
   m_IsFaceUp = !(m_IsFaceUp);
}

class Hand
{
public:
   Hand();

   virtual ~Hand();

   void Add(Card* pCard);

   void Clear();

   int GetTotal() const;

protected:
   vector<Card*> m_Cards;
};

Hand::Hand()
{
   m_Cards.reserve(7);
}

Hand::~Hand()
{
   Clear();
}

void Hand::Add(Card* pCard)
{
   m_Cards.push_back(pCard);
}

void Hand::Clear()
{
   vector<Card*>::iterator iter = m_Cards.begin();
   for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
   {
      delete *iter;
      *iter = 0;
```

```
    }
    m_Cards.clear();
}

int Hand::GetTotal() const
{
    if (m_Cards.empty())
        return 0;

    if (m_Cards[0]->GetValue() == 0)
        return 0;

    int total = 0;
    vector<Card*>::const_iterator iter;
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
        total += (*iter)->GetValue();

    bool containsAce = false;
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
        if ((*iter)->GetValue() == Card::ACE)
            containsAce = true;

    if (containsAce && total <= 11)
        total += 10;

    return total;
}

class GenericPlayer : public Hand
{
    friend ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer);

public:
    GenericPlayer(const string& name = "");

    virtual ~GenericPlayer();

    virtual bool IsHitting() const = 0;

    bool IsBusted() const;

    void Bust() const;

protected:
    string m_Name;
};
```

```
GenericPlayer::GenericPlayer(const string& name): m_Name(name)
{}

GenericPlayer::~GenericPlayer()
{}

bool GenericPlayer::IsBusted() const
{
   return (GetTotal() > 21);
}

void GenericPlayer::Bust() const
{
   cout << m_Name << " busts.\n";
}

class Player : public GenericPlayer
{
public:
   Player(const string& name = "");

   virtual ~Player();

   virtual bool IsHitting() const;

   void Win() const;

   void Lose() const;

   void Push() const;
};

Player::Player(const string& name): GenericPlayer(name)
{}

Player::~Player()
{}

bool Player::IsHitting() const
{
   cout << m_Name << ", do you want a hit? (Y/N): ";
   char response;
   cin >> response;
   return (response == 'y' || response == 'Y');
}
```

```
void Player::Win() const
{
   cout << m_Name <<  " wins.\n";
}

void Player::Lose() const
{
   cout << m_Name <<  " loses.\n";
}

void Player::Push() const
{
   cout << m_Name <<  " pushes.\n";
}

class House : public GenericPlayer
{
public:
   House(const string& name = "House");

   virtual ~House();

   virtual bool IsHitting() const;

   void FlipFirstCard();
};

House::House(const string& name): GenericPlayer(name)
{}

House::~House()
{}

bool House::IsHitting() const
{
   return (GetTotal() <= 16);
}

void House::FlipFirstCard()
{
   if (!(m_Cards.empty()))
      m_Cards[0]->Flip();
   else cout << "No card to flip!\n";
}
```

```
class Deck : public Hand
{
public:
   Deck();

   virtual ~Deck();

   int GetSize() const;     // return the number of cards in deck

   void Populate();

   void Shuffle();

   void Deal(Hand& aHand);

   void AdditionalCards(GenericPlayer& aGenericPlayer);
};

Deck::Deck()
{
   m_Cards.reserve(52);
   Populate();
}

Deck::~Deck()
{}

int Deck::GetSize() const    // return the number of cards in deck
{
   return m_Cards.size();
}

void Deck::Populate()
{
   Clear();
   for (int s = Card::CLUBS; s <= Card::SPADES; ++s)
         for (int r = Card::ACE; r <= Card::KING; ++r)
            Add(new Card(static_cast<Card::rank>(r), static_cast<Card::suit>(s)));
}

void Deck::Shuffle()
{
   random_shuffle(m_Cards.begin(), m_Cards.end());
}

void Deck::Deal(Hand& aHand)
```

```
{
   if (!m_Cards.empty())
   {
      aHand.Add(m_Cards.back());
      m_Cards.pop_back();
   }
   else
   {
      cout << "Out of cards. Unable to deal.";
   }
}

void Deck::AdditionalCards(GenericPlayer& aGenericPlayer)
{
   cout << endl;
   while ( !(aGenericPlayer.IsBusted()) && aGenericPlayer.IsHitting() )
   {
      Deal(aGenericPlayer);
      cout << aGenericPlayer << endl;

      if (aGenericPlayer.IsBusted())
         aGenericPlayer.Bust();
   }
}

class Game
{
public:
   Game(const vector<string>& names);

   ~Game();

   void Play();

private:
   Deck m_Deck;
   House m_House;
   vector<Player> m_Players;
};

