

Table of Contents

Table of Contents	1
Course Overview	2
Review	4
1.0 Design Documents	30
1.1 Defensive Programming.....	43
1.2 Exception Handling.....	55
1.3 Structures	65
1.4 Separate Compilation.....	79
1.5 Function: Advanced Topics.....	92
2.0 Encapsulation Design.....	108
2.1 Building a Class	119
2.2 Class Syntax.....	126
2.3 Accessors & Mutators	136
2.4 Constructors & Destructors	143
2.5 Static	155
2.6 Non-Member Operator Overloading.....	165
2.7 Friends	180
2.8 Member Operator Overloading.....	188
3.0 Class Relations.....	203
3.1 Building Polymorphism	212
3.2 Inheritance.....	221
3.3 Inheritance Qualifiers.....	231
3.4 Virtual Functions.....	241
3.5 Pure Virtual Functions.....	253
4.0 Type-Independent Design.....	264
4.1 Void Pointers and Callbacks.....	274
4.2 Function Templates	286
4.3 Class Templates	294
4.4 Linked Lists	301
4.5 Iterators	314
4.6 Standard Template Library.....	325
A. Elements of Style.....	338
B. C + Syntax Reference Guide	343
C. Glossary	347
D. Index	362

Course Overview

Procedural programming, the subject of CS 124 preceding this class, is a style of programming where the focus is on the function. This includes how to subdivide a program into functions (in a process called modularization) and what goes on inside functions. Virtually any programming project can be completed with procedural programming tools.

Object-Oriented programming, on the other hand, is a style of programming where the focus is on the class. A class is a construct containing a collection of functions as well as the data associated with them. Topics such as encapsulation (how to create well-designed classes), inheritance (the relationship between classes), and polymorphism (working with many versions of related classes) form the backbone of Object-Oriented programming.

This class will teach Object-Oriented programming in C++ as well as software development methodologies that enable programmers to work with large projects and many individuals.

Goals

By the end of this semester, you will be able to:

- Generate a design document describing an approach to solve a given program definition
- Predict the output of Object-Oriented C++ code
- Identify syntax errors
- Identify code matching a given output example
- Identify code matching a given output description
- Identify the best design for a given problem
- Write Object-Oriented C++ code conforming to a problem definition
- List and define the terms and concepts of Object-Oriented design

These goals will be explored in the context of C++ using the Linux operating system.

Course layout

This course will be broken into four units:

1. **Using Classes.** This is a preparatory unit, ensuring we have the programming skills to learn Object-Oriented programming and to lay the foundation for the topics to come.
2. **Encapsulation.** Our first undertaking into the world of Object-Oriented programming, the encapsulation unit introduces us to classes. Here we will learn how to design a program with classes and create our first class.
3. **Inheritance & Polymorphism.** With knowledge of how to create and use classes, the next unit is concerned with leveraging the relationships between similar classes to minimize code duplication and to provide a new perspective on class design.
4. **Abstract Types.** After spending virtually the entire semester concerning ourselves with designing and creating new data types, the final unit is about defining functions and classes that operate independent of the data type passed to them.

How to use this textbook

This textbook is closely aligned with CS 165. All the topics, problems, and technology used in CS 165 are described in detail with these pages.

Sam and Sue

You may notice two characters present in various sections of this text. They embody two common archetypes representative of people you will probably encounter in industry.



Sue's Tips

Sue is a pragmatist. She cares only about getting the job done at a high quality level and with the minimum amount of effort. In other words, she cares little for the art of programming, focusing instead on the engineering of the craft.

Sue's Tips tend to focus on the following topics:

- Pitfalls: How to avoid common programming pitfalls that cause bugs
- Effort: How to do the same work with less time and effort
- Robustness: How to make code more resistant to bugs
- Efficiency: How to make code execute with fewer resources



Sam's Corner

Sam is a technology nerd. He likes solving hard problems for their own sake, not necessarily because they even need to be solved. Sam enjoys getting to the heart of a problem and finding the most elegant solution. It is easy for Sam to get hung up on a problem for hours even though he found a working solution long ago.

The following topics are commonly discussed in Sam's Corner:

- Details: The details of how various operations work, even though this knowledge is not necessary to get the job done
- Tidbits: Interesting tidbits explaining why things are the way they are

Neither Sue's Tips nor Sam's Corner are required knowledge for this course. However, you may discover your inner Sam or Sue and find yourself reading one of their columns.

Need Help

Occasionally, each of us reaches a roadblock or feels like help is needed. This textbook offers two ways to bail yourself out of trouble.

If you find you are not able to understand the problems we do in class, work through them at home before class. This will give you time for reflection and help ask better questions in class.

If you find the programming problems to be too hard, take the time to type out all the examples by hand. Once you finish them, work through the challenge associated with each example. There is something about typing code by hand that helps it seep into our brains. I hope this helps.

Review

Procedural programming is a method of programming where the fundamental unit is a function (or procedure). Topics such as variables, loops, and functions are central to procedural programming. Object-oriented programming, on the other hand, is a methodology where the fundamental unit is an object which is built from procedural tools. It is therefore necessary to have a firm grasp of procedural programming before object-oriented programming can be learned.

This section is a brief review of procedural programming. Please use this section as a gauge indicating whether you need to review some procedural topics before continuing with this book.

Variables and data types

The first generation of computers could only work with integers. A few decades later, scientists figured out how to store decimal numbers in a computer. Today, most computers natively handle many data types. The built-in data types supported by the C++ language are:

Data type	Use	Size	Range of values
bool	Logic	1	true, false
char	Letters and symbols	1	-128 to 127 ... or 'a', 'b', etc.
short	Small numbers, Unicode characters	2	-32,768 to 32,767
int	Counting	4	--2,147,483,648 to 2,147,483,647
long (long int)	Larger Numbers	8	$\pm 9,223,372,036,854,775,808$
float	Numbers with decimals	4	$\approx 10^{-38}$ to 10^{38} accurate to 7 digits
double	Larger numbers with decimals	8	$\approx 10^{-308}$ to 10^{308} accurate to 15 digits
long double	Huge Numbers	16	$\approx 10^{-4932}$ to 10^{4932} accurate to 19 digits

Variable declarations

To use one of these data types, it is necessary to declare a variable. The syntax for declaring a variable is:

```
<DataType> <variableName>;
```

Observe how the variable name is **camelCase**: multiple words are commonly used in variable names with the first letter of each word Capitalized except for the first word.

The range of values for an integer is -2 billion to positive 2 billion for a total of 4 billion values. You can declare an integer to be positive by using the `unsigned` modifier. Note that this only works for integral data types; floating point numbers do not accept the `unsigned` modifier.

Data type	Use	Size	Range of values
unsigned char	Small numbers	1	0 to 255
unsigned short	Small numbers	2	0 to 65,535
unsigned int	Counting	4	0 to 4,294,967,295
unsigned long int	Larger Numbers	8	0 to 18,446,744,073,709,551,615

You can also make a variable constant, thereby making it impossible to change the value. This can be achieved with the `const` modifier:

```
const float PI = 3.14159;
```

Observe how we make constant variables ALL_CAPS with an underscore separating the words. This is done for all constant variables except for function parameters.

Data conversion

When converting a value from one data type to another, the compiler inserts code into your program to make the conversion. When converting from a relatively small data type like a `char` into a larger one like an `int`, there is no data loss from the conversion. The entire range for a `char` fits within that of an `int`.

```
{
    char character = 100;                      // fits within the range of -128 to 127
    int integer;                                // range is -2 billion to +2 billion

    integer = character;                         // since 100 is within the int range,
                                                // this does not present a problem
}
```

However, when converting from a large data type like a `long double` into a smaller one like a `float`, it is possible that data will be lost.

```
{
    long double bigNum = 3.141592653589793238; // 19 digits for a long double
    float smallNum;                            // 7 digits for a float

    smallNum = bigNum;                        // we lose 12 digits here!
}
```

Data conversion happens automatically whenever a value from one data type is assigned to that of another. The programmer can also explicitly perform a data conversion through casting:

```
{
    float value = 3.14159;
    cout <<      value << endl;           // display "3.14159"
    cout << (int)value << endl;         // display "3"
}
```

Observe that the `(int)` cast signals to the compiler to convert the floating point `value` into an integer. C++ adds four special types of casts:

Cast	Use
<code>static_cast<Type>(expression)</code>	Same as the simple cast described above
<code>const_cast<Type>(expression)</code>	Useful for making a <code>const</code> value no-longer <code>const</code> .
<code>dynamic_cast<Type>(expression)</code>	Useful for downcasting. See chapter 3.2 for details.
<code>reinterpret_cast<Type>(expression)</code>	Allows conversion between pointer types.

Of these four, the most common is the `static_cast`. You can use it interchangeably with the simple casting we used in previous semesters:

```
{
    float value = 3.14159;
    cout <<      value << endl;           // display "3.14159"
    cout << static_cast<int>(value) << endl; // display "3"
    cout <<      (int) value << endl;       // display "3"
}
```

Sam's Corner



Though `static_cast<int>(value)` is more of a C++ way of doing a cast than the C way of doing things `(int)value`, they do the same thing. In this text and in all the examples, we will use the C casting convention simply because it requires less typing.

Expressions

An expression is an equation that is evaluated to a single value. This equation can be in the form of a mathematical expression, the result of a function call, or any combination thereof. Evaluation of these expressions occurs in the following order:

1. Variables are replaced with the values they contain
2. The order of operations are evaluated: parentheses first and assignment last
3. When there is an `int` being compared/computed with a `float`, convert it to a `float` just before evaluation

The order of operations is:

Name	Operator	Example
Array indexing	[]	<code>array[4]</code>
Function call	()	<code>function()</code>
Postfix increment and decrement	<code>++ --</code>	<code>count++ count--</code>
Prefix increment and decrement	<code>++ --</code>	<code>++count --count</code>
Not	!	<code>!married</code>
Negative	-	<code>-4</code>
Dereference	*	<code>*pValue</code>
Address-of	&	<code>&value</code>
Allocate with new	<code>new</code>	<code>new int</code>
Free with delete	<code>delete</code>	<code>delete pValue</code>
Casting	()	<code>(int)4.2</code>
Get size of	<code>sizeof</code>	<code>sizeof(int)</code>
Multiplication	*	<code>3 * 4</code>
Division	/	<code>3 / 4</code>
Modulus	%	<code>3 % 4</code>
Addition	+	<code>3 + 4</code>
Subtraction	-	<code>3 - 4</code>
Insertion	<code><<</code>	<code>cout << value</code>
Extraction	<code>>></code>	<code>cin >> value</code>
Greater than, etc.	<code>>= <= > <</code>	<code>3 >= 4</code>
Equal to, not equal to	<code>== !=</code>	<code>3 != 4</code>
Logical And	<code>&&</code>	<code>passed && juniorStatus</code>
Logical OR	<code> </code>	<code>passed juniorStatus</code>
Assignment, etc.	<code>= += *= -= /= %=</code>	<code>value += 4</code>
Conditional expression	<code>? :</code>	<code>passed ? "happy" : "sad"</code>

Arithmetic operators

Most of the arithmetic operators such as addition and multiplication work the same in C++ as they do in algebra. There are a few exceptions: integer division, modulus, and the increment operator.

Division /

Floating point division (/) behaves the way it does in mathematics. Integer division, on the other hand, does not. The evaluation of integer division is always an integer. In each case, the remainder is thrown away. To illustrate this, consider the following:

```
{
    int answer = 19 / 10;
    cout << answer;
}
```

In this case, the output is not 1.9. The variable `answer` cannot store a floating point value. When 19 is divided by 10, the result is 1 with a remainder of 9. Therefore, `answer` will get the value 1 and the remainder is discarded.

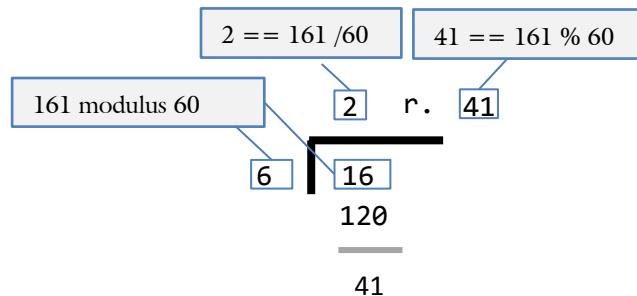
Modulus %

Recall that integer division drops the remainder of the division problem. What if you want to know the remainder? This is the purpose of the modulus operator (%). Consider the following code:

```
{
    int remainder = 19 % 10;
    cout << remainder;
}
```

In this case, when you divide 19 by 10, the remainder is 9. Therefore, the value of `remainder` will be 9 in this case. For example, consider the following problem:

```
{
    int totalMinutes = 161;           // The movie "Out of Africa" is 161 minutes
    int numHours   = totalMinutes / 60; // The movie is 2 hours long ...
    int numMinutes = totalMinutes % 60; // ... plus 41 minutes
}
```



Increment ++

There are two flavors of the increment (and decrement of course) operators: increment before the expression is evaluated and increment after. To illustrate, consider the following example:

```
{
    int age = 10;
    cout << age++ << endl;      // display the value of 10 because the expression
}                                //     is evaluated before age is incremented
```

In this example, we increment the value of `age` *after* the expression is evaluated (as indicated by the `age++` rather than `++age` where we would evaluate *before*). Therefore, the output would be 10 although the value of `age` would be 11 at the end of execution. This would not be true with:

```
{
    int age = 10;
    cout << ++age << endl;      // display the value of 11 because the expression
}                                //     is evaluated after age is incremented
```

Streams

The standard tools to perform text input and output with C++ is with the standard stream libraries. This includes writing text to the screen and accepting text from the keyboard as well as interacting with files.

Output

Text output on the screen is performed with the `cout` object. You can setup your program to write text to the screen by including the `iostream` library and using the standard namespace.

```
#include <iostream>           // where cout and cin live
using namespace std;          // cout and cin are part of the standard namespace
```

Simple text and number output can be performed with `cout` and the insertion operator `<<`.

```
{
    int variable = 10;

    cout << "text"           // c-string literals can be displayed
        << 42                 // numeric literals can be displayed
        << variable           // variables can be evaluated
        << 4 + 6               // expressions can be evaluated
        << endl;               // signify the end of a line
}
```

You can left-align text with the tab '\t' character, you can right-align text with the `setw()` function:

```
{
    cout << "\tFirst line aligns with\n"
        << "\tSecond line\n"
        << setw(20) << "right\n"
        << setw(20) << "align this text\n";
}
```

Note that `setw()` needs the `iomanip` library:

```
#include <iomanip>
```

Input

Input is primarily accomplished through the `cin` object and the extraction `>>` operator.

```
{
    int number;
    char text[256];

    cin >> number >> text;           // first fetch a number, then a string
}
```

It is also possible to fetch an entire line of text:

```
{
    char text[256];

    cin.getline(text, 256);          // fetch an entire line of text up to the \n
}
```

Finally, it is possible to fetch only a single character from the input stream, including a white space:

```
{
    char character;

    character = cin.get();          // fetch a single character. This could be a
                                    // letter, digit, symbol, or even a space
}
```

File

To read or write from a file, it is necessary to use the `fstream` library:

```
#include <fstream>
```

To read data from a file, it is necessary to open the file, fetch the text, and close the file:

```
int getNumber(const char * filename)
{
    ifstream fin(filename);        // the fin object will point to the file in filename
    if (fin.fail())                // always check to see if the file correctly opened
        return 0;                  // if we failed, do not continue on

    // fetch the data
    int data;                     // reading from a file is the same as accepting
                                    // input from the keyboard
    fin >> data;
    fin.close();                  // do not forget to close the file when finished
    return data;
}
```

Writing to a file follows the same pattern except we create an `ofstream` object:

```
void writeNumber(const char * filename, int data)
{
    ofstream fout(filename);       // just like with fin, we need to open the file
    if (fout.fail())              // always check the error state
        return;

    // write the data
    fout << data << endl;         // write to a file exactly the same as you would
                                    // output data to the screen
    fout.close();                 // do not forget to close the file!
}
```

Loops

Loops are mechanisms to allow a program to execute the same section of code more than once. There are three types of loops in C++: WHILE, DO-WHILE, and FOR:

while	do-while	for
A WHILE loop is good for repeating through a given block of code multiple times. <pre>{\n while (x > 0)\n {\n x--;\n cout << x << endl;\n }\n}</pre>	Same as WHILE except we always execute the body of the loop at least once. <pre>{\n do\n {\n x--;\n cout << x << endl;\n } while (x > 0);\n}</pre>	Designed for counting, usually meaning we know where we start, where we end and what changes. <pre>{\n for (x = 10;\n x > 0;\n x--)\n {\n cout << x << endl;\n }\n}</pre>

While

The simplest loop is the WHILE statement. The WHILE loop will continue executing the body of the loop until the controlling Boolean expression evaluates to `false`. The syntax is:

```
while (<Boolean expression>)\n    <body statement>;
```

An example of the WHILE loop in action is to verify that the user input is a valid letter grade:

```
char getGrade()\n{\n    char grade; // the value we will be returning\n\n    // initial prompt\n    cout << "Please enter your letter grade: ";\n    cin >> grade;\n\n    // validate the value\n    while (grade != 'A' && grade != 'B' && grade != 'C' &&\n          grade != 'D' && grade != 'F')\n    {\n        cout << "Invalid grade. Please enter a letter grade {A,B,C,D,F} ";\n        cin >> grade;\n    }\n\n    // return when done\n    return grade;\n}
```

Do-while

The DO-WHILE loop is the same as the WHILE loop except the controlling Boolean expression is checked after the body of the loop is executed. The syntax is:

```
do
    <body statement>;
while (<Boolean expression>);
```

An example of the DO-WHILE loop in action is to verify that the user's age is greater than zero:

```
{
    int age;
    do
    {
        cout << "What is your age? ";
        cin >> age;
    }
    while (age < 0);
}
```



Sue's Tips

We commonly use WHILE and DO-WHILE loops in event-controlled loops, a loop that continues until a given event occurs. With these loops, the number of repetitions is typically not known before the program starts.

For

The final loop is designed for counting. The syntax is:

```
for (<initialization statement>; <Boolean expression>; <increment statement>)
    <body statement>;
```

Here the syntax is quite a bit more complex than its WHILE and DO-WHILE brethren.

```
for (int count = 0; count < 5; count++)
    cout << count << endl;
```

Initialization:

The first statement to be executed in a loop.

- Can be any statement.
- We can declare and initialize a variable inside the loop:

```
for (int i = 0; ...
```

- We can initialize more than one variable:

```
for (j = 0, k = 0; ...
```

- We can also leave it empty:

```
for (; i < 10; i++)
```

Boolean expression:

Is executed immediately before the body of the loop.

- Can be any expression.
- As long as the expression evaluates to **true**, the loop continues:
- If it is left empty, the expression evaluates to **true**. This means it will loop forever:

```
for (i = 0; ; i++)
```

Increment:

Is executed immediately after the body of the loop.

- Can be any statement.
- Usually we put a **++** or **--** here:
- You can put more than one statement here:

```
for (... ; ...; i++, j--)
```

- Can be left empty:

```
for (; i < 10; )
```

While the syntax of the FOR loop may look quite complex, it has the three things any counting problem needs: where to start (initialization), where to end (Boolean expression), and how much to count by (the increment statement). For example, a FOR loop to give a countdown from 10 to zero would be:

```
{
    // a countdown, just like what Cape Kennedy uses
    for (int countDown = 10; countDown >= 0; countDown--)
        cout << countDown << endl;
}
```



Sue's Tips

We commonly use FOR loops in counter-controlled loops, loops executing a fixed number of times. Counter-controlled loops are readily identified by the presence of a single variable that moves through a range of values. In other words, counter-controlled loops do not exclusively increment by one: they might increment by 10 or powers of 3.

When designing counter-controlled loops, it is helpful to answer the following four questions:

- How does the loop start
- How does the loop end
- What do you count by
- What happens each iteration

Functions

A function is a small part of a larger program. In fact, it is the fundamental unit of organization for a procedural program (the subject of CS 124). When designing writing a program with functions, three things need to be taken into account: modularization, the syntax of a function, and parameter passing.

Modularization

Modularization is the process of splitting a large program into smaller chunks. There are, of course, good ways of doing this and bad one. We have two metrics by which we can measure the quality of modularization in a given program: cohesion and coupling.

Cohesion is the quality of a function performing one and only one task. The seven levels of Cohesion (from highest to lowest) are:

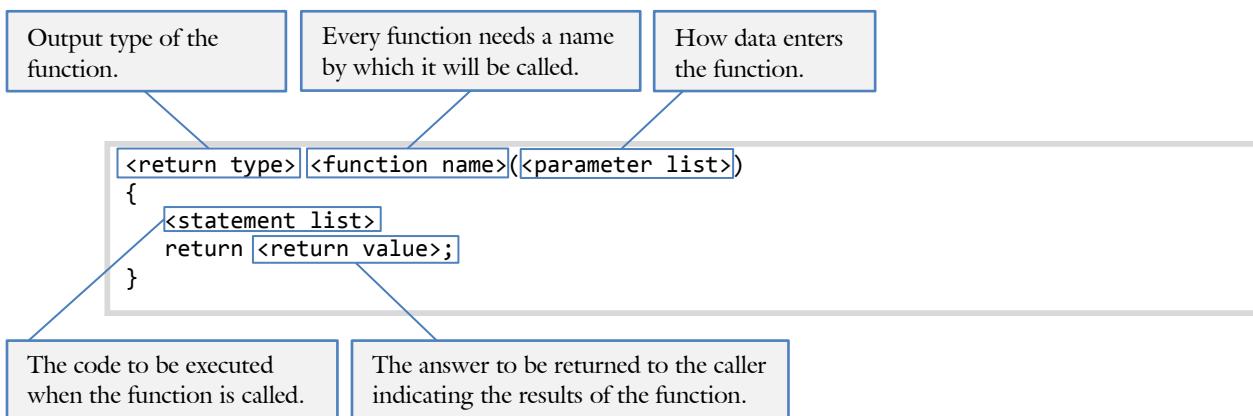
Cohesion Name	Description
Functional	Every item in a function or module is related to a single task.
Sequential	Operations in a module must occur in a certain order. Here operations depend on results generated from preceding operations.
Communicational	All elements work on the same piece of data.
Procedural	All related items must be performed in a certain order.
Temporal	Items are grouped in a module because the items need to occur at nearly the same time. What they do or how they do it is not important.
Logical	Items are grouped in a module because they do the same kinds of things. What they operate on, however, is totally different.
Coincidental	Items are in a module simply because they happen to fall together. There is no relationship.

Coupling is the quality of the information interchange between functions. Loose coupling, represented by simple data being passed between functions, is desirable over tight. The four levels of Coupling are:

Coupling Name	Description
Data	Occurs when the data passed between functions is very simple. This occurs when a single atomic data-item is passed, or when highly cohesive data-items are passed.
Stamp	Occurs when complex data or a collection of unrelated data-items are passed between modules.
Control	Occurs when one module passes data to another that is interpreted as a command.
External	Occurs when two modules communicate through a global variable or another external communication avenue.

Function syntax

The syntax for declaring a function is:



An example function for converting feet to meters is:

```

/******************
 * CONVERT FEET TO METERS
 * Convert imperial feet to metric meters
 *****/
double convertFeetToMeters(double feet)
{
    double meters = feet * 0.3048;
    return meters;
}
  
```

Calling a function occurs by naming the function and providing the required parameters. To call the `convertFeetToMeters()` function above, see the following example:

```

{
    double heightFeet = 5.9;
    double heightMeters = convertFeetToMeters(heightFeet);
}
  
```

Parameters

Parameter passing is the process of sending data between functions. The programmer is able to specify the parameters needed in a function with a comma-separated list. For example, consider the scenario where the programmer is sending a row and column coordinate to a display function. The display function will need to accept two parameters.

```
*****
 * DISPLAY COORDINATES
 * Display the row and column coordinates on the screen
 *****/
void displayCoordinates(int row, int column) // two parameters are expected
{
    cout << "(" << row // the row parameter is the first passed
        << ", " << column // the column parameter is the second
        << ")\n";
    return;
}
```

For this function to be called, two values need to be provided.

```
displayCoordinates(5, 10);
```

Parameter matchup occurs by order, not by name.

There are two ways to send data between functions: pass-by-value and pass-by-reference. Pass-by-reference” is the process of indicating to the compiler that a given parameter variable is shared between the caller and the callee. We use the ampersand & to indicate the parameter is pass-by-reference.

Pass By Value	Pass By Reference
<p>Pass-by-value makes a copy so two independent variables are created.</p> <p>Any change to the variable by the function will not affect the caller.</p> <pre>***** * * Pass-by-value * No change to the caller ***** / void notChange(int number) { number++; }</pre>	<p>Pass-by-reference uses the same variable in the caller and the callee.</p> <p>Any change to the variable by the function will affect the caller.</p> <pre>***** * * Pass-by-reference * Will change the caller ***** / void change(int &number) { number++; }</pre>

We use pass-by-reference to enable a callee to send more than one piece of data back to the caller. An example of this would be:

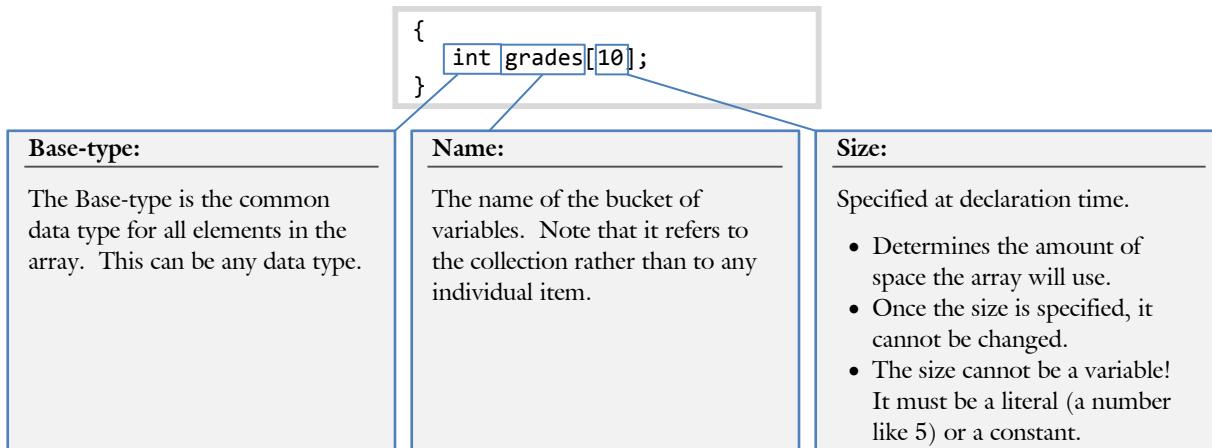
```
void getCoordinates(int & row, int & column)
{
    cout << "Please specify the coordinates x y: ";
    cin >> row >> column;
    return; // no data is sent using return
}
```

Arrays

In the simplest form, an array is a “bucket of variables.” Rather than having many variables to represent the values in a collection, we can have a single variable representing the bucket. There are two main tasks we do with arrays: create lists of data, and looking up values from a table. In call cases, it is necessary to know how to declare an array, reference individual items, and to pass arrays as parameters.

Declare

A normal variable declaration asks the compiler to reserve the necessary amount of memory and allows the user to reference the memory by the variable name. Arrays are slightly different. The amount of memory reserved is computed by the size of each member in the list multiplied by the number of items in the list.



It is also possible to initialize an array at declaration time:

Declaration	In memory	Description						
<code>int array[6];</code>	<table border="1"><tr><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table>	?	?	?	?	?	?	Though six slots were set aside, they remain uninitialized. All slots are filled with unknown values.
?	?	?	?	?	?			
<code>int array[6] = { 3, 6, 2, 9, 1, 8 };</code>	<table border="1"><tr><td>3</td><td>6</td><td>2</td><td>9</td><td>1</td><td>8</td></tr></table>	3	6	2	9	1	8	The initialized size is the same as the declared size so every slot has a known value.
3	6	2	9	1	8			
<code>int array[6] = { 3, 6 };</code>	<table border="1"><tr><td>3</td><td>6</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	3	6	0	0	0	0	The first 2 slots are initialized, the balance are filled with 0. This is a partially filled array.
3	6	0	0	0	0			
<code>int array[] = { 3, 6, 2, 9, 1, 8 };</code>	<table border="1"><tr><td>3</td><td>6</td><td>2</td><td>9</td><td>1</td><td>8</td></tr></table>	3	6	2	9	1	8	Declared to exactly the size necessary to fit the list of numbers. The compiler will count the number of slots
3	6	2	9	1	8			
<code>int array[6] = {};</code>	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	This is the easiest way to initialize an array with zeros in all the slots
0	0	0	0	0	0			

It is also possible to declare a multi-dimensional array. This is an array with more than one index:

```
<base-type> <variable>[<number of rows>][<number of columns>];
```

A grid of integers that is 3×4 can be declared as:

```
int grid[4][3];
```

Referencing an array

We access individual items in an array with the square-bracket operator [].

```
out << list[3] << endl;           // Access the fourth item in the list
```

One important difference between arrays and mathematical sequences is that the indexing of arrays starts at zero. In other words, the first item is `list[0]` and the second is `list[1]`. Since indexing for arrays starts at zero, the valid indices for an array of 10 items is 0 ... 9. This brings us to our standard FOR loop for arrays:

```
for (int i = 0; i < num; i++)
    cout << list[i];
```

From this loop we notice several things. First, the array index variable is commonly the letter `i` or some version of it (such as `iList`). This is one of the few times we can get away with a one letter variable name.

When referencing a multi-dimensional array, it is important to specify each of the dimensions. Again, we use the vertical dimension first so we use (Row, Column) variables rather than (X, Y). Back to our 3×4 grid example:

```
{
    int grid[4][3] =
    { // col 0   1   2
        { 8, 12, -5 }, // row 0
        { 421, 4, 153 }, // row 1
        { -15, 20, 91 }, // row 2
        { 4, -15, 182 }, // row 3
    };

    // two indices for 2d arrays
    int row;                                // vertical dimension
    int col;                                 // horizontal dimension

    // fetch the coordinates
    cout << "Specify the coordinates (X, Y) "; // people think in terms of X,Y
    cin >> col >> row;

    // paranoia!
    assert(row >= 0 && row < 4);           // a loop would be a better tool here
    assert(col >= 0 && col < 3);          // always check before indexing into
                                            // an array
    // actually display the contents
    cout << grid[row][col] << endl;
}
```

Observe how we add an `assert` immediately before we reference the items in the grid. Though we will learn more about asserts in Chapter 1.1, the important thing to note here is that there is nothing in the C++ language to prevent the programmer from accessing memory outside the valid range of the array. It is up to the programmer to provide those checks!

Passing an array as a parameter

Passing arrays as parameters is quite different than passing other data types. The reason for this is a bit subtle. When passing an integer, a copy of the value is sent to the callee. When passing an array, however, the data itself does not move. Instead, only the address of the data is sent. This means, in effect, that passing arrays is always pass-by-reference.

```
*****
* DISPLAY NAME
* Display a user's name on the screen
*****
void displayName(char lastName[], bool isMale) // no number inside the brackets!
{
    if (isMale)
        cout << "Brother ";
    else
        cout << "Sister ";
    cout << lastName;                                // treated like any other string
    return;
}
```

Passing multi-dimensional arrays as parameters works much the same for their single-dimensional brethren with one exception: it is necessary to specify the size of all the dimensions except the left-most dimension. Back to our 3×4 example, a prototype might be:

```
void displayGrid(int array[][3]); // column size must be present
```

Note that c-strings are arrays. Therefore, if you would like to create an array of c-strings, multi-dimensional arrays are necessary. Consider the following example:

```
void promptNames(char names[][256])           // the column dimension must be the
{                                              // buffer size
    // prompt for name (first, middle, last)
    cout << "What is your first name? ";
    cin >> names[0];                          // passing one instance of the array
    cout << "What is your middle name? ";
    cin >> names[1];                          // Note that the data type is
    cout << "What is your last name? ";
    cin >> names[2];                          // a pointer to a character,
                                                // what CIN expects
}

int main()
{
    char names[3][256];                      // arrays of strings are multi-
                                                // dimensional arrays of chars
    // fill the array
    promptNames(names);                      // pass the entire array

    // first name:
    cout << names[0] << endl;                // again, an array of characters

    // middle initial
    cout << names[1][0] << endl;              // first letter of second string

    // loop through the names for output
    for (int i = 0; i < 3; i++)
        cout << names[i] << endl;

    return 0;
}
```

Pointers

A pointer is a variable that does not hold data, but rather an address. All pointers have a data type, namely “a pointer to an integer” or “a pointer to a character.” Arrays and c-strings are pointer variables in C++.

Pointer syntax

When declaring a normal data variable, it is necessary to specify the data type.

```
<DataType> * <pointerVariable>;
```

The following is an example of a pointer to a `float`:

```
float * pGPA;
```

The first part of the declaration is the data type we are pointing to (`float`). This is important because, after we dereference the pointer, the compiler needs to know what type of data we are working with.

The address of any variable can be ascertained with the address-of operator:

```
{
    int variable = 42;
    cout << &variable;           // this will display a value such as 0x7fff9d235d74, the
}                                //      address of "variable" rather than the value which is 42
```

We can always retrieve the data from a given address using the dereference operator (*).

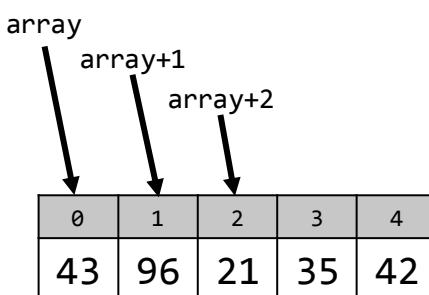
```
{
    int speed = 65;                  // the location in memory we will be pointing to
    int * pSpeed;                   // currently uninitialized. Don't dereference it!
    pSpeed = &speed;                // now it is initialized to the address of speed
    cout << *pSpeed << endl;       // need to use the * to get the data
}
```

Pointer arithmetic

An array is a pointer to a block of memory. Consider the following code:

```
{
    int array[5] =
    {
        43, 96, 21, 35, 42
    };
}
```

This corresponds to:



It is possible to iterate through all the elements in an array by incrementing a pointer:

```
{
    int array[5] =
    {
        43, 96, 21, 35, 42           // initialize the array
    };

    int * p = array;                // both p and array point to 43
    for (int i = 0; i < 5; i++)     // go through the loop five times
    {
        cout << *p << endl;        // display whatever p is pointing to
        p++;                      // advance p to point to the next item in array
    }
}
```

We can do something similar when iterating through a c-string, except we end the loop when we encounter the null-character.

```
{
    char text[] = "Pointer arithmetic";      // initialize a c-string
    for (char * p = text; *p; p++)          // standard FOR loop for iterating
        cout << *p << endl;                 // through a c-string
}
```

Allocating memory

It is possible to determine the size of a buffer at run-time using dynamic memory allocation with `new` and `delete`. When doing this, it is important to check for errors by catching the `bad_alloc` exception:

```
{
    // buffer size determined at run-time
    int bufferSize;
    cout << "Please specify the buffer size: ";
    cin  >> bufferSize;

    // attempt to allocate the memory
    try
    {
        double * array = new double[bufferSize];      // allocation done here

        for (int i = 0; i < bufferSize; i++)          // treat the array as you would
            cin >> array[i];                         // any other at this point

        delete [] array;                            // do not forget to free your memory
    }
    catch (bad_alloc)                           // new throws the bad_alloc
    {                                           // exception
        cout << "We failed to allocated "
            << bufferSize * sizeof(double)
            << " bytes of memory\n";
    }
}
```

Tools

A collection of tools, called libraries, are written to help us with the programming task. Some of these include the c-c-type library, the c-string library, the math library, the standard library, the string class, and the Standard Template Library (STL) vector container.

C-c-type

The c-c-type library allows the programmer to test the properties of characters. Examples include:

```
bool isalpha(char);      // is the character an alpha ('a' - 'b' or 'A' - 'Z')?
bool isdigit(char);     // is the character a number ('0' - '9')?
bool isspace(char);     // is the character a space (' ' or '\t' or '\n')?
bool ispunct(char);     // is the character a symbol (%#$!-_@.,? )?
bool isupper(char);     // is the character uppercase ('A' - 'Z')?
bool islower(char);     // is the character lowercase ('a' - 'b')?
```

The c-c-type library also offers the ability to change a character. Two examples are:

```
int toupper(char);      // convert lowercase character to uppercase. Rest unchanged
int tolower(char);      // convert uppercase character to lowercase. Rest unchanged
```

In order to use any of these functions, it is necessary to include the cc-type library:

```
#include <cctype>
```

C-string

The c-string library allows the programmer to perform common operations with c-strings, including:

```
int    strlen(const char *);           // find the length of a c-string
int    strcmp(const char *, const char *); // 0 if the two strings are the same
char *strcpy(char *<dest>, const char *<src>); // copies src onto dest
```

In order to use any of these functions, it is necessary to include the c-string library:

```
#include <cstring>
```

C-standard-library

The c-standard-library is a virtual grab-bag of helpful functions. Some of the most common are:

```
double atof(const char *); // parses input for a floating point number and returns it
int    atoi(const char *); // parses input for an integer number and returns it
int    rand();            // generate a random number between 0 and RAND_MAX
void   srand(unsigned int ); // initialize the random number generator rand()
int    system(const char *); // execute a system command as if typed on command prompt
int    abs(int);          // returns the absolute value of a number
```

In order to use any of these functions, it is necessary to include the c-standard-library:

```
#include <cstdlib>
```

String class

The `string` class is a data type designed to make text manipulation easier. An example using the `string` class is the following code:

```
#include <string>                                // don't forget the string library

int main()
{
    string lastName;                            // the data type is "string," no []
    cout << "What is your last name? ";
    cin  >> lastName;                          // as were needed with c-strings
                                                // cin works the way you expect

    string fullName = "Mr. " + lastName;        // the + operator appends

    cout << "Hello " << fullName << endl;      // cout works the way you expect
    return 0;
}
```

STL containers

There are a set of tools collectively called the Standard Template Library which facilitate storing groups or lists of items. Perhaps the most useful of these is the `vector` class. The `vector` class behaves the same as an array in many ways with one important difference: a `vector` object can change its size. Consider the following example:

```
#include <vector>                                // you need to include the vector library first

int main()
{
    vector <int> items;                           // it is necessary to specify the base-type in
                                                // the <>s, but not the number of items

    // fill the list
    for (int i = 0; i < 10; i++)
    {
        int number;
        cout << "Enter a number: ";
        cin  >> number;
        items.push_back(number);                  // the push_back() method allows us to add an
                                                // item onto the end of the vector
    }

    // display the results
    for (int i = 0;
         i < items.size();                      // the size() method retrieves the number of
         i++)
        cout << items[i] << endl;                // items currently in the vector
                                                // the [] operator works as you expect: fetching
                                                // the ith item from the list
    }
```

The `vector` class will grow to accommodate as many items as is added to it through the `push_back()` mechanism.

Problem 1 - 6

For each of the following, indicate where the parentheses go to disambiguate the order of operations.

1. `a && b || c && d`
2. `c ++ < ! 4 + 2`
3. `a || b && c + d * e`
4. `a += * b ++ * 7 || ! c + 5 > 2`
5. `1 < x < 10`

Please see page 6 for a hint.

Problem 6

If the tab stops are set to 8 spaces, what will be the output of the following code?

```
{
    cout << "\taa\n";
    cout << "aa\taa\n";
}
```

Please see page 8 for a hint.

Problem 7

How much space in memory does each variable take?

- `bool value;` _____
- `char value[256];` _____
- `char value;` _____
- `long double value;` _____

Please see page 4 for a hint.

Problem 8

What is the value of `a` at the end of execution?

```
float a = 1.0 + 2 * 3 / 4;
```

Please see page 6 for a hint.