Game::Game(const vector<string>& names)
{
   vector<string>::const_iterator pName;
   for (pName = names.begin(); pName != names.end(); ++pName)
      m_Players.push_back(Player(*pName));
```

```
    srand(time(0));
    m_Deck.Populate();
    m_Deck.Shuffle();
}

Game::~Game()
{}

void Game::Play()
{
     const int MAX_CARDS_PER_HAND = 6;
    // check if have 6 cards for every player; if not, reshuffle
    if (m_Deck.GetSize() < MAX_CARDS_PER_HAND * m_Players.size())
    {
        cout << "Using new deck.\n";
        m_Deck.Populate();
        m_Deck.Shuffle();
    }

    vector<Player>::iterator pPlayer;
    for (int i = 0; i < 2; ++i)
    {
        for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
            m_Deck.Deal(*pPlayer);
        m_Deck.Deal(m_House);
    }

    m_House.FlipFirstCard();

    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
        cout << *pPlayer << endl;
    cout << m_House << endl;

    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
        m_Deck.AdditionalCards(*pPlayer);

    m_House.FlipFirstCard();
    cout << endl << m_House;

    m_Deck.AdditionalCards(m_House);

    if (m_House.IsBusted())
    {
        for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
            if ( !(pPlayer->IsBusted()) )
                pPlayer->Win();
```

```
    }
    else
    {
        for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
            if ( !(pPlayer->IsBusted()) )
            {
                if (pPlayer->GetTotal() > m_House.GetTotal())
                    pPlayer->Win();
                else if (pPlayer->GetTotal() < m_House.GetTotal())
                    pPlayer->Lose();
                else
                    pPlayer->Push();
            }
    }

    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
        pPlayer->Clear();
    m_House.Clear();
}

ostream& operator<<(ostream& os, const Card& aCard);
ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer);

int main()
{
    cout << "\t\tWelcome to Blackjack!\n\n";

    int numPlayers = 0;
    while (numPlayers < 1 || numPlayers > 7)
    {
        cout << "How many players? (1 - 7): ";
        cin >> numPlayers;
    }

    vector<string> names;
    string name;
    for (int i = 0; i < numPlayers; ++i)
    {
        cout << "Enter player name: ";
        cin >> name;
        names.push_back(name);
    }
    cout << endl;

    Game aGame(names);
    char again = 'y';
```

```
    while (again != 'n' && again != 'N')
    {
        aGame.Play();
        cout << "\nDo you want to play again? (Y/N): ";
        cin >> again;
    }

    return 0;
}

ostream& operator<<(ostream& os, const Card& aCard)
{
    const string RANKS[] = {"0", "A", "2", "3", "4", "5", "6", "7", "8", "9",
                            "10", "J", "Q", "K"};
    const string SUITS[] = {"c", "d", "h", "s"};

    if (aCard.m_IsFaceUp)
        os << RANKS[aCard.m_Rank] << SUITS[aCard.m_Suit];
    else
        os << "XX";

    return os;
}

ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer)
{
    os << aGenericPlayer.m_Name << ":\t";

    vector<Card*>::const_iterator pCard;
    if (!aGenericPlayer.m_Cards.empty())
    {
        for (pCard = aGenericPlayer.m_Cards.begin();
             pCard != aGenericPlayer.m_Cards.end(); ++pCard)
            os << *(*pCard) << "\t";

        if (aGenericPlayer.GetTotal() != 0)
            cout << "(" << aGenericPlayer.GetTotal() << ")";
    }
    else
    {
        os << "<empty>";
    }

    return os;
}
```

```
//Abstract Creature improved - Chapter 10, Exercise 3
//Added OrcBoss class that inherits from Orc

#include <iostream>
using namespace std;

class Creature
{
public:
   Creature(int health = 100): m_Health(health)
   {}

   virtual void Greet() const = 0;

   virtual void DisplayHealth() const
   { cout << "Health: " << m_Health << endl; }

protected:
   int m_Health;
};

class Orc : public Creature
{
public:
   Orc(int health = 120): Creature(health)
   {}

   virtual void Greet() const
   { cout << "The orc grunts hello.\n"; }
};

// New class, OrcBoss
class OrcBoss : public Orc
{
public:
   // Constructor calls base class constructor
   OrcBoss(int health = 180): Orc(health)
   {}

   // Virtual functions overrides Orc's Greet() member function
   virtual void Greet() const
   { cout << "The orc boss growls hello.\n"; }
};

int main()
{
```

```
    Creature* pCreature = new OrcBoss();
    pCreature->Greet();
    pCreature->DisplayHealth();

    delete pCreature;
    pCreature = 0;

    int x; cin >> x;

    return 0;
}
```