Problem 9

What is the value of `b` at the end of execution?

```
int b = (float) 1 / 4 * 10;
```

Please see page 5 for a hint.

Problem 10 - 16

What are the values of the following variables?

{

```
bool a = false && true || false && true;
```

10.

```
bool b = false || true && false || true;
```

11.

```
bool c = true && true && true && false;
```

12.

```
bool d = false || false || false || true;
```

13.

```
bool e = 100 > 90 > 80;
```

14.

```
bool f = 90 < 80 || 70;
```

15.

```
bool g = 10 + 2 - false;
```

16.

}

Please see page 6 for a hint.

Problem 17

What is the output?

```
char value = 'a';

int main()
{
    char value = 'b';

    if (true)
    {
        char value = 'c';
    }

    cout << value << endl;

    return 0;
}
```

Problem 18

What is the output?

```
{  
    bool failedClass = false;  
    int grade = 95;  
  
    // pass or fail?  
    if (grade < 60);  
        failedClass = true;  
  
    // output grade  
    cout << grade << "%\n";  
  
    // output status  
    if (failedClass)  
        cout << "You need to take "  
            << "the class again\n";  
}
```

Problem 19

What is the output when the user inputs the number 5?

```
{  
    int number;  
  
    // prompt for number  
    cout << "number? ";  
    cin >> number;  
  
    // crazy math  
    if (number = 0)  
        number += 2;  
  
    // output  
    cout << number << endl;  
}
```

Problem 20

What is the output?

```
void weird(int a, int & b)  
{  
    a = 1;  
    b = 2;  
}  
  
int main()  
{  
    int a = 3;  
    int b = 4;  
  
    weird(a, b);  
  
    cout << a * b << endl;  
    return 0;  
}
```

Please see page 14 for a hint.

Problem 21

What is the output?

```
int setZero()
{
    int value = 0;
    return value;
}

int main()
{
    int value = 10;

    setZero();

    cout << value << endl;
    return 0;
}
```

Please see page 13 for a hint.

Problem 22-26

Write the function prototype to:

22. Update the bill to include the 15% tip

23. Display the price of a used car

24. Convert meters to feet

25. Prompt the user for his name

26. Display the contents of a Sudoku board

Please see page 14 for a hint.

Problem 27

What is the output?

```
{
    int a[] = {2, 4, 6, 8, 10};
    int b = 0;

    for (int c = 1; c < 4; c++)
        b += a[c];

    cout << b << endl;
}
```

Please see page 15 for a hint.

Problem 28

What is the output of the following code fragment?

```
{
    int array[2][2] =
        { {3, 4}, {1, 2} };

    cout << array[1][0];
}
```

Please see page 15 for a hint.

Problem 29

Given the following code:

```
{
    int array[] = {7, 14, 21, 28};
}
```

How can you output the 3rd item in the list without using the square bracket operator []?

Please see page 18 for a hint.

Problem 30-33

Describe what each of the following functions do:

```
void mystery(char * p)
{
    while (*p)
        cout << *(p++);
}
```

```
void mystery(char * p1, char * p2)
{
    while (*(p1++) == *(p2++))
        ;
}
```

```
int mystery(char * p1)
{
    char * p2 = p1;
    while (*(p2++))
        ;

    return p2 - p1 - 1;
}
```

```
bool mystery(char * p1, char * p2)
{
    while (*p1 == *p2 && *p1)
    {
        p1++;
        p2++;
    }

    return (*p1 == *p2);
}
```

Please see page 18 for a hint.

Problem 34

Write the code to find the sum of all the items in the following array of integers:

```
{  
    int array[10] = {5, 4, 7, 3, 5, 9, 8, 1, 3, 2};  
    int sum = 0;  
  
}
```

Please see page 7 for a hint.

Problem 35

Write the code to display the contents of a string, one character on each line:

```
void display(const char * text)  
{  
  
}
```

Please see page 7 for a hint.

Problem 36

Match the declaration with the type of data:

int & a;
int a;
int @ a;
int * a;
int ** a;

Pointer to an integer
Error
Pointer to a pointer to an integer
A reference to an integer
An integer variable

Please see page 18, 101 for a hint.

Problem 37

Match the description of the statement with the code:

int * p = new int;
int * p = new int *;
int * p = new int(7);
int * p = new int(int);
int * p = new int[7][7];
int * p = new int[7]

Allocate a pointer to a function
Allocate an array of 7 integers
Allocate an integer and leave the memory un-initialized
Error
Allocate an integer and initialize the memory to 7
Allocate a 2-dimensional array: 7x7

Please see page 19 for a hint.

Problem 38

Fibonacci is a sequence of numbers where each number is the sum of the previous two:

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n - 1) + F(n - 2) & \text{if } n > 1 \end{cases}$$

Write the code to generate the first 100 numbers in the Fibonacci sequence

Please see page 15 for a hint.



Unit 1. Using Objects

1.0 Design Documents	30
1.1 Defensive Programming.....	43
1.2 Exception Handling.....	55
1.3 Structures	65
1.4 Separate Compilation.....	79
1.5 Function: Advanced Topics.....	92

1.0 Design Documents

Sam is anxious to begin work on a new programming project. Though the project is large and complex (the game of Tic-Tac-Toe), Sam knows how to do some of the easier things and wants to get them out of the way. As he feverously starts typing, Sue enters the lab. Sue also wants to begin her project early and is surprised by Sam's progress. "How are you going to solve the file-format problem" Sue asks him. "Umm, I haven't figured that out yet." "How are you going to handle user input" she asks. Again, Sam has no answer. Finally, truly bewildered, she asks Sam how he can type any code without knowing how to solve the key problems of the project. With this, Sam replies "Well, you have to begin somewhere." Sue chews on this for a few minutes. How do you begin a large project such as this?

Objectives

By the end of this chapter, you will be able to:

- See the benefit of designing a program before writing the code
- List and define the components of a design document
- Understand how the components of a design document work together to help the programmer solve key problems

Prerequisites

Before reading this chapter, please make sure you are able to:

- Write the pseudocode for a given algorithm
(Procedural Programming in C++, Chapter 2.2)
- Create a structure chart describing the dataflow and modularization of a program
(Procedural Programming in C++, Chapter 2.0)

If you are unsure of any of these prerequisites, please take a moment to review them before proceeding.

What is a design document and why you should care

Computer languages such as C++ force programmers to focus on a myriad of little details when writing code. This makes it difficult to see the big picture, to design with broad strokes and to make big decisions. It is like building an airplane and people keep asking you what kind of screw you want to use. What kind of screw!? You don't even know how many engines will be needed!

The **design document** is meant to free you from these details so you can identify and solve the key problems. Once these big decisions are made, then we can address the details required when writing code. Perhaps more importantly, the design document helps us identify which path we do not want to pursue before we get too committed to a given solution. The danger of getting bogged down in details is that large parts of the problem remain unknown. Often the value of a decision is not realized until long after the decision is made. By drafting out a solution to the entire problem, however, we can see how all our design decisions interact with each other early in the design process. In other words, the design document gives the programmer the ability to see and address the big issues of a project before committing to any one solution.

It is perhaps best to think of the design document as a suite of tools, each designed to address a specific aspect of a programming project. Though each of these tools has a purpose, it is not uncommon to encounter a problem where some of the tools are not needed. The parts of a design document are:

Part	Purpose
Problem Description	Clearly state what problem is being solved so everyone involved is on the same page
Design Overview	Summarize to the reader of how the problem will be solved. Highlight the big design decisions so the reader will know what to focus on
Interface Design	Make sure the I/O functionality of the program is understood early in the design process
Structure Chart	Make an inventory of the functions of the program
Algorithms	Identify and draft the algorithms of the key functions
Data-structures	Describe how data will be stored in the program while it is running
File Format	Describe how data will be read from and written to a file
Error Handling	Identify sources of user, file, and internal errors the program will likely encounter

If the programmer solves all of these problems before a line of code is written, then the actual process of writing code will be relatively straight forward.

One final note: there is no such thing as a standard template for a design document. All programmers do this a little differently according to the needs of their project and their personal taste. For the purpose of this class, we will use the above format so you can be familiar with the tools that are commonly used in a design document. After you have mastered these tools, you can decide for yourself the composition of your design documents.



Sue's Tips

Generally, the more detailed the design document, the smaller the chance for unforeseen problems to derail the project. It is not uncommon in industry to spend a day on design for every two days writing code.

At the beginning of every project, the programmer is asked to make an estimate of how long a given feature will take to implement. This is very difficult: if the programmer over-estimates the time to complete the project then he will be perceived as being less valuable. If the programmer under-estimates then he will need to work extra hours to meet his ambitious timeline. The design document helps the programmer deliver an accurate estimate because it helps the programmer to better understand the solution.

1. Problem description

The first part of the design document is the program description. In an academic setting, this is the easiest part of the design document. The only thing required is to summarize the assignment in a sentence or two. That is it! In the workplace, however, this is much more complex.

Often the most difficult part of a project is just identifying what the project is all about. Perhaps this is best explained by example. Consider a young programmer who decides to make a new mobile calendar application. Being a diligent software engineer, he starts with the program description. Suddenly he is confronted with such questions as:

- What is the point of this project?
- Who is going to use it, and why?
- What makes it different/better than the hundreds of similar programs out there currently?

The whole point of the program description is to get clarity, so there is no question as to what problem the program is meant to solve.

What: The design overview is a brief description of the problem the program is designed to solve. If you cannot describe it in a couple of sentences, then you probably don't understand it yourself.

How: Try to tell a classmate or friend what the project is all about. If you can do it in a single sentence, then write that sentence down in the program description. If not, keep working on it!

Example: Consider the classic game of Tic-Tac-Toe. A program description might be:

This program will allow the two users to play Tic-Tac-Toe. The game will be saved in a file and the users will be able to interact with their game using a text-based interface.

Sam's Corner



The program description should be concise and largely technology free. Read and re-read the problem description from the assignment and summarize it in a single sentence. If you are spending more than a minute on it, you are over-thinking it!

2. Design overview

The design overview serves the same purpose as the introduction to an essay: it gives the reader a fighting chance of understanding what is to follow. In other words, the design document consists of a variety of tools designed to describe various parts of the solution. Because these tools describe different aspects of the solution, it is not easy to connect the dots. One part of the solution may be described in the structure chart, in algorithms, and in the file I/O section. The design overview describes the entire approach to solve the problem holistically so the reader knows what to look for when the other tools are described.

The single biggest mistake students make when writing a design document (aside from not allocating enough time to accomplish the task) is to confuse the design overview with the problem description. The problem description describes what problem you are solving. The design overview describes how the problem is to be solved. Thus the design overview does not address the behavior of the program (that is for the problem description). It will address what strategy the programmer should follow to write the code.

The problem description describes what problem you are solving.

The design overview describes how the problem is to be solved.

What: The design overview should briefly describe the elements of your program. This includes how you will go about solving the problem described in the problem definition. Often this includes the main modules, the data-structures that will be used, and the algorithms that will be needed. The detail here will be sufficient for the reader to understand how the problem will be solved, but not necessarily detailed enough to actually build it.

How: Describe the highlights of how the code will be written, focusing on the major decisions you made in the other parts of the design document. For most assignments in college, the design overview will be one or two short paragraphs. In the workplace, the design overview can be a full page.

Example: Back to our Tic-Tac-Toe program, a design overview might be:

The game will be stored in a 3x3 array of characters. All user interactions will be directed towards modifying this array. The board will also be stored in a file in a 3 row, 3 column grid.



Sue's Tips

Though the design overview is near the beginning of the design document, it is the last part written. You often don't have a fully developed idea of how you will solve the problem until the design document is complete.

One way to think of this is: if you were to describe to a friend how you are going to solve this program in one minute, how would you do that? What are the big decisions you made? What is the gist of your basic approach?

3. Interface design

The interface design describes everything the user will encounter with the program. This is the most tedious but the easiest part of the design document. It is tedious because you need to discover and describe every single output the program could generate and every input the user could enter.

In industry, the interface design can be obtained from the spec (short for “specification”). The spec is a document describing how a given feature or product is to work and is provided to the programmer by some designer. If the spec is written correctly, the programmer needs only to search the depths of the spec for the interface design and paste it into his design document. If the spec is not done correctly, then the programmer will need to ask the designer to provide those interface details. In either case, the programmer simply cannot begin the project without a clear understanding of what the interface will be like.

In academia, the interface design is usually provided by the teacher. Either it is part of the program description, it is implied with a working program that the student needs to explore, or the interface is up to the student to describe. In either case, the student cannot begin the project without a clear understanding of the interface.

What: A detailed description of the input the user may enter, the output the program needs to produce, and all the error messages the program may generate.

How: Three tables are usually required:

- **Output:** If the program uses text input and output, provide the actual text the program will generate. If the program uses graphical output, provide a screen shot.
- **Input:** Create a table with two columns. The first describes the input and the second describes how the program will respond to it.
- **Errors:** Create a table with two columns. The first describes the text of the error message and the second describes when the error message will be presented to the user.

Example

With our Tic-Tac-Toe program, a sample from the Output section is:

Output screen-shot	What it is used for
	Displaying the board after every turn

A small sample of the Input section is:

Input value	How the program responds
s	Save a board to a file
e	Edit a square
q	Quit the game

One example error is:

Error message	When the message appears
Unable to open file file.txt for reading.	File missing or otherwise unavailable for reading

4. Structure chart

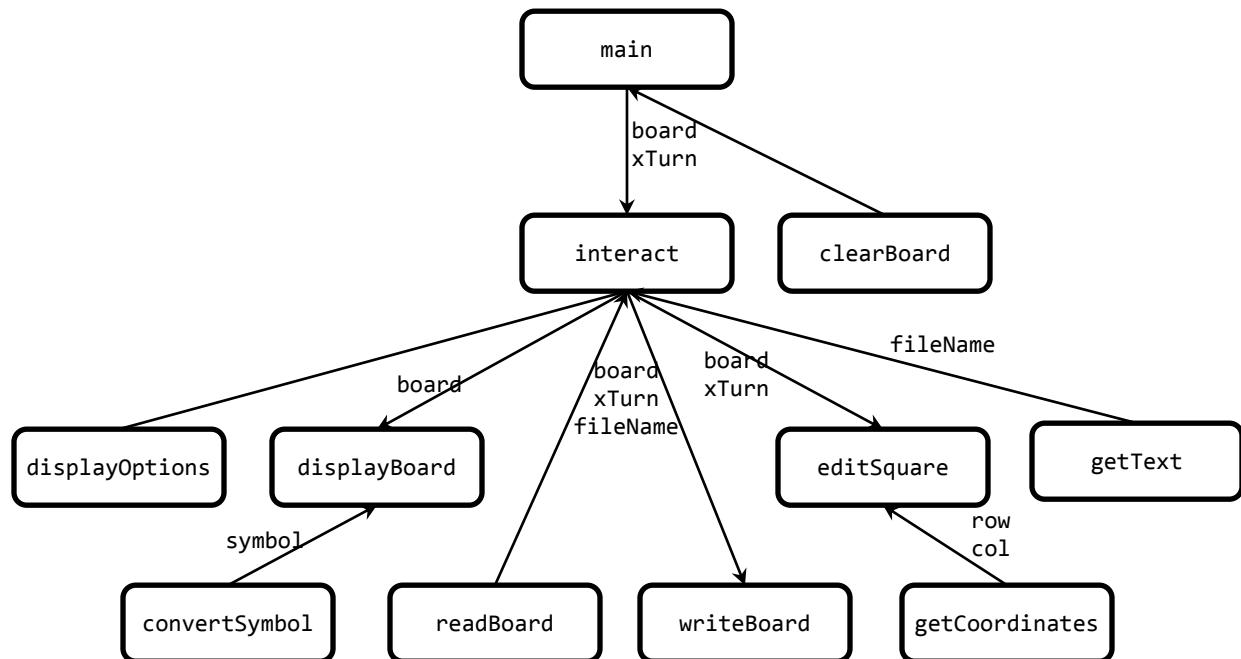
Any reasonably complex design consists of myriad of parts, all working together as a single whole. The purpose of the structure chart is to identify these parts and describe how they interact. Note that the function of the parts themselves is not described (thereby underscoring why self-descriptive function names are so important). For programming problems, the tool of choice for designing at this level is the structure chart.

What: A structure chart is a graphical representation of the modules in a program and how information passes between them. A structure chart is the primary design tool for procedural designs.

How: The structure chart is a graph containing three components:

- Functions: the functions of the program with a circle drawn around them.
- Lines: lines connecting functions that call each other.
- Parameters: labels on the lines connecting functions if data flows between them.

Example: Back to our Tic-Tac-Toe program, the Structure Chart would be the following:



Sue's Tips

Though it is possible to draw a structure chart electronically, this can be a time-consuming and frustrating experience. It is best to leave a half page blank in your design document and draw it by hand. Before you do this, however, take a moment and draw a few drafts on scratch paper. Make sure that `main()` is on top and functions get called top-to-bottom like the above example.

When you are finished with your structure chart, make sure it is neat and readable. Sloppy work in a structure chart often is a reflection of sloppy thinking.

5. Algorithms

While the structure chart describes how the functions of the program interact with each other but do not describe what occurs within a function, the algorithms section has the opposite purpose. Here we look at individual functions and briefly forget about the rest of the program. Note that you must have a structure chart before you can do this; otherwise you won't know what algorithms will need to be written!

What: The algorithms section describes the critical algorithms of the program. Trivial functions, such as `getFileName()`, have no place in the design document. Instead, focus on the handful of hard functions. If you can't just sit down and write the code for a function, then draft it out in the algorithms section. This is perhaps the most common mistake programmers make in their design documents.

How: Pseudocode is the tool of choice for algorithms. Remember to make it detailed enough that we have actual variable names and equations, but not so detailed that the language of C++ comes through.

Example: One non-trivial function from the Tic-Tac-Toe game is `readBoard()`. The tricky part is determining whose turn it is. If there are an equal number of X's and O's, it must be X's turn; otherwise it is O's turn to go next.

```
readBoard(fileName, board, xTurn)

fin(fileName)
IF fin.fail()
    PUT error

numXoverO = 0
FOR row = 0...2
    FOR col = 0...2
        READ board[row][col]
        IF board[row][col] = X
            numXoverO + 1
        IF board[row][col] = O
            numXoverO - 1

xTurn ← (numXoverO == 0)
```

The next non-trivial function is `editSquare()`, how the user is allowed to put an X or O on the board. Again the function needs to keep track of whose turn it is.

```
editSquare(board, xTurn)
getCoordinates(row, col)

IF board[row][col] ≠ '.'
    RETURN

IF xTurn
    board[row][col] = X
    xTurn = FALSE
ELSE
    board[row][col] = O
    xTurn = TRUE
```



Sue's Tips

In practice, any algorithm that could be the source of bugs or prove difficult to implement should be specified in pseudocode so difficulties could be worked out before the code is written. The sooner the bugs are found, the easier they will be to fix

6. Data structure

Most data-structure decisions in a program are trivial: use an integer for counting, a Boolean for logical data, and a string for text. It is often the case when there is no obvious built-in data-structure that can solve a given problem. Consider, for example, a program designed to play a MadLib® game from Project 3 of last semester. Should the story be stored as a single string? As an array of lines? As an array of words? Each of these possibilities has different ramifications for the rest of the program. In many ways, the choice of data-structure is the most critical design decision for a MadLib® program.

What: The data-structures addresses how the data in the program will be stored. Don't bother with trivial items such as file names or indexes. Instead, focus on the data that drives the program. If you have an array, structure (Chapter 1.3), class (Chapter 2.0), or any other type of data-structure, it should be described here.

How: For procedural programs, describe how the user's data will be stored in the program. Later in the semester we will use UML class diagrams to describe Object-Oriented constructs.

Example: There are two non-trivial data-structures in our Tic-Tac-Toe game: a coordinate and a board.

- Coordinate: a structure with two member variables: row and column.

Coordinate
row
col

- Board: a 3x3 array of characters were the possible tokens are:

Token	Meaning
X	Space occupied by X
O	Space occupied by O
.	Unoccupied space

Sam's Corner



When defining a data-structure for a given problem, it is important to consider how much memory the data-structure will take. This is an important consideration when we have many instances of that data-structure in a program. A pixel consisting of three chars, for example, is much more compact than one consisting of three integers. This is especially true when there are millions of pixels in a given image.

7. File format

Whereas the data-structures section of the design document is concerned with how to store the user's data while the program is executing, the file format section is concerned with how to persist that data between executions of the program.

What: Describe the format of data in the file as well as how that data is converted into a run-time data-structure at file-read and file-write time.

How: Give an example of the file format. Describe any translation that needs to happen to convert this file format into a data-structure. Typically this includes a diagram as well as an English description.

Example: An example file for our Tic-Tac-Toe game is the following:

```
x0.  
...  
.x.
```

The 'x' and 'o' token map directly to the corresponding token in the file format. The '.' token maps to a space.



Sue's Tips

There are two overriding concerns when defining a file format: the amount of space required to store the user's data, and how easy it is to perform the read/write operations.

When faced with a fixed file format, namely a file format that the programmer does not get to specify, then the programmer often must choose a data-structure that facilitates the read/write process. Also, it is important that the programmer fully understands the associated standards and requirements for the file format before attempting to write the code.

A more common scenario occurs when the programmer is given the freedom to define an appropriate way to store the data in a file. In these scenarios, pay special attention to space and translation issues.

8. Error handling

In the ideal world users never make mistakes, files always have data in the expected format, and programmers never make mistakes. Of course we do not live in the perfect world. Our programs need to be designed to withstand errors from any source and still function properly. The best way to accomplish this is to think about error handling early in the design process.

What: The error handling section will represent your best guess as to the types of errors that your program will need to handle. This is always incomplete; some errors are difficult to predict until you write the program or test the software. However, every error you catch early in the process will save a great deal of time further down the road.

How: Enumerate the three types of errors that a program is likely to handle:

- User Errors: unexpected input coming from the user. This could be anywhere from an incorrect filename to an invalid coordinate. User errors are typically handled with an IF statement and an error message.
- File Errors: data in a file is in a different order or a different format than is expected. This could be caused by a bug in the file creation code or by the file itself being changed by the user. File errors are usually handled with an IF statement and an error message.
- Internal Errors: errors caused by bugs within a program. These are all detected with asserts (Chapter 1.1) or other types of debug code.

Example: The user errors that the Tic-Tac-Toe program will have to handle include:

Error	Condition	Handling
Invalid option	User input something other than r,s, or q	Re-prompt and error message
Invalid coordinate	Coordinate specification that is not [a..c][1..3]	Re-prompt
Invalid filename	Unable to open file with specified filename	Re-prompt and error message

The file errors in the Tic-Tac-Toe program include:

Error	Condition	Handling
Unexpected token	Token in file that is not X, 0, or .	Error message and stop loading file
Too few tokens	Reach EOF before 9 tokens are read	Error message and stop loading file
Impossible game	There should be the same number of xs and os or one more X than o	Error message and stop loading file

The final type of errors is internal:

Error	Condition	Handling
Malformed coordinate	coord.x > 2 or coord.x < 0 or coord.y > 2 or coord.y < 0	Assert
Malformed board	Any value in the board that is not X, 0, or .	Assert



Sue's Tips

The condition of every error, be that user, file, or internal, should be readily convertible into a Boolean expression. If it is not, you don't understand the error well enough yet.

Example 1.0 – Tic-Tac-Toe Design Document	
Demo	This demo is meant to illustrate a complete design document for a program approaching the complexity of what one would expect to see in CS 124.
Problem	Write a design document for a simple game to play Tic-Tac-Toe. The game should be able to allow two users to play the game and the program should persist a game in a file.
Solution	<p>The important components of the design document are presented in the preceding section. It is important to observe how each section addresses a different aspect of the overall design.</p> <p>The screenshot shows the 'Tic Tac Toe' application. On the left, the 'Program Description' and 'Design Overview' sections provide context. The 'Interface Design' section shows the application's user interface with various input fields and their descriptions. On the right, there is a 'Structure Chart' showing the flow from 'main' to 'interact', and an 'Algorithms' section with pseudocode for reading and writing boards from files.</p> <pre> Structure Chart graph TD main --> interact main --> clearBoard interact --> board interact --> xTurn interact --> displayOptions interact --> displayBoard interact --> board interact --> xTurn interact --> fileName interact --> editSquare interact --> writeBoard interact --> getCoordinates board --> symbol board --> readBoard board --> writeBoard xTurn --> row xTurn --> col fileName --> editSquare fileName --> getCoordinates editSquare --> convertSymbol convertSymbol --> readBoard readBoard --> board writeBoard --> board getCoordinates --> row getCoordinates --> col </pre> <pre> Algorithms readBoard(fileName, board, xTurn) FILE(fileName) IF (file fail) PUT error numRow = 0 FOR row = 0, 2 FOR col = 0, 2 READ board[row][col] IF board[row][col] == 'X' numXoverQ += 1 IF board[row][col] == 'O' numXoverO -= 1 xTurn ← (numXoverQ == 0) getCoordinate(row, col) row ← col ← -1 PROMPT for value GET input IF 'A' ≤ input[0] ≤ 'C' col ← input[0] - 'A' IF '1' ≤ input[1] ≤ '3' row ← input[1] - '1' </pre>
Challenge	As a challenge, see if you can complete a design document for a project you have worked on in a previous semester. Pay special attention to the new sections (Design Overview, Interface Design, Data-structures, File Format, and Error Handling).
See Also	<p>The complete solution is available at:</p> <p>Example 1.0 – Tic-Tac-Toe.pdf</p> <p>The source-code for the game is available at 1-0-ticTacToe.html or:</p> <pre>/home/cs165/examples/1-0-ticTacToe.cpp</pre>

Problem 1

What typically goes at the top of a structure chart?

Answer:

Please see page 35 for a hint.

Problem 2

List and define the three components of the Interface Design section of the Design Document

- _____
- _____
- _____

Please see page 34 for a hint.

Problem 3

List and define all the components of the Design Document.

- _____
- _____
- _____
- _____
- _____
- _____
- _____
- _____

Please see page 31 for a hint.

Problem 4

Define each of the following Design Document tools:

- Structure Chart: _____
- UML _____
- Pseudocode _____

Please see page 35, 36, and 37 for a hint.

Challenge 5

Create a complete design document from the Calendar project (Project 2) from CS 124. This will obviously take more space than is provided here in the textbook...

Please see page 40 for a hint.

1.1 Defensive Programming

Sue is working on a design document for a project and is stumped by the “Error Handling” section. While she understands the importance of properly handling all types of errors to her program, she is unsure exactly how that is to work. Sue needs a set of tools to help her identify and deal with file, user, and internal errors.

Objectives

By the end of this chapter, you will be able to:

- Use asserts to catch internal errors
- Catch user errors with `cin.fail()`
- Recover from user errors with `cin.ignore()` and `cin.clear()`

Prerequisites

Before reading this chapter, please make sure you are able to:

- Describe the purpose of the Error Handling section of a Design Document (chapter 1.0)
- Make logical assertions use Boolean algebra (Procedural Programming in C++, chapter 1.5)
- Accept user input with `cin` (Procedural Programming in C++, chapter 1.2)
- Be able to use the `string` class (Procedural Programming in C++, chapter 4.2)

What is defensive programming and why you should care

Defensive driving is the process of driving your car in such a way to be constantly ready to handle the worst possible conditions. This is an important skill for keeping the occupants of your car safe at all points in time. Defensive programming is much the same way. This is about writing code so there is “no way possible” for the program to malfunction, regardless of the input of the user or other system faults that may occur. While this is clearly an impossible goal, the closer we get the more reliable our code becomes. A well written program should never malfunction regardless of the environment it is put in. It should resist failure to any combination of user input, be that malicious, ignorant, or accidental. When software falls short of this goal, the following eventualities may result:

- **Instability:** The user may perceive the software to be unreliable and untrustworthy.
- **Incompatibility:** The software may work on less systems than the user needs.
- **Insecurity:** The software may be vulnerable to malicious attacks compromising the user’s system or his confidential data.

Each of these eventualities is clearly undesirable. Though the sources of these defects are legion, the most common include: file errors, user errors, and internal errors.



Sue's Tips

The earlier you start thinking about errors, the easier they will be to catch. This is why the Error Handling section exists in a design document. It is well worth your time to brainstorm about the most likely sources of errors in your program. In many cases, significant sources of errors can be mitigated in the design process long before actual code is written.

File errors

When reading from a file, several things could happen causing a program to malfunction:

- **Missing file:** The requested file might not exist. It is common for the user to misspell a file name or specify a file that does not exist. A program should handle these errors gracefully.
- **Insufficient permissions:** The user might not have sufficient privileges to access the file.
- **Corrupt file:** The file may not be in the expected format.

In each case, checks must be in place to catch the errors. These are typically accomplished with extensive use of the `fail()` method. Back to our `readBoard()` function from the Tic-Tac-Toe example:

```
void readBoard(const string & fileName, char board[][3], bool & xTurn)
{
    // open the file
    ifstream fin(fileName.c_str());
    if (fin.fail())                                // always check for errors when
    {                                                 // opening a file
        cout << "Unable to open file "
            << fileName << " for reading.\n";
        return;                                       // see chapter 1.2 for ways to report
    }                                                 // these errors to the caller

    // read the contents of the file
    int numXover0 = 0;
    for (int row = 0; row < 3; row++)
        for (int col = 0; col < 3; col++)
    {
        // read the item from the board
        fin >> board[row][col];

        // check for failure.
        if (fin.fail())                                // a failure can also occur in the middle
        {                                                 // of reading a file. Check them too!
            cout << "Error reading file "
                << fileName << ".\n";                  // tell the user that the failure
            fin.close();                               // happened, and why!
            return;                                     // don't forget to close the file!
        }
    }

    if (board[row][col] == 'X')
        numXover0++;
    else if (board[row][col] == 'O')
        numXover0--;
    else if (board[row][col] != '.')      // make sure the file only consists
        cout << "Unexpected symbol "           // of X, O, and . symbols.
            << board[row][col]
            << "in file: " << fileName << ".\n";
}

// determine whose turn it is by the xOver0 count
xTurn = (numXover0 == 0 ? true : false);
if (numXover0 != 0 && numXover0 != 1)      // there must be an equal number of
    cout << "Invalid board in file "
        << fileName << ".\n";                  // X's and O's or there must be
                                                // one more X.

// close the file
fin.close();
}
```

Observe how the added code verifies that the file exists and the data in the file is in the expected format. We need to do all we can to detect every possible misconfiguration of the file.

User errors

One must always assume that the user will enter the worst possible input to a given prompt. While most users are not malicious, ignorant or careless users often find ways to make programs malfunction. The two most common form of user errors are out-of-bounds errors, buffer overrun errors, and unexpected input.

Bounds checking

A program prompting the user for a row on a Tic-Tac-Toe board expects the value to be between 1 and 3, and a column value to be between ‘A’ and ‘C’. Thus, if the user enters a value outside that range, the user must be re-prompted. The process of verifying that user input is within a pre-defined range is called “bounds checking.”

Back to our Tic-Tac-Toe example, one of the bugs with the original program is that the bounds checking is missing. To address this issue, the following code was added:

```
void getCoordinates(int & row, int & col)
{
    char colLetter;

    do
    {
        // prompt with instructions
        cout << "Please specify the coordinates: ";
        cin >> colLetter >> row;           // an error may occur here too!
    }
    while (colLetter < 'A' || colLetter > 'C' ||
           row < 1 || row > 3);

    // convert for a letter to a number
    col = colLetter - 'A';

    // convert from 1's based to 0's based
    row--;
}
```

Observe how the added DO-WHILE loop serves to catch the case where user input is outside the valid range. This guarantees that the variable `row` and `col` are in the expected range.



Sue's Tips

As a general rule, every time the user is given the opportunity to input a number, the program should verify that the number is in the valid range before the value is used. It is therefore a very common pattern to use a DO-WHILE loop in a prompt function.

Buffer overruns

A **buffer** is like an array, a block of memory set aside for a task. If the programmer sets aside ten slots of memory, what happens when eleven are used? What happens when the programmer uses memory that is not reserved?

A buffer is an array of data to be filled with user input

```
{
    int array[10];           // ten integers

    for (int i = 0; i < 20; i++)
        array[i] = 0;         // what happens when twenty numbers are put
                            // into an array that can hold ten?
}
```

The answer to this question is: bad things will happen. It is akin to building an Olympic sized swimming pool on a small city lot: the pool will extend into your neighbor's land and he will not be happy! With a computer program, a buffer overrun like this will probably cause the program to crash.

Often, however, array bounds bugs are more difficult to find. Observe that c-strings are arrays of characters with a fixed length (say 256 characters). Thus the following code from our Tic-Tac-Toe game has a buffer overrun bug.

```
void getText(const char * prompt, char * input)
{
    cout << prompt;
    cin >> input;           // ERROR: could be longer than 255!
}
```

In this case, the assumption is that the user will not input more than 255 characters for his file name. While few individuals would enter such a long name, a malicious user may attempt an especially long string. This defect can be fixed by using the `string` class rather than c-strings.

```
string getText(const char * prompt)
{
    cout << prompt;
    string input;           // notice that we do not specify the size here!
    cin >> input;           // the string object "input" will grow to accommodate
                           // as much text as the user provides
}
```

Sam's Corner



The `string` class is able to handle any-sized input because of the way it stores data in its buffer. Initially, the buffer size of text is small. As the user enters more data into the buffer, the `string` object checks to make sure the buffer size is not exceeded. When it is exceeded, a new buffer twice the size is allocated and the old data is copied into the new data. This process is continued as long as more data is entered into the buffer. Thus, as long as there is enough memory in the computer, it is impossible to exceed the buffer size of a `string` object.

Unexpected input

What happens when the user enters a non-digit into a variable that expects numbers? Consider, for example the following code:

```
void getCoordinates(int & row, int & col)
{
    char colLetter;

    do
    {
        // prompt with instructions
        cout << "Please specify the coordinates: ";
        cin >> colLetter >> row;                                // ERROR! User could enter a letter
                                                               //      for the row variable

        while (colLetter < 'A' || colLetter > 'C' ||
               row < 1 || row > 3);

        // convert to letter to a number
        col = colLetter - 'A';

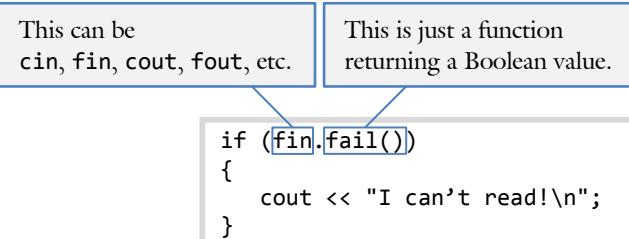
        // convert from 1's based to 0's based
        row--;
    }
}
```

What will be the output if the user enters a letter instead of a digit? To see an example of this, try compiling the program `/home/cs165/examples/1-0-ticTacToe.cpp` and enter “zz” as a coordinate.

The problem here is that `cin` enters an error state when a non-digit appears at the beginning of the input stream. We need to detect this state (for example when a letter is put into an integer), clear the state, and handle the error (skip the invalid data).

Detecting errors

The first step to recovering from user input errors is to detect that an error has occurred. The stream variables (such as `cin`, `cout`, `fin`, and `fout`) contain a member variable representing the error state of the stream. For a file stream, these errors could include missing file, no permission to open a file, unable to write to a file, or a host of other sources. Each of these errors sets the error variable in the stream object. We can detect these errors with the `fail()` method:



Console streams also have error member variables that can be set due to a variety of error events. One of these events occurs when a non-integer is read into an integer variable. We can detect this class of errors with:

```
if (cin.fail())
...
}
```

Clearing the error state

The final stage in handing stream errors is to clear the error state. Recall that each stream object has an error member variable. Once an error is encountered, the error member variable remains set until it is explicitly cleared. This can be accomplished through the `clear()` method:

```
cin.clear();
```

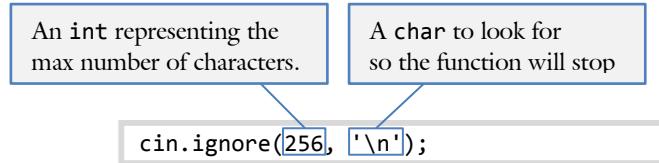
After this has been called, normal stream operations can continue and `cin.fail()` function will return `false`.

Handling errors

Once it has been determined that the input stream contains at least one erroneous character, it is necessary to skip over them so normal user input can be accepted. If, for example, the program prompts the user for an integer and the user enters “five” into the input stream, the stream will look like:

'f'	'i'	'v'	'e'	'\n'	
-----	-----	-----	-----	------	--

From here, we need to skip over the text “five” and move the insertion stream pointer to the newline character. This can be accomplished through the `ignore()` method. By default, `ignore()` will skip over one character in the input stream. You can also configure `ignore()` to skip over any number of characters until a given character is encountered in the input stream. For example, the following code will skip up to 256 characters until a newline is encountered:



Using `cin.fail()`, `cin.clear()`, and `cin.ignore()`

In order to make our `getCoordinates()` function correctly handle errors, we need to carefully incorporate error detection (`cin.fail()`), error recovery (`cin.ignore()`), and clearing the error state (`cin.clear()`). To bring it all together:

```
void getCoordinates(int & row, int & col)
{
    char colLetter;

    do
    {
        // prompt with instructions
        cout << "Please specify the coordinates: ";
        cin >> colLetter >> row;

        if (cin.fail())                                // detect we are in an error state
        {
            cin.clear();                             // clear the error state
            cin.ignore(256, '\n');                  // ignore the rest of the characters
        }
    }
    while (colLetter < 'A' || colLetter > 'C' ||
          row < 1 || row > 3);

    // convert to a letter to a number
    col = colLetter - 'A';

    // convert from 1's based to 0's based
    row--;
}
```

Internal errors

When writing a program, we often make a ton of assumptions. We assume that a function was able to perform its task correctly; we assume the parameters in a function are set up correctly; and we assume our data-structures are correctly configured. A diligent programmer would check all these assumptions to make sure his code is robust. Unfortunately, the vast majority of these checks are redundant and, to make matters worse, can be a drain on performance. A method is needed to allow a programmer to state all his assumptions, get notified when the assumptions are violated, and have the checks not influence the speed or stability of the customer's program. Assertions are designed to fill this need.

Overview of asserts

An **assert** is a check placed in a program representing an assumption the developer thinks is always true. In other words, the developer does not believe the assumption will ever be proven false and, if it does, definitely wants to be notified. An assert is expressed as a Boolean expression where the true evaluation indicates the assumption proved correct and the false assumption indicates violation of the assumption. Asserts are evaluated at run-time verifying the integrity of an assumption with each execution of the check.

An assert is a check placed in a program representing an assumption the developer thinks is always true

An assert is said to fire when, during the execution of the program, the Boolean expression evaluates to false. In most cases, the firing of an assert triggers termination of the program. Typically the assert will tell the programmer where the assert is located (program name, file name, function, and line number) as well as the assumption that was violated.

Assertions have several purposes:

- **Identify logical errors:** While writing a program, assertions can be helpful for the developer to identify errors in the program due to invalid assumptions. Though many of these can be found through more thorough investigation of the algorithm, the use of assertions can be a time saver.
- **Find special-case bugs:** Testers can help find assumption violations while testing the product when their copy of the software has embedded asserts. Typically, developers love this class of bugs because the assert will tell the developer where to start looking for the cause of the bug.
- **Highlight integration issues:** During component integration activities or when enhancements are being made, well-constructed assertions can help prevent problems and speed development time.

Assertions are not designed for:

- **User error:** The user should never see an assert fire. Asserts are designed to detect internal errors, not invalid input provided by the user.
- **File errors:** Like user-errors, a program must gracefully recover from file errors without asserts firing.

Syntax

Asserts in C++ are in the `cassert` library. You can include asserts with:

```
#include <cassert>
```

Since asserts are simply C++ statements (more precisely, they are function calls), they can be put in just about any location in the code. An example `assert` ensuring an index is not negative would be:

A Boolean expression we assume to be true.

```
assert(theChurch == true);
```

If this assert were in a file called `testimony.cpp` as part of a program called `grades` in the function called `templeInterview`, then the following output would appear if the assumption proved to be invalid because the Boolean variable was set incorrectly:

```
interview: testimony.cpp:164: bool templeInterview(std:: string&): Assertion `theChurch == true' failed.  
Aborted
```

It is important that the client never sees a build of the product containing asserts. Fortunately, it is easy to remove all the asserts in a product by defining the `NDEBUG` macro. Since asserts are defined with pre-processor directives, the `NDEBUG` macro will effectively remove all assert code from the product at compilation time. This can be achieved with the following compiler switch:

```
g++ -DNDEBUG file.cpp
```

When to use asserts

As a general rule, an assert should be added every time an assumption is made in the code. Typically, a program should consist of about 50% asserts. Common places to put asserts include: at the beginning of functions, before a function is called, and after a function is called.

Beginning of a function

Asserts should be placed at the beginning of a function to verify that the passed parameters are valid. If the assert fires, the problem is not to be found in the function, but rather in the caller. This is very useful information for debugging a program. For example, the `convertSymbol()` function in the Tic-Tac-Toe function will convert '.' into ' ' for display purposes. The code with an assert would be:

```
char convertSymbol(char letter)  
{  
    assert(letter == '.' || letter == 'O' || letter == 'X');  
    return (letter == '.') ? ' ' : letter;  
}
```

End of a function

Asserts should be placed at the end of a function to verify that the data passed back to the caller is valid. If the assert fires, it is not the caller's fault but the function's fault. For example, the `editSquare()` function needs place an X or O in a square. We can add an assert to verify this:

```
void editSquare(char board[][3], bool & xTurn)
{
    int row;
    int col;

    // get the coordinates
    getCoordinates(row, col);
    if (board[row][col] != '.')
        return;

    // change the state of the board
    board[row][col] = (xTurn ? 'X' : 'O');
    xTurn = !xTurn;

    assert(board[row][col] == 'X' || board[row][col] == 'O');
}
```

Note that the `assert` in the above example may seem redundant. After all, we just set `board[row][col]` to a valid value! However, what would happen if the function was changed by another programmer who does not fully understand the `board[][]` data-structure? This `assert` is not meant to catch bugs that currently exist in the function, but rather bugs that may be introduced in the future.

After a function is called

When a function is called, the caller needs an assurance that the provided data was as expected. It is therefore common to the return values of a function. Back to the `editSquare()` function:

```
void editSquare(char board[][3], bool & xTurn)
{
    int row;
    int col;

    // get the coordinates
    getCoordinates(row, col);
    assert(row >= 0 && row <= 2);
    assert(col >= 0 && col <= 2);
    if (board[row][col] != '.')
        return;

    // change the state of the board
    board[row][col] = (xTurn ? 'X' : 'O');
    xTurn = !xTurn;

    assert(board[row][col] == 'X' || board[row][col] == 'O');
}
```

Other instances

There are many other instances that an `assert` may come in handy. This includes any time an assumption is made in a program. Common examples are: valid pointers (Assume that a passed pointer is not `NULL` before dereferencing it), valid indices (assume that the index to an array is within an acceptable range), and well-formed data (assume that the data (say our Tic-Tac-Toe board) is in the expected format). A programmer should never hesitate to add an `assert`. Even situations where the assumption “could not possibly be violated” can prove to be incorrect.

Example 1.1 – Get Index

This program will demonstrate how to fully integrate asserts and user error handling into a function that prompts the user for an index.

Write a program to prompt the user for an index and display the results. The program should function normally regardless of the input the user enters.

```
Please enter the index. The acceptable range is 1 <= index <= 10.  
> 0  
ERROR: value is outside the accepted range  
> five  
ERROR: non-digit entered  
> 5  
The user's index is: 5
```

```
int getIndex(int min, int max)  
{  
    // before we do anything, validate the input.  
    assert(min <= max);  
    bool done = false;  
    int index = min - 1; // some invalid state.  
  
    // instructions  
    cout << "Please enter the index. The acceptable range is "  
        << min << " <= index <= " << max << ".\n";  
  
    do  
    {  
        // we should be all clear at this point  
        assert(cin.good());  
        cout << "> ";  
        cin >> index;  
  
        // check for a value that is not an integer  
        if (cin.fail())  
        {  
            cout << "ERROR: non-digit entered\n";  
            cin.clear(); // clear the error state  
            cin.ignore(256, '\n'); // ignore all the characters in the buffer  
        }  
  
        // check we are within range  
        else if (index < min || index > max)  
            cout << "ERROR: value is outside the accepted range\n";  
        else  
            done = true;  
    }  
    while (!done);  
  
    // ensure we are good before we even think of leaving  
    assert(index >= min && index <= max);  
    return index;  
}
```

The complete solution is available at [1-1-getIndex.html](#) or:

```
/home/cs165/examples/1-1-getIndex.cpp
```



See Also

Example 1.1 – Tic-Tac-Toe

Demo

This program will demonstrate how to fully integrate asserts and user error handling into project. Specifically, the Tic-Tac-Toe project from Chapter 1.0

All three types of errors are handled in Tic-Tac-Toe:

File Errors

Please see `readBoard()`. Observe how failure-to-open failures are handled as well as invalid data in the board. All the conditions of the Error Handling section of the design document are checked:

Error	Condition	Handling
Unexpected token	Token in file that is not X, 0, or .	Error message and stop loading file
Too few tokens	Reach EOF before 9 tokens are read	Error message and stop loading file
Impossible game	There should be the same number of Xs and Os or one more X than O	Error message and stop loading file

Solution

User Errors

Users errors are checked in the `interact()` function for the times when the user specified an invalid option, `getCoordinates()` when the user is prompted to change the game, and in the file functions when the user specifies a filename. Thus all the conditions of the Error Handling section of the design document are checked.

Error	Condition	Handling
Invalid option	User input something other than r,s, or q	Re-prompt and error message
Invalid coordinate	Coordinate specification that is not [a..c][1..3]	Re-prompt
Invalid filename	Unable to open file with specified filename	Re-prompt and error message

Internal Errors

Seven asserts are placed in the project to ensure the filename is correctly formed, the row & column variables are valid, and that the board itself has the expected tokens in it. Thus all the conditions of the Error Handling section are checked:

Error	Condition	Handling
Malformed coordinate	<code>coord.x > 2</code> or <code>coord.x < 0</code> or <code>coord.y > 2</code> or <code>coord.y < 0</code>	Assert
Malformed board	Any value in the board that is not X, 0, or .	Assert

See Also

The complete solution is available at [1-1-ticTacToe.html](#):

/home/cs165/examples/1-1-ticTacToe.cpp



Review 1

List the three types of errors that frequently need to be described in the Error Handling section of the Design Document.

- _____
- _____
- _____

Please see page 39 for a hint.

Review 2

For each of the three error types listed in Review 1, how should the program handle these errors?

- _____
- _____
- _____

Please see page 39 for a hint.

Problem 3

Consider the following function, add asserts to catch as many bugs as possible:

```
bool isLeapYear(int year)
{
    if (year % 4)
        return false;

    if (year % 100)
        return true;

    if (year % 400)
        return false;

    return true;
}
```

Please see page 50 for a hint.

Problem 4

Write a function to prompt the user for his GPA. Make sure the function will always return valid data and will catch all errors. The stub function is:

```
float getGPA
{
```



```
}
```

Please see page 45 for a hint.

1.2 Exception Handling

Sam has just finished a large project and, at his professor's insistence, needs to retrofit error handling to his code. This is proving to be much more difficult than he envisioned. It is one thing to catch an error in a function, it is another thing entirely to let the caller of that function know what happened. This is forcing Sam to rethink his entire program! If only he thought about error handling from the beginning.

Objectives

By the end of this chapter, you will be able to:

- List and define the components to C++ exceptions
- Write the code necessary to throw and catch an exception within a single function
- Write the code necessary to throw an exception in one function and catch it in another

Prerequisites

Before reading this chapter, please make sure you are able to:

- Be able to create and use a #define (Procedural Programming in C++, Chapter 2.1)
- Author the Error Handling section of a design document (Chapter 1.0)
- Write the code necessary to detect a file, user and internal error (Chapter 1.1)

What are exceptions and why you should care

In all but the simplest problems, programs are expected to gracefully recover from errors originating from a wide variety of sources. Some errors may come from the user such as improperly formed input, some may come from the system such as resources no longer being accessible, and some may come from the program itself when a logical error is encountered. In each of these cases, the user expects the program to continue to function normally.

There are three main ways that errors are handled in a program: error flags, error IDs, and exception handling. Of these three, exception handling is the only mechanism built into C++ specifically to facilitate the handling of errors. Exception handling affords the programmer an easy, efficient, and standard way to work with errors. Though the syntax may initially appear awkward and unusual, it serves to clearly delineate the part of a program dedicated to normal operation and the part tasked with recovering from errors.



Sue's Tips

Understanding exception handling is a required skill for all programmers. Though you will certainly encounter it many times in your career, there are other ways to handle errors. However, the more tools you have in your tool-bag of programmer tricks, the more effective you will be. This is one more thing that makes our job as programmers that much easier.

Error flags

Possibly the easiest way to propagate an error is to have a function return a `bool`: `true` means the function succeeded in performing its task and `false` means that it failed.

```
bool readBoard(const string & fileName, char board[][3], bool & xTurn);
```

This methodology is called an “error flag.” In the context of programming, a flag is a Boolean variable that contains two-state data. In many ways, this is similar to a signal flag used to communicate between ships in years past.

Back to our `readBoard()` example, consider the following code:

```
bool readBoard(const string & fileName, char board[][3], bool & xTurn)
{
    assert(fileName.length() != 0);

    // open the file
    ifstream fin(fileName.c_str());
    if (fin.fail())                                // always check for errors when
    {                                                 // opening a file
        cout << "Unable to open file "
            << fileName << " for reading.\n";
        return false;
    }

    ... code removed for brevity ...

    // determine whose turn it is by the xOver0 count
    xTurn = (numXover0 == 0 ? true : false);
    if (numXover0 != 0 && numXover0 != 1) // there must be an equal number of
    {
        cout << "Invalid board in file "      //      X's and O's or there must be
            << fileName << ".\n";                //      one more X.
        fin.close();
        return false;
    }

    // close the file
    fin.close();
    return true;
}
```

In this example, the function returns `false` if one of many conditions are met: we failed to open the file, we failed to read a number from the file, or if the board is invalid. Only if everything goes as expected does the function return `true`.

There are several problems with error flags. The first is that we only know that an error has occurred. No information is passed back to the caller indicating what the error was. Thus the caller is forced to treat all errors the same.

The second problem is that it is up to the caller to handle the error. If the caller cannot recover from the error (by re-prompting the user for a new file name in this example), then the caller will be forced to return `false` as-well. In other words, the caller must propagate the error. This means the error information must be passed down to the caller’s caller if it expected the function to succeed.

The complete solution is available at [1-2-ticTacToe-flags.html](#) or:

```
/home/cs165/examples/1-2-ticTacToe-flags.cpp
```

Error ID

An error-ID (EID) works much the same as an error flag with one important difference: EIDs are integers and are able to differentiate between a wide number of errors. Consider the following `#defines`.

```
#define EID_NONE      0
#define EID_NO_FILE   1
#define EID_CORRUPT   2
#define EID_INVALID   3
```

With these codes, we can re-write our file reading function from the previous example:

```
int readBoard(const string & fileName, char board[][3], bool & xTurn)
{
    // open the file
    ifstream fin(fileName.c_str());
    if (fin.fail())
        return EID_NO_FILE;           // send NO-FILE error
    ... code removed for brevity ...

    if (numXover0 != 0 && numXover0 != 1)
    {
        fin.close();                // send INVALID error because
        return EID_INVALID;         //      the game does not make sense
    }

    // close the file
    fin.close();
    return EID_NONE;               // success!
}
```

The complete solution is available at [1-2-ticTacToe-eid.html](#) or:

```
/home/cs165/examples/1-2-ticTacToe-eid.cpp
```

Observe how much more detailed the error message is with this example than with the flag example. This detail also yields an increase in complexity: both the caller and the callee must share the same EID vocabulary. That complexity is present in the function `interact()` which calls `readBoard()`:

```
switch(readBoard(fileName, board, xTurn))
{
    case EID_NO_FILE:
        cout << "Error: Unable to open file "
              << fileName << ".\n";
        break;
    case EID_CORRUPT:
        cout << "Error: The file "
              << fileName << " contains unexpected data.\n";
        clearBoard(board, xTurn);
        break;
    case EID_INVALID:
        cout << "Error: The file "
              << fileName << " does not contain a valid game.\n";
        clearBoard(board, xTurn);
}
```

In other words, an additional EID used by the callee `readBoard()` must be understood by `interact()`. This is a form of control coupling. A second problem with the EID method is that, like with the error flags, it is the responsibility of the caller to either handle the error or propagate it. This can be a source of programming errors if not done correctly.

Exception handling

With the inherit shortcomings of the error flag method and the EID method of error handling, C++ developed a specialized mechanism to work with errors: exceptions. An **exception** is mechanism to alter the flow of a program in the case of a problem to special-purpose error-handling code. All C++ exceptions have the same three components: the `throw`, the `try`, and the `catch`.

An exception is a special error-handling mechanism built into the C++ language

Throw

When an error is detected, an exception is thrown. If, for example, I wish to handle the case where an array variable consists of a `NULL` pointer, I could throw a c-string indicating the error state:

```
{
    // check if the array is invalid
    if (array == NULL)
        throw "Invalid pointer address";           // the exception is thrown from here

    // use the array variable now that it is safe
    array[0] = 42;
}
```

In this example, the c-string "Invalid pointer address" will be thrown if the pointer `array` consists of the `NULL` address. This means that the code after the `throw` statement (`array[0] = 42;` in this case) will not be executed. The syntax of the `throw` statement is:

`throw ("down");`

Expression to be thrown.
Must match the data type in the `catch` statement.

It is possible to throw any data type you like, though it is most common to throw a c-string, a `string` object, or an integer. It is also possible to throw more than one type of exception in a given function. For example, one statement could throw an `int` and another a `float`.

Try

The second part of exception handling is the `try` statement. The purpose of the `try` statement is to delineate the part of the code where an exception could be thrown. Since the `try` statement combined with the curly braces delineate a block of code wherein an exception may be thrown, we most commonly call the region a try-block. Back to our example of the `NULL` pointer detection, the code may be:

```
try                                // the try block indicates that an
{                                     //      exception may be thrown here
    // check if the array is invalid
    if (array == NULL)
        throw "Invalid pointer address";

    // use the array variable now that it is safe
    array[0] = 42;
}
```

Sam's Corner



Exception handling code is heavily optimized for the non-error case. This means that you pay very little performance penalty for using exceptions in the case when it is not thrown. It also means that a thrown exception is rather expensive. C++ exceptions are designed for exceptional circumstances, not main-stream ones. They should only be used in the case of an unusual error.

Catch

The final part of the exception handling mechanism is the `catch` statement. The purpose of `catch` is to receive the error message sent by `throw`. In other words, execution of the program jumps from `throw` to `catch` in the case where an exception is thrown. Thus, in the following example, if `array == NULL`, the statement `array[0] = 42;` will be skipped when the "Invalid pointer address" exception is thrown.

```
{
    try
    {
        // check if the array is invalid
        if (array == NULL)
            throw "Invalid pointer address";

        // use the array variable now that it is safe
        array[0] = 42;
    }
    catch (const char * message)           // in the case of an error, the
    {                                     //     program will jump from the
        cout << "Error: " << message << endl; //     throw to this code
    }
}
```

The data type in the `catch` statement must match the data type in the `throw` statement for the thrown exception to be caught. In the above example, a c-string constant is thrown and the variable `message` in the `catch` statement is of the same data type.

In the case when more than one data type is thrown, multiple `catch` statements may be required. Consider the following code:

```
{
    try
    {
        if (text == NULL)
            throw string("NULL pointer in text variable"); // throw a string object
        if (text[0] == '\0')
            throw 0;                                         // throw an integer
        cout << text << endl;
    }
    catch (const string message)                      // catch the string
    {
        cout << message << endl;
    }
    catch (int value)                                // catch the integer
    {
        cout << "text contains an empty string!\n";
    }
}
```

With two types of exceptions thrown (a `string` object and an integer), two catches are required. There is also a special type of catch statement akin to the `default` statement in a SWITCH-CASE. This `catch` statement will catch any type exception, regardless of the data type. This is called the `catch-all`:

```
catch (...)                                // catch-all
{
    cout << "Some exception was thrown, I just don't know which one!\n";
}
```

Note that exception handling may occur within a function or between functions. Although the mechanisms are similar, there are subtle differences between these cases.

Exceptions within a function

One kind of exception handling occurs when an exception is thrown and caught in the same function. Back to the `readBoard()` example used earlier:

```

void readBoard(const string & fileName, char board[][3], bool & xTurn)
{
    assert(fileName.length() != 0);

    try
    {
        // open the file
        ifstream fin(fileName.c_str());
        if (fin.fail())                                // throw if we fail to open
            throw "Unable to open file";               //      the file for any reason

        // read the contents of the file
        int numXover0 = 0;
        for (int row = 0; row < 3; row++)
            for (int col = 0; col < 3; col++)
            {
                // read the item from the board
                fin >> board[row][col];

                // check for failure.
                if (fin.fail())                                // throw if we fail to read data
                    throw "Error reading file ";             //      from the file

                if (board[row][col] == 'X')
                    numXover0++;
                else if (board[row][col] == 'O')
                    numXover0--;
                else if (board[row][col] != '.')              // throw if the symbol we read
                    throw "Unexpected symbol";              //      is not a X O or .
            }

        // determine whose turn it is by the xOver0 count
        xTurn = (numXover0 == 0 ? true : false);
        if (numXover0 != 0 && numXover0 != 1)          // throw if the board does not
            throw "Invalid board";                     //      make sense
    }
    catch (const char * message)                      // all errors are sent here
    {                                                 //      so we only need to
        cout << "Error: " << message                 //      construct the error
        << " in file " << fileName                  //      message once
        << ".\n";
    }

    // close the file
    fin.close();                                     // this is called regardless of
                                                    //      whether there was an error
}

```

In this example, no code in the `catch` block will get executed if an exception is not thrown. However, if one of the four exceptions is thrown, control of the program will be redirected to the `catch` block immediately.

The complete solution is available at [1-2-ticTacToe-exception1.html](#) or:

```
/home/cs165/examples/1-2-ticTacToe-exception1.cpp
```

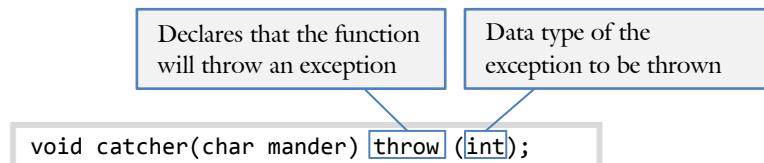
Exceptions between functions

It is common to want to throw an exception in one function and catch it in another. To accomplish this, three things are needed. First, the function throwing the exception needs to advertise the types of exceptions that are thrown in a **throw list**. Thus the prototype of `readBoard()` becomes:

```
void readBoard(const string & fileName,
               char board[][3],
               bool & xTurn) throw (const char *);
```

A throw list is a list of all the possible un-caught exceptions a function may throw

The syntax for a function prototype including the throw list is:



The second part of the syntax is to throw the exception in the function itself. Because `readBoard()` will not be catching its own exception, there is no try-block and there is no catch-block.

```
void readBoard(const string & fileName,
               char board[][3],
               bool & xTurn) throw (const char *)
{
    // open the file
    ifstream fin(fileName.c_str());
    if (fin.fail())                                // throw if we fail to open
        throw "Unable to open file";                //     the file for any reason
    ... code removed for brevity ...
    if (numXover0 != 0 && numXover0 != 1)          // throw if the board does not
        throw "Invalid board";                      //     make sense

    // close the file
    fin.close();                                     // this is called regardless of
}
```

The final part of the syntax is to catch the exception in the caller. In this case, the caller is `interact()`:

```
... code removed for brevity ...
case 'r':
    fileName = getText("What file would you like to read the board from? ");
    if (fileName.length() != 0)
    {
        try
        {
            readBoard(fileName, board, xTurn);
            displayBoard(board);
        }
        catch (const char * message)           // catch the exception
        {                                     //     thrown in the
            cout << "Error: " << message      //     readBoard() function
            << " in file " << fileName
            << ".\n";
        }
    }
    break;
... code removed for brevity ...
```

The complete solution is available at [1-2-ticTacToe-exception2.html](#) or:

```
/home/cs165/examples/1-2-ticTacToe-exception2.cpp
```

Example 1.2 – Multiple exceptions

Demo The purpose of this demo is three-fold. First, it is meant to demonstrate how multiple exception types can be thrown in a single function. Next, the exceptions are to be caught in a different function than they are thrown from. Finally, a catch-all will demonstrate how to catch un-expected exceptions.

The function throwing the exceptions:

```
void exceptionalFunction() throw (int, float, string, char)
{
    Switch (prompt())
    {
        case 1:
            throw 0;
        case 2:
            throw float(0.0);
        case 3:
            throw string("zero");
        case 4:
            break;
        default:
            throw '0';
    }
    cout << "End of the exceptional function\n";
}
```

Next, `main()` which will catch the exceptions:

```
int main()
{
    try
    {
        exceptionalFunction();
        cout << "No exception was thrown\n";
    }
    catch (int integer)                                // for throw 0;
    {
        cout << "An integer was thrown!\n";
    }
    catch (float floatingPoint)                      // for throw float(0.0);
    {
        cout << "A floating point number was thrown!\n";
    }
    catch (string text)                             // for throw string("zero");
    {
        cout << "Text was thrown!\n";
    }
    catch (...)
    {
        cout << "Error! Unexpected exception was thrown!\n";
    }
    return 0;
}
```

The complete solution is available at [1-2-multipleThrows.html](#) or:

/home/cs165/examples/1-2-multipleThrows.cpp



Review 1

Write an assert to verify that the piece in a chess board (`board[row][col]`) is one of the following:
 {K, Q, R, B, N, P, space}

Please see page 50 for a hint.

Review 2

Modify the above code from Review 1 so the entire board is checked to make sure it is one of the valid pieces.

Please see page 50 for a hint.

Problem 3

What exceptions are thrown from the following function?

```
double funky(string text) throw (int, bool)
```

Please see page 61 for a hint.

Problem 4

What is the most correct function definition for the following code?

```
{
    cout << "Which item would you like to edit? "
    int index;
    cin >> index;

    if (cin.error())
        throw "Non-digit entered";
    if (index < 0)
        throw index;
    return index;
}
```

Please see page 61 for a hint.

Problem 5

What is the output when the user enters the value 0?

```

try
{
    int value;
    cin >> value;

    if (value < 0)
        throw "Negative!";
    if (value == 0)
        throw 0;
}
catch (char *message)
{
    cout << message << endl;
}
catch (...)
{
    cout << "Unhandled exception!";
}
catch (int num)
{
    cout << num;
}

```

Please see page 59 for a hint.

Problem 6

Up to this point, we call the `new` function to allocate memory with the following code:

```
double * list = new(no_throw) double[10];
```

It turns out that `new` throws the `bad_alloc` exception. Write some code to allocate 10 `doubles` and display an error message if the allocation failed by catching the `bad_alloc` exception.

```

{
    double * list = new double[10];
}
```

Please see page 60 for a hint.

Problem 7

Match the scenario on the left with the best error handling technique on the right

Scenario

- User responds to a prompt with data that is outside the expected range
- The file you are attempting to read from does not exist
- The code depends on a pointer being initialized, but you want to check it just to make sure
- The user enters a file-name that has illegal characters
- In a grades program, the user entered -100% as an earned grade

Technique

- Throw an exception
- Use an assert
- While loop and try again
- Display an error message and re-prompt
- Exit the program immediately

Please see page 39 for a hint.

1.3 Structures

Sue is working on a program to play Sudoku and is getting tired of passing row and column parameters to almost every function. In her mind the two variables should be one because they represent a single concept. However, there is no “Coordinate” data type in C++. Surely there must be a better way!

Objectives

By the end of this chapter, you will be able to:

- List and define the following terms: structure, member variable, and member value.
- Create a structure to represent a given concept.
- Declare, initialize, pass as a parameter, and dereference a structure variable.
- See the value in using structures to simplify a wide range of programming problems.

Prerequisites

Before reading this chapter, please make sure you are able to:

- Choose the most appropriate data type for a given programming problem
(Procedural Programming in C++, Chapter 1.2)
- Pass a variable to a function by-value, by-reference, and by-pointer
(Procedural Programming in C++, Chapter 1.4, 3.3)
- Define cohesion and use it to better define functions
(Procedural Programming in C++, Chapter 2.0)
- Declare, initialize, pass as a parameter, and dereference an array variable
(Procedural Programming in C++, Chapter 3.0)

What is a structure and why you should care

A **structure** is a mechanism to create a custom data type specifically tailored for the needs of a given program. In other words, we do not have to be limited to the built-in data types provided by the C++ language (such as `int`, `float`, `char`, `bool`, `double`, etc.).

There are several reasons why structures are an invaluable tool in your programming toolbox. First, there are a collection of problems that can only be addressed with structures. Thus the topic of structures completes your training in procedural programming techniques. Second, a big part of programming, especially using the Object-Oriented design methodology, is to create custom data types designed especially for a given application. Our first baby-step into this space is structures. Every single program we write for the remainder of the semester will build off of this chapter.

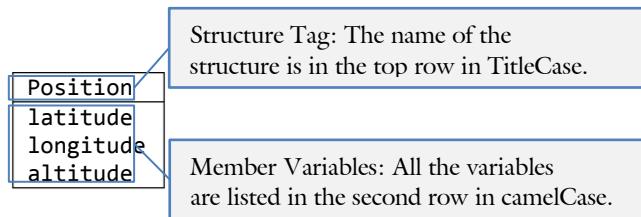
Sue's Tips



The structure is the second *aggregate* data type. This means that it consists of more than one primitive (or built-in) data types. Our first aggregate data type was an array, consisting of a collection of items with the same base type and where each instance is accessed with an index. Structures are similar to arrays in many ways but with two important exceptions: elements in a structure can be of different data types where arrays must be the same, and elements in a structure are referenced by name whereas arrays are referenced by index.

Designing with structures

Perhaps the easiest way to think of structures is as a “bucket of variables.” There are two parts to this model: the variables in the bucket and the name of the bucket itself. When designing a structure, we use a simple two-row table where the first row corresponds to the name of the structure and the second consists of the list of variables that are to be contained:



We call this simple design tool **UML**, an acronym meaning “Unified Markup Language.” UML is actually a suite of five tools, of which this the class diagram presented here is only one. (Booch, 1994) (Rumbaugh, Blaha, Premerlani, Eddy, & Lorensen, 1991)

The first part of the design process is to answer the question “What is this structure for?” Just as when we design functions, the concept of cohesion comes into play. If we can answer this question well, if the member variables together describe a single cohesive concept, and if a simple name completely encapsulates the concept the member variables represent, then we will have an easy time designing and using our new structure.

The second part of the design process is to identify the member variables necessary to characterize the concept represented with the structure tag. Again, the concept of cohesion plays an important role. All the member variables should work together to describe one concept. Resist the temptation to throw in extra or unnecessary variables. These will just make the code more difficult to maintain and use.

The following are examples of structures you may want to create:

Account	Complex	Name	Address
centsBalance centsMinimumBalance interestRate accountNumber	real imaginary	first middleInitial last title	street city state zip

An example of a bad structure definition exhibiting weak cohesion might be:

Stuff
counter
buffer
fileName



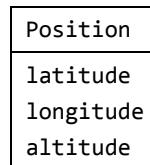
Sue's Tips

UML is an easy tool to quickly visualize and design structures. It is also the primary tool we use in the Data-structures section of the design document. Take a moment and sketch out the structures that may come in handy for the Chess project presented at the end of this unit.

Defining a structure

The first step in working with a structure is to define it. A structure definition is a template for a variable; it is not a variable itself. The integer date-type `int`, for example, is not a variable. Only when you declare a variable with `int count`; then the data type gets realized as a variable.

The structure definition has several parts: the “`struct`” keyword indicating we are defining a structure, the name of the structure (“`Position`” in this case), the curly braces `{}`, and a list of variable declarations comprising the member variables of the structure. Consider the following UML class diagram for a `Position` structure:

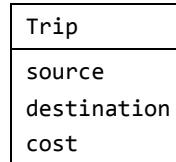


The definition for `Position` is:

```
struct Position           // corresponds to the structure tag
{
    float latitude;      // corresponds to the member variable latitude
    float longitude;
    int altitude;         // note member variables can be of different data types
};                         // do not forget the semi-colon!
```

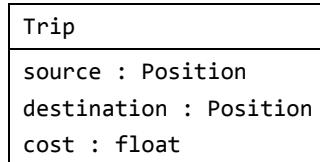
With the **UML** description of a structure, it should be a very straight-forward process to create the structure definition. The only thing that is needed is to provide the data types and remember the syntax. Keep in mind that this only defines the structure. It is a blue-print telling the compiler what a `Position` variable will look like. You can use this definition to create as many `Position` variables as you like, each of which will have its own copy of the data.

Note that it is possible to create one structure out of another. This is called a **nested structure**. If, for example, I wished to create a structure capturing the notion of the leg of a journey, I would want to use my newly created `Position` structure. The UML for the `Trip` structure would be:



Sue's Tips

You can also specify data types in your UML if you feel it adds clarity to your design. Our `Trip` structure above could also be represented as:



Based on this UML, the definition for `Leg` is:

```
struct Trip           // the structure tag should be TitleCased
{
    Position source;      // we don't have to use built-in data types for
    Position destination; //   member variables; we can use structures!
    float cost;          // it is possible to mix built-in data types
};                      // with structures
```

Declaring a structure

To turn the template of a data type into a variable, you need to declare the variable. This is true with a built-in data type as it is with a structure. Therefore, the syntax of declaring a variable of type `Position` is as one would expect:

```
{
    Position positionRexburg;      // a variable of type Position. Make sure that
};                          //     Position is defined before here
```

It is possible to initialize a variable at declaration time (`int count = 0;`). It is similarly possible to initialize a structure variable at declaration time. The main difference is that three member variables need to be initialized at once:

```
{
    Position positionRexburg = {43.82937, -111.7828, 4865}; // define Position before this line of code
    {                                // use curly braces to surround the values
        43.82937,                  // latitude member variable
        -111.7828,                 // longitude
        4865                        // altitude
    };                                // don't forget the semi-colon!
}
```

You can copy the contents of one structure onto another with the assignment operator `=`: Remember, every time you declare a structure, you are creating a brand new copy or instance. This means you can change the values of one instance without affecting the others.

```
{
    Position rexburg = {43.8231, -111.7828, 4865};
    Position disney = {33.8121, -117.9190, 141};
    Position copyRexburg;

    copyRexburg = rexburg;           // this is legal! You can copy the contents of one
                                    // one struct into another any time you want.

    cout << "Difference in altitude between Rexburg and Disney: "
        << copyRexburg.altitude - disney.altitude
        << endl;
}
```

Notice how the assignment operator copies all the data from the curly braces into the new structure variable. It turns out that you can perform a structure copy any time you want.

Sam's Corner



While it is legal to copy one structure onto another, it is also expensive. If the structure has any member variables, a large amount of data may be copied. This means a seemingly innocent assignment operator may be the source of many assembly instructions taking a lot of CPU time. In other words, use this construct sparingly.

It is easy and convenient to copy structures with the assignment operator. Remember that we cannot do the same thing quite as easily with an array:

```
{
    int array1[] = {4, 6, 8, 1};
    int array2[4];
    array1 = array2;                                // ERROR: this does not work! You need to use a
                                                    //      loop to copy the contents of an array
}
```

Referencing member variables

With an array, it is possible to access individual elements with the square bracket operator `[]`. The 4th item in an array, for example, can be accessed with `array[3]`. Structures do not access member variables by index but rather by name. This is accomplished with the **dot operator** (also known as the member access operator). Back to our `positionRexburg` variable declared previously, we can declare the `Position` variable and initialize the member variables separately:

```
{
    Position positionRexburg;                      // declared but not initialized
                                                    // use the dot operator to access the
    positionRexburg.altitude = 4865;                //      member variables directly
    cout << positionRexburg.altitude;
}
```

This also works if a member variable is a structure itself. Recall our `Trip` structure that had a source member variable of type `Position`.

Trip
Source
destination
cost

This can be initialized in much the same way:

```
{
    Trip romeToVenice;

    romeToVenice.cost = 25.32;                      // the cost member variable is a float
    romeToVenice.source.latitude = 41.9000;           // source is a Position which has three
    romeToVenice.source.longitude = 12.5000;          //      member variables. Each of these
    romeToVenice.source.altitude = 456;               //      can be accessed by name with the
}                                              //      dot operator .
```

Sam's Corner



With arrays, it is easy to write a FOR loop to iterate through all the elements in the collection.

This is impossible with a structure! Because member variables are referenced by name rather than by index, the only way to access all the member variables is to laboriously enumerate them individually in the code.

A structure is called an “early-binding” mechanism. The compiler has all the information necessary to verify that the referencing of a member variable is done correctly. You cannot iterate off the end of the buffer in a structure as you can with an array!

As with any data type, we can also make arrays of structures:

```
{  
    Position manyPositions[10]; // an array of 10 positions  
  
    // initialize the 5th item  
    manyPositions[4].latitude = 43.8293; // access the member variables after  
    manyPositions[4].longitude = -111.7928; // the [] with the dot operator  
    manyPositions[4].altitude = 4865;  
  
    // copy the 5th into the 1st  
    manyPositions[0] = manyPositions[4]; // copy all the data  
}
```

Pointers and structures

It is possible to create a pointer to a structure in much the same way we can create a pointer to any other data type. Unfortunately the syntax of accessing member variables gets a bit complex. We need to first dereference the structure pointer with the * operator before we can access the member variables with the dot operator. The problem arises from the fact that the dot operator comes before the dereference operator in the order of operations. To make our intentions clear, we need to use parentheses. Consider the following example with the `Position` structure:

```
{  
    Position pos; // pos is a variable of type Position  
    Position * pPos = &pos; // pPos is a pointer to a Position  
  
    pos.altitude = 4865; // access the altitude member variable with the dot  
    (*pPos).altitude = 5260; // we need the () operator to override the order of  
    // operations. This is ugly  
}
```

To avoid this nasty syntax involving the dot, parentheses, and dereference operator, a more convenient syntax was developed: the **arrow operator** `->`. It allows us to access member variables from a pointer to a structure without the unpleasant syntax.

```
{  
    Position pos;  
    Position * pPos = &pos;  
  
    pos.altitude = 4865; // dot operator for a structure variable  
    pPos->altitude = 5260; // arrow operator for a pointer to a structure variable  
}
```

Why do you suppose they developed a special syntax for a pointer to a structure? How common could it be? The answer is: very common.

Structures can be large, taking many bytes of memory. Therefore, it is expensive to copy them. Just like with arrays, programmers go through great lengths to ensure that there is only one copy of a structure in memory. Imagine how slow a computer would be if all the data was copied every time an array or structure got passed to a function. To prevent this inefficiency, it is quite common to pass structures as pointers between functions. This way, the callee will have access to the member variables without the callee having to make a copy of the structure. As you can see, if we are commonly passing pointers to structures between functions, the arrow operator would come in handy to access the member variables.

Passing a structure to a function

As previously discussed, making copies of structures can be an expensive practice. How then do we pass structures to functions as parameters? The answer is: the same as any other data type with one exception. We almost never pass a structure by-value.

```
void function(Position byValue,           // avoid passing a structure by-value
              Position & byReference,    // very common way to pass a structure
              Position * byPointer)     // less common but still very useful
{
    byValue.altitude      = 5260;        // use the dot operator here
    byReference.altitude = 5260;        // also use the dot operator
    byPointer->altitude  = 5260;        // pointer necessitates the use of the arrow
}
```

When passing a structure variable to a parameter and the function is not to change the variable, then pass the structure by-reference as a constant:

```
void display(const Position & pos)           // pass a constant struct by reference
{
    cout.setf(ios::fixed | ios::showpoint);
    cout.precision(4);
    cout << "(" << pos.latitude             // reference each member variable with the
         << ", " << pos.longitude          //      dot operator. Since the variable is
         << ", " << pos.altitude           //      a const, there is no chance of
         << ")";                         //      accidentally changing the value of pos
}
```

When passing a structure variable and the intention is to change the variable, then pass the structure by-reference:

```
void prompt(Position & pos)           // pass a struct by reference
{
    cout << "Please enter the position as latitude longitude altitude: ";
    cin  >> pos.latitude
         >> pos.longitude
         >> pos.altitude;
}
```

Sam's Corner



The C++ language was derived from C which had many of the same constructs. One important difference is that there was no pass-by-reference in the C language. If you wanted the caller to change the variable, you had to use pass-by-pointer. This means that an entire generation of programmers were trained to pass structures by pointer instead of by reference. The above function prototypes would thus be:

```
void display(const Position * pPos);
void prompt(Position * pPos);
```

Of course the body of the functions would access the member variables with the arrow operator.

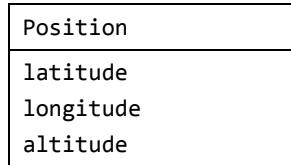
Example 1.3 – Position

Demo

This program will demonstrate how to define a structure, declare a variable, reference the member variables, and pass a structure as a parameter.

Problem

Create a structure representing a position on the globe:



Solution

```
int main()
{
    // First we will declare a simple Position variable
    Position pos1;

    // Next we will fill it with the prompt function
    prompt(pos1);

    // Now to display the results
    cout << "Original value of pos1: ";
    display(pos1);

    // We will initialize a position as we declare it this time
    Position pos2 =
    {
        43.82937,                      // use curly braces to surround the values
        -111.7828,                     // latitude member variable
        4865                            // longitude
    };

    // we can perform a structure-copy with the assignment operator
    pos1 = pos2;

    // display the results to verify
    cout << "New value of pos1: ";
    display(pos1);

    // one final prompt
    promptPointer(&pos1);

    return 0;
}
```

See Also

The complete solution is available at [1-3-position.html](#) or:

/home/cs165/examples/1-3-position.cpp



Example 1.3 – Address

This program will demonstrate how to define a structure, declare a variable, reference the member variables, and pass a structure as a parameter.

Write a program to prompt a user for his address and display it back to him.



The declaration of the structure is:

```
struct Address           // structure tag is TitleCased
{
    string street;      // the street can be long, use the string class
    string city;         // same is true with the city
    char state[3];       // states use a two-letter abbreviation, plus \0
    int zip;             // the zip is five digit integer
};
```

To initialize the structure, we need a prompt function:

```
void prompt(Address & input)          // we are changing the input so it is
{                                         // pass-by-reference
    cout << "What is your street address? ";
    getline(cin, input.street);          // use the dot operator to access the
    cout << "What is your city? ";
    getline(cin, input.city);            //     member variables here
    cout << "What is your state and zip? ";
    cin  >> input.state >> input.zip;
}
```

Finally, we display the Address by passing it as a constant by reference.

```
void display(const Address & output)    // we are not changing the output
{                                         // so the parameter is a const
    cout << output.street << endl        // we access the member variables
    << output.city   << ", "
    << output.state << " "
    << output.zip << endl;                 //     with the dot operator. Because
                                            //     the variable is a const, we
                                            //     cannot change the values
}
```

The complete solution is available at [1-3-address.html](#) or:

```
/home/cs165/examples/1-3-address.cpp
```



See Also

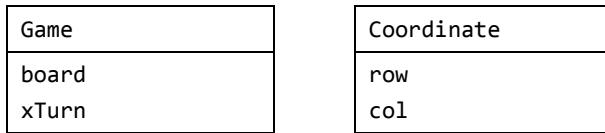
Example 1.3 – Tic-Tac-Toe

Demo

This program will demonstrate how to integrate structures into a large project. In this case, the project is Tic-Tac-Toe.

Problem

Integrate two structures into our Tic-Tac-Toe example from the previous chapters.



Solution

The declaration of the Game structure is:

```
struct Game
{
    char board[][3];                                // the same 3x3 array as before
    bool xTurn;                                     // also whose turn it is
};
```

With this change, `main()` is greatly streamlined:

```
int main()
{
    // set up the board
    Game game;                                         // only one variable needed, not 2
    clearBoard(game);                                  // it is also much easier to
                                                       //     pass as a parameter

    // let the users play the game
    interact(game);

    return 0;
}
```

Though many functions will change with this instruction of the `Game` structure, most will follow the pattern of the `clearBoard()` function.

```
void clearBoard(Game & game)                      // pass the structure by reference
{
    // remove all the markings off the board
    for (int row = 0; row < 3; row++)                // the loops are the same with a
        for (int col = 0; col < 3; col++)            //     struct as they were before
            game.board[row][col] = '.';                 // note the dot operator

    // set the turn to X's
    game.xTurn = true;                               // again, the dot operator
}
```

See Also

The complete solution is available at [1-3-ticTacToe.html](#) or:

```
/home/cs165/examples/1-3-ticTacToe.cpp
```



Problem 1

Given the following code:

```
{  
    struct Position  
    {  
        double latitude;  
        double longitude;  
        double altitude;  
    };  
  
    Position myHouse;  
  
    myHouse.latitude = 43.83388;  
    myHouse.longitude = -111.80500;  
    myHouse.altitude = 5260.00;  
}
```

List all the member variables.

Please see page 65 for a hint.

Problem 2

Write the code for a structure definition matching the following UML:

Name
firstName
lastName

Please see page 67 for a hint.

Problem 3

Given the following structure definition:

```
struct Position  
{  
    double latitude;  
    double longitude;  
    int altitude;  
};
```

How would you declare a variable of type `Position`?

Please see page 68 for a hint.

Problem 4

Given the following code:

```
struct Position
{
    double latitude;
    double longitude;
    int altitude;
};

Position myHouse;
```

How do you initialize the member variable `altitude`?

Please see page 69 for a hint.

Problem 5

What is the output?

```
{
    struct Person
    {
        int age;
        char name[100];
        float height;
    };

    Person jack;

    cout << sizeof(jack) << endl;
}
```

Please see page 68 for a hint.

Problem 6

Given the following structure definition and variable declarations:

```
struct Name
{
    char first[256];
    char middleInitial;
};

struct Family
{
    char last[256];
    Name father;
    Name mother;
};

int main()
{
    Family smithFamily;
    Name bob;

    <your code goes here>
}
```

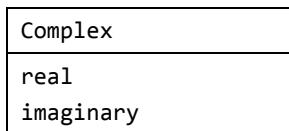
Write the code to display the contents of all the member variables. For example:

```
cout << bob.first;
```

Please see page 69 for a hint.

Problem 7

Consider the following UML:



Please do the following:

1. Create a structure matching the UML
2. Create a variable of type Complex. Name it `x` and fill it with `5.2 + 36.1i`
3. Create another variable. Name it `y` and copy `x` into `y`
4. Write an IF statement comparing the value of `x` with the value of `y`

Please see page 74 for a hint.

1.4 Separate Compilation

Sam and Sue are working on a large project with several other programmers. Immediately it is apparent to them that this will be very difficult to coordinate. With so many people editing the same file, how will they ever keep the project compiling? How will they ever keep people from editing the file at the same time? To mitigate these issues, they decide to break the project up into several smaller files.

Objectives

By the end of this chapter, you will be able to:

- Name and define the terminology of separate compilation: header files, implementation or source files, makefile, linking, and assembly.
- Break a large program in a single file into many smaller and more manageable files.
- Design a project with separate compilation in mind.
- See the value of using separate compilation to reduce the complexity of a project.

Prerequisites

Before reading this chapter, please make sure you are able to:

- Write a function prototype so functions can appear in any order in a file
(Procedural Programming in C++, Chapter 1.4)

What is separate compilation and why you should care

Separate compilation is the process of designing a program with multiple files each of which is a manageable size. Virtually every commercial software you run, be it on your phone, on your laptop, or on the web, consists of many source files. It is such a common practice in industry that most professional programmers cannot fathom working with just a single source code file.

There are many reasons why even beginner programmers should start thinking in terms of separate compilation. The first is that it drastically reduces the complexity of a program. When all the functions pertaining to a given aspect of a program are located in a single file, it becomes much easier to find the function you are looking for and easier to localize bugs.

Another benefit to separate compilation is that it facilitates code re-use. If you perfect a function that is general-purpose (such as the `getIndex()` function from Chapter 1.1), it is easy to put the function in its own file. Then, when you wish to incorporate that function in another program, it is only necessary to include the file in the project rather than copy-paste the code. In other words, large chunks of code can be reused into a new project with just a single statement. In this way, you can start building your own personal libraries that are as useful as `iostream` and `string` that you have been using since the beginning of the semester.

The final benefit of separate compilation is that it makes it much easier to work on large projects with other programmers. When a programmer is given a part of a project to work on independently, he can implement and often test his code independently from modules being developed by team members.

Modularizing files

The first step in working with multiple files is to determine the best way to break the project into manageable components. If this sounds similar to the problem of breaking a large function into a collection of smaller ones, it should. In both cases, the principles of cohesion and coupling come to play.

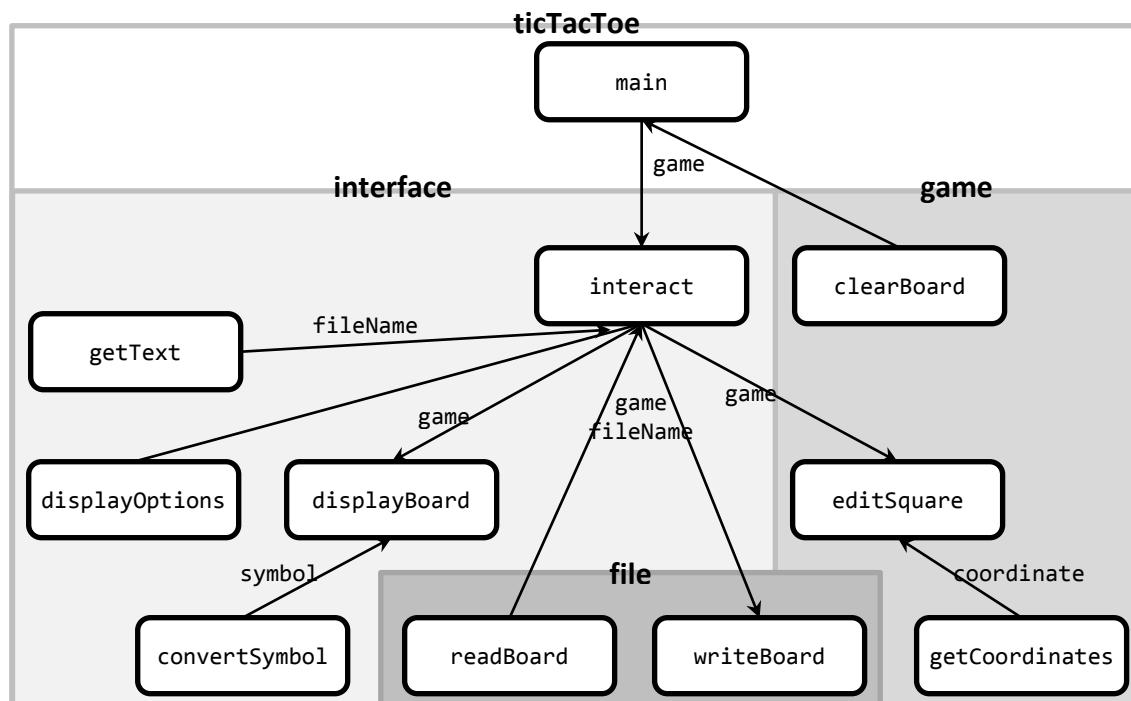
File cohesion

Cohesion, as you recall, is the quality of all the parts of a given component being directed towards a single purpose. This is true with files as it is with functions. Each file should only contain code designed to accomplish one task. Similarly, all the code for the task should be present in the single file. Consider, for example, our Tic-Tac-Toe program from previous chapters. A cohesive design would be to have one file containing all the functions working with the user interface, one file working with file I/O, and another with working with the rules of the game. This way only the user interface file needs to be changed if a different user interface is to be used.

Component coupling

Coupling is a measure of information interchange. With functions, coupling is primarily concerned with the parameters passed to and from functions. With files, coupling is primarily concerned with how easy it would be for someone to use the functionality contained within the file. In other words, the less complicated the interface of the functions, the easier it will be to use these functions. As a general rule, we try to shield the client from the complexity of our algorithms by minimizing how much he need to know.

Back to our Tic-Tac-Toe example, we might be tempted to sub-divide the program into the following parts:



Thus the file I/O is in one component, being the only thing that needs to know about the file format. The game logic is in another component, the only file that understands the rules of Tic-Tac-Toe. Finally the user interface is in the interface component. The complete program listing is available at [1-4-ticTacToe.html](#) or:

/home/cs165/examples/1-4-ticTacToe

Note that this is a directory with several files.

Source files and headers

Up to this point in time, all the code necessary to compile a given program has existed in a single file. We call this file a **source file** (a.k.a. **implementation file**) because the file contains the source code of the program. For a C++ program, source files end with the .cpp suffix.

One factor complicating compilation of a source file is that, at compile time, all references need to be known. In other words, before you can call a function, the function needs to be defined or prototyped. How can this happen when the function might be defined in another file? Header files are designed to address this issue.

Header files

A header file is in many ways like a cargo ship's manifest. It contains a list of all the contents so clients know what they can expect to find inside. This is important in the context of separate compilation because a given source file needs to know what tools (such as functions) are defined in other files.

In C++, header files end with the .h suffix. With few exceptions, header files do not contain code. Instead, they contain the necessary information for source files to use code existing in other files. Header files consist of:

- **Function prototypes:** Prototypes of the functions intended to be used by the client.
- **Type definitions:** Structure definitions intended to be used by the client.

Typically every component source file has its own header file by the same name. For example, `file.cpp` will have `file.h` as a companion. In order to use the functions in the source file, the header file will need to be included.

Including header files

In order to include the code from a header file into a source file, it is necessary to use the `#include` pre-processor directive. This directive tells the pre-processor to copy-paste the code from the specified file directly into the source file. Consider the following line of code:

```
#include <iostream>
```

This instructs the pre-processor to include a standard template library (`iostream` in this case) into the program we are writing.

We can also include files from our own directory using a similar syntax:

```
#include "file.h"
```

There is no mystery what this line of code does. The above line of code is exactly the same as the programmer physically copying the code from `file.h` and pasting it into the program at that point.

Notice that there is a slight difference between including the `iostream` library and the `file.h` header. In the former case, there are chevrons `<>` around the file name. This tells the pre-processor to look in the standard library directory for the file. In the latter case, there are double quotes `""` around the file name. This tells the pre-processor to look in the current directory for the file.



Sue's Tips

It is a good habit to design header files in such a way that it is as easy and convenient as possible for the client to use your code. One part of that is to avoid the following statement:

```
using namespace std;
```

By using that statement, you are forcing your client to use the standard namespace. use the long version of the common data types in your function prototypes:

```
std::string getText(); // instead of "string getText();"
```

Ensuring only one copy is used

There is one level of complexity that needs to be addressed in order to fully understand header files. Consider a header file that contains a structure representing a Tic-Tac-Toe board combined with whose turn it currently is:

```
struct Game
{
    char board[3][3];
    bool xTurn;
};
```

Because this structure will be needed by almost every function in the program, we decide to put it in its own header file called `ticTacToe.h`. So far, everything works great. Now let's say that I wish to use this structure as a parameter in the `interact()` function whose prototype exists in `interface.h`. This means that part of `interface.h` will appear as follows:

```
#include "ticTacToe.h" // need Game structure for interact()

// the main interaction loop
void interact(Game & game);
```

I will also need the `Game` structure in the file I/O functions as well. Thus part of `file.h` will appear as:

```
#include "ticTacToe.h" // need Game for both read and write
#include <string> // need the string class

// read the board from a file
void readBoard(const std::string & fileName, Game & game);

// write the board to a file
void writeBoard(const std::string & fileName, const Game & game);
```

Now I wish to use both `interact()` and `writeBoard()` in the same file. This will necessitate including both `interface.h` and `file.h`:

```
#include "file.h"
#include "interface.h"

int main()
{
    Game game;
    readBoard(string("game.txt"), game);
    interact(game);
    return 0;
}
```

This will generate a very unusual compile error: apparently we are attempting to define the `Game` structure twice! At first this may seem unusual. We only have one copy of `Game` and that copy is in `ticTacToe.h`. However, since `interface.h` does a `#include "ticTacToe.h"` as does `file.h`, that means that two copies of the structure definition are included. How can this be avoided?

The answer is to put a special `#ifndef` around each header file:

```
#ifndef TIC_TAC_TOE_H
#define TIC_TAC_TOE_H

/******************
 * GAME
 * A game is a combination of the 3x3
 * board and whose turn it currently is
 *****************/
struct Game
{
    char board[][][3];
    bool xTurn;
};

#endif // TIC_TAC_TOE_H
```

The first time this file is included, `TIC_TAC_TOE_H` is not defined in the code so it gets included. The second time, however, `TIC_TAC_TOE_H` is defined so the entire body of the header file is skipped. This mechanism ensures only one copy of a header file is ever included in the compilation of a source file. In other words, it enables programmers to include header files without worrying about including the same file twice.

Compiling

Possibly the easiest way to compile a project consisting of multiple files is to list all the source files on the compiler's command line:

```
g++ ticTacToe.cpp file.cpp interface.cpp game.cpp
```

When doing this, it is important that there is exactly one `main()` in the source files. If there is more or less, the compiler will be unsure from which function to start the program. It is therefore a common practice to name the file containing `main()` with the name of the resulting program. In this case, the program is called `ticTacToe` so the file containing `main()` is `ticTacToe.cpp`.

Observe how there are no header files specified on the command line. The reason for this is that the header files are included with the `#include` mechanism.

Compilation script

As the number of files in a project get larger and larger, it becomes more and more tedious to type this long command when compiling the project. A simple way to avoid this is to create a small program called a script to compile the project for you.

To create a compile script (traditionally called `compile`), simply edit a file called `compile`:

```
g++ ticTacToe.cpp file.cpp interface.cpp game.cpp
```

```
/home/cs165/examples/1-4-ticTacToe ls
compile file.h game.h interface.h ticTacToe.h
file.cpp game.cpp interface.cpp ticTacToe.cpp
```

In this file, type the command that you need to execute to compile your project. This means that as you add new files to your project, you will need to edit your `compile` file to include the new file.

The final step of creating a compile script is to tell the operating system that the newly-created file is executable. You do this by adding “execute” permissions to the file. Again, from the command line, type the following command:

```
chmod +x compile
```

Observe how, in the above screen-shot, the “`compile`” file is green while the rest of the files are grey. This is the operating system’s way of indicating that the `compile` file is executable. We can simply type the command “`compile`” on the command prompt and our project will be compiled. All examples on the course site using separate compilation will have a `compile` script.

Object files

One advantage of using multiple source files is that compilation can become much quicker. This may not seem like a big deal now; small programs take only a moment to compile. As programs get larger, compilation times matter. For truly large projects, it may take several hours for the program to compile. The key to speeding up the compilation process is to break it into multiple phases. The first phases involves the generation of object files. An object file is a *partially compiled* file, assembly code rather than the machine language required by the CPU. The second phase involves linking together all the object files into a single executable.

You can generate an object file with the following command:

```
g++ -c file.cpp
```

Notice the `-c` compiler switch. This indicates to the compiler that the target will be an object file (`file.o` in this case) and that we are not trying to create an executable such as `a.out`.

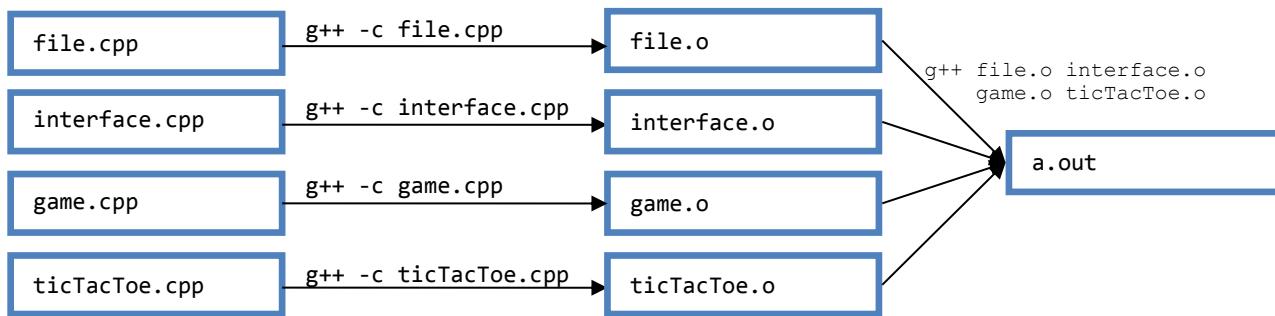
After all the object files are generated, there are `.o` files for every `.cpp` file:

```
/home/cs165/examples/1-4-ticTacToe ls
compile file.o game.o interface.o ticTacToe.o
file.cpp game.cpp interface.cpp ticTacToe.cpp
file.h game.h interface.h ticTacToe.h
```

The last step of the compilation process is to link together all the object files into a single executable. To do this, the object files are sent to the compiler the same way the source files are sent:

```
g++ file.o game.o interface.o ticTacToe.o
```

The following illustration graphically illustrates what occurred in the two stages.



Makefile

A **makefile** is another mechanism to compile multiple source files. It accomplishes this by selectively compiling the object files of a project according to what has changed since the last compilation. The essence of **makefile** is this selection process.

A **makefile** is a file that contains the dependencies for a given project. In the **makefile**, the programmer indicates how to tell if a given file needs to be rebuilt, and how. Each dependency, called a rule, has three components:

- **Target**: the name of the file that might need to be rebuilt.
- **Dependency List**: the list of dependencies used to determine if target needs to be rebuilt.
- **Recipe**: instructions how to build the target.

The syntax for a recipe is:

```
<target> : <Dependency List>
  <Recipe>
```

Note that there must be a tab (not spaces) before recipe.

For example, we will need to re-build the file “`interface.o`” if the source-file has changed (`interface.cpp`), if the header file has changed (`interface.h`), or if the file containing the board definition has changed (`ticTacToe.h`). If any of these has changed, then we will need to rebuild the `interface.o`.

```
interface.o : interface.cpp interface.h ticTacToe.h
           g++ -c interface.cpp
```

One final thing to know about the **makefile** is that line comments start with the `#` rather than `//`. The complete **makefile** for the Tic-Tac-Toe game is:

```
a.out: file.o interface.o game.o ticTacToe.o
      g++ file.o interface.o game.o ticTacToe.o

file.o : file.cpp file.h ticTacToe.h
        g++ -c file.cpp

interface.o : interface.cpp interface.h ticTacToe.h
             g++ -c interface.cpp

game.o : game.cpp game.h ticTacToe.h
        g++ -c game.cpp

ticTacToe.o : ticTacToe.cpp ticTacToe.h
              g++ -c ticTacToe.cpp
```

This rule has only one command but it is possible to have many commands to the rule. The next question is “how do we know *when* to build `interface.o`?” The prerequisite list will answer this question. If any of the files listed (`interface.cpp`, `interface.h`, `ticTacToe.h`) are *newer* than the target file (`interface.o`), then the rule gets executed. The entire project can get compiled with the following command:

```
make
```

Thus `makefile` is the name of the file containing the build instructions, and `make` is the name of the tool that reads the `makefile` configuration file. `make` will open `makefile` in the current working directory and execute the rules from top to bottom. If all the rules are properly formed, then the end result will be an update to all the targets listed in the `makefile`.

The following sections of the [GNU reference for the makefile](#) are worth the read:

- [2. An Introduction to Makefiles](#)
- [2.1 What a Rule Looks Like](#)
- [2.2 A Simple Makefile](#)
- [2.3 How make Processes a Makefile](#)

Submission tools

In our classes, we turn in assignments using the `submit` program. This program, unfortunately, only accepts one file and changes the filename. The combination of these two things requires us to use an extra tool in order to turn in assignment consisting of multiple files. This tool is called TAR.

TAR (Tape ARchiving utility, originally designed for creating an archive on the old-style reel-to-reel tape storage devices), is a program conceptually similar to ZIP: it combines multiple files into one and compresses them. We will use TAR to combine our separate source files for the purpose of submission.

Creating a TAR file

We only submit those files necessary to compile a program. This means we do not turn in compiled files (ex: `a.out`), object files (ex: `date.o`), or backup files (ex: `date.cpp~`). For the purposes of CS 165, we will only submit source files (`*.cpp`), header files (`*.h`) and the `makefile` itself (`makefile`). To create a TAR file, we use the `tar` command, the `-cf` switch, the name of the target file, and the name of the files to be included. For example, if I were to create a file called `homework1.tar`, then I would use the following command:

```
tar -cf homework1.tar *.h *.cpp makefile
```

Running test-bed on a TAR file

Test-bed is aware of the TAR file format. There are two assumptions that test-bed makes on your file:

- All the files, including the source files, header files, and the `makefile`, must not exist in a directory. This means we should not TAR a *folder*, but rather the *files* themselves as illustrated above.
- The output file should be called `a.out`. After test-bed runs your `makefile` to compile your project, it looks for an executable called `a.out`.

When test-bed runs, it performs the following steps:

1. Creates a folder from the current directory called [username].dir.
2. Copies the TAR file to [username].dir.
3. Extracts the files from the TAR file with:

```
tar -xf [username].tar
```

4. Compiles the files according to the `makefile` instructions

```
make a.out
```

5. Compares the test result of `a.out` with those in the test-bed specification for the assignment.
6. Removes [username].dir.

If test-bed malfunctions, you can debug the process by running through the steps manually.

Turning in a TAR file

For a standard C++ file, `submit` reads the program header to decide where the program is to go. Unfortunately, `submit` cannot read a TAR file. Thus, when we turn in a program containing multiple files, we can expect `submit` to prompt us for the instructor name, the class number, and the assignment name. Note that this does not excuse us from putting a well-formed comment block at the head of each of our files!

Best practices

When working with large, multi-file projects, things can get complicated without taking a few elementary steps. The following best practices are designed to make life easier for programmers:

1. Divide files carefully

Time spent thoughtfully dividing a large project into files, or specifying files for a new project, is well spent. Good modularization practices can pay large dividends throughout the life of the project through easier collaboration, simplified debugging, reduced development time, and easier code re-use.

2. Build driver programs

Create a driver program for the file every time a file is created. For example, a source file called `file.cpp` should have a header called `file.h` and also a driver program called `fileTest.cpp`. The driver program contains a `main()` and should make it easy to exercise the interfaces in the source file. This will facilitate finding bugs and give you some idea as to how easy the interfaces are to work with.

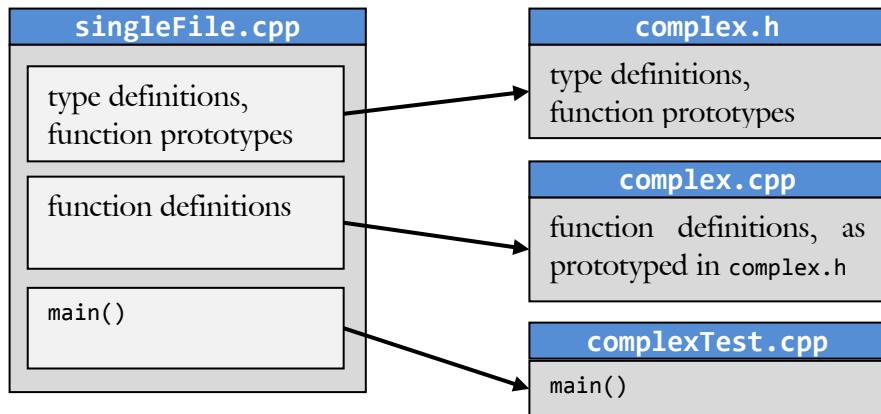
3. Negotiate interfaces with your collaborators

When collaborating with other programmers, you can save a lot of time and frustration by carefully defining your interfaces. Since collaboration typically occurs at the file level, changes affecting header files are much more painful than those affecting only source files. Periodic meetings with your collaborators can be best spent reviewing and negotiating interfaces. These interfaces often serve as contracts between different aspects of the program.

Example 1.4 – Complex Number

This example will demonstrate how to break a single large file to several, more manageable chunks. The single large file is in `singleFile.cpp`.

Demo



The header file is:

```
#ifndef COMPLEX_H
#define COMPLEX_H
struct Complex
{
    double real;
    double imaginary;
};

void prompt(Complex & complex);
void display(const Complex & complex);
void addTo(Complex & lhs, const Complex & rhs);
bool isZero(const Complex & num);

#endif // COMPLEX_H
```

Solution

The compile script is:

```
g++ complex.cpp complexTest.cpp
```

The makefile is:

```
### the main rule
a.out: complexTest.o complex.o
    g++ -o a.out complexTest.o complex.o
    tar -cf example14.tar *.h *.cpp makefile

### for the object files
complexTest.o: complexTest.cpp complex.h
    g++ -c complexTest.cpp

complex.o: complex.cpp complex.h
    g++ -c complex.cpp
```

See Also

The complete solution is available at [1-4-complexNumbers.html](#) or:

```
/home/cs165/examples/1-4-complexNumbers/
```



Review 1

Given the following structure definition and variable declaration, enumerate all possible ways to access the member variables:

```
struct A1
{
    int a;
    int b;
};

struct A2
{
    int c;
    A1 d;
};

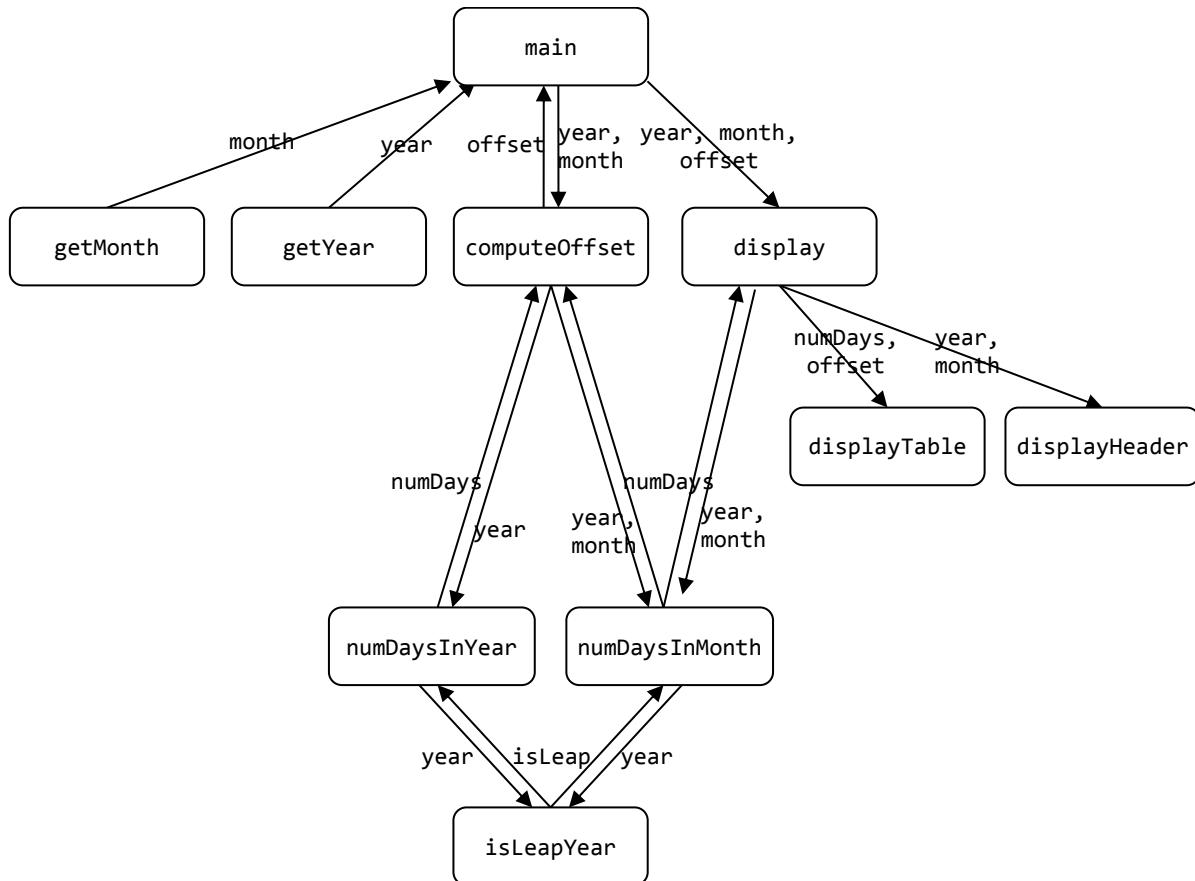
struct A3
{
    A2 e;
    A1 f;
    int g;
};
```

```
{
    A1 x;
    A2 y;
    A3 z;
}
```

Please see page 69 for a hint.

Problem 2

Consider the following structure chart for the Calendar program from CS 124. What would be the best way to break the project up into separate files?



Please see page 80 for a hint.

Problem 3

Write the header files from Problem 2

Filename: _____

Filename: _____

Filename: _____

Please see page 81 for a hint.

Problem 4

Write the `makefile` for Problem 2

Please see page 85 for a hint.

Problem 5, 6, 7, 8

Given the following `makefile`:

```
a.out : sukoku.o game.o interface.o file.o
g++ sukoku.o game.o interface.o file.o

gameTest : gameTest.o game.o
g++ -o gameTest.o game.o

sukoku.o : sudoku.cpp sudoku.h
g++ -c sudoku.cpp

game.o : game.h game.cpp sudoku.h
g++ -c game.cpp

interface.o : interface.cpp interface.h sudoku.h
g++ -c interface.cpp

file.o : file.h file.cpp sudoku.h
g++ -c file.cpp
```

5. List all the executables that will be built from the above `makefile`.
6. List all the object files that will be built from the above `makefile`.
7. List, in order, what would be built if `interface.h` were changed.
8. List, in order, what would be built if `sudoku.h` were changed.

Please see page 85 for a hint.

1.5 Function: Advanced Topics

Sam is working on a project with Sue and is getting a bit exasperated. So many of the functions he needs to write are nearly identical! He was just about to give up when Sue shows him a few tricks to make his job much easier.

Objectives

By the end of this chapter, you will be able to:

- Define and use a collection of overloaded functions
- Create a function with default parameters
- Make a function inline and articulate when it would be useful to do so
- Create a pointer to a function

Prerequisites

Before reading this chapter, please make sure you are able to:

- Write a user-defined function to perform a task
(Procedural Programming in C++, Chapter 1.4)
- Pass parameters to a function by-value, by-reference, and by-pointer
(Procedural Programming in C++, Chapter 1.4, 3.3)
- Create a pointer variable, initialize it, dereference it, and pass it to a function
(Procedural Programming in C++, Chapter 3.3)

The advanced topics of functions and why you should care

As we learned from CS 124, functions are a great tool to simplify the job of writing programs. They allow us to break large and complex projects into more manageable chunks and they facilitate code re-use. While these are all true, they are a very procedural way of approaching a project. In the object-oriented world, it is often useful to think in terms of the **client**, some programming collaborator who will use the code you produce. As we begin to think in terms of producing code that others consume, it becomes apparent that we need to find ways to make using our code more convenient for the client.

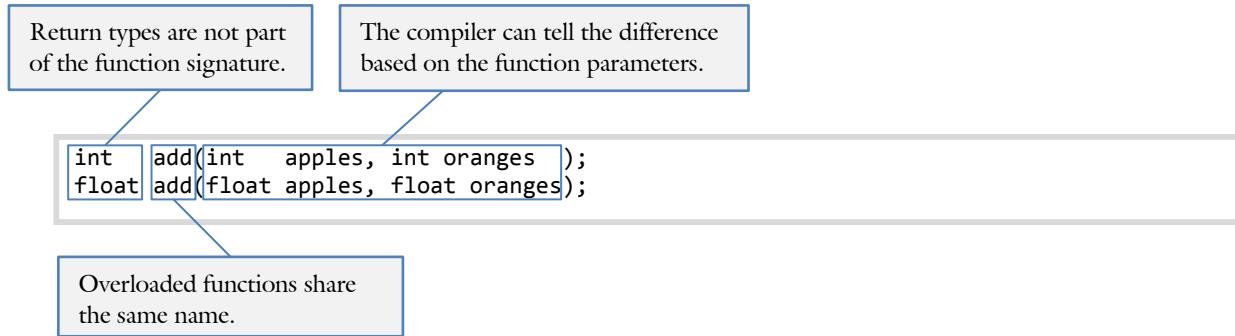
There are four topics concerning functions that, from a procedural perspective, may not seem to have much value. As we learn about object-oriented constructs in the coming units, the hope is that the value of these topics will become apparent. These topics are overloading, default parameters, inline, and pointers to functions.

Overloading

Overloading is the process of having more than one function with the same name. The only differences between the functions are the parameters they take. Perhaps this is best explained by example:

```
int addIntegers(int value1, int value2);
float addFloats( float value1, float value2);
```

Both of these functions perform essentially the same task though with different parameters. Overloading allows us to simplify the names of these functions:



We specify which version of `add()` we want through the data type of the parameter we pass. Thus calling `add(4, 5)` will access the integer version of the function while calling `add(4.0, 5.0)` will access the float version. This matching occurs by comparing the function signatures of the overloaded functions.

Function signature

A **function signature** is the combination of the name of the function with the data type of the parameters it takes. Thus the function signature of “`int add(int value1, int value2)`” is “`add-int-int`”. Two functions can be said to be overloaded when they have the same name but different signatures. Note that the notion of a function signature does not include the return type. Thus the following is in error:

```
string prompt();           // ERROR: we cannot overload prompt() this way because the
int    prompt();           //             function signatures are identical.
```

Sam's Corner



The compiler is able to handle overloading in the following way. When a file or program is compiled, a table is created with all the function names and the location in memory where the code for the function is to reside. The function name the compiler uses, however, is a special one. It contains the complete function signature. For example, if we were to create the following function:

```
int add(int value1, int value2);
```

The compiler would name this function “`add@int@int`.” Thus an `add()` function taking `floats` as parameters (`add@float@float`) would clearly have a different name to the compiler.

When to use overloading

Overloading should be used whenever a collection of functions has the same semantic meaning but operate on different parameters. Common examples include mathematical operations:

```
void setZero(float      & value);
void setZero(Complex    & value);
void setZero(Coordinate & value);
```

Another example might be an operator that works with a variety of data types:

```
void sort(int array[], int size);
void sort(float array[], int size);
void sort(string array[], int size);
```

In general, whenever the same action is to be applied to many things, overloading is a good choice.

When to not use overloading

Consider the case when there are three overloaded versions of the `setZero()` function from the above example. How does the compiler choose which to call? The answer, of course, depends on the parameters that are passed to the function. If a `float` is passed to `setZero()`, then the float version will be called.

We cannot overload functions where the only difference in the function signature is a constant modifier or a by-reference modifier. To illustrate this point, consider the following code:

```
void overload(int x)                      // overload() with an integer parameter
{
    cout << "Integer\n";
}
void overload(const int x)                 // overload() with a constant integer parameter
{
    cout << "Constant integer\n";
}
void overload(int & x)                     // overload() with an integer by-reference
{
    cout << "By-reference integer\n";
}

int main()
{
    int value;
    overload(x);                          // ERROR: which version of overload() do you
    return;                                //     think will be called? We call all three
                                            //     versions the same way! This is an error
                                            //
```

In the above example, the compiler will generate the following errors:

```
1-5-overload.cpp: In function “void overload(int)”:
1-5-overload.cpp:22: error: redefinition of “void overload(int)”
1-5-overload.cpp:18: error: “void overload(int)” previously defined here
1-5-overload.cpp: In function “void overload(int)”:
1-5-overload.cpp:57: error: redefinition of “void
overload(int)”
1-5-overload.cpp:18: error: “void overload(int)” previously defined here
```

Example 1.5 – Overloading

This example will demonstrate several ways to overload a function. In each case, the function is called “overload()” though different parameters will be used.

Demo

There are three versions of the function `overload()`:

```
void overload(int value)
{
    cout << "The integer version of the overload() function\n";
}
void overload(int value1, int value2)
{
    cout << "The multi-integer version of the overload() function\n";
}
void overload(const char * value)
{
    cout << "The text version of the overload() function\n";
}
```

Solution

Now we will call these functions:

```
int main()
{
    cout << "overload(7):      ";
    overload(7);

    cout << "overload(7, 7):   ";
    overload(7, 7);

    cout << "overload(\"seven\"): ";
    overload("seven");

    return 0;
}
```

The output of the above program is:

```
overload(7):      The integer version of the overload() function
overload(7, 7):   The multi-integer version of the overload() function
overload("seven"): The text version of the overload() function
```

Challenge

As a challenge, see if you can add to this example by adding another overloaded function:

```
void overload(int * pointer);
```

Notice how the pointer modifier is part of the function signature.

See Also

The complete solution is available at [1-5-overload.html](#) or:

```
/home/cs165/examples/1-5-overload.cpp
```



Example 1.5 – Swap

This example will demonstrate several ways to overload a function taking two parameters. In each case, the name of the function will remain the same but the parameters will be different.

Write three versions of a function called `swap()`: one that swaps two integers, one that swap two strings, and one that swaps two doubles.

There are three versions of the function `swap()`:

```
void swap(int & value1, int & value2)
{
    int tmp = value1;
    value1 = value2;
    value2 = tmp;
}
void swap(string & s1, string & s2)
{
    string tmp = s1;
    s1 = s2;
    s2 = tmp;
}
void swap(double & value1, double & value2)
{
    double tmp = value1;
    value1 = value2;
    value2 = tmp;
}
```

Now we will call these functions:

```
int main()
{
    string last = "Thomas";           // oh no! I got the first name and
    string first = "Ricks";          //   the last name mixed up

    swap(last, first);              // calls the “string &” version

    cout << first << ' ' << last << endl; // all is better now!
    return 0;
}
```

As a challenge, can you think of other versions of `swap()` that may be useful? Implement them and write a driver program to test them.

The complete solution is available at [1-5-swap.html](#) or:

```
/home/cs165/examples/1-5-swap.cpp
```



Default parameters

Often we write a function with multiple parameters but when we call the function we pass exactly the same values again and again. Wouldn't it be great if we could tell the compiler "If I don't pass anything, just use the default value." Default parameters give us that capability. It is exactly the same thing as overloading a function with different numbers of parameters. For example, consider the `getline()` function to fetch a line of text from the keyboard. The prototype could be:

```
void getline(char * text, int bufferSize);
```

Note that we use a buffer size of 256 the vast majority of the time. It might behoove us to then provide the following code to make it more convenient for the user of our code:

```
void getline(char * text)
{
    getline(text, 256);
}
```

Observe how the client can now either call the two parameter version of the function or the one parameter version. If they call the one parameter version, then 256 is supplied for the buffer size. C++ provides a mechanism to make this process much easier:

```
void getline(char * text,           // required parameter
            int bufferSize = 256); // default or optional parameter
```

Order of the parameters

When designing a function with default parameters, it is important to carefully order the parameters. If, for example, there is one required parameter and one default parameter, the required parameter must be the left-most of the two. Similarly, if there are three default parameters and the user specifies one, it is the left-most parameter that is matched with the user value.

```
void function(int value1 = 0, int value2 = 0);
function(43)
```

Prototypes

One final quirk of default parameters: if you are using a function prototype, the default parameter specification goes in the function prototype rather than in the function declaration. For example:

```
// prototype
int getIndex(int max,           // required parameter
             int min = 0); // default parameter specified here

// declaration
int getIndex(int max, int min) // the default value is not indicated
{
    int input;
    do
    {
        cout << "Enter a value between "
            << min << " and " << max << ": ";
        cin >> input; // notice how we use the variable min
                      // exactly the same regardless
                      // of whether the default value
                      // is used or a user-specified
                      // value is used
    }
    while (input > max || input < min);
    return input;
}
```

Demo

Example 1.5 – Default Parameter

This example will demonstrate the use of default parameters. Note how we will have one function declaration but it will be called four different ways.

The prototype for our default parameter function is:

```
void defaultParam(int one = 0, int two = 0, int three = 0);
```

The declaration does not include the default parameter specification:

```
void defaultParam(int one, int two, int three)
{
    cout << "defaultParam(" << one << ", " << two << ", " << three << ")\n";
}
```

Now to test the function, we will call it four different ways:

```
int main()
{
    // zero parameters
    cout << "defaultParam()      : ";
    defaultParam();                                // one == 0, two == 0, three == 0

    // one parameter
    cout << "defaultParam(1)    : ";
    defaultParam(1);                               // one == 1, two == 0, three == 0

    // two parameters
    cout << "defaultParam(1, 2)  : ";
    defaultParam(1, 2);                            // one == 1, two == 2, three == 0

    // three parameters
    cout << "defaultParam(1, 2, 3) : ";
    defaultParam(1, 2, 3);                          // one == 1, two == 2, three == 3

    return 0;
}
```

The output for this program is:

```
defaultParam()      : defaultParam(0, 0, 0)
defaultParam(1)     : defaultParam(1, 0, 0)
defaultParam(1, 2)   : defaultParam(1, 2, 0)
defaultParam(1, 2, 3) : defaultParam(1, 2, 3)
```

Challenge

As a challenge, add a fourth default parameter to this function. Make it a c-string. There are several key issues that will need to be addressed: what should the default value be, and how will you handle the NULL pointer case in the program.

See Also

The complete solution is available at [1-5-defaultParam.html](#) or:

```
/home/cs165/examples/1-5-defaultParam.cpp
```



Inline

Often we wish to write a trivial function with just one or two lines of code. Unfortunately, with standard functions, this trivial function incurs a small performance penalty. Function calls take about a dozen clock cycles or more, depending on the number of parameters and local variables. This may seem insignificant, but they will pile up when performed over and over again.

Inline functions are designed to alleviate this performance penalty. When a function is called, instead of leaving the current function and jumping to the called function, the compiler simply pastes the code from the inline function directly into the caller. In other words, there is exactly one copy of each standard function in a compiled program. With inline functions, there are many copies because the compiler makes duplicate copies of your code.

Inline functions represent a net performance win because the overhead of calling a function is avoided. However, it could actually involve a memory penalty if the amount of code duplicated is large. For this reason, we only use inline functions when there are just a few lines of code in the function body.

To make a function inline, simple add the `inline` keyword to the beginning of the function:

The `inline` keyword tells the compiler that it would be more efficient to copy-paste the code of the function rather than use a traditional function call.

```
inline int skates()
{
    return 42;
}
```

Header files

There is one quirk of inline functions: they must be specified in the header file rather than in the source file. The reason for this is that, at compile time, the complete body of the function must be available when the source file is compiled. This means that if we wish to make the `convertSymbol()` function from our Tic-Tac-Toe program inline, we would need to modify `interface.h`:

```
#ifndef _INTERFACE_H
#define _INTERFACE_H

#include "ticTacToe.h"      // need Game structure for interact()

// the main interaction loop
void interact(Game & game);

/******************
 * CONVERT SYMBOL
 * Convert symbol for the space '.' into
 * the display representation ' '
*****************/
inline char convertSymbol(char letter)
{
    assert(letter == '.' || letter == '0' || letter == 'X');
    return (letter == '.') ? ' ' : letter;
}

#endif // _INTERFACE_H
```

Example 1.0 – Inline

This example will demonstrate the user of inline functions. Note how they behave exactly the same as non-inline functions. The only difference is performance.

Demo

The inline function (multiply in this case) is:

```
inline int multiplyInline(int value1, int value2)
{
    return value1 * value2;
}
```

The standard or non-inline version of the same function is:

```
int multiplyStandard(int value1, int value2)
{
    return value1 * value2;
}
```

We will call these two functions the same way:

```
int main()
{
    // the inline version
    cout << "multiplyInline(6, 7) = "
        << multiplyInline(6, 7)
        << endl;

    // the standard version
    cout << "multiplyStandard(6, 7) = "
        << multiplyStandard(6, 7)
        << endl;

    return 0;
}
```

We also expect these two functions to behave the same way:

```
multiplyInline(6, 7) = 42
multiplyStandard(6, 7) = 42
```

Challenge

As a challenge, go back to an old project from CS 124 or from earlier this semester and see how many of your trivial functions should be inline. Remember, functions that are more than a few lines should not be made inline.

See Also

The complete solution is available at [1-5-inline.html](#) or:

```
/home/cs165/examples/1-5-inline.cpp
```



Pointers to functions

A pointer, as you recall, is the address of data rather than the data itself. For example, a URL is not the web page but rather the address of the web page somewhere on the internet. Similarly, we can create a new variable holding *data* with:

```
int data;
```

Or we can create a new variable to hold the *address of data* with:

```
int * pointer;
```

All variables reside in memory. We can retrieve the address of a variable with the address-of operator:

```
{
    int data;                                // "data" resides in memory
    cout << &data << endl;                  // "&data" will return the address of "data"
}
```

It turns out that functions also reside in memory. We can retrieve the address of a function, store the address of a function, and de-reference the address of a function in much the same way we do other pointers.

Getting the address of a function

When your program gets loaded by the operating system in memory, it is given an address. Therefore, the addresses of all the functions in the program are known at program execution time. We can, at any time, find the address of a function with the address-of operator (`&`). If, for example, there was a function with the name `display()`, then we could find the address of the function with:

```
&display
```

Note how we do not include the `()`s here. When we put the parentheses after a function, we are indicating we want to call the function.

Declaring a function pointer

Declaring a pointer to a function is quite a bit more complex. The data type of the pointer needs to include the complete **function signature**, including:

- **Name:** The name of the function.
- **Return type:** What type of data is returned, even if the type is `void`
- **Parameter list:** All the parameters the function takes.

The syntax for a pointer to a function is:

The parenthesis and asterisk indicate a pointer to a function.

```
void (*dropBass)(int volume, int bass);
```

Consider, for example, the following function prototype and pointer of the same type:

```
void display(double value);           // prototype

int main()
{
    void (*p)(double);             // pointer to a function. Variable 'p'
    p = &display;                  // initialize the pointer to display()
```

Using function pointers

Once you have an initialized pointer to a function, you use it much as any other pointer variable: using the dereference operator (*). The important difference, however, is the requirement to specify the parameters as well. One might assume the following would be the correct syntax:

```
*p(value); // ERROR! calling a function named 'p' and dereferencing
           // the return value
```

This is an error. In this case, the order of operations for the parentheses () is before that of the dereference operator *. As a result, the compiler thinks you are calling a function named p() returning a pointer which is to be dereferenced. To make your intentions clear, a slightly more heavy syntax is required:

```
(*p)(value); // CORRECT, though the p(value) convention is more convenient
```

In this case, we are first dereferencing the pointer variable p before attempting to call the function, exactly what is needed. It turns out that the dereference operator (*) is optional here. As long as there is not another function named p, we can simply say:

```
p(value); // CORRECT, though one might expect to need the * to dereference 'p'
```

This is the preferred way to access a pointer to a function.

Passing a function pointer as a parameter

It turns out there are no tricks or complications when passing a function pointer to another function as a parameter. Consider the following function prototype from the previous examples:

```
void display(double);
```

An example of the code to accept as a parameter a function pointer matching the above signature:

```
void function(void (*pointer)(double), double value)
{
    pointer(value);                // we could also say (*pointer)(value);
```

Observe how the first parameter p is a function pointer. The easiest way to call this function is by specifying the address of the target function directly:

```
{
    function(display, 3.14159);      // we could also say function(&display, 3.14159);
}
```

Example 1.5 – Function Pointer

Demo

This example will demonstrate how to create a variable that is a pointer to a function, and pass it as a parameter to another function.

Problem

Write a program to prompt the user for a value. Then prompt the user if the value is a GPA (displayed with one digit after the decimal) or an amount of money (displayed like \$100.32).

Solution

First, we need two display functions to handle the GPA and the money case. The prototypes are:

```
void displayGPA (float gpa );
void displayMoney(float money);
```

Next we need to write a generic display function to handle either case. We will delegate the code that actually handles the number to one of our two display functions:

```
void display(void (*pDisplay)(float), float value)
{
    cout << "The answer is: ";
    pDisplay(value);
    cout << endl;
}
```

Finally, we will ask the user which version of the code we should call.

```
int main()
{
    // prompt for the value
    float value;
    cout << "What is the amount? ";
    cin >> value;
    // prompt for the type of value it is
    char input;
    cout << "Is this money (y/n)";
    cin >> input;

    // choose the appropriate display function
    void (*pDisplay)(float);
    if (input == 'Y' || input == 'y')
        pDisplay = displayMoney;
    else
        pDisplay = displayGPA;

    // call the generic display function
    display(pDisplay, value);    return 0;
}
```

Challenge

As a challenge, add a third display type: weight to the nearest pound:

```
155 lbs
```

You will need to change the IF statement in `main()` to a SWITCH to accommodate this challenge.

See Also

The complete solution is available at [1-5-pointerToFunction.html](#) or:

```
/home/cs165/examples/1-5-pointerToFunction.cpp
```



Problem 1, 2

Given the following makefile:

```
a.out : suokoku.o game.o interface.o file.o
g++ suokoku.o game.o interface.o file.o

gameTest.out : gameTest.o game.o
g++ -o gameTest.out gameTest.o game.o

sudoku.o : sudoku.cpp sudoku.h
g++ -c sudoku.cpp

game.o : game.h game.cpp sudoku.h
g++ -c game.cpp

interface.o : interface.cpp interface.h sudoku.h
g++ -c interface.cpp

file.o : file.h file.cpp sudoku.h
g++ -c file.cpp
```

1. List, in order, what would be built if `file.cpp` were changed.

1. List, in order, what would be built if `game.h` were changed.

Please see page 85 for a hint.

Problem 3

Write four functions to return the larger of two values passed as parameters.

- Two integers
- Two characters
- Two floats
- Two strings

Please see page 95 for a hint.

Problem 4

Make the following function inline:

```
int max(int value1, int value2)
{
    return (value1 > value2) ? value1 : value2;
}
```

Please see page 99 for a hint.

Problem 5

Given these three functions:

```
int getValue()

{
    int value = -1;
    while (value < 0 || value > 100);
        cin >> value;
    return value;
}
```

```
int getValue(int max)

{
    int value = -1;
    while (value < 0 || value > max);
        cin >> value;
    return value;
}
```

```
int getValue(int max,
            int min)

{
    int value = -1;
    while (value < min || value > max);
        cin >> value;
    return value;
}
```

Make one function out of them using default parameters.

Please see page 98 for a hint.

Challenge 6

Given the following function:

```
bool compare(int n1, int n2)
{
    return n1 == n2;
}
```

Add a third parameter to `isEqual()` below. This parameter is a pointer to a function of the same type as `compare()`. Then use that pointer to compare whether the two passed arrays are the same.

```
bool isEqual(const int array1[], int n1,
             const int array2[], int n2,
             _____)
{
    for (int i = 0; i < n1 && i < n2; i++)
    {
        if (!_____)
            return false;
    }
    return true;
}
```

Finally, call the function `isEqual()` with the third parameter being the function `compare()`.

```
int main()
{
    int array1[] = {4, 6, 3, 2, 1};
    int array2[] = {4, 6, 7, 2, 1};

    cout << isEqual(array1, 5,
                    array2, 5,
                    _____)
        << endl;
}
```

Please see page 103 for a hint.



Unit 2. Encapsulation

2.0 Encapsulation Design.....	108
2.1 Building a Class	119
2.2 Class Syntax	126
2.3 Accessors & Mutators	136
2.4 Constructors & Destructors	143
2.5 Static	155
2.6 Non-Member Operator Overloading.....	165
2.7 Friends	180
2.8 Member Operator Overloading.....	188

2.0 Encapsulation Design

Sam is working on a new programming project: to play the card-game War. As he works through the design of the project, he is having difficulty settling on the data type for a single playing card. Should it be an integer with the value 0-51? Should it be a structure with two member variables? While C++ offers him many built-in data types to choose from, there is nothing which is exactly a playing card. This project would be so much easier if the designers of the C++ language added a special data type for him.

Objectives

By the end of this chapter, you will be able to:

- List and define the components of a UML class diagram
- Enumerate the four encapsulation design rules
- Design a new data type to fit the needs of a given project
- Explain why custom data types simplify the design of large projects

Prerequisites

Before reading this chapter, please make sure you are able to:

- Use UML class diagrams to describe a structure (Chapter 1.3)
- Define functional cohesion and explain how cohesion helps programmers design better functions (Procedural Programming in C++ Chapter 2.0)

What is encapsulation and why you should care

Encapsulation is the process of separating the use of an item from its implementation. Perhaps this is best explained by analogy. The owner of a car can become well-versed in driving a car (use) without understanding the engineering behind how it is built (implementation). If the interface between the internal workings of the car and the driver is sufficiently clear, and if the engineering of the car is sufficiently reliable, a car-owner can use a car for decades without knowing the compression ratio or spark plug firing sequence of the engine. The designers of the car went to great lengths to shield the driver from this knowledge so the driver can focus on the one thing he cares about: getting the car safely from point A to point B.

This principle of encapsulation is also at work in the programming world. However, instead of just shielding the user from the internal workings of our code, we strive to shield the client (other programmers we are collaborating with) from the internal workings of our code. Programmers should design their functions and custom data types in such a way that the other programmers who use our code need to only concern themselves with what the code does, not how it does it.

Two types of encapsulation

There are two main forms of encapsulation: procedural encapsulation commonly known as modularization, and data encapsulation (Schach, 2007, p. 196).

Procedural encapsulation

Procedural encapsulation is the process of modularizing a collection of related operations into a single function. A well modularized design will free the client of a function from having to know about how the function was implemented. Similarly, a well modularized design will allow the programmer to alter how a given function solves a problem without concern as to how the clients will behave.

The main tools we have to measure procedural encapsulation are cohesion and coupling (Robertson, 2004). As you recall, **cohesion** is a measure of the internal strength of a module. The stronger the cohesion, the more focused the module is on performing a single task. **Coupling** is a measure of the information interchange between modules. The simpler the interface, the easier it is to use the function in more contexts.

Functions exhibiting sound modularization techniques have strong procedural abstraction. This facilitates both repurposing the function and changing the function implementation without side effects in the rest of the program.

Data encapsulation

Object oriented design adds another form of abstraction: data encapsulation. **Data encapsulation** is the process of creating new data types with interfaces useful to the clients of the type yet whose implementation is not necessarily known. This technique allows the programmer to focus on using the data rather than maintaining the data. Data encapsulation is also called **data abstraction** and **information hiding**.

Data encapsulation is the process of creating a new type, called a class. A **class** is the collection of data-items and the operations on those items. These are called member variables and member functions.

- **Member variables.** A member variable is one of a collection of items in a class necessary to represent a given abstract data type. While these could consist of built-in types, they could also consist of other custom data types.
- **Member functions.** A member function, typically called a **method**, is a part of the class enabling clients of the class to access member variables, and manipulate and/or compute with them.

When a variable is declared of a class type, it is called an **object**. The user of the class, or the client, is able to treat this class like any other variable to help him satisfy the needs of his program. The author of the class tries to design the class in such a way as to maximize the utility of the new data type to the client.



Sue's Tips

In CS 124 we were introduced to c-strings, a way to represent text as an array of characters. We also learned about a collection of functions making string manipulation easier: `strlen()`, `strcat()`, and `strcpy()`. One convenient property of classes is that it is possible to bundle the data type and all the functions that operate on the data type into a single package: a class. This is the essence of Object Oriented design.

UML class diagrams

Recall from Chapter 1.3 the UML class diagram we used to describe structures.

Complex
real
imaginary

This consisted of two parts: the first row is the name of the structure (`Complex` in this case) and the second is a list of the member variables (`real` and `imaginary`). The class diagram to describe a class is similar except a third row is added to describe the functions (called **member functions** or **methods**) associated with the class.

Complex
real
imaginary
display
set

There is one additional level of complexity to the UML class diagram: **access modifiers**. An access modifier allows the designer of the class to specify whether a given member variable or method is accessible to clients of the class or just to methods of the class itself. If a member variable or method can be freely accessed by anyone, including the client or methods within the class, then it is called **public**. A public method or member variable is signified with the `+` sign on the UML class diagram. However, if access is restricted to only a method, the member variable or member function is called **private** and the `-` sign is used in the UML class diagram:

ClassName
+ publicVariable
- privateVariable
+ publicMethod
- privateMethod

For example, consider the `string` class built into the STL. There are externally-facing methods (`size()` and `c_str()`, for example), and methods not visible to the client (`reallocate()`). There are externally facing variables (well, not really, but we will say so now for the sake of argument that the `array` is externally facing) and private variables (such as `length`). In this case, the class diagram would look like:

String
+ array
- length
+ size
+ c_str
- reallocate

Sue's Tips



It is common for programmers new to UML to confuse the public/private designations with the variable/method parts of the class diagram. If you are ever unsure what goes where on the class diagram, remember this: a structure only has two rows (one for the name and the other for the member variables) and everything is public.

Designing classes

When designing a class, the foremost concern that should be in the programmers mind is how convenient the class will be for the client to use. In other words, can the class be designed in such a way that it is as useful as possible and requires the smallest amount of work to integrate? To accomplish this goal, one simple thing is needed:

Design the interface of the class before worrying about implementing the class

Typically it is best to design the publically visible methods before the implementation details are considered. A primary goal of the class author is to make the class as useful to the clients as possible. This can be achieved through careful consideration of all the possible needs of the clients, making a method custom designed for each of these needs. In other words, focus on the interface of the class rather than the data.

A common mistake for new class designers is to start the other way around: with the data representation. It is easy to lose sight of the needs of the client when we adopt a data-centric perspective.

As a general rule, start with the class diagram and design all the publicly visible methods (with the + sign) first. Only after this and all other classes are designed do we begin to explore how those methods will work. At this point in time, the private methods and the data representation strategies are addressed.



Sue's Tips

All this talk about making the client's job easier may not make much sense to you if you are working alone on a project. However, there are still many benefits to the client-centered approach even when you are going solo. The main benefit is this: the more encapsulated the program, the easier it is to find bugs. By simplifying the interfaces between parts of a program, we reduce complex interactions that often breed bugs. In other words, you are often your own client! Encapsulation can make your job easier too.

Because encapsulation is such a core part of using classes effectively, four design rules have been developed. (Wick, Stevenson, & Phillips, 2004). Each of these rules are intended to help the programmer focus his efforts on making the class as easy for the client to use as possible. These rules are:

1. **Define properties independent of data.** In other words, consider first how to receive information from the client and send information to the client in a way that is as convenient for the client as possible.
2. **Use only interval data if it is truly interval.** Do not force the client to have to know that you start the week on Sunday rather than Monday. He should not have to be burdened with implementation details such as that.
3. **Use getters and setters if we have interval data.** When data needs to be validated, make sure that the only public interfaces go through validation checks.
4. **Don't use getters and setters if you don't validate.** Why make access to data more inconvenient than necessary if getters and setters do not validate?

Rule #1: Define properties independent of data

Define all properties of an object in the language of the domain and independent of the member data used to implement the object

When defining an object, first design all the publically visible interfaces in a way that makes them as easy to use as possible. All data returned to the client must be immediately useful without further processing. Similarly, all data requested by the object should be readily producible by the client.

A fundamental principle in understanding data encapsulation is the difference between object properties and member data (Wick, Stevenson, & Phillips, 2004).

- **Properties:** An object property is a conceptual characteristic of the object. It is the client conceptualizes or thinks about the object. Another way to think about it is the properties are how the client will want to use or manipulate the object.
- **Data:** The object data is how the object is actually implemented. The author of an object cares deeply about the data representation of the object data because he will have to interface with it when he writes the methods. Note that if any of the implementation details of the data representation are visible outside the object, then clients of the object could likely depend on these details. This will effectively prohibit the author of the object from making any meaningful changes to the data representation and thus defeats the goal of data encapsulation.

One way to see the difference between an object's properties and data is the built-in data type `int`. Programmers are encouraged to think of an integer as a number without a decimal. This understanding is sufficient in most cases; you can perform integral math and perform integral comparisons in a way that is highly predictable. The data representation is quite different. Internally, an integer is a collection of 1's and 0's corresponding to powers of 2. Because all the operations working with integers are built into C++, programmer does not have to be aware of the internal representation of integers. This results in robust data encapsulation of the integer type.

For example, consider an object holding the time of day. The client of the object will want to set the hour and minute, compare if two times are equal, and display the time on the screen. All these are properties of the time. The author of the object may choose to implement the member data as a single integer specifying the number of seconds since midnight. While the data representation should be very elegant and efficient, it should be completely opaque to the client.

Time
- <code>secondsSinceMidnight</code>
+ <code>setHour</code>
+ <code>setMinute</code>
+ <code>compare</code>
+ <code>display</code>
- <code>validate</code>

Rule #2: Only use interval data if the data is truly interval

If a property has a small range of possible values and that range is not likely to change in the future, define “test()” and “changeTo()” methods for that purpose that test for or change to each possible value. Do not define getter and setter methods.

In order to understand this rule, it is first necessary to discuss some types of numbers. Continuous data is defined across a continuum, such as GPA (0.0 – 4.0) and Celsius temperature (-273.15° - ∞). Note that there are an infinite number of possible GPAs even though the range is limited. Discrete data is where every value is distinct, such as ACT score (1 – 36). There is no 24.5 ACT score. Interval data can be either continuous or discrete but there must be a well-defined zero point. In other words, everyone must agree what a GPA of 3.7, 27 °C, and a 24 ACT score means. Numbers that are not interval include the day of the week and the “ace of spades.” Note that we can easily assign either of these values to an integer, but we will need to make an arbitrary decision of what the zero point means. For example, is Sunday the zero value or possibly Monday? The choice is arbitrary. Is the “ace of spades” represented as zero or possibly is the “2 of clubs” represented as zero? If we are going to expose to the client some data that is not truly interval, then we are exposing our arbitrary zero decision.

A better form of encapsulation would be to hide any underlying encoding and present the interface to the client on his terms. For example, consider an object holding the day of the week. While the implementation of this object will probably use an `int` (or `char`) to represent the day, a few other implementation details need to be solved (such as which day is zero?). All these design decisions should be opaque to the client of the class. The client would want to perform the following tasks:

- `test()`: Determine if two days are the same.
- `set(TUESDAY)`: Set a day to a given value.
- `adjust(4)`: Advance by a fixed number of days. Note that we will handle week wrapping (Saturday → Sunday, for example).
- `display()`: Put “Tuesday” on the screen.

DayOfWeek
- day
+ test
+ set
+ adjust
+ display
- validate

Rule #3: Use setters and getters if we have interval data

If a property has a large range of possible values, or if the set of values is likely to change in the future, define public access methods for that property. When defining these access methods, follow the traditional `getX()` and `setX()` naming conventions.

When working with interval data (a range of values with a large number of intermediate values, such as a GPA or the temperature outside), allow the client of the class to get and set values directly. There is one caveat to this rule however. Good encapsulation design dictates that we should not burden the client with the work to keep the data of a class valid. It should be impossible for the client to set data into an invalid state. Fortunately, the class author has a pair of tools to accomplish this: getters and setters.

A **getter** (traditionally called `get()` or `getProperty()`) is a method whose sole purpose is to give the client access to member data. Note that it may be that the member data is stored in a different format than the client expects. In this case, it is the job of the getter to translate member data into client properties. Consider, for example, a class representing the time of day. The data for the class is a single member variable storing the number of seconds since midnight. The properties of the class are the second, minute, and hour of the day. Thus the pseudocode for the three getters would be:

```
Time::getSecond()
RETURN time % 60
```

```
Time::getMinute()
RETURN (time / 60) % 60
```

```
Time::getHours()
RETURN time / 3600
```

A **setter** (traditionally called `set()` or `setProperty()`) is a method serving two purposes: to translate client properties into member data, and to validate that the member data is within the valid range (this will be mentioned in more detail in Chapter 2.3). Thus one of the setters for our `Time` class would be:

```
Time::setSeconds(seconds)
IF (seconds < 0 OR seconds ≥ 60)
    THROW
    time ← seconds + getMinute() * 60 + getHours() * 3600
```

Another example is a class storing a student's GPA. Since the data is interval ($0.0 \rightarrow 4.0$) and the property is the same as the data (both floats), getters and setters are great candidates. The getter will simply return the value in the member data variable (no translation from data to properties required). The setter, however, will first validate the parameter to ensure it is in the correct range ($4.0 \geq gpa \geq 0.0$) before modifying the member variable.

GPA
- gpa
+ get
+ set
- validate

The `set()` function for our GPA class would be:

```
GPA::set(input)
IF 0.0 ≤ input ≤ 4.0
    gpa ← input
```

Rule #4: Don't use getters and setters if you don't validate

If no validation is necessary, use public visibility to provide access to member data of a pure data-structure. Use traditional getters/setters only if validation is necessary.

One principle of data encapsulation is to ensure the data is always valid. If your implementation has no data validation requirements, then using methods to access the data serves little purpose. In these very limited cases, public access to member data is faster than function calls and provides no loss in data hiding.

For example, consider a class consisting of complex numbers. The two member data variables (the real and the imaginary component in this example) have no data validation associated with them because they are simply numbers. In this case, it serves no purpose to restrict client access to getters and setters. It would be more efficient to publicly expose the variables to the clients.

Complex
+ real
+ imaginary
+ display
+ add
+ subtract
+ multiply
+ divide

Final thoughts

When designing programs with procedural tools, structure charts and pseudocode were the best tools to draft out a solution. Using the Object-Oriented design tools and methodologies, we still use structure charts and pseudocode in the design process. However, the main design tools are UML class diagrams. You may find that the data-structure part of the design document becomes the focus of your design efforts.

Example 2.0 – Playing card	
Demo	This example will demonstrate how to use the UML class diagram as an important design tool for an Object Oriented problem. Here, we will implement the data-structures part of a design document.
Problem	Design a program to play the card game of War. The game of War is played by dealing out an entire deck of cards to two players. Each player will turn the top card over at the same time. The winner will take both cards and put them in the bottom of his stack. There is also a special procedure for dealing with the case when both cards are of the same rank and thus a tie.
Solution	<p>First we need a class to describe a playing card.</p> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> <pre>Card - card + display + setRank + setSuit + getRank + getSuit + isSameRank + isLargerRank + isSmallerRank + isSameSuit + isLargerSuit + isSmallerSuit - validate</pre> </div> <p>Note how the design of the class tries to take into account every possible use of the <code>Card</code> for the game of War. It might be that some of the methods are not used by the game. We also need a class to describe a collection of cards called a hand:</p> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> <pre>Hand - cards + getTopCard + setToBottom + getSize</pre> </div> <p>The client does not need to know how the collection of cards is stored. In fact, the only things he will need to do is to get the top card off the stack, put a new card on the bottom, and find out how many there are in the collection.</p>
Challenge	As a challenge, modify the <code>Card</code> class to handle the game of Klondike solitaire. What additional methods would be necessary for this?
See Also	<p>The complete design document for the card game of War:</p> <p style="text-align: center;">Example 2.0: Card Game War</p> <p>The working program can be played at:</p> <div style="border: 1px solid gray; padding: 2px; width: fit-content; margin-top: 5px;"> <code>/home/cs165/examples/2-0-war.out</code> </div>

Problem 1

Consider the following UML class diagram:

Point
+ x
+ y
- invalid
+ get
+ set
+ display
- clear

Find the corresponding variable, function, and class name for each of the following:

- Private member function: _____
- Private member variable: _____
- Class name: _____
- Public member variable: _____
- Public member function: _____

Please see page 110 for a hint.

Problem 2

Identify the name corresponding to the following descriptions:

- The measure of information interchange between functions _____
- A collection of tools to facilitate OO design _____
- The process of creating a protective layer around data _____
- The data associated with a class _____
- A subroutine or procedure associated with a class _____
- The name of a function designed to access data in a class _____
- The definition for a new data type _____
- The name of an instantiated class _____
- The name of a function designed to alter data in a class _____
- The process of breaking a large problem into smaller ones _____
- The extent in which a function does one thing only _____

Please see page 109 for a hint.

Problem 3

Design a class to hold the position on a chess board.

Please see page 114 for a hint.

Problem 4

Design a class to hold a date (month, day, and year).

Please see page 112 for a hint.

Problem 5

Design a class to hold a playing card. Note that there are 4 suits (♠ ♥ ♣ ♦) and 13 ranks (2-10, jack, queen, king, and ace)

Please see page 112 for a hint.

Challenge 6

Design a class to hold an integral number (number without a decimal) with a potentially infinite number of digits.

Please see page 113 for a hint.

2.1 Building a Class

Sam loves Object-Oriented programming. Unfortunately, he needs to use a language that does not support classes. What can he do? Is he forced to go back to his old procedural ways? Just as he was about to give up all hope, he remembered that you can always create a class from a structure. With relief, he continues with his project.

Objectives

By the end of this chapter, you will be able to:

- See the relationship between a structure and a class
- Appreciate how simple it is to build a class from a structure
- Know what a v-table is and how to create one
- See the necessity of the `this` pointer

Prerequisites

Before reading this chapter, please make sure you are able to:

- Create and use a structure (Chapter 1.3)
- Create and use a pointer to a function (Chapter 1.5)
- Understand the components of a UML class diagram and know what each component is used for (Chapter 2.0)

What is a v-table and why you should care

While the value of classes in programming problems is readily apparent, it is less obvious how to create one. The answer, it turns out, is actually quite simple. While a structure consists only of member variables, a class consists of member variables and member functions. If we can find a way to add functions to a structure, we would have a class. The most straightforward way to do this is through function pointers.

A **virtual function table**, also known as a **v-table**, is a structure containing function pointers to all the methods in a class. By adding a v-table as a member variable to a structure, the structure becomes a class.

This brings us to the “why do I care?” part of the chapter. Most programming languages such as C++ hid the v-table from the programmer. A programmer could go through his entire career without directly seeing or using a v-table. Why, then, would we want to learn about v-tables? The answer is subtle. All the major concepts in Object-Oriented programming (encapsulation, inheritance, and polymorphism) are directly or indirectly influenced by the v-table. If these concepts were easy to understand, then there would be little need to discuss the v-table. Unfortunately they are not! Many of their behaviors seem mysterious and are difficult to predict. However, by understanding how the v-table works behind the scenes, the mystery and unpredictability disappear.

Building a class with function pointers

Perhaps the best way to explain how to build a class is to do so by example. Consider the `Card` class from Chapter 2.0:

Card
card
set
getRank
getSuit

We will start by implementing the three methods as though they were not members of the class. Since all three methods access the member variable `card`, each method must take a `Card` as a parameter. By convention, we pass this object by-pointer rather than by reference as we normally would:

```
void set(Card * pThis, int iSuit, int iRank) // we need to pass pThis as well as iSuit
{
    pThis->card = iRank + iSuit * 13; // and iRank because pThis changes
}

int getRank(const Card * pThis) // pThis is const because getRank does not
{
    return pThis->card % 13; // change pThis
}

int getSuit(const Card * pThis) // here too pThis is const because getSuit
{
    return pThis->card / 13; // does not change pThis
}
```

Now, to add these three member functions to `Card`, we need to add three function pointers:

```
struct Card
{
    int card;
    void (*set)(Card * pThis, int iSuit, int iRank); // wow the function
    int (*getRank)(const Card * pThis); // pointer syntax
    int (*getSuit)(const Card * pThis); // is ugly!
};
```

The final step is to instantiate a `card` object. This means it will be necessary to initialize all the member variables. Unfortunately, this is a bit tedious:

```
{
    Card cardAce;
    cardAce.set = &set; // this is tedious. Every time we want to
    cardAce.getRank = &getRank; // instantiate a card object, we need to
    cardAce.getSuit = &getSuit; // hook up all these function pointers!

    cardAce.set(&cardAce, 3, 0); // calling the function is sure easy!
}
```

The full code for this example is available at [2-1-cardFunctionPtr.html](#):

```
/home/cs165/examples/2-1-cardFunctionPtr.cpp
```

Building a class with v-tables

The most tedious thing about building a class with function pointers is that, with every instantiated object, it is necessary to individually hook up all the function pointers. This can be quite a pain if the class has dozens of methods. We can alleviate much of this pain by bundling all the function pointers into a single structure. We call this structure a v-table.

A virtual method table or v-table is a list or table of methods relating to a given object (the “virtual” adjective will be explained later in the semester). By tradition, the name for this table in the data-structure is `__vptr`. Back to our `Card` example, the v-table might be:

```
struct VTableCard
{
    void (*set)(Card * pThis, int iSuit, int iRank); // exactly the same
    int (*getRank)(const Card * pThis); // function pointers
    int (*getSuit)(const Card * pThis); // as before
};
```

From here, we can create a single global instance of `VTableCard` to which all `Card` objects will point:

```
const VTableCard V_TABLE_CARD = // global const which all Card objects will point to
{
    &set, &getRank, &getSuit // here we hook up all the function pointers once,
}; // when we instantiate V_TABLE_CARD
```

Now the definition of the `Card` is much easier:

```
struct Card
{
    int card; // the single member variable
    const VTableCard * __vptr; // all the member functions are here!
};
```

So how does this change the use of the `Card` class? Well, instantiating a `Card` object becomes much easier but the syntax for accessing the member functions is much more complex:

```
{
    Card cardAce;
    cardAce.__vptr = &V_TABLE_CARD; // with this one line, all the function pointers
                                    // are connected in a single assembly command

    cardAce.__vptr->set(&cardAce, 3, 0);
}
```

Note that while `__vptr` is a member variable of `Card` and, as such, requires the dot operator. However, `__vptr` itself is a pointer. It is necessary to either dereference it with the dereference operator `*` or use the arrow operator `->` when accessing its member variables.

Sam's Corner



On the surface, the VTable implementation of an object does not appear to have a clear advantage to the simpler function pointer. Though it is smaller (only one copy of the function pointer table is required), it is slower to reference and much more inconvenient to use. These disadvantages are mitigated with classes where the syntax is greatly simplified and the compiler handles most of the housekeeping details.

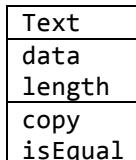
The full code for this example is available at [2-1-cardVTable.html](#):

```
/home/cs165/examples/2-1-cardVTable.cpp
```

Example 2.1 – Text with Function Pointer

This first example will demonstrate how to implement a class using function pointers. This class will have two member variables and two methods.

Create a class to implement the notion of text. Use the following UML class diagram as a guide:



The first step is to define the `Text` class with two member variables and two function pointers:

```
struct Text
{
    char data[256];
    int length;
    void (*copy)(Text * pDest, const Text * pSrc);
    bool (*isEqual)(const Text * pText1, const Text * pText2);
};
```

Note that the function pointers are not initialized. Each time a new instance of `Text` is initialized, the `copy()` and `isEqual()` function pointers will need to be assigned to the appropriate functions:

```
{
    // create two text objects
    Text text1;
    text1.copy = &textCopy;
    text1.isEqual = &textIsEqual;

    Text text2;
    text2.copy = &textCopy;
    text2.isEqual = &textIsEqual;

    // initialize text1
    cin >> text1.data;
    text1.length = strlen(text1.data);

    // copy into text2
    text2.copy(&text2, &text1);
    assert(text2.isEqual(&text2, &text1));
}
```

Observe how the member functions are referenced in exactly the same way one would reference the member functions of `cout.getline()` and `fin.fail()`: with the dot operator.

As a challenge, add the method `prompt` to the `Text` class.

```
void prompt(Text & pThis, const char * message);
```

The complete solution is available at [2-1-textFunctionPtr.html](#) or:

```
/home/cs165/examples/2-1-textFunctionPtr.cpp
```



Example 2.1 – Text with V-Table

Demo

This second example will demonstrate how to implement a class using V Tables. This class will have two member variables and two methods.

Problem

This problem will implement the same UML as the previous example. The output is:

```
Text 1: Join the Dark Side  
Text 2: Join the Dark Side  
They are the same!
```

...and...

```
Text 1: No, Luke, I am your father!  
Text 2: Nooooooooooooooo!  
They are not the same!
```

Solution

We start by building a struct called `Text` (which contains the variables, in other words, the nouns):

```
struct Text  
{  
    char data[MAX];  
    int length;  
    const VTableText * __vptr;  
};
```

The body of `VTableText` (which contained the functions, or in other words, the verbs) is:

```
struct VTableText  
{  
    void (*copy)(Text * pDest, const Text * pSrc);  
    bool (*isEqual)(const Text * pText1, const Text * pText2);  
};
```

Now this will only contain the member functions that will be in `Text`. Now to initialize `VTableText`:

```
const VTableText V_TABLE_TEXT =  
{  
    &textCopy,  
    &textisEqual  
};
```

To instantiate a `Text` object, we need to initialize the `__vptr` member variable:

```
{  
    Text text1;  
    text1.pVTable = &V_TABLE_TEXT;  
}
```

Challenge

As a challenge, add the method `prompt` to the `Text` class.

```
void prompt(Text & pThis, const char * message);
```

See Also

The complete solution is available at [2-1-textVTable.html](#) or:

```
/home/cs165/examples/2-1-textVTable.cpp
```



Problem 1 - 3

Problem 1 through 3 will be done using the function pointer method of implementing a class. Giving the following class definition:

Position
row
col
set
display

1. Define the `Position` class using a structure and function pointers.
2. Implement the `set()` and `display()` methods. Do not forget to pass the `this` pointer as the first parameter!
3. Create a variable of type `Position`, set the value to `(4, 3)`, and display the results. Assume that there are functions defined called `set(Position *, int, int)` and `display(const Position *)`.

Please see page 122 for a hint.

Problem 4 - 5

Problem 4 and 5 will be done using the v-table method of implementing a class. Giving the following class definition:

Position
row
col
set
display

4. Define the `Position` class using a structure and the `VTablePosition` structure.

5. Create a variable of type `Position`, set the value to (4, 3), and display the results. Assume there the following global constant variable is defined:

```
VTablePosition V_TABLE_POSITION =
{
    &set;
    &display;
};
```

Please see page 123 for a hint.

2.2 Class Syntax

Sue is anxious to work on her programming project but is unsure of the syntax of a class. As she shuffles through her textbook for the hundredth time, it occurs to her that she would save a ton of time by memorizing the syntax.

Objectives

By the end of this chapter, you will be able to:

- Write the code defining a class
- Be able to convert a UML class diagram into a class
- Know how to make a variable public or private

Prerequisites

Before reading this chapter, please make sure you are able to:

- Define a structure (Chapter 1.3)
- Create the UML class diagram describing a class matching a given problem definition (Chapter 2.0)

What is the syntax of a class and why you should care

It is possible to write code in C++ (or any other computer language for that matter) without memorizing the syntax of the basic constructs. This can be accomplished by looking at sample code similar to the problem you are trying to solve or by leafing through a textbook describing this syntax. While this works, it is extremely tedious and inefficient. In many ways, this is like trying to speak French in Paris without knowing the vocabulary. Standing in front of a Parisian with a French-to-English dictionary in your hand is not likely to win you any friends. Furthermore, with the language being such an obstacle, it is unlikely you will be able to communicate much more than the simplest ideas. Preach my Gospel emphasizes this:

Do not stop improving your language skills once people begin to understand you. As your ability to speak the language grows, people will listen more to what you say than to how you say it (p. 128).

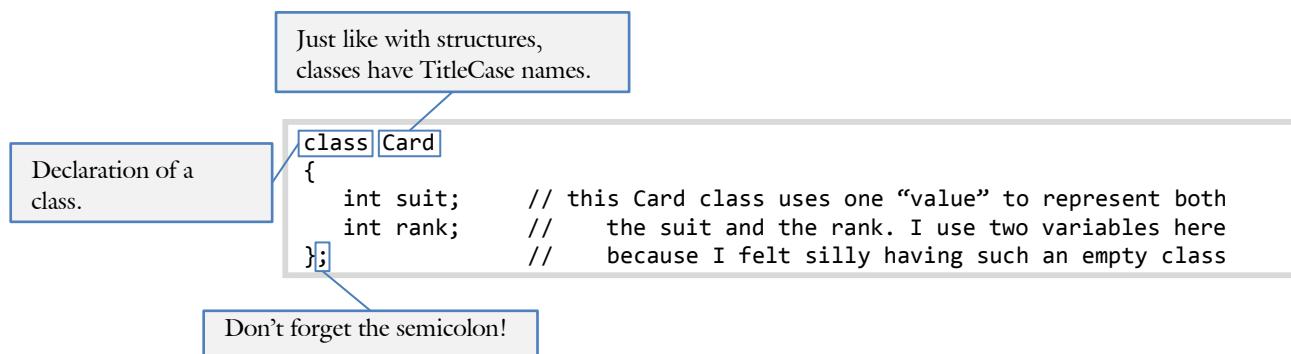
Computer languages are no different than this. When you are fluent with a computer language, you can easily translate your ideas into working code without having to focus too much attention on the syntactic details of the language. In other words, it is definitely worth your time to memorize the syntax of basic constructs such as a class.

Defining a class

There are four parts of the class syntax: the `class` tag itself, the member functions, the access modifiers, and incorporating classes into header & source files.

The class tag

The syntax of defining a class is nearly identical to that of a structure. The main difference is that we use the keyword “`class`” instead of “`struct`”.



Note that a class is named exactly the same as a structure: with a `TitleCase` label. Note that the member variables are surrounded with curly braces. Don’t forget that trailing semicolon. Just like with structures, the class definition is just a blueprint for the class. We can use this blueprint to create as many instances of the class as we like. Class instances are also called objects.

Sam's Corner

The similarity between a class and a structure are more than skin deep. There is a strong family relationship between the two constructs. Just as we build our own class from structures in Chapter 2.1, the inventors of C++ also built classes upon the foundation that structures afforded.



Unit 2

Member functions

A method can be added to a class by including the function prototype in the class definition.

```
class Card
{
    // member variables
    int value;                                // unlike the above Card class, we will
                                                // use only one variable instead of two

    // member functions a.k.a. methods
    void display();                            // use the same syntax to indicate
    void setSuit(int suit);                  // member functions as you would
    void setRank(int rank);                  // for any other prototype
    int getSuit();
    int getRank();
};
```

All these member functions can access the member variable of the class the same way they would access their own local variables. This will be discussed in more detail later in the chapter as we discuss methods.



While structures typically lack member functions, classes just about always contain a few. The C++ language chooses to implement structures and classes almost the same way. This means that it is legal to define a structure with member functions and to define a class with them. Just because it is possible does not mean you should do it! Conceptually a structure contains just member variables, nothing more! On the other hand, a class contains both member variables and member functions.

While the structure and the class clearly have different semantic meaning, C++ chooses to implement them the same. To the compiler, the *only* difference between a structure and a class is that, lacking any access modifiers, the member variables and functions of a structure are public by default while in a class they are private. In other words, syntactically speaking, the two are interchangeable. That being said, you should use a structure when there are no methods and a class when there are! It is less confusing that way.

Access modifiers

Recall the UML we defined for our card class from Chapter 2.0:

Card
- value
+ display
+ setRank
+ setSuit
+ getRank
+ getSuit
- validate

The plus and minus signs signify whether the variable or method is accessible to everyone or just to member functions. In the above example, the client can access the getters, the setters, and the display function. However only the member functions can access the member variable `value` and the method `validate()`. In C++ we implement these access modifiers with the `public` and `private` keyword:

```
class Card
{
    private:                                // "private" means that everything from here
        int value;                           //      down to the next access modifier is private

    public:                                 // "public" means that the next five methods
        void display();                     //      are public, accessible by both the client
        void setSuit(int suit);           //      and the member functions themselves
        void setRank(int rank);
        int getSuit();
        int getRank();

    private:                                // we can have more than one "private" access
        bool validate();                  //      modifier. Don't use too many, it can
                                         //      get confusing if you do
};
```

The `public` access modifier signifies to the compiler that anyone can access the member variable or member function. For example, if we create an object from the `Card` class (called “**instantiate**” a `Card`), we can access the public method `display()` but not the private method `validate()`:

```
{
    Card card;                         // instantiate a Card object

    card.display();                    // LEGAL because display() is a public method
    card.validate();                  // ERROR because validate is private
}
```

The `private` access modifier signifies only that another method can access the member variable or member function. This is an important tool helping us provide guarantees that the data contained in the member variables is always in a valid and well-formed state.

Header and source files

Recall the difficult time we had trying to divide a large procedural project into multiple files. The difficulty originated from trying to find which functions were “related enough” to justify a separate file. Fortunately this process is greatly simplified with classes.

Generally speaking, each class should be in its own file. This means that the class definition goes in the header file (where we put structure definitions and prototypes before) and the method implementation goes in the source files. This means that it is only necessary for the client to `#include` the appropriate header file and modify his `makefile` in order to use a class. This is how we build our own libraries. The header file for our card class is thus:

```
#ifndef CARD_H
#define CARD_H

#include <string> // because many of the functions take or return strings
#include <iostream> // because of the insertion and extraction operator

/*****************
 * RANK: The ordering of the cards
 *****/
#define FIRST_RANK 0
#define LAST_RANK 12
const char RANKS[] = "234567890jqka";

/*****************
 * SUIT: The ordering of the ranks
 *****/
#define FIRST_SUIT 0
#define LAST_SUIT 3
const char SUITS[] = "shcd";

// the current card is invalid
#define INVALID 255
#define NO_CARD "--"

/*****************
 * CARD
 * Card class
 *****/
class Card
{
public:
    void initialize(); // set to two of spades
    void setCard(int iSuit, int iRank); // combination of setSuit and setRank
    void display(); // display the contents of the card

    bool isValid() const { return (value == INVALID);}

private:
    // holds the value. Though there are 256 possible, only 52 are used
    unsigned char value; // internal representation

    // private functions
    bool validate() const; // are we in a valid state?
};

#endif // CARD_H
```

Defining methods

Defining a member function is much like defining any other function with the exception of how the method relates to the rest of the class. There are three parts to this syntax: specifying class membership through the scope resolution operator (::), accessing member variables, and the `this` pointer.

Specifying class membership

When implementing a method, it is necessary to indicate to the compiler which class the method is associated with. This is done with the name of the class and the **scope resolution operator** ::

```
int Card :: getSuit()
{
    return value / 13;
}
```

The same syntax is used regardless of the return type or the parameters passed to the function:

Name of the class that
“owns” the method.

Scope Resolution
Operator.

```
void Card :: setRank(int rank)
{
    value = getSuit() * 13 + rank;
}
```

Accessing member variables

There are three types of variables that are accessible from within procedural functions: global variables accessible from anywhere in the program, parameters that are accessible only from within the function but provide a conduit through which data passes between functions, and a local variable accessible only from within a function. With classes, a forth type of variable is available: member variables.

Member variables are like local variables except they are shared between all the member functions. They are instantiated when the class itself is instantiated and are destroyed when the class falls out of scope. This means that member variables have a longer life-span than the typical local variable. A member variable is accessible exactly the same as any other variable: by name:

```
bool Card :: validate()
{
    return (value >= 0 && value < 52);           // “value” is a private member variable,
}                                              //      accessible by all member functions
```



Sue's Tips

Recall from last semester (Procedural Programming in C++, Chapter 1.4) that it is desirable to minimize the scope of a variable. The longer the variable is alive, the greater the chance it will cause a bug. Since member variables have greater scope than local variables (because they are accessible by more than one function), they are a potential source of bugs.

Never use a member variable when a local variable or parameter will do the job.

Member variables should only be used when it is essential to the purpose and identity of the class. Everything else should be a local variable or a parameter.

this

The final component of the syntax of a method is the `this` pointer. Recall from Chapter 2.1 how it is necessary to pass a reference or a pointer to a structure if the function is to change the structure.

```
void set(Card * pThis, int iSuit, int iRank) // we need to pass pThis as well as iSuit
{
    pThis->value = iRank + iSuit * 13;
}
```

We don't need to pass the reference with a member function:

```
void Card :: set(int iSuit, int iRank)          // no Card pointer is passed because we
{
    value = iRank + iSuit * 13;                  // have access to the member variables
}
```

The way this works is a bit insidious. When a function is associated with a class through the scope resolution operator (namely the “`Card ::`” part of the method syntax), the compiler inserts another hidden parameter. The hidden parameter is a pointer to the class and it is called `this`. We can always access a member variable or a member function from a method in a class by using the `this` pointer:

```
void Card :: set(int iSuit, int iRank)          // though we don't see it, there is a
{
    this->value = iRank + iSuit * 13;           // hidden parameter called "this"
                                                // works exactly the same as "pThis" in
                                                // the top example
```

So why exactly is `this` necessary (besides the obvious opportunity for word-play)? Consider the following example:

```
class Temperature
{
public:
    bool set(int temp);
    int get();
private:
    int temp;
};
```

In the method `Temperature::set()`, notice how the parameter `temp` has the same name as the member variable `temp`. How are we to keep from getting them mixed-up? We could give them separate names but that would be silly; if two variables mean exactly the same thing they should have the same name. The name collision problem disappears with the `this` pointer.

```
bool Temperature :: set(int temp)          // same name as the member variable!
{
    // check for absolute zero
    if (temp > -273)                      // in this case, temp is the parameter
    {
        this->temp = temp;                // because it has the innermost scope
        return true;                     // this->temp is the member variable,
    }                                     // temp is the local variable
    return false;
}
```

Sam's Corner

You don't need to know how member variables are passed into the member functions nor how the `this` pointer is passed as a parameter in order to use it correctly. That being said, being aware of this hidden parameter sure makes understanding `this` much easier.



Using a class

The whole point of creating new data types and encapsulating code into classes is to make the job of the programmer easier. To see if C++'s implementation of classes accomplishes this goal, let's try to use our new Card class.

```
#include "card.h"                                // get access to the Card class through
                                                //   including the card header file

//****************************************************************************
* MAIN: a simple driver program for our card class
*****
int main()
{
    Card card;                                // how many member variables are used here?
                                                //   How is the playing card stored?
                                                //   Frankly, the client here does not care!

    // we need to initialize the card
    card.initialize();                         // call the method with the dot operator

    cout << "Instructions: keep prompting the user for a suit and rank\n"
        << "                      until the suit is specified as 0.\n";

    for (;;)                                // forever. We will return when done.  We usually do
    {                                       //   this only in driver programs
        //prompt for suit
        char chSuit;                          // this simple driver program will just
        cout << "Suit: ";                   //   get user input in the most
        cin  >> chSuit;                    //   convenient way so we can exercise
        if (chSuit == '0')                  //   the methods of Card
            return 0;                       // it is usually better to not return out of main()
                                                //   like this, but in a driver program...
        // prompt for rank
        char chRank;                         // 
        cout << "Rank: ";
        cin  >> chRank;

        // set the card
        card.setCard(card.iSuit(chSuit),
                     card.iRank(chRank));

        // display the results
        cout << endl;                      // notice how we are able to call the
        card.display();                    //   display function without passing any
        cout << endl;                      //   parameters.  The data is silently
    }                                         //   passed through the hidden "this"

    return 0;
}
```

The complete solution for the card class, including the header file, source file, driver program, and `makefile`, are available at [2-2-card.html](#) or:

```
/home/cs165/examples/2-2-card/
```

Instantiating Classes

Remember, we can use our `Car` class definition to create as many `Car` objects as we like, each of which will act independently from one another, containing their own copies of the member variables and member functions:

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

/************************************************
 * CAR: the definition of a car - just the model and the cost for now.
 *****/
class Car
{
    private:
        string model;                      // the model name, can be any string
        int    cost;                        // the cost rounded to the nearest dollar

    public:
        void setModel(const string & model)   // this function only serves to set the
        {                                     //     model member variable of the class
            this->model = model;
        }

        void setCost(int cost)               // this function will set the cost
        {
            this->cost = cost;
        }

        void display()                     // display a summary of the car
        {
            cout << "This " << model           //     in a nice, easy-to-read format
            << " is worth $" << cost << endl;
        }
};

/************************************************
 * MAIN: a simple driver program for our Car class
 *****/
int main()
{
    // I will buy a minivan for $30,000
    Car familyCar;
    familyCar.setModel("Honda Odyssey");
    familyCar.setCost(30000);

    // If it was only this easy to own a Ferrari!
    Car sportsCar;
    sportsCar.setModel("Ferrari 458");
    sportsCar.setCost(260000);

    // Show off your new wheels to some friends. Which will they like more?
    familyCar.display();
    sportsCar.display();
    return 0;
}
```

Even though the `familyCar` and `raceCar` objects are both instances of the `Car` class, their variables are independent, so calling the `display()` function on the `familyCar` object provides a different result than calling the `display()` function on the `raceCar` object.

Example 2.2 – Time

This example will demonstrate how to create a simple class with public and private methods as well as private member variables.

Demo

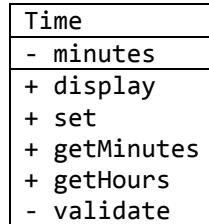
Problem

Solution

Challenge

See Also

Write a class to represent the notion of time to match the following UML class diagram:



First, the class definition matching exactly the UML description:

```
class Time
{
    public:
        void set(int hours = 0, int minutes = 0);
        void display();
        int getMinutes();
        int getHours();

    private:
        bool validate();
        int minutes;
};
```

The `getMinutes()` function needs to convert the internal data representation (minutes since midnight) to the properties (minutes since the beginning of the hour):

```
int Time :: getMinutes()
{
    // paranoia...
    assert(validate());

    return minutes % 60;
}
```

Finally we have a driver program to test the new `Time` class:

```
{
    Time time;
    Time.set(3, 5);
    Time.display();
}
```

As a challenge, modify the class so it also stores seconds. This means that the internal data representation will be “seconds since midnight” rather than “hours since midnight.”

The complete solution is available at [2-2-time.html](#) or:

```
/home/cs165/examples/2-2-time.cpp
```



Problem 1 – 5

Given the following UML class diagram:

Complex
- real
- imaginary
+ set
+ display

1. Write the class definition to match the above UML class diagram. Leave some extra space for problem 2 below.
2. Turn the above class definition into a header file with the appropriate `#ifdefs`.
3. Implement the `display()` method.
4. Implement the `set()` method.
5. Write a program to instantiate a `Complex` object, initialize it to the value “`4.5 + 3.6i`”, and display the results on the screen.

Please see page 134 for a hint.

2.3 Accessors & Mutators

Sue can see the value of designing class interfaces with the properties the client has in mind. However, she is very concerned about the resulting performance penalty. If a function call needs to be made every time the data from a class is accessed, won't this be a huge drain on the entire program? While mulling over this dilemma, Sam stops by and reminds Sue of the `inline` construct. "Yes!" she says, "that will definitely solve this problem..."

Objectives

By the end of this chapter, you will be able to:

- Make setters and getters inline to improve program performance
- Make getters constant to reduce the chance of accidentally modifying a member variable

Prerequisites

Before reading this chapter, please make sure you are able to:

- Recite and explain the four design rules for encapsulation (Chapter 2.0)
- Convert a UML class diagram into a class definition (Chapter 2.2)
- Implement member functions (Chapter 2.2)
- Use the `this` pointer to access member variables and methods (Chapter 2.2)
- Make a function inline using the `inline` keyword (Chapter 1.5)

What are accessors and mutators and why you should care

An **accessor** is a method providing the client access to data from member variables. Accessors are also commonly known as "getters" because the function names tend to be variations of `get()`. A `Date` class, for example, might have three accessors: `getDay()`, `getMonth()`, and `getYear()`.

A **mutator** is a method enabling the client to modify or change the data stored in member variables. Mutators are also commonly known as "setters" because the function names tend to be variations of `set()`. A `Date` class, for example, might have four mutators: `set()` taking three parameters, `setDay()`, `setMonth()`, and `setYear()`.

Just about every class you use or write will have some variation of a getter or setter. This should not be surprising: most classes store data and the client will want to have access to the data or change it. Because they are so important, it behooves us to learn how to make our getters and setters as safe and efficient as possible. That is the purpose of this chapter.

There are two tools we commonly use in getters and setters to accomplish these goals: making them `inline` and making them `constant`.

Inline

As you may have noticed, most getters and setters tend to be trivial and consist of just a couple lines of code. This makes them great candidates for being `inline` methods. Recall that making a function inline is a signal to the compiler to make a special optimization. Instead of the function being called in the normal way, the compiler copies the function body code directly into that of the caller. While the behavior remains identical, performance is improved.

We can make a method inline though including the `inline` keyword just as we did with a standard function:

```
inline int Card :: getRank() // the inline keyword makes this method
{                           //     inline as it would for any other
    return value % 13;      //     function
}
```

Remember that inline functions need to go in the header file. This is true whether the inline function is a method or a stand-alone function.

It is so common to want to make a method inline (probably because so many methods are trivial) that C++ has made an easier way to do this. We can provide the body of a member function directly in the class definition, making the method inline.

```
class Card
{
public:
    void display(); // not inline because it is not trivial

    // CARD :: GET RANK // use an abbreviated comment block.
    int getRank() // traditionally, getters are even simpler
    {           // than setters because they don't
        return value % 13; // perform any validation
    }

    // CARD :: SET RANK // use an abbreviated comment block.
    void setRank(int rank) // because setRank() is a trivial function
    {                   // consisting of just one statement,
        value = rank + getSuit() * 13; // it is a great candidate for inline
    }

    ... code removed for brevity ...

private:
    bool validate(); // not inline because it is not trivial
    int value;       // member variables are unchanged
};
```

The inline keyword must be missing! If the method is defined in the class, it is inline automatically

The “Card ::” is not used in the class definition



Sue’s Tips

It is not uncommon to have a class consisting completely of inline methods, making it unnecessary to write a source file: everything is in the header!

Constant methods

As we discussed in Chapter 2.2, member variables have much larger scopes than local variables. Local variables are limited to the function in which they are defined but member variables are accessible to all the methods in the class. This is a potential source of bugs: the more accessible a variable is the larger the chance that it will be misused. If, for example, a local variable has an unexpected value that results in a bug, the programmer need only look in the body of the function to find the source. However, if offending variable is a member variable, any of the methods in the class could be to blame.

Wouldn't it be great if we could restrict access to member variables, giving fewer methods access to make changes? That mechanism exists with the `const` method modifier.

When the `const` modifier is applied to a variable, the compiler provides guarantees that the variable will not be changed. When the `const` modifier is applied to a method, the compiler provides guarantees that the member variables will not be changed in the method.

```
class Card
{
    ... code removed for brevity ...
    int getSuit() const // because this method has the const modifier,
    {                  // it is illegal for it to change the
        return value / 13; // member variable "value"
    }
    ... code removed for brevity ...
    private:
        int value;
};
```

The `const` modifier makes it illegal to change member variables, such as “`value`”.

If the programmer made a mistake and attempted to change `value` in the above function, the following compiler error would result:

```
In file included from cardTest.cpp:8:
card.h: In member function “int Card::getSuit() const”:
card.h:50: error: assignment of data-member “Card::value” in read-only structure
```

Warning!

The `const` modifier is a two-edged sword. It is a great help for finding bugs because the compiler will point to an unintended assignment (such as “`if (value = 0)`” where “`if (value == 0)`” was intended). However, it can also be really annoying. Consider the following function:

```
... code removed for brevity ...
bool Card :: validate();           // NOT const...
int getSuit() const               // const so we cannot change value
{
    assert(validate());           // because validate() is not const, we cannot
    return value / 13;            // guarantee that "value" isn't changed in
}                                // getSuit(). This will be a compile error
... code removed for brevity ...
```

We get a very cryptic error message:

```
In file included from cardTest.cpp:8:
card.h: In member function “int Card::getSuit() const”:
card.h:50: error: passing “const Card” as “this” argument of “bool Card::validate()”
discards qualifiers
```

The problem here is that `validate()` is not a `const` method! Thus using `const` is an all-or-nothing proposition. A `const` method cannot call a method in the class that is not `const`.

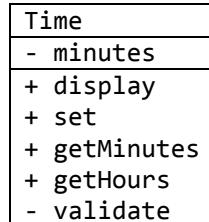
Example 2.3 – Time

Demo

This example will demonstrate how to use the `const` and `inline` modifiers in a simple class definition. Most getters and setters should follow this pattern.

Problem

With the same UML diagram we used from Example 2.2, implement the getters and setters as `inline` and `const` as appropriate.



Solution

```
class Time
{
public:
    // TIME :: SET : Set the time according to user input
    void set(int hours = 0, int minutes = 0)
    {
        // first just trust them
        this->minutes = hours * 60 + minutes;

        // set to midnight if invalid
        if (!validate())
            this->minutes = 0;
    }
    ... code removed for brevity...
    // TIME :: GET MINUTES : Fetch the minutes since midnight
    int getMinutes() const
    {
        // more paranoia...
        assert(validate());
        return minutes % 60;
    }
    ... code removed for brevity ...
    // TIME :: VALIDATE : Ensure that the member variable is set correctly
    bool validate() const           // this must be const or it cannot be called by
                                    //      getMinutes() which is const!
    {
        return (minutes >= 0 && minutes < 24 * 60);
    }
};
```

Challenge

As a challenge, modify the class so it also stores seconds as we did in Chapter 2.2. This means that the internal data representation will be “seconds since midnight” rather than “hours since midnight.”

See Also

The complete solution is available at [2-3-time.html](#) or:

/home/cs165/examples/2-3-time.cpp



Example 2.3 – Card

Demo

This example will demonstrate how to use the `const` and `inline` modifiers in a more complex class definition.

Problem

Write a class to represent a playing card. Include all the convenient getters and setters that the client may need to use this class.

The most interesting code for this problem is in the class definition in the `card.h` header.

```
#ifndef CARD_H
#define CARD_H
... code removed for brevity...
class Card
{
public:
    // initialize
    void initialize() { value = 0; } // trivial functions can be on one line

    // getters and setters
    bool setSuit(char chSuit);
    bool setRank(char chRank);

    // CARD :: GET SUIT fetch the suit {s,h,c,d}
    char getSuit() const
    {
        // paranoia
        assert(this->validate());
        return SUITS[value / 13];
    }

    // CARD :: GET RANK fetch the rank {2,3,4,...j,q,k,a}
    char getRank() const
    {
        // paranoia
        assert(this->validate());
        return RANKS[value % 13];
    }

    bool isValid() const { return value == INVALID; }
    void display();

private:
    // holds the value. Though there are 256 possible, only 52 are used
    unsigned char value;           //internal representation

};

... code removed for brevity...
#endif // CARD_H
```

Solution

The complete solution is available at [2-3-card.html](#) or:

/home/cs165/examples/2-3-card/



See Also

Review 1

Given the following class:

```
class Coordinate
{
    public:
        int x;
    private:
        int y;
};

int main()
{
    Coordinate coord;

    coord.y = 3;
    cout << coord.y << endl;

    return 0;
}
```

What is the output?

Please see page 128 for a hint.

Review 2

In the following class:

```
class Number
{
public:
    void set(int num)
    {
        <code goes here>
    }
private:
    int num;
};
```

What code will complete `set()`?

Please see page 131 for a hint.

Problem 3 - 6

All four of these problems are to solve the same problem: create a class to hold an individual's grade on a test: (0 → 100% or F → A+). Note that the properties must be both number and letter grades. It is up to the author of the class to determine the best data representation.

3. Create the UML class diagram to describe a class to hold an individual's grade

4. Write the class definition to match the above UML class diagram

5. Implement the method `set()` from your class definition

6. Write a driver program to exercise your `Grade` class

Please see page 139 for a hint.

2.4 Constructors & Destructors

Sue is working on her `Date` class but seems a bit confused. If the client does not call her `initialize()` function, the member variables in her class will remain uninitialized. What can she do to guarantee her `initialize()` function gets called? As she ponders this point, she stumbles upon constructors in the reading...

Objectives

By the end of this chapter, you will be able to:

- Describe situations when a constructor would be helpful in a class design
- Create a class with a default constructor, non-default constructor, and a copy constructor
- Create a destructor and describe when one might be used
- Be able to predict which constructors are called in a given situation

Prerequisites

Before reading this chapter, please make sure you are able to:

- Create a class with member functions (Chapter 2.2)
- Use accessors and mutators in a class definition (Chapter 2.3)
- Explain the differences between pass-by-value, pass-by-pointer, and pass-by-reference (Procedural Programming in C++ Chapter 1.4, 3.3)

What are constructors and why you should care

A **constructor** is a method in a class that is guaranteed to be called when an object is instantiated. A **destructor** is a method in a class that is guaranteed to be called when an object is destroyed such as when it falls out of scope. These two methods are essential to the principle of encapsulation.

If a class designer is truly going to absolve the client from having to know anything about the implementation details of a class, then there must be some way to make such housecleaning chores as initializing variables invisible. Back to our `Date` class from Chapter 2.3, if the client fails to call `initialize()` then nothing will work!

There are several scenarios where constructors and destructors are very convenient. The need to initialize member variables necessitates constructors for almost every class. Classes allocating and freeing memory usually need both constructors and destructors to prevent memory leaks. Tools initiating complex operations such as setting up and tearing down graphics windows use constructors and destructors to ensure the client does not forget important procedures. There are so many uses for constructors and destructors that it is difficult to think of an application that would not benefit from them.

Constructor

A constructor is a special method in a class that is guaranteed to be called when an object is instantiated. It is unique because it has the following properties:

- **Name:** Constructors must have the same name as the class in which they are a member
- **Return value:** Constructors have no return type, not even `void`!

When an object is instantiated, the compiler selects the most appropriate constructor defined in the class to be called. The three types of constructors are the default constructor, non-default constructors, and the copy constructor.

Default constructor

The default constructor is the constructor lacking any parameters. Back to our `Card` class, the default constructor might be:

```
class Card
{
public:
    Card() // default constructor
    {
        value = 0;
        assert(validate()); // make sure our Card is well formed
    }
... code removed for brevity ...;
```

Default Constructors are declared with no return type and no parameters.

Constructors can call functions or methods just like any other member function.

Observe how the method takes on the name of its class, that there is no return type, and this constructor takes no parameters.

Constructors have other syntactic nicety that the C++ language affords us: the **initialization section**. It provides a special way to initialize member variables outside the function body. The initialization section affords us this convenience; just list the member variables after the function name and specify the default value:

```
class Card
{
public:
    Card() : value(0) // this constructor is inline
    {
        assert(validate());
    }
... code removed for brevity ...;
```

Variable to be initialized and the value to be assigned to it.

The variables in the initialization section get set immediately before the code in the body of the function is executed. Observe in the above example that `validate()` checks `value`. Clearly `value` must be set to zero first or our assert will fire.

Of course we do not need to make our constructors inline. The following will work as well:

```
Card :: Card() : value(0) // non-inline constructors can also have
{                         // initialization sections. The functionality
    assert(validate());    // is exactly the same as the non-inline variant
}
```

Non-default constructors

Frequently we wish to instantiate an object and initialize it in one step. Consider, for example, the `ifstream` class. We can either create a `fin` object first and then attach it to a file...

```
{
    ifstream fin;           // instantiate the fin object
    fin.open("data.txt");   // initialize it in a separate step
    ...
}
```

... or we could instantiate `fin` and attach it to a file at the same time...

```
{
    ifstream fin("data.txt"); // instantiate and initialize all at once
    ...
}
```

How is this done? The answer is that a special constructor was created for `ifstream` that takes a c-string as a parameter. This is a non-default constructor.

A non-default constructor is a constructor that takes a parameter. The parameter can be really anything, though it is usually a good idea to make it related to how the client would want to initialize an object. Back to our `Card` class, one default constructor might be to set the suit and the rank.

```
Card :: Card(int iSuit, int iRank)
{
    // make sure we are in the valid range
    if (iSuit >= (sizeof(SUITS) / sizeof(SUITS[0])) || iSuit < 0)
        iSuit = 0;
    if (iRank >= (sizeof(RANKS) / sizeof(RANKS[0])) || iRank < 0)
        iRank = 0;

    // assign
    value = iSuit * 13 + iRank;

    // paranoia
    assert(validate());
}
```

We can have as many non-default constructors as we choose. Another might take a string as a parameter.

```
Card :: Card(const string & s) : value(0)
{
    parse(s);
    assert(validate());
}
```

The client can create an object using a non-default constructor as you might expect:

```
{
    // call the "int, int" constructor
    Card sixHearts(1, 4);           // iSuit == 1 which is a Heart
    ...                                // iRank == 4 which is the six

    // call the "const string by-reference" constructor
    string s("8c");                 // eight of clubs
    Card eightClubs(s);             // call with string("8c") as a parameter
    ...
}
```

Copy constructor

A copy-constructor is a special constructor designed to make an exact copy of an object. Every copy-constructor must have the following properties:

- It must be a constructor.
- It takes exactly one parameter.
- The parameter must be the same data type as the class itself
- The parameter must be constant and by reference.

Thus there can be no more than one copy constructor for a given class. Back to our `Card` example, the declaration of the copy constructor would be:

```
class Card
{
    This prevents values in the parameters from changing.     The object to be copied is passed as a parameter by reference.
    struct
        Card(const std::string & s);           // non default constructor - const string reference
        Card(int iSuit, int iRank);           // non default constructor - int, int
        Card(const Card & rhs);             // copy constructor
    ... code removed for brevity ...
};
```

The complete solution for this example is available at [2-4-card.html](#):

```
/home/cs165/examples/2-4-card.cpp
```

Observe how it accepts as a parameter a constant `Card` by reference. The implementation would be:

```
Card :: Card(const Card & rhs)
{
    assert(rhs.validate());           // call the validate method of the "rhs" parameter
    value = rhs.value;              // do the actual work
    assert(validate());             // call the validate method of "this"
}
```

By convention, we call the parameter “rhs” which stands for “Right Hand Side.” The reason for this will become apparent in Chapter 2.6 through 2.8. There are several scenarios when the copy constructor is called. The first is when the client wishes to make a copy of a given object:

```
{
    Card card1(string("qh"));       // create a queen of hearts with the non-default
                                    // constructor: const string reference
    Card card2(card1);            // create a copy!
```

Another time the copy constructor is called is when an object is passed by-value. Recall that pass-by-value makes a copy of the parameter. How does the compiler know how to make a copy of an object? With the copy constructor of course!

```
void function(Card card)          // make a copy of Card. Are you sure you want to
{                                //      do that? This should be const by-reference
    ...
}
```

The final way the copy constructor is called is when an object is returned by-value:

```
Card makeCard(Card & card)        // it would be better to return a reference here:
{                                //      Card & makeCard(Card & card)
    ...
    return card;                // return-by-value so we make copy is made here.
}                                //      This should be return-by-reference instead.
```

Using constructors

As mentioned previously, constructors get called automatically when an object is instantiated. When no constructor is present, none gets called:

```
class Simple
{
public:
    void display() { cout << value << endl; }
private:
    int value;
};

int main()
{
    Simple s;           // no constructor is present so none gets called
    s.display();        // s.value is not initialized.
}
```

In other words, a default constructor is not created for classes lacking one and the object is instantiated without a constructor being called. However, if a constructor does exist for a class but the client does not specify one, an error will occur. Consider the case where a non-default constructor exists for our `Simple` class. Note that there is no default constructor:

```
class Simple
{
public:
    Simple(int value) : value(value) {} // non-default constructor
private:
    int value;
};
int main()
{
    Simple s;           // ERROR: no default constructor
}
```

In this example, we are trying to summon the default constructor but none exists, yielding a compile error:

```
2-4-errors.cpp: In function “int main()”:
2-4-errors.cpp:22: error: no matching function for call to “Simple::Simple()”
2-4-errors.cpp:12: note: candidates are: Simple::Simple(int)
2-4-errors.cpp:10: note:           Simple::Simple(const Simple&)
```

Notice that, though only one constructor was provided (the non-default constructor taking an integer as a parameter), two are candidates. This is because, lacking a copy constructor, one is created that performs a simple copy of member variables. This may be what you want. However, if there is a dynamically allocated buffer in the class, a bug will likely result:

```
class String
{
private:
    char * buffer;      // Warning: dynamically allocated array
... code removed for brevity ...
};
```

Instead of creating a new buffer, it will instead just copy the address. Thus both the new object and the old will refer to the same buffer. This means that changes made to one object will also be reflected in the other. You must always write a copy constructor for classes with dynamically allocated memory.

The complete solution is available at [2-2-errors.html](#) or:

```
/home/cs165/examples/2-2-errors/
```

Destructor

While constructors are special methods that are called when an object is created, a destructor is a method that is called when an object is destroyed. Unlike with constructors, however, there can never be more than one destructor. Destructors have the following properties:

- The name is a combination of the class name with a tilde ~ on the front.
- There is no return type, not even void.
- There are no parameters.

Destructors are commonly used to free allocated memory, close files, terminate communications with external libraries such as graphic canvases, and a host of other things. None of these really apply to our `Card` class so the following destructor is a bit contrived:

```
class Card
{
public:
    // various constructors
    Card();                                // default constructor
    Card(const std::string & s);            // non default constructor - const string reference
    Card(int iSuit, int iRank);             // non default constructor - int, int
    Card(const Card & rhs);                // copy constructor

    // destructor
    ~Card()                                 // never more than one destructor
    {
        assert(validate());
    }
    ... code removed for brevity ...
};
```

Of course we could also define the destructor as a non-inline method:

```
Card :: ~Card()                         // notice the ~ before the name
{
    assert(validate());                  // any code can go here, even a function call
                                         // we can't return anything because no return type
```

Example 2.4 – Silly

Demo This example will demonstrate how to predict the output of code involving creating objects from classes with constructors.

Problem

Consider the following class:

```
class Silly
{
public:
    Silly() { cout << "Default constructor\n"; }
    Silly(const Silly & s) { cout << "Copy constructor\n"; }
    ~Silly() { cout << "Destructor\n"; }
};
```

What is the output of the following code:

```
int main()
{
    Silly s1; // line 1
    Silly s2(s1); // line 2
    return 0; // line 3
}
```

Solution

In Line 1, the default constructor is called because there are no parameters specified beside the `s1` variable. This means the following output will result:

Default constructor

In Line 2, we are creating another `Silly` object. This one will use the copy constructor because `s1` is passed to the constructor as a parameter. Therefore, the copy constructor will be called:

Copy Constructor

In Line 3, we leave the `main()` function and terminate the program. This means that the destructor for both `s1` and `s2` will be called:

Destructor
Destructor

See Also

The complete solution is available at [2-4-silly.html](#) or:

/home/cs165/examples/2-4-silly.cpp

Example 2.4 – Silly with Functions

Demo

This example will demonstrate how to predict the output of code involving creating objects from classes with constructors when functions are called.

Problem

Consider the following class:

```
class Silly
{
public:
    Silly() { cout << "Default constructor\n"; }
    Silly(const Silly & s) { cout << "Copy constructor\n"; }
    ~Silly() { cout << "Destructor\n"; }
};
```

What is the output of the following code:

```
Silly function(Silly s)
{
    cout << "Function\n";           // function line 1
    return s;                      // function line 2
}

int main()
{
    Silly s;                      // main line 1
    function(s);                  // main line 2
    return 0;                      // main line 3
}
```

In “main line 1,” the default constructor is called because there are no parameters specified:

Default constructor

In “main line 2,” `function()` is called passing `s` as a by-value parameter. This calls the copy-constructor:

Copy Constructor

In “function line 1” we encounter the `cout` line in `function()`.

Function

In “function line 2,” because `function()` is return-by-value, we need to create another copy of `s`. Additionally, the parameter that was pass-by-value needs to be destroyed.

Copy constructor
Destructor

We are back to “main line 2” where the return-by-value object needs to be destroyed.

Destructor

In “main line 3,” the object `s` needs to be destroyed because we are exiting the function `main()`:

Destructor

The complete solution is available at [2-4-sillyWithFunctions.html](#) or:

/home/cs165/examples/2-4-sillyWithFunctions.cpp

See Also

Example 2.4 – Time

Demo

This example will demonstrate how to use the default constructor, a non-default constructor, and the copy constructor in our `Time` class from previous chapters.

Problem

Modify the `Time` class from “Example 2.3 – Time” on page 139 to include a default constructor that sets the time to midnight, a non-default constructor taking hours and minutes as a parameter, and a copy constructor.

Solution

The class definition describing the three constructors:

```
class Time
{
public:
    // Constructors
    Time() : minutes(0) {}                      // default constructor
    Time(int hours, int minutes = 0)             // non-default constructor
    {
        set(hours, minutes);
    }
    Time(const Time & rhs)                      // copy constructor
    {
        assert(rhs.validate());
        minutes = rhs.minutes;
        assert(validate());
    }
    ... code removed for brevity ...
};
```

The driver program is:

```
int main()
{
    // exercise the default constructor
    Time time1;
    cout << "Time1 is midnight - ";
    time1.display();
    cout << endl;

    // the non-default constructor
    Time time2(10 /*hours*/, 11 /*minutes*/);
    cout << "Time2 is in the morning - ";
    time2.display();
    cout << endl;

    // the copy constructor
    Time time3(time2);
    cout << "Time3 is the same as time2 - ";
    time3.display();
    cout << endl;

    return 0;
}
```

See Also

The complete solution is available at [2-4-time.html](#) or:

b/home/cs165/examples/2-4-time.cpp



Example 2.4 – Write

Demo This example will demonstrate how to use constructors and destructors for something other than initializing variables.

Problem Create a class to write text to a file. The client of this class should not have to think about opening or closing a file; it should happen automatically.

Solution The header file is the following:

```
#ifndef WRITE_H
#define WRITE_H

#include <fstream>           // necessary for the ofstream object
#include <string>             // necessary for the string parameters

class Write
{
public:
    Write() : isOpen(false) { }
    Write(const std::string & fileName);
    ~Write();
    void writeToFile(const std::string & text);
private:
    bool isOpen;               // did we successfully open the file?
    std::ofstream fout;        // the file stream object
};

#endif // _WRITE_H
```

Notice how the constructors and destructors take care of opening and closing the file. The implementation file is the following:

```
#include "write.h"
using namespace std;

Write::Write(const string & fileName) : isOpen(false)
{
    fout.open(fileName.c_str());
    isOpen = !(fout.fail());
}

void Write::writeToFile(const string & text)
{
    if (isOpen)
        fout << text << endl;
}

Write::~Write()
{
    if (isOpen)
        fout.close();
}
```

See Also The complete solution is available at [2-4-write.html](#) or:

```
/home/cs165/examples/2-4-write/
```



Problem 1

Given the following code:

```
class Silly
{
public:
    Silly() { cout << "Default constructor\n"; }
    Silly(const Silly & s) { cout << "Copy constructor\n"; }
    ~Silly() { cout << "Destructor\n"; }
    void method() { cout << "Method\n"; }
};

int main()
{
    Silly s1;
    if (true)
    {
        Silly s2(s1);
        s2.method();
    }
    s1.method();
    return 0;
}
```

What is the output?

Please see page 149 for a hint.

Problem 2

Given the following code:

```
class Silly
{
public:
    Silly() { cout << "Default constructor\n"; }
    Silly(const Silly & s) { cout << "Copy constructor\n"; }
    ~Silly() { cout << "Destructor\n"; }
    void method() { cout << "Method\n"; }
};

Silly function(Silly & s)
{
    s.method();
    return s;
}

int main()
{
    Silly s1;
    s1.method();
    function(s1);
    return 0;
}
```

What is the output?

Please see page 150 for a hint.

Problem 3 & 4

For the following class diagram:

Temp
- degrees
+ Temp
+ get
+ set
+ display

3. Write the header file for the `Temp` class. Ensure the temperature is not below -273°C

4. Write the source file for the `Temp` class that implements any of the methods not defined as inline in the class definition.

Please see page 151 for a hint.

2.5 Static

Sam is writing a program to keep track of the employees in a medium sized company. He decides to use a class to represent an employee. One property of this class will be the employee number. It doesn't matter what the number is for each employee, it is just important that each employee has a unique number. How can he do that? As he mulls over the problem for a while, he remembers reading something about the `static` modifier...

Objectives

By the end of this chapter, you will be able to:

- Explain what the `static` modifier does and when it can be useful
- Use `static` to solve programming problems in classes and in functions

Prerequisites

Before reading this chapter, please make sure you are able to:

- Describe the difference between a global, local, and member variable (Procedural Programming in C++ Chapter 1.2 and 1.4)
- Create a class definition to match a UML class diagram (Chapter 2.2)
- Articulate the difference between public and private member variables (Chapter 2.2)

What is static and why you should care

The `static` keyword is a modifier attached to a variable to indicate that only one copy of the variable will exist in a program. Perhaps this is best explained by example. Consider a function with a single integer local variable. The first time the function is called, the variable is uninitialized until a value is assigned. The second time the function is called, it is again uninitialized until a value is assigned. If, on the other hand, the `static` keyword is used, everything changes. The second time the function is called, the variable will remember the value from the first time it was called. This is because, no matter how many times the function is called, only one copy of the variable exists in memory. All instances of the function share the same variable.

While `static` is a useful programming tool, it is infrequently used. The reason for this is twofold: first a programmer rarely encounters the situation when `static` would benefit him, and second there are other ways to accomplish the same thing. That being said, a member variable made `static` is a far more useful construct than a `static` local variable. The situation often arises when all the objects created from a given class need to share the same member variable. The `static` modifier makes this possible.

Static local variables

A local variable can be made static by the use of the `static` keyword:

```
void display()
{
    static int count = 0;                      // count is initialize to zero only
                                                //      the first time display() is called
    cout << "This function has been called "
        << ++count << " times\n";           // every time display() is called, we
                                                //      will add one to count
}
```

So how does this work exactly? When the program begins execution, space is reserved for the static local variables. This space stays reserved until the program exits. Therefore, unlike a local variable that is created when the function is called and destroyed when the variable falls out of scope, static variables are “alive” the entire length of the program execution. However, unlike a global variable, they are only accessible from within the function in which they were defined.

There are a few instances when `static` might come in handy. One is for performance reasons. Since static variables only get initialized once, we can save the initialization cost by making it `static`.

```
bool playAgain()
{
    static string input;                      // because this is static, we will only
                                                //      initialize this variable once.
    cout << "Do you want to play again? ";     // 
    cin >> input;
    return (input == "yes");
}
```

Static can also be used to keep track of how the function was previously used.

```
int getScore()
{
    // fetch the current score
    int score;
    cout << "What is your score? ";
    cin >> score;

    // is this the highest score yet?
    static maxScore = 0;                      // only this function cares what the
                                                //      highest current score is. Why
    if (score > maxScore)                    //      would we want to declare it in
    {                                         //      the caller function?
        cout << "Highest score yet!\n";
        maxScore = score;
    }
    return score;
}
```



Sue's Tips

One interesting thing about `static` local variables is that they automatically get initialized to zero even if the programmer neglects to explicitly do it. However, while you can depend on this initialization, it is unwise to do so. It is far better to be clear about your intentions and initialize it yourself.

Static member variables

Member variables can also be made `static`. If a class has a member variable that is made `static`, all objects made from that class will share that static member variable. The UML class diagram for a static is underline:

ClassName
+ <u>publicStaticMemberVariable</u>
+ publicMemberVariable
- <u>privateStaticMemberVariable</u>
- privateMemberVariable
+ publicMethod
- privateMethod

However, unlike their local variable cousins, declaration and initialization must occur separately.

Declaring static member variables

In the following example, we will create a class that has a member variable called `temp` and a static member variable called `highest`. Every object made from this class will have its own `temp` member variable, but they will all share the `highest` member variable.

```
class Temperature
{
    public:
        Temperature() : temp(0.0) { }
        void set(float temp)
        {
            if (temp > this->temp)
                highest = temp;
            this->temp = temp;
        }
        float get() const { return temp; }
        float getHighest() const { return highest; }
    private:
        float temp;                                // "temp" is not shared with
                                                    // other Temperature objects
        static float highest;                         // "highest" is static so it is
                                                    // shared with other objects
};
```

The size of an object is computed by summing the size of all the member variables. At first glance, it may appear that `sizeof(Temperature) == 8` because each of the two `floats` takes four bytes of memory. However, since the `highest` member variable is shared, it is not part of `Temperature`'s size. Thus `sizeof(Temperature) == 4`.

Initializing static member variables

Static member variables are initialized outside the class definition. If, for example, the above `Temperature` class were to be used in a program, the `highest` member variable would have to be initialized:

```
#include "temperature.h"

float Temperature :: highest = 0.0;    // initialization of static member variables
                                         // occurs outside any function
int main()
{
    Temperature t;
    ... code removed for brevity ...
    return 0;
}
```

When to use static

The `static` keyword is a scope modifier. It serves to increase the scope from just one instance of a function or a class to all instances of the function or class. As we learned from CS 124, this can be a two-edged sword. The larger the scope, the harder it can be to find a bug with the variable. If, for example, a variable is global, it is difficult to tell what code is looking at the value or which code changes it. These questions are much easier to answer with local variables.

With `static`, there are now several levels of scope:

global variables	The largest level of scope. A global variable is accessible from anywhere in the program.
<code>static</code> member variable	All objects instantiated from a class share the same variable.
<code>public</code> member variable	All the methods in a class have access to the variable as well as functions having an object from that class.
<code>private</code> member variables	All the methods in a class have access to the variable.
<code>static</code> local variables	All instances of a single function share access to the variable.
By-reference parameter	Both the caller and the callee share the same variable.
Local variable	All the statements in a given function have access to the variable. Note that by-value parameters have essentially the same scope as local variables.
Blocks	It is possible to declare a variable that is only visible inside the body of an IF statement or in a FOR loop. These represent the smallest scope of any variable.

As a general rule, a programmer should use the smallest possible scope to solve a given programming problem. For example, never use a member variable when a parameter or a local variable will suffice. Try to declare counter variables inside the FOR loop rather than use a local variable for the same purpose.

One final note: `static` member variables have the largest scope of any variable with the exception of globals. Therefore, they should be used cautiously and when there is no better solution.

Example 2.5 – Point

Demo

This example will demonstrate how to use `static` to make all objects share the same limits. This is one of the most common uses of `static` in a class.

Problem

Consider a computer game where the size of the window is adjustable. Create a class called `Point` representing the position of an item in the game. The `Point` class should also know the bounds of the window (`xMin` through `xMax` and `yMin` through `yMax`) so it can determine if the item is off the screen.

Solution

First, the declarations of `xMin` and company in the `Point` class definition are:

```
class Point
{
    ... code removed for brevity...
private:
    float x;           // horizontal position
    float y;           // vertical position
    bool dead;         // have we exceed our bounds?
    static float xMin; // minimum extent of the x position
    static float xMax; // maximum extent of the x position
    static float yMin; // minimum extent of the y position
    static float yMax; // maximum extent of the y position
};
```

To test this class, we need to instantiate a couple `Point` objects:

```
float Point::xMin = -10.0;           // initialize the static member
float Point::yMin = -10.0;           // variables. Though these
float Point::xMax = 10.0;            // look like global variables,
float Point::yMax = 10.0;            // they are not

int main()
{
    // create a legal point at zero, zero
    Point pt1;                      // bounds set to (-10, -10)
    cout << "Initial value: ";        //      to (10, 10)
    pt1.display();
    cout << endl;

    // move it to an illegal point
    pt1.setX(-20.0);                // outside the xMin bounds
    cout << "After setting to (-20.0, 0.0), "
        << (pt1.isDead() ? "invalid" : "valid")
        << endl;

    // create another point that is also invalid
    Point pt2(0, 20.0);              // also sharing the same bounds
    cout << "Second point at (0.0, 20.0) is " // as pt1, so this is
        << (pt2.isDead() ? "invalid" : "valid") // also invalid
        << endl;

    return 0;
}
```

See Also

The complete solution is available at [2-5-point.html](#) or:

```
/home/cs165/examples/2-5-point/
```

Example 2.5 – Card

Demo

Another common use of `static` is to configure a class for use in a single application. This is useful when all objects from the class share the same configuration setting.

Problem

Some card games consider the Ace to be higher than the King while others consider it to be lower than the Two. Create a card class that allows for either configuration.

Solution

There are two changes to `card.h` from “Example 2.4 – Card.” The first is that we need two strings for the ranks.

```
const char RANKS_HIGH[] = "234567890jqka"; // aces high
const char RANKS_LOW[] = "a234567890jqk"; // aces low
```

The second is the addition of the static member variable called `acesHigh`:

```
class Card
{
    ... code removed for brevity ...
private:
    // holds the value. Though there are 256 possible, only 52 are used
    unsigned char value; // internal representation
    static bool acesHigh; // is an Ace high, or low?
};
```

Next the `getRank()` method needs to be adjusted to point to the correct string:

```
char Card :: getRank() const
{
    // this is static because we should only need to initialize it once
    static const char * pRank = (acesHigh ? RANKS_HIGH : RANKS_LOW);
    return pRank[value % 13]; // point to the appropriate string
}
```

Finally we shall test the new class.

```
bool Card :: acesHigh = true; // initialized outside main()

int main()
{
    // where is the Ace of Diamonds?
    string sAceDiamonds("ad");
    Card cardAceDiamonds(sAceDiamonds);
    cout << "The Ace of Diamonds is at rank: "
        << cardAceDiamonds.iRank('a')
        << endl;

    return 0;
}
```

See Also

The complete solution is available at [2-5-card.html](#) or:

```
/home/cs165/examples/2-5-card/
```

Example 2.5 – Time

Demo	This example is much like “Example 2.5 – Card” in that the static member variable will be used to configure all the objects in the class.
Problem	The most common way to represent dinner time is “6:00pm.” The military uses a 24-hour clock and represents the same time as “18:00.” Modify the <code>Time</code> class to either display military time or the traditional am/pm time.
Solution	<p>The first change to the time class from “Example 2.4 – Time” is the static member variable <code>isMilitary</code>:</p> <pre>class Time { ... code removed for brevity ... static bool isMilitary; };</pre> <p>Next we need to modify <code>display()</code> to call two variants <code>displayMilitary()</code> or <code>displayCivilian()</code>:</p> <pre>void display() const { // paranoia... assert(validate()); if (isMilitary) displayMilitary(); else displayCivilian(); }</pre> <p>Finally it is necessary to test our new <code>Time</code> class:</p> <pre>bool Time :: isMilitary = false; // Configure the time class int main() { Time time1; // start with midnight cout << "Time1 is midnight - "; time1.display(); Time time2(9 /*hours*/, 11 /*minutes*/); // Next 9:11 am cout << "Time2 is in the morning - "; time2.display(); Time time3(18 /*hours*/, 2 /*minutes*/); // Finally 6:02 pm cout << "Time3 is the afternoon - "; time3.display(); return 0; }</pre>
Challenge	As a challenge, modify the driver program so it displays military time. This is accomplished by setting the <code>isMilitary</code> member variable.
See Also	<p>The complete solution is available at 2-5-time.html or:</p> <pre>/home/cs165/examples/2-5-time.cpp</pre>

Review 1

Given the following code:

```
class Silly
{
public:
    Silly() { cout << "Default constructor\n"; }
    Silly(const Silly & s) { cout << "Copy constructor\n"; }
    ~Silly() { cout << "Destructor\n"; }
    void method() { cout << "Method\n"; }

};

Silly & function(Silly & s)
{
    cout << "Function\n";
    return s;
}

int main()
{
    Silly s1;
    s1.method();
    function(s1);
    s1.method();
    return 0;
}
```

What is the output?

Please see page 150 for a hint.

Problem 2

What is the output of the following code:

```
int addto(int value)
{
    static int sum = 0;
    return sum += value;
}

int main()
{
    int array[] = { 3, 1, -5, 2 };

    for (int i = 0; i < 4; i++)
        addTo(array[i]);

    cout << addTo(0) << endl;
}
```

Please see page 156 for a hint.

Problem 3

Given the following class:

```
class Bullet
{
    public:
        Bullet() { numBullet++; }
        ~Bullet() { --numBullet; }
        int getBullets() { return numBullet; }
    private:
        static int numBullet;
};
```

What is the output of the following code:

```
int Bullet :: numBullet = 0;

int main()
{
    Bullet b1;
    Bullet b2;

    {
        Bullet b3;
        Bullet b4;
        cout << b3.getBullets() << endl;
    }

    cout << b1.getBullets() << endl;
    return 0;
}
```

Please see page 157 for a hint.

Challenge 4, 5, 6

You would like to build a class to handle group texting. While each object will have its own variables (ID, for example), they would share the same display text (an array of the last 20 messages).

5. Create a UML class diagram of the `GroupText` class.

6. From the above UML class diagram, define the class.

7. Implement any method which is not defined inline above.

2.6 Non-Member Operator Overloading

Sue is putting the finishing touches on her new `Date` class. She has got it working the way it is supposed to, but it still does not look as polished as she wishes it would. Why can she not make it as slick as the `string` class? Instead of calling `date.display()`, why can she not use `cout << date;` like the `string` class does? As she mulls this over, she flips through the textbook and discovers operator overloading.

Objectives

By the end of this chapter, you will be able to:

- Explain what is operator overloading and why you would want to use it
- Write the functions necessary to overload the common arithmetic operators
- Explain which parameters and return values are constant, and why

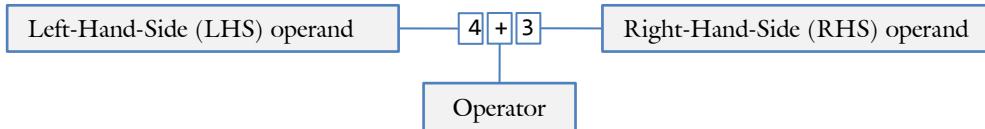
Prerequisites

Before reading this chapter, please make sure you are able to:

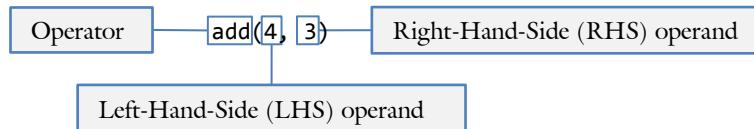
- Create a class definition matching a given UML class diagram (Chapter 2.2-2.5)
- Pass an object as a parameter to another function (Chapter 1.3 and 2.4)
- Explain the difference between pass-by-value & return-by-value and pass-by-reference & return-by-reference (Procedural Programming in C++ Chapter 1.4 and 3.3)

What is operator overloading and why you should care

Operator overloading is the process of using a more convenient and human-readable notation for calling a function (such as “`4 + 3`”…)



... than the functional notation common in programming languages (such as “`add(4, 3)`”).



In other words, operator overloading does not add any power to C++; there is nothing you can do with operator overloading that cannot be done with traditional functions. Instead, operator overloading makes it easier and more convenient for you and others to use your code.

To understand the benefit of operator overloading, it is first necessary to embrace the differences between working on a project alone and sharing code with others. When you are working on a project alone, it is equally important to make a function easy to write as it is to make a function easy to use. After all, you are the only person calling the function! When you are sharing code with others, one person’s code can be used by dozens (or even thousands) of other people. Thus it is often worthwhile to go to great lengths to make things even slightly easier to use. Operator overloading is among the most powerful tools we have to that end.

Infix notation

Most programming languages such as C++ use prefix notation to represent a function or operator. This notation is characterized by using the verb (or function name or operator) before the parameters (or operands). An example of prefix notation is “add four and three” in English or “`add(4, 3)`” in C++. In both cases, the operator (“add”) precedes the operands (four and three).

Infix notation, on the other hand, is characterized by situating the operator between the operands. Examples of infix notation include “four plus three” or “ $4 + 3$.”

While most people find infix notation easier to understand than prefix notation, it is not without limitations. Prefix notation allows for an unlimited number of parameters while infix notation allows for only two. For example, “`add(4, 3, 7)`” uses only one operator while “ $4 + 3 + 7$ ” uses two. Similarly, there is no ambiguity in the order in which functions are called with prefix notation whereas infix notation introduces ambiguity. Does “ $4 + 3 + 7$ ” mean “ $(4 + 3) + 7$ ” or “ $4 + (3 + 7)$ ”? We need parentheses and the order-of-operations to disambiguate how to evaluate expressions in infix notation, but neither are required for prefix.

Operator overloading is the process of using infix notation for a collection of special operators. These operations are:

Operator	Example	Use and metaphor
insertion	<code>a << b</code>	Bitwise left-shift, send-to-left, display
extraction	<code>a >> b</code>	Bitwise right-shift, send-to-right, get input from keyboard
addition	<code>a + b</code>	Sum, add, adding onto, etc.
subtraction	<code>a - b</code>	Difference, subtraction, distance between, removing from, etc.
multiplication	<code>a * b</code>	Multiplying, extending, etc.
division	<code>a / b</code>	Division, dividing, reducing, etc.
modulus	<code>a % b</code>	Remainder
add onto	<code>a += b</code>	Add onto, increase by
subtract from	<code>a -= b</code>	Subtract from, decrease by
multiply onto	<code>a *= b</code>	Multiplying onto, extending, etc.
divide from	<code>a /= b</code>	Division, dividing, reducing, etc.
modulus from	<code>a %= b</code>	Apply the remainder to
increment	<code>a++ ++a</code>	Add one, advance, move forward, increase
decrement	<code>a-- --a</code>	Subtract one, retreat, move backwards, decrease
negative	<code>-a</code>	Negative, opposite, disjoint set
not	<code>!a</code>	Logical not, opposite
equivalence	<code>a == b</code>	Are they the same?
different	<code>a != b</code>	Are they different?
greater than	<code>a > b</code>	Is one larger than another?
greater than or equal to	<code>a >= b</code>	Is one larger or equal to another?
less than	<code>a < b</code>	Is one smaller than another?
less than or equal to	<code>a <= b</code>	Is one smaller or equal to another?
and	<code>a && b</code>	Logical “and”, subset
or	<code>a b</code>	Logic “or”, superset

Because the infix operator goes between the operands, the two parameters are typically called `lhs` (for left-hand-side) and `rhs` (for right-hand-side).

Insertion operator

The insertion operator (`<<`) is traditionally used to send (or insert) output to the screen. This makes it very convenient for just about all classes because it enables us to quickly and easily view the contents of an object.

Using the operator

When displaying content on the console with `cout`, the insertion operator (`<<`) is the function, not `cout`. Each insertion operator in a statement constitutes a function call:

```
{
    Complex c(3.1, 4.5);
    cout << c << endl;
}
```

`// 3.0 + 4.5i`
`// insertion operator called twice`

The left-hand-side of the insertion operator is `cout`. It turns out that `cout` is itself an object of type `std::ostream` and it must be passed to the insertion operator by-reference. The `ofstream` class that we use to write data to a file is also a derivative of `ostream`.

The right-hand-side is the object to be displayed. In this case, it is the `Complex` class. Since the insertion operator should not need to change `Complex`, we pass it by-reference as a constant.

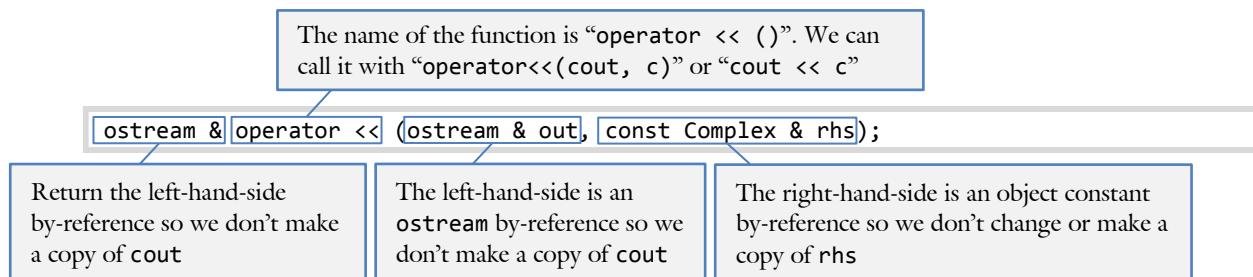
The return value needs to be the `ostream` passed on the left-hand side. The reason for this is that the insertion operator can be “stacked” or “chained.” Thus the `cout` statement can be re-written as:

```
(cout << c) << endl;
```

In order for `cout` to be passed to the second insertion operator (the one displaying the `endl`), the left-hand-side needs to be returned.

Prototype

The prototype for the insertion operator defined with the `Complex` class on the right-hand-side is:



Implementation

The following is the implementation of the extraction operator for the `Complex` class:

```
ostream & operator << (ostream & out, const Complex & rhs)
{
    out << rhs.getReal();                                // use “out” like “cout”
    if (rhs.getImaginary() != 0.0)
        out << " + " << rhs.getImaginary() << "i";    // only display the imaginary
                                                       // component if non-zero
    return out;                                         // always return “out”
}
```



The most common mistake is to forget to return `out`, thereby making it impossible to chain the insertion operator.

Extraction operator

The extraction operator (`>>`) is traditionally used to get input (or extract) from a file or the keyboard. As with the insertion operator, we use it with just about every class.

Using the operator

When receiving input from the keyboard with `cin`, the extraction operator (`>>`) is the function, not `cin`.

```
{
    Complex c1;                                // the default constructor initializes both c1
    Complex c2;                                // and c2 as (0.0 + 0.0i)
    cin >> c1 >> c2;                          // first fill c1 then fill c2
}
```

The left-hand-side of the extraction operator is `cin`. It turns out that `cin` is itself an object of type `std::istream` and it must be passed to the extraction operator by-reference. The `ifstream` class that we use to read data from a file is also a derivative of `istream`.

The right-hand-side is the object to be filled with input. In this case, it is the `Complex` class. Since the extraction operator will be changing `Complex`, we pass it by-reference and not as a constant.

The return value needs to be the `istream` passed on the left-hand side. The reason for this is that the extraction operator can be “stacked” just like the insertion operator. Thus the `cin` statement can be re-written as:

```
(cin >> c1) >> c2;
```

In order for `cin` to be passed to the second extraction operator (the one filling `c2`), the left-hand-side needs to be returned.

Prototype

The prototype for the extraction operator defined with the `Complex` class on the right-hand-side is:

The name of the function is “operator `>>` ()”. We can call it with “operator`>>`(`cin`, `c`)” or “`cin` `>>` `c`”

```
istream & operator >> (istream & in, Complex & rhs);
```

Return the left-hand-side by-reference so we don't make a copy of `cin`

The left-hand-side is an `istream` by-reference so we don't make a copy of `cin`

The right-hand-side is an object by-reference because we will need to be changing `rhs`

Implementation

The following is the implementation of the extraction operator for the `Complex` class:



```
inline istream & operator >> (istream & in, Complex & rhs)
{
    float real;
    float imaginary;
    in >> real >> imaginary;           // first fetch input into local variables
    rhs.set(real, imaginary);          // second set rhs with the data (real & imaginary)
    return in;                         // finally, do not forget to return "in"
}
```

The most common mistake is to forget to return `in`, thereby making it impossible to chain the extraction operator.

Arithmetic operators

The arithmetic operators include operators traditionally used for mathematical operations. There are many arithmetic operators supported in C++: addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). Additionally there are the bit-wise operators of bit-wise and (&), bit-wise or (|), and bitwise xor (^).

When to use

Arithmetic operators are commonly used for mathematical functions like addition. Many other metaphors may apply. For example, the plus operator can also work for string concatenation (`string1 + string2`) and for advancement (`position + velocity`). The minus operator between two points can mean “the distance.”

Using the operator

In algebra, the notation for adding two complex numbers is the following:

$$(3 + 7i) + (2 + 9i) = (5 + 16i)$$

To represent this in C++, we need two `Complex` objects and the addition operator:

```
{
    Complex c1(3.0, 7.0);           // (3.0 + 7.0i)
    Complex c2(2.0, 9.0);           // (2.0 + 9.0i)
    cout << (c1 + c2) << endl;     // displays "5.0 + 16.0i"
}
```

Both the left-hand-side and the right-hand-side are `Complex` objects. Since we don't want to needlessly make a copy of `lhs` or `rhs`, both are passed by-reference. Notice that we do not expect either `c1` or `c2` to change. This means that both `lhs` and `rhs` must be constant as well.

The return value of the addition operator is neither `c1` nor `c2`, but rather a newly generated object. Therefore we will return a `Complex` by-value.

Prototype

The prototype for the addition operator defined with the `Complex` class is the following:

The name of the function is “operator + ()”. We can call it with “operator+(c1, c2)” or “c1 + c2”

```
Complex operator + (const Complex & lhs, const Complex & rhs);
```

Return a new object by-value because it will not be `lhs` or `rhs`

Both the left-hand-side and the right-hand-side are pass-by-reference so we don't make a copy. They are also constant so we don't change `lhs` or `rhs`

Implementation

The following is the implementation of the addition operator for the `Complex` class:



```
inline Complex operator + (const Complex & lhs, const Complex & rhs)
{
    return Complex(lhs.getReal()      + rhs.getReal(),           // use the non-default
                  lhs.getImaginary() + rhs.getImaginary());   // constructor to
                                                               // save a step here
}
```

Note that since many of these operators are just one or two lines of code, it is common to make them inline.

Add onto

While the arithmetic operators do not change either operand, the add-onto operator (`+=`) does. The whole point, after all, is to modify the left-hand-side of the equation. Other flavors of the “add-onto” include subtract-from (`-=`), multiply-onto (`*=`), divide-from (`/=`), modulus-onto (`%=`), bitwise-and-onto (`&=`), bitwise-or-onto (`|=`), bitwise-exclusive-or-onto (`^=`), bitwise-shift-left-onto (`<<=`), and bitwise-shift-right-onto (`>>=`). Clearly the utility of overloading some of these operators is greater than others.

Using the operator

The add-onto operator increases the value of the left-hand-side by the amount on the right-hand-side. Additionally, the left-hand-side is returned from the expression:

```
{
    Complex c1(3.0, 7.0);           // (3.0 + 7.0i)
    Complex c2(2.0, 9.0);           // (2.0 + 9.0i)
    cout << c1 += c2 << endl;      // displays the new value for c1: "5.0 + 16.0i"
    cout << c1 << endl;             // since c1 changes, this displays "5.0 + 16.0i"
}
```

The left-hand-side is the object to be changed. This means it must be passed by-reference.

The right-hand-side is the amount that the left-hand-side is to be changed by. Since we do not want to copy the right-hand-side, it is also passed by-reference. However, since the right-hand-side does not change, it is passed as a constant.

The return value is the newly updated left-hand-side. This means it must be returned by-reference.

Prototype

The prototype for the add-onto operator defined with the `Complex` class is the following:

The name of the function is “operator `+=` ()”. We can call it with “`operator+=(c1, c2)`” or “`c1 += c2`”

`Complex & operator += (Complex & lhs, const Complex & rhs);`

Return the left-hand-side by-reference so the add-onto operator can be chained

The left-hand-side is the object to be changed so it must be by-reference

The right-hand-side is by-reference so we don’t copy `rhs`. It is also constant because we will not be changing `rhs`

Implementation

The following is the implementation of the add-onto operator for the `Complex` class:



```
inline Complex & operator += (Complex & lhs, const Complex & rhs)
{
    // (a + bi) + (c + di) = (a + c) + (b + d)i
    lhs.set(lhs.getReal() + rhs.getReal(),
            lhs.getImaginary() + rhs.getImaginary());
    return lhs;
}
```

Note that the right-hand-side does not need to be the same data type as the left-hand-side. We might choose to say “`c1 += 7.0`”. In this case, the right-hand-side might be a `double`.

Prefix increment & decrement

The prefix increment and decrement operators (`++x` and `--x`) are much like the add-onto operators except there is no right-hand-side. This means that they are not truly infix operators and they take only one parameter.

When to use

While the obvious uses for the increment and decrement operators are for adding or subtracting one, other metaphors apply. For example, `++` could mean “move ahead” or “next turn” or even “redo.”

Using the operator

The prefix increment and decrement operators update the object before the expression is evaluated. This means that if we display the results of `++c`, we expect to see the new value instead of the old:

```
{
    Complex c(3.0, 7.0);           // (3.0 + 7.0i)
    cout << ++c << endl;          // displays "4.0 + 7.0i"
    cout <<     c << endl;         // since c changes, this also displays "4.0 + 7.0i"
}
```

Since the increment and decrement operators are unary operators (only one operand), there is only one parameter. This parameter, the right-hand-side, must be by-reference because it changes.

The return value is the newly updated right-hand-side. This means it must be returned by-reference.

Prototype

The prototype for the prefix increment operator defined with the `Complex` class is the following:

The name of the function is “operator `++ ()`”. We can call it with “`operator++(c)`” or “`++c`”

`Complex & operator ++ (Complex & rhs);`

Return the right-hand-side by-reference because we return the newly updated copy

The right-hand-side is the object to be changed so it must be by-reference

Implementation

The following is the implementation of the prefix increment operator for the `Complex` class:



```
inline Complex & operator ++ (Complex & rhs)
{
    return rhs += 1.0;           // prefix ++ is exactly the same as += 1
}
```

It is very common to define the prefix-increment operator in terms of the add-onto operator. Just make sure that the add-onto operator is defined first.

Postfix increment & decrement

The postfix increment and decrement operators (`x++` and `x--`) are much like the prefix versions (`+x` and `-x`) with one important exception. While the prefix versions return the newly updated value of the variable, the postfix versions return the old value.

Using the operator

The postfix increment and decrement operators update the object after the expression is evaluated. This means that if we display the results of `c++`, we expect to see the old value instead of the new:

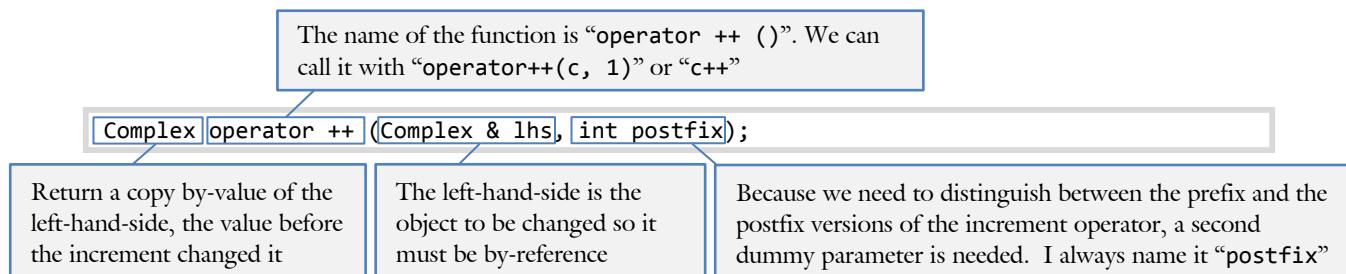
```
{
    Complex c(3.0, 7.0);           // (3.0 + 7.0i)
    cout << c++ << endl;          // displays the old value "3.0 + 7.0i"
    cout << c   << endl;          // now we display the new value "4.0 + 7.0i"
}
```

Since the increment and decrement operators are unary operators (only one operand), it seems like there is only need for one parameter: the left-hand-side that is being changed. This parameter must be by-reference. However, C++ requires us to define a second parameter so we can tell the difference between the two.

The return value is the old value of left-hand-side, before the update happened. This means we must return a copy of the old value by-value.

Prototype

The prototype for the post-fix increment operator defined with the `Complex` class is the following:



Implementation

The following is the implementation of the postfix increment operator for the `Complex` class:



```
inline Complex operator ++ (Complex & lhs, int postfix)
{
    Complex old(lhs);           // the easiest way is to remember the old value
    lhs += 1.0;                 // and return it after we have done
    return old;                 // the increment
```

Note that the postfix version of the the increment operator is necessairely less efficient than the prefix version. This is because the copy constructor must be called at least once!

Negative and not

The negative (-) and not (!) operators, like the increment and decrement operators before them, are unary. This means that there is only one operand. In addition to negation and not, there are other unary operators supported in C++: bitwise not (~), the dereference operator (*), and the address-of operator (&).

When to use

The negative operator can be used in the traditional algebraic manner or it could be used as “opposite” or “backwards” or even “turn around.”

Using the operator

The negative and not operators do not change the variable or object; they only return the result of the expression:

```
{
    Complex c(3.0, 7.0);           // (3.0 + 7.0i)
    cout << -c << endl;          // displays "-3.0 - 7.0i"
    cout << !c << endl;          // displays "0" because "3.0 + 7.0i" isn't 0
    cout << c << endl;           // displays "3.0 + 7.0i" because c is unchanged
}
```

Since negative and not are unary operators, there is only one operand. We pass the right-hand-side by-reference because we don’t want to change the object. The parameter is a constant because it does not change.

The return value is a new object of the same type as the right-hand-side. This means it must be returned by-value.

Prototype

The prototype for the negative operator defined with the `Complex` class is the following:

The name of the function is “operator - ()”. We can call it with “operator-(c)” or “-c”

`Complex operator - (const Complex & rhs);`

Return a new object so it must be by-value. It does not matter if it is constant or not

The right-hand-side is not to be changed so it must be a constant. It should not be copied so it should be by-reference

Implementation

The following is the implementation of the negative increment operator for the `Complex` class:

```
inline Complex operator - (const Complex & rhs)
{
    return rhs * -1.0;                      // -c is the same as c * -1.0
}
```

The logical-not operator is similar to negative:

```
inline bool operator ! (const Complex & rhs)
{
    return (rhs.getReal() == 0.0 && rhs.getImaginary() == 0.0);
```

Comparision operators

The comparisons operators such as equivalence (`==`) and greater-than (`>`) take two parameters and return a Boolean value.

Using the operator

In algebra, the notation for comparing two complex numbers is the following:

```
(3 + 7i) ≠ (2 + 9i)
```

Notice how the expression is either true or it isn't. To represent this in C++, we need two `Complex` objects and the comparison operators:

```
{
    Complex c1(3.0, 7.0);           // (3.0 + 7.0i)
    Complex c2(2.0, 9.0);           // (2.0 + 9.0i)
    if (c1 == c2)
        cout << "Same!\n";
    if (c1 != c2)
        cout << "Different!\n";
}
```

Both the left-hand-side and the right-hand-side are `Complex` objects. Since we don't want to needlessly make a copy of `lhs` or `rhs`, both are passed by-reference. Notice that we do not expect either `c1` or `c2` to change. This means that both `lhs` and `rhs` must be constant as well.

The return value of the comparison operator is a Boolean value.

Prototype

The prototype for the equivalence operator defined with the `Complex` class is the following:

The name of the function is “operator `==` ()”. We can call it with “`operator==(c1, c2)`” or “`c1 == c2`”

```
bool operator == (const Complex & lhs, const Complex & rhs);
```

Return `true` or `false`, depending on whether the expression is true.

Both the left-hand-side and the right-hand-side are pass-by-reference so we don't make a copy. They are also constant so we don't change `lhs` or `rhs`

Implementation

The following is the implementation of the equivalence operator for the `Complex` class:



```
inline bool operator == (const Complex & lhs, const Complex & rhs)
{
    return (lhs.getReal()      == rhs.getReal() &&
            lhs.getImaginary() == rhs.getImaginary());
```

For the most part, we do not need to implement all the comparison operators independently. We can do it in terms of the other operators we previously defined. Note that if we define some, we should probably define them all. For example, if we define the less-than operator, we should also define the greater-than operator.

All the comparison operators associated with our `Complex` class are:

```
inline bool operator == (const Complex & lhs, const Complex & rhs)
{
    return (lhs.getReal()      == rhs.getReal() &&
            lhs.getImaginary() == rhs.getImaginary());
}

inline bool operator != (const Complex & lhs, const Complex & rhs)
{
    return !(lhs == rhs);
}

inline bool operator > (const Complex & lhs, const Complex & rhs)
{ // the distance from the origin
    return (lhs.getReal()      * lhs.getReal() +
            lhs.getImaginary() * lhs.getImaginary()) >
           (rhs.getReal()      * rhs.getReal() +
            rhs.getImaginary() * rhs.getImaginary());
}

inline bool operator >= (const Complex & lhs, const Complex & rhs)
{
    return (lhs > rhs) || (lhs == rhs);
}

inline bool operator < (const Complex & lhs, const Complex & rhs)
{
    return !(lhs >= rhs);
}

inline bool operator <= (const Complex & lhs, const Complex & rhs)
{
    return !(lhs > rhs);
}
```



Sue's Tips

Defining a comparison operator in terms of other comparison operators is easy and, if the functions are inline, incurs no performance penalty. However, there is a hidden pitfall. What would happen if the `>` operator was defined in terms of the `<` operator and the `<` operator was defined in terms of the `>` operator? The result would be a circular reference. To avoid this, only reference functions defined above the current location in the file.

Example 2.6 – Complex

Unit 2

Demo

This example will demonstrate how to build a class with number-like properties implementing most of the arithmetic operators.

Problem

A complex number can be thought of as a two-dimensional number: the first dimension is the real component similar to floats in C++ and the second dimension is the imaginary component. Write a class to represent complex numbers and implement all the convenient operators. This includes the insertion (`<<`) and extraction (`>>`), increment (`++`) and decrement (`--`), addition (`+`) and subtraction (`-`), multiplication (`*`) and division (`/`), negative (`-`) and logical not (`!`), add-onto (`+=`) and subtract-from (`-=`), and the comparision (`>=`, `>`, `<=`, `<`, `==`, and `!=`) operators. .

Solution

The driver program for this class is the following:

```
int main()
{
    Complex c1;

    for (;;)
    {
        cout << "Old value: " << c1 << endl;

        // prompt for new value
        Complex c2;
        cout << "\n\nEnter a complex number: ";
        cin >> c2;
        cout << "New value: " << c2 << endl << endl;

        // plus
        cout << '(' << c1 << ")+" << c2 << ") == " << c1 + c2 << endl;

        // minus
        cout << '(' << c1 << ")-(" << c2 << ") == " << c1 - c2 << endl;

        // times
        cout << '(' << c1 << ")*(" << c2 << ") == " << c1 * c2 << endl;

        // c1 = c2 and loop again...
        c1.set(c2.getReal(), c2.getImaginary());
    }

    return 0;
}
```

When the program is run, the output is:

```
Old value: 4 + 5i

Enter a complex number: 7 2
New value: 7 + 2i
(4 + 5i)+(7 + 2i) == 11 + 7i
(4 + 5i)-(7 + 2i) == -3 + 3i
(4 + 5i)*(7 + 2i) == 18 + 43i
```

See Also

The complete solution is available at [2-6-complex.html](#) or:

```
/home/cs165/examples/2-6-complex/
```

Example 2.6 – Card

Demo

This example will demonstrate how operator overloading can work with a class that does not contain values that work like traditional numbers.

Problem

Write a card class that implements all the convenient operators. This includes the insertion (`<<`) and extraction (`>>`), increment (`++`) and decrement (`--`), addition (`+`) and subtraction (`-`), add-onto (`+=`) and subtract-from (`-=`), and the comparision (`>=`, `>`, `<=`, `<`, `==`, and `!=`) operators. .

Solution

The insertion operator is the following:

```
istream & operator >> (istream & in, Card & card)
{
    // input comes in the form of a string
    string input;
    in >> input;

    // do the actual work
    card.parse(input);
    assert(card.validate());

    // return the input stream
    return in;
}
```

The extraction operator is:

```
ostream & operator << (ostream & out, const Card & card)
{
    if (card.isInvalid())
        out << NO_CARD;
    else
        out << card.getSuit() << card.getRank();

    // return the output stream
    return out;
}
```

The greater-than operator is:

```
bool operator > (const Card & lhs, const Card & rhs)
{
    //only comparing the ranks
    unsigned char iLHS = lhs.getValue() % 13;
    unsigned char iRHS = rhs.getValue() % 13;

    return iLHS > iRHS;
}
```

See Also

The complete solution is available at [2-6-card.html](#) or:

```
/home/cs165/examples/2-6-card/
```

Problem 1

Given the following class:

```
class Angle
{
    public:
        Angle() : angle(0) {}
        Angle(float a) : angle(a) {}
        Angle(const Angle & a);
        void setAngle(float a);
        float getAngle() const;
    private:
        float angle;
};
```

Which operators should be overloaded?

Please see page 166 for a hint.

Problem 2

Given the following class:

```
class Velocity
{
    public:
        Velocity() : angle(0.0), speed(0.0), pos() { }
        Velocity(const Velocity & v);
        float getAngle() const;
        float getSpeed() const;
        Position getPosition() const;
        void setAngle(float a);
        void setSpeed(float s);
        void setPosition(const Position & pos);
    private:
        float angle;
        float speed;
        Position pos;
};
```

Which operators should be overloaded?

Please see page 166 for a hint.

Problem 3

Given the following class:

```
class Coordinate
{
    public:
        Coordinate();
        Coordinate(int r, int c);
        Coordinate(const Coordinate &);

        float getR() const;
        float getc() const;
        void set(int r, int c);

    private:
        int row;
        int col;
};
```

Overload the insertion and the extraction operators.

Please see page 167, 168 for a hint.

Problem 4

Given the following class:

```
class Gpa
{
    public:
        Gpa();
        Gpa(float gpa);
        Gpa(const Gpa & gpa);

        float get() const;
        void set(float);

    private:
        float gpa;
};
```

Overload the insertion, extraction, equivalence, and greater-than operators.

Please see page 176 for a hint.

2.7 Friends

Sam found adding operator overloading functions to his `Date` class to be tedious. Because these functions were not members of his class, he was restricted to using public methods. This meant he needed to use the publicly-exposed properties rather than the privately-available data which would be more convenient.

Objectives

By the end of this chapter, you will be able to:

- Define and explain the uses for the `friend` modifier.
- Use the `friend` modifier to make a non-member function a `friend`.
- Explain the pros and cons of using `friends`.

Prerequisites

Before reading this chapter, please make sure you are able to:

- List and define the rules of encapsulation (Chapter 2.0)
- Create a class matching a UML definition (Chapter 2.2 – 2.5)
- Overload common operators using non-member functions (Chapter 2.6)

What are friends and why you should care

Friends are special functions in C++ that have access to the private member variables and private methods of a class. Many would say that this violates the rules of encapsulation: giving non-members access to privates. After all, only member functions should have access to private member variables and functions!

The advantage of friends is that it is often desirable to allow some functions to operate directly on the data of a class without having to work with public properties. This is especially true when a non-trivial computation needs to happen that translate private data to public properties. It may be undesirable to expose these properties to the client. With friends, you don't. The class enumerates all the functions that have access to the privates thereby avoiding the unchecked public access to privates. In other words, the class gets to indicate which functions act like methods.

Syntax of friends

As you may recall, function prototypes typically go in header files. Member functions are also in the form of prototypes in the header file except they go inside the class definition. This is an important detail; classes have complete control over which functions are their friends.

To illustrate this point, consider the following class definition of our `Complex` class from last chapter:

```
class Complex
{
    ... code removed for brevity ...
};

// this inline function is defined outside the class definition
inline std::ostream & operator << (std::ostream & out, const Complex & rhs)
{
    out << rhs.getReal();                                // access the public methods
    if (rhs.getImaginary() != 0.0)                        //     getReal() and getImaginary()
        out << " + " << rhs.getImaginary() << "i";   //     just like all other non-member
    return out;                                         //     functions
}
```

In this example, observe how the inline function “operator `<<`” is defined outside the `Complex` class and how it only has access to the public methods `getReal()` and `getImaginary()`. To make the same function a `friend`, the prototype would be moved inside the `Complex` class definition and the `friend` keyword would be added:

This designates the function as a friend.
This will be defined inside the class definition.

```
class Complex
{
    ... code removed for brevity ...
public:
    inline friend std::ostream & operator << (std::ostream & out,
                                                const Complex & rhs)
    {
        out << rhs.r;                                     // now we have access to the private
        if (rhs.i != 0.0)                                 //     member variables 'r' and 'i'
            out << " + " << rhs.i << "i";             //     without having to go through
        return out;                                       //     the public methods
    }
    ... code removed for brevity ...
};
```

From this example, we can observe the three things that are unique about `friends`:

1. The `friend` keyword is added before the return type to the function prototype or inline function definition. This identifies it as a `friend`.
2. The function prototype or inline function definition resides inside the class definition
3. The function has access to private member variables and private methods

Note that if a friend function is defined inside a class definition, the `inline` keyword is optional. The compiler knows the function should be inline simply because the function is defined inside a class definition.

Friends and operator overloading

Friends and operator overloading often occur hand-in-hand because operator overloading functions often feel like member functions. All the operator overloading functions taking an object as either the left-hand-side or the right-hand-side are tightly connected to the class they serve. Consider, for example, the non-friend version of the equivalence operator (==) defined for the `Card` class from previous chapters:

```
// Equivalence without using friends
bool operator == (const Card & lhs, const Card & rhs)
{
    // only same if both suit and rank are the same
    return lhs.getSuit() == rhs.getSuit() &&
           lhs.getRank() == rhs.getRank();
}
```

This is quite inconvenient. After all, we know that the private data implementation of the `Card` is a single `unsigned char` representing a value from 0 – 51. It would be much easier to just compare that value directly. Unfortunately, in order to do so, it is necessary to write another method exposing data:

```
class Card:
{
    public:
    ... code removed for brevity ...
    // functions that are public but really should be private
    int getValue() const { return value; }
    void setValue(int value) { this->value = value; }
    ... code removed for brevity ...
};
```

This would allow for a more convenient definition of the equivalence operator:

```
// Equivalence using a private method getValue()
bool operator == (const Card & lhs, const Card & rhs)
{
    // only same if the values are the same
    return lhs.getValue() == rhs.getValue();
}
```

If, on the other hand, we make the equivalence operator a friend, we can avoid defining `getValue()`:

```
class Card:
{
    public:
    ... code removed for brevity ...
    // there is now no need for getValue()
    inline friend bool operator == (const Card & lhs, const Card & rhs)
    {
        // only same if the values are the same
        return lhs.value == rhs.value;
    }
    ... code removed for brevity ...
};
```

Are friends bad?

Some object-oriented purists believe that friends represent a violation of encapsulation. This, strictly speaking, is true. Only member functions should have access to private member functions and member variables. Every function given access to privates represents more code that is made aware of the private and internal workings of a class. The rules of encapsulation stipulate that no one aside from members of the class should know or care how the internal workings of the class operate.

There are two counter-arguments to this point. First, only the class itself can designate whether a given function is a friend. In other words, a function cannot designate itself to be a friend of a class willy-nilly. First approval must be granted! Therefore friends do not represent wide-spread and wanton expansion of scope. The number of functions (member functions or friends) having access to privates are strictly controlled by the class.

The second counter-argument is significantly more powerful than the first. There are some operations that can only be effectively done with direct access to member variables. If a non-member function is to perform these operations, then it either must be made a `friend` or a special method must be written to accommodate it. The question is: which is a greater violation of encapsulation? Is it worse to give a handful of functions `friend` status or is it worse to make a data-centric (as opposed to a property-centric) method publically accessible? For example, our `Card` class needed to implement a `getValue()` method to efficiently implement the equivalence operator. This method exposed the internal workings of the `Card` class: that Spades came before Hearts in the suit order. The client should never be exposed to this implementation detail!

Because operator-overloading functions are closely tied to a class and because they typically need to have direct access to a class's private member variables, it is frequently best to make them friends.

Example 2.7 – Card

Demo	This example will demonstrate how to implement all the common non-member operators using friends. It would be helpful to compare this with the same code from last chapter to see the differences.
Problem	Modify the <code>Card</code> class to implement the common operators without having to expose the private data implementation. In other words, remove the <code>getValue()</code> and <code>setValue()</code> methods that were necessary before. . .
Solution	<p>The class definition is:</p> <pre>class Card { public: ... code removed for brevity ... // insertion and extraction operators friend std::ostream & operator << (std::ostream & out, const Card & card); friend std::istream & operator >> (std::istream & in, Card & card); // increment and decrement ... only changing rank friend Card & operator ++ (Card & lhs); // prefix friend Card & operator -- (Card & lhs); // prefix friend Card operator ++ (Card & lhs, int postfix); // postfix friend Card operator -- (Card & lhs, int postfix); // postfix // change a card by adding or subtracting one friend Card operator + (const Card & lhs, const int input); friend Card operator + (const int input, const Card & lhs); friend Card operator - (const Card & lhs, const int input); friend Card & operator += (Card & lhs, const int input); friend Card & operator -= (Card & lhs, const int input); // Relative comparison... only comparing rank friend bool operator >= (const Card & lhs, const Card & rhs); friend bool operator > (const Card & lhs, const Card & rhs); friend bool operator <= (const Card & lhs, const Card & rhs); friend bool operator < (const Card & lhs, const Card & rhs); // Absolute comparison... comparing both rank and suit friend bool operator == (const Card & lhs, const Card & rhs); friend bool operator != (const Card & lhs, const Card & rhs); };</pre>
Challenge	Observe how all the prototypes are in the class definition.
See Also	As a challenge, see if you can make all the operators <code>inline</code> whereas currently they are not.
See Also	The complete solution is available at 2-7-card.html or: <code>/home/cs165/examples/2-7-card/</code>

Example 2.7 – Complex

Demo

This is a second example on how to implement all the common operators with friends. Unlike “Example 2.7 – Card,” all the overloaded functions are done as inlines.

Problem

Modify the `Complex` class to implement the common operators without having to expose the private data implementation.

Solution

The class definition is:

```
class Complex
{
public:
    ... code removed for brevity ...
    inline friend std::ostream & operator << (std::ostream & out,
                                                const Complex & rhs)
    {
        out << rhs.r;
        if (rhs.i != 0.0)                                // only display the imaginary
            out << " + " << rhs.i << "i";           // component if non-zero
        return out;                                       // return "out"
    }
    inline friend std::istream & operator >> (std::istream & in,
                                                Complex & rhs)
    {
        in >> rhs.r >> rhs.i;          // input directly into the member variables
        return in;                                     // do not forget to return "in"
    }

    inline friend Complex operator + (const Complex & lhs,
                                      const Complex & rhs)
    { // (a + bi) + (c + di) = (a + c) + (b + d)i
        return Complex(lhs.r + rhs.r, lhs.i + rhs.i);
    }

    inline friend Complex & operator += (Complex & lhs, const Complex & rhs)
    { // (a + bi) + (c + di) = (a + c) + (b + d)i
        lhs.r += rhs.r;
        lhs.i += rhs.i;
        return lhs;
    }

    inline friend bool operator == (const Complex & lhs, const Complex & rhs)
    {
        return (lhs.r == rhs.r && lhs.i == rhs.i);
    }

    ... code removed for brevity ...
};
```

Observe how all the prototypes are in the class definition.

See Also

The complete solution is available at [2-7-complex.html](#) or:

/home/cs165/examples/2-7-complex/



Review 1

Given the following UML class diagram:

Money
- dollars
- cents
+ get
+ set
+ display
- normalize

Which operators should be overloaded?

Please see page 166 for a hint.

Problem 2

Given the following class definition:

```
class Money
{
public:
    Money() : dollars(0), cents(0) { }
    Money(const Money & rhs) : dollars(rhs.dollars), cents(rhs.cents) {}
    double get() const { return (double)dollars + (double)cents / 100.0 }
    void set(double money);
private:
    int dollars;
    int cents;
    void normalize();
};
```

Write the insertion and extraction operators using friends.

Please see page 185 for a hint.

Problem 3-6

Given the following class definition:

```
class Money
{
    public:
        Money() : dollars(0), cents(0) { }
        Money(const Money & rhs) : dollars(rhs.dollars), cents(rhs.cents) {}
        double get() const { return (double)dollars + (double)cents / 100.0 }
        void set(double money);
    private:
        int dollars;
        int cents;
        void normalize();
};
```

3. Write the addition (+) operator using friends.
4. Write the add-onto (+=) operator using friends.
5. Write the both the increment (++) operators (prefix and postfix) using friends.
6. Write the equivalence (==) and greater-than (>) operators using friends.

Please see page 185 for a hint.

2.8 Member Operator Overloading

Sue is working on her Date class but can't seem to get the assignment operator working. Why is that? Of all the operators, the assignment seems to be the most useful (with the exception of the insertion and the extraction operator). While investigating the cause of this obvious oversite, she stumbles upon member operator overloading.

Objectives

By the end of this chapter, you will be able to:

- Implement operator loading as non-member, non-member with friends, or as member functions
- List which operators should and must be overloaded as member functions
- Be able to use either the prefix or the infix notation for an overloaded operator

Prerequisites

Before reading this chapter, please make sure you are able to:

- Explain the differences between infix and prefix function notation (Chapter 2.6)
- Create a function to implement non-member operator overloading (Chapter 2.6)

What is member overloading and why you should care

Member operator overloading is the process of using infix notation of methods. This is similar to non-member operator overloading except the left-hand-side part of the equation is always `this`. It also allows operator functions to access member variables without using the `friend` modifier.

Most operators can be overloaded using non-members or as members. There are a few operators that can only be overloaded as non-members and a few that can only be overloaded as members. The non-member variety are those whose left-hand-side is not `this`. The classic examples of this are the insertion and extraction operators. For the insertion operator the left-hand-side is an `ostream` object; for the extraction operator the left-hand-side is an `istream` object.

The assignment operator can only be overloaded as a member function. The same is true with the array index operator also known as the square bracket operator (`[]`). That being said, the following rules-of-thumb are applied when considering whether to make an operator member or non-member:

- Member if it is a unary operator (like the negative `(-)`, not `(!)`, dereference `(*)`, address-of- `(&)`, increment `(++)` or decrement `(--)`).
- Non-member if both operands are treated equally and are left unchanged (like addition `(+)`, multiplication `(*)`, equivalence `(== !=)`, comparison `(> >= < <=)`, and `(&&)` and or `(||)`).
- Member if the binary operator does not treat both operands equally (like assignment `(=)`, add-onto `(+= -= *= /= %= &= !=)`, and array index `([])`).

Syntax overview

The syntax of member operator overloading is identical to non-member with two important distinctions. First, the function is part of a class. Therefore it is defined as a member function, either prototyped in the class definition or completely defined as inline in the class definition. Second, the left-hand-side parameter is hidden. Recall from Chapter 2.1 that a function modifying an object needs to pass the object as a parameter. In the case of member functions, this parameter is hidden and given the name `this`. The same is true with member operator overloading..

The following operators **must** be overloaded as **member** functions:

Operator	Example	Use
assignment	<code>a = b</code>	Assign the value of <code>b</code> onto <code>a</code>
square bracket	<code>a[b]</code>	Retrieve an item from a collection
function call	<code>a(b)</code>	Can take any number of parameters, used similarly to non-default constructors

The following operators **must** be overloaded as **non-member** functions:

Operator	Example	Use
insertion	<code>a << b</code>	When the left-hand-side is an <code>ostream</code> object
extraction	<code>a >> b</code>	When the left-hand-side is an <code>istream</code> object
addition	<code>a + b</code>	When the returned data type is not the same as the left-hand-side data type
multiplication	<code>a * b</code>	When the returned data type is not the same as the left-hand-side data type

The following **should** be overloaded as **member** functions:

Operator	Example	Use
add onto	<code>a += b</code>	Right-hand-side can be any data type
subtract from	<code>a -= b</code>	Right-hand-side can be any data type
multiply onto	<code>a *= b</code>	Right-hand-side can be any data type
divide by	<code>a /= b</code>	Right-hand-side can be any data type
modulus from	<code>a %= b</code>	Right-hand-side can be any data type
increment	<code>a++ ++a</code>	Add one, postfix or prefix
decrement	<code>a-- --a</code>	Subtract one, postfix or prefix
negative	<code>-a</code>	Negative, opposite, disjoint set
not	<code>!a</code>	Logical not, opposite

The following **should** be overloaded as **non-member** functions

Operator	Example	Use
addition	<code>a + b</code>	Sum, add, adding onto, etc.
subtraction	<code>a - b</code>	Difference, subtraction, distance between, removing from, etc.
multiplication	<code>a * b</code>	Multiplying, extending, etc.
division	<code>a / b</code>	Division, dividing, reducing, etc.
modulus	<code>a % b</code>	Remainder
equivalence	<code>a == b</code>	Are they the same?
different	<code>a != b</code>	Are they different?
greater than	<code>a > b</code>	Is one larger than another?
greater than or equal to	<code>a >= b</code>	Is one larger or equal to another?
less than	<code>a < b</code>	Is one smaller than another?
less than or equal to	<code>a <= b</code>	Is one smaller or equal to another?
and	<code>a && b</code>	Logical “and”, subset
or	<code>a b</code>	Logic “or”, superset

Assignment

Conceptually the assignment (=) operator works the same as the copy constructor with the exception that a new object is not created. Instead, an exact copy of the object on the left-hand-side is made to the object on the right-hand side. The one exception to this general rule is those times when the left-hand-side represents a different data type than the object on the right-hand-side. Be careful with those situations; make sure the overloading the assignment operator makes sense in that context.

Using the operator

With the assignment operator (as with the += operator), the left-hand-side gets changed:

```
{
    Complex c1(10.0, 9.6);           // initialize to 10.0 + 9.6i
    Complex c2;                   // initially 0 + 0i
    Complex c3;                   // same here
    c3 = c2 = c1;                // c2 assigned to 4.5 + 9.6i. Next
                                //   c3 is assigned to 4.5 + 9.6i
}
```

The right-hand-side of the assignment operator is the thing being copied. This means it should not change, thus being a constant. Since we should not make a copy of the right-hand-side, it additionally should be by-reference.

The left-hand-side of the assignment operator is being changed. Since the assignment operator must be a member function, it comes through as `this`. Since `this` changes, the method cannot be constant.

Finally, we need to be able to chain or stack assignment operators. The return value should be the newly changed left-hand-side. Thus the assignment operator in the above code can be re-written as:

```
c3 = (c2 = c1);
```

Prototype

The prototype for the insertion operator defined with the `Complex` class on the right-hand-side is:

The name of the function is “`Complex :: operator = ()`”.
We can call it with “`c1.operator=(c2)`” or “`c1 = c2`”

```
Complex & Complex :: operator = (const Complex & rhs);
```

Return the left-hand-side
by-reference so we don't make
a copy of `this`

The left-hand-side is `this`
because we are overloading
a member operator

The right-hand-side is an object constant
by-reference so we don't change or make a
copy of `rhs`

Implementation

The following is the implementation of the assignment operator for the `Complex` class:



```
Complex & Complex :: operator = (const Complex & rhs)
{
    this->r = rhs.r;
    this->i = rhs.i;
    return *this;
}
```

// the left-hand-side of the operator
// comes in as “`this`”
// return `*this` because “`this`” is
// a pointer to a `Complex`

The most common mistake is to forget to return `*this`, thereby making it impossible to chain the assignment operator.

Square bracket

The square bracket operator (`[]`), also known as the array index operator, is commonly used to retrieve an item from a collection. The interesting thing about this operator, however, is that the parameter does not need to be an integer.

Using the operator

The square bracket operator is commonly used with containers (classes storing data). To demonstrate, we will start with the simple container class `Array`:

```
class Array
{
    ... code removed for brevity ...
private:
    double * data;           // where the data for the array is stored
    int     sizeArray;       // the number of items currently in the array
};
```

We can use the square-bracket operator both as an accessor and as a mutator:

```
{
    Array array(10);          // initial capacity of 10
    array[0] = 9.9;           // change the 1st item by setting it to 9.9
    cout << array[0] << endl; // access the 1st item, the value 9.9
}
```

The right-hand-side of the square bracket operator traditionally is an integer, corresponding to the index of the item in the container. Note, however, that this can be any data type.

The left-hand-side is the class itself, represented as this. If you wish the square-bracket operator to function only as a getter, make `this` constant by making the method itself constant.

The return value is the data being accessed from the container. If the square bracket operator is to function as just a getter, either return by-value or a constant by-reference. If the square bracket operator is to function as both a getter and a setter, then the function must return by-reference.

Prototype

The prototype for the square-bracket operator defined with the `Array` class on the right-hand-side is:

The name of the function is “`Complex :: operator [] ()`”.
We can call it with “`c.operator[](7)`” or “`c[7]`”

```
double & Array :: operator [] (int index);
```

Return value by-reference so
we this operator can be both a
getter and a setter

The left-hand-side is `this`
because we are overloading
a member operator

The right-hand-side is typically an index.
Since we usually use a built-in data type,
there is no need to make it constant.

Implementation

The following is the implementation of the square-bracket operator for the `Array` class:

```
double & Array :: operator [] (int index) throw(bool)
{
    if (index < 0 || index >= sizeArray)
        throw false;
    return data[index];
}
```

Function call

The function call operator () can also be overloaded as a member function. While this is not commonly done, it does have a few rather unusual benefits. First, it can be overloaded with any number of operands. Most operators are unary or binary, but the function call operator can be anything. The second use for the function call operator is how it works similarly to a non-default constructor. Recall that a constructor can take any number of parameters, serving to initialize the class with the passed values. The function call operator can look and work the same, except a new object is not created. In other words, it can be a `set()` function.

Using the operator

It is rare to overload the function operator. However, in those situations when initializing or setting a value to an object needs to occur frequently and with many parameters, it might be the most convenient tool for the job at hand.

```
{
    Complex c1(10.0, 9.6);           // initialize to 10.0 + 9.6i
    c1(3.4, 2.1);                  // set the value to 3.4 + 2.1i
}
```

The left-hand-side is the object being operated on. Since we usually change the left-hand-side, the method is frequently not a constant.

There can be any number of parameters to the function call operator. Usually they come in as constant by-reference, but there is no hard-and-fast rule here.

The return value of the function call operator is usually `void`. That being said, it can be anything.

Prototype

The prototype for the square-bracket operator defined with the `Array` class on the right-hand-side is:

The name of the function is “`Complex :: operator () ()`”. We can call it with “`c.operator()(3.4, 2.1)`” or “`c(3.4, 2.1)`”

`void Complex :: operator () (float real, float imaginary);`

Return value can be anything.
In this case, the operator works like a setter

The left-hand-side is `this` because we are overloading a member operator

There can be any number of parameters here. In this case, we are sending to built-in parameters by-value

Implementation

The following is the implementation of the function-call operator for the `Complex` class:



```
void Complex :: operator () (float real, float imaginary = 0.0)
{
    this->r = real;
    this->i = imaginary;
}
```

Logical operators

The logical “and” (`&&`) and logical “or” (`||`) operators can be overloaded as with any other operator. The problem with overloading these operators originates in how they are usually used. Traditionally the “and” and “or” operators take a Boolean value on both the left-hand-side and the right-hand-side.

```
{  
    bool value1 = false;  
    bool value2 = true;  
  
    bool logicalAnd = value1 && value2;      // set intersection n  
    bool logicalOr  = value1 || value2;        // set union U  
}
```

Note that we can also use “and” to mean set intersection (the list of elements two sets have in common) and “or” to mean union (the list of elements in either of the two sets).

First, we will write a function to compute all the elements in common between two Arrays:

```
Array intersection(const Array & lhs, const Array & rhs) // return a new set, by value  
{  
    Array array;                                // the new Array to be returned  
  
    for (int i = 0; i < lhs.size(); i++)          // go though all the items  
        if (rhs.isMember(lhs.get(i)))              // if it is not present  
            array += lhs.get(i);                  // add it.  
  
    return array;                                // return the new array by-value  
}
```

Traditional functions

Operator overloading

Unit 2

The “and” operator can be implemented as:

```
Array Array :: operator && (const Array & rhs) const  
{  
    Array array;                                // the new Array to be returned  
  
    for (int i = 0; i < size(); i++)          // access size() or this->size()  
        if (rhs.isMember(data[i]))              // access the private member variable data[]  
            array += data[i];  
  
    return array;                                // return the same way: by-value  
}
```

The “or” operator (`||`) works in much the same way. While it is uncommon to want to overload the logical “and” or “or” operator, the union and intersection metaphor is commonly needed.

Member vs. non-member

As mentioned previously, most operators can be overloaded as a member or as a non-member. A few examples are:

Addition

Non-member	<pre>class Complex { ... code removed for brevity ... inline friend Complex operator + (const Complex & lhs, const Complex & rhs) { // (a + bi) + (c + di) = (a + c) + (b + d)i return Complex(lhs.r + rhs.r, lhs.i + rhs.i); } };</pre>
Member	<pre>class Complex { ... code removed for brevity ... Complex operator + (const Complex & rhs) const { // (a + bi) + (c + di) = (a + c) + (b + d)i return Complex(r + rhs.r, i + rhs.i); } };</pre>

Observe how we have one less parameter because `lhs` becomes `*this`. Also, if the `lhs` parameter with the non-member function is a `const`, the corresponding method is a `const` with member operator overloading.

Non-member	<pre>class Complex { ... code removed for brevity ... inline friend Complex & operator += (Complex & lhs, const Complex & rhs) { // (a + bi) + (c + di) = (a + c) + (b + d)i lhs.r += rhs.r; lhs.i += rhs.i; return lhs; } };</pre>
Member	<pre>class Complex { ... code removed for brevity ... Complex & operator += (const Complex & rhs) { // (a + bi) + (c + di) = (a + c) + (b + d)i r += rhs.r; i += rhs.i; return *this; } };</pre>

Because member functions have access to member variables without using the dot operator, the member variables `r` and `i` are directly accessible in the member version of the `+=` function. Also, since `this` changes in the member function, the method is not a `const`.

Increment prefix

Non-member

```
class Complex
{
    ... code removed for brevity ...
    inline friend Complex & operator ++ (Complex & c)
    {
        return c += 1.0;           // prefix ++ is exactly the same as += 1
    }
};
```

Member

```
class Complex
{
    ... code removed for brevity ...
    Complex & operator ++ ()
    {
        return *this += 1.0;       // prefix ++ is exactly the same as += 1
    }
};
```

In the non-member function, we returned “`c += 1.0;`”. Since we have no ‘`c`’ parameter, we have to use `*this` instead. Note that we could also have said “`return operator+=(1.0);`”.

Comparison

Non-member

```
class Complex
{
    ... code removed for brevity ...
    inline friend bool operator == (const Complex & lhs, const Complex & rhs)
    {
        return (lhs.r == rhs.r && lhs.i == rhs.i);
    }
};
```

Member

```
class Complex
{
    ... code removed for brevity ...
    bool operator == (const Complex & rhs) const
    {
        return (r == rhs.r && i == rhs.i);
    }
};
```

In each of the above examples, less code needs to be written to implement the operator as a member.

Calling operator functions

Operator overloading is designed to make it easier and more convenient for the client to call a common function. That being said, C++ allows the programmer to call the function either with the infix notation or the prefix notation. Therefore, the following are equivalent:

```
int main()
{
    cout << "Hello world";
    return 0;
}
```

```
int main()
{
    operator << (cout, "Hello world");
    return 0;
}
```

Of course, one would never want to do this. However, it does more clearly illustrate how functions work. Consider the following code for the `Complex` class:

```
class Complex
{
... code removed for brevity ...
    inline friend Complex & operator += (Complex & lhs, const Complex & rhs)
    { // (a + bi) + (c + di) = (a + c) + (b + d)i
        lhs.r += rhs.r;
        lhs.i += rhs.i;
        return lhs;
    }
};
```

In this example, it is possible to call the `+=` operator several ways:

```
{
    Complex c1(4, 5);
    Complex c2(6, 1);
    Complex c3(6, 0);

    c1 += c2 += c3;
}
```

```
{
    Complex c1(4, 5);
    Complex c2(6, 1);
    Complex c3(6, 0);

    operator += (c1, operator += (c2,
        c3));
}
```

From this example, it becomes clear why the `+=` operator needs to return a `Complex` object by-reference. If it was lacking, it would be impossible to chain more than one `+=` operator on the same line.

Sam's Corner



It is very important to implement operator overloading so the functions work the way the client expects. If an unfamiliar metaphor is used or if the parameters are not implemented the way they do for the built-in data types, the whole purpose of operator overloading will be defeated: the class will be harder not easier to work with!

Example 2.8 – Complex

Demo This example will demonstrate how to implement the common operators as member functions. In this example, every function that can be overloaded as a method will be, regardless of whether it is easier for the client to work. All the operators will be overloaded as `inline` functions.

Problem Take the code from “Example 2.7 – Complex” and convert every possible non-member operator to a member operator.

In this example, all the methods are implemented as `inline`.

```
class Complex
{
    ... code removed for brevity ...
public:
    Complex operator + (const Complex & rhs) const
    {
        return Complex(r + rhs.r, i + rhs.i);
    }
    Complex operator + (double rhs) const
    {
        return Complex(r + rhs, i);
    }
    // cannot be a method because the left-hand-side is not a Complex
    inline friend Complex operator + (int lhs, const Complex & rhs)
    {
        return rhs + lhs;
    }

    Complex & operator += (const Complex & rhs)
    {   // (a + bi) + (c + di) = (a + c) + (b + d)i
        r += rhs.r;
        i += rhs.i;
        return *this;
    }
    Complex & operator += (double rhs)
    {   // (a + bi) + c = (a + c) + bi
        r += rhs;
        return *this;
    }

    Complex & operator ++ ()
    {
        return *this += 1.0;
    }
    Complex operator ++ (int postfix)
    {
        Complex old(*this);
        *this += 1.0;
        return old;
    }
};
```

The complete solution is available at [2-8-complex.html](#) or:

```
/home/cs165/examples/2-8-complex/
```

See Also

Example 2.8 – Card

Demo

This example will demonstrate how to implement the common operators as member functions. In this example, every function that can be overloaded as a method will be, regardless of whether it is easier for the client to work. All the operators will be overloaded as non-`inline` functions.

Problem

Take the code from “Example 2.7 – Card” and convert every possible non-member operator to a member operator.

Solution

In this example, all the methods are not implemented as `inline`.

```
class Card
{
    ... code removed for brevity ...
    // insertion and extraction operators
    friend std::ostream & operator << (std::ostream & out, const Card & card);
    friend std::istream & operator >> (std::istream & in, Card & card);

    // increment and decrement ... only changing rank
    Card & operator ++(); // prefix
    Card & operator --(); // prefix
    Card operator ++(int postfix); // postfix
    Card operator --(int postfix); // postfix

    // change a card by adding or subtracting one
    Card operator +(const int input) const;
    friend Card operator +(const int input, const Card & lhs);
    Card operator -(const int input) const;
    Card & operator +=(const int input);
    Card & operator -=(const int input);

    // assignment
    Card & operator =(const Card & rhs);
    Card & operator ()(int iSuit, int iRank);

    // Relative comparison... only comparing rank
    bool operator >=(const Card & rhs) const;
    bool operator >(const Card & rhs) const;
    bool operator <=(const Card & rhs) const;
    bool operator <(const Card & rhs) const;

    // Absolute comparision... comparing both rank and suit
    bool operator ==(const Card & rhs) const;
    bool operator !=(const Card & rhs) const;
};
```

See Also

The complete solution is available at [2-8-card.html](#) or:

```
/home/cs165/examples/2-8-card/
```

Example 2.8 - Array

Demo	This example will demonstrate how to use the array-index operator as well as use the <code>+=</code> operator to append, the <code>&&</code> operator to find set intersection, and the <code> </code> operator to find set union.
Problem	Write a function to implement an <code>Array</code> . This will behave much like the STL vector class except it will append with the <code>+=</code> operator rather than <code>push_back()</code> and it will display the contents with the insertion operator.
Solution	<p>In this example, all the methods are implemented as <code>inline</code>.</p> <pre>class Array { ... code removed for brevity ... public: // access a given item for getting and setting double & operator [](int index) throw(bool) { if (index < 0 index >= sizeArray) throw false; return data[index]; } // push an item onto the back of the list Array & operator += (double value) throw(bool) { grow(sizeArray + 1); data[sizeArray++] = value; return *this; } // copy the contents of one Array onto another Array & operator = (const Array & rhs) throw(bool) { grow(rhs.capacity()); sizeArray = 0; for (int i = 0; i < rhs.size(); i++) (*this) += rhs.get(i); return *this; } // grow the array to a given size void grow(int capacity) throw(bool); // set intersection Array operator && (const Array & rhs) const; // set union Array operator (const Array & rhs) const; // fetch the size or capacity int size() const { return sizeArray; } int capacity() const { return capacityArray; } private: double * data; // where the data for the array is stored int sizeArray; // the number of items currently in the array int capacityArray; // the capacity of the array };</pre>
See Also	The complete solution is available at 2-8-array.html or: <code>/home/cs165/examples/2-8-array/</code>

Review 1-8

What are the data types for the return value, left-hand-side, and right-hand-side for the following operators for the `Complex` class?

Expression	Return value	Left-Hand-Side	Right-Hand-Side
1. <code>lhs + rhs</code>			
2. <code>lhs += rhs</code>			
3. <code>lhs++</code>			
4. <code>++lhs</code>			
5. <code>lhs == rhs</code>			
6. <code>lhs << rhs</code>			
7. <code>lhs >> rhs</code>			
8. <code>-lhs</code>			

Please see page 197 for a hint.

Problem 9

Given the following friend non-member operator overloading:

```
Time operator + (const Time & lhs, int secondsToAdd)
{
    return Time(lhs.secondsSinceMidnight + secondsToAdd);
}
```

Define the member version of the addition operator.

Please see page 194 for a hint.

Problem 10

Given the following friend non-member operator overloading:

```
bool operator == (const Time & lhs, const Time & rhs)
{
    return (lhs.secondsSinceMidnight == rhs.secondsSinceMidnight);
}
```

Define the member version of the equivalence operator.

Please see page 194 for a hint.

Problem 11

Write an assignment operator for the following class:

```
class Time
{
public:
    Time() : secondsSinceMidnight(0) {}
    Time(int rhs) : secondsSinceMidnight(rhs) {}
    Time(const Time & rhs)
    {
        secondsSinceMidnight = rhs.secondsSinceMidnight;
    }
private:
    int secondsSinceMidnight;
};
```

Please see page 190 for a hint.

Problem 12

Given the following `Position` structure:

```
struct Position
{
    int row;
    int col;
};
```

Given the following `TicTacToe` class:

```
class TicTacToe
{
    ... code removed for brevity ...
private:
    char board[3][3];
};
```

And given the following method:

```
char & TicTacToe :: getPiece(const Position & pos)
{
    return board[pos.row][pos.col];
}
```

Write the array-index operator for the `TicTacToe` class.

Please see page 191 for a hint.



Unit 3. Inheritance & Polymorphism

3.0 Class Relations.....	203
3.1 Building Polymorphism	212
3.2 Inheritance.....	221
3.3 Inheritance Qualifiers.....	231
3.4 Virtual Functions.....	241
3.5 Pure Virtual Functions.....	253

Unit 3. Inheritance & Polymorphism

3.0 Class Relations

There is something that has been bothering Sue since she began working on the Skeet project. Though she can see the value in creating a separate class for each of the many game entities, a couple of the classes look a lot alike. Particularly, she noticed that 90% of her Bullet class and her Bird class are the same. Surely there has got to be a way to leverage these similarities and avoid all of this redundant code.

Objectives

By the end of this chapter, you will be able to:

- Describe the difference between “is-a” and “has-a” relations
- Draw the UML class diagram describing “is-a” and “has-a” relations
- Explain why a programmer would want to use class relations in a program design

Prerequisites

Before reading this chapter, please make sure you are able to:

- Draw the UML class diagram for a single class (Chapter 2.0)
- Recite, describe, and use the four encapsulation design rules (Chapter 2.0)

What are class relations and why you should care

Often there exist relationships between the custom data types we create in a programming project. In the context of a Graphical User Interface (GUI), a window may consist of a collection of widgets, where a widget could be a button, scrollbar, or edit control. While it is possible to design a GUI from a set of disconnected objects, this design would not be leveraging the similarities of the classes and would have a great deal of redundant code. Class relations encompass all the different ways in which two classes can be related.

This chapter will help us to identify the basic types of class relations, learn design tools enabling us to identify and visualize these relationships, and discover guidelines to help us find the best way to capture these relationships in our data-structure designs.

Design patterns

On the simplest level, a class can be built from primitives (built in types such as integers and floats) or from other classes. When a class is defined in terms of another class, two possible relationships exist: “**has-a**” or “**composition**” where one class consists of a collection of subordinate classes, and “**is-a**” or “**derivation**” (also called “**inheritance**”) where one class is a manifestation or type of a parent class.

Has-A

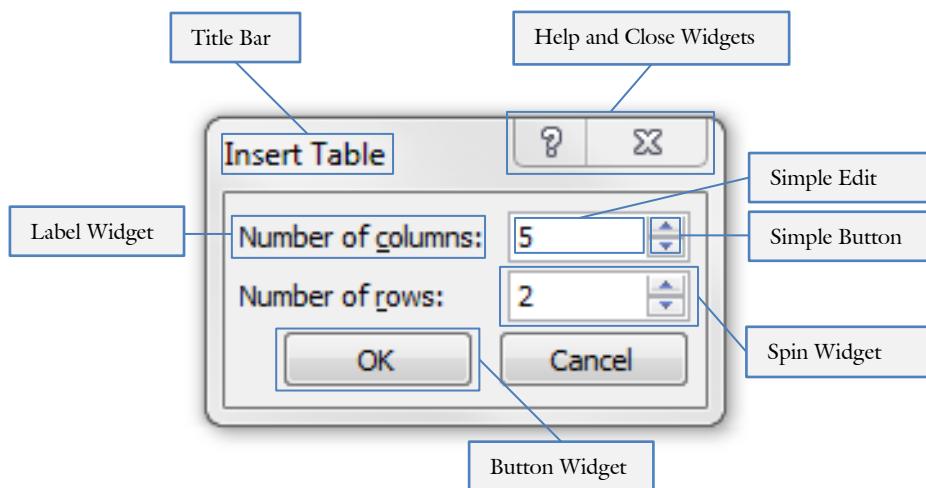
The simplest type of relationship is when one class is defined in terms of a collection of other classes. Back to our GUI example, a scrollbar consists of two buttons (up and down arrows), the moveable button (slider), and the track (where the moveable button slides). This is commonly called a “has-a” relationship because the scrollbar “has a” down-arrow button, “has an” up-arrow button, “has a” slider, and “has a” track.

Note that in composition relationships, one class has a distinct and persistent relationship with the parents. The up-arrow button, for example, behaves exactly the same in the scrollbar as it would if it were by itself.

Is-A

A more complex relationship between classes is when one class inherits many (but not necessarily all) of the properties of a parent (or the class we are building a new class off of). In many cases, it is a “type of” the parent. Back to our GUI example, a window has a collection of widgets. Every widget has a collection of properties, including the position (x, y), size ($xSize, ySize$), and some draw functionality. A button “is a” widget. It inherits the position and size properties from its widget parent, but it also has a collection of other properties (text label, click behavior, and enabling state to name a few). An edit control is another widget. It also inherits the position and size properties from its widget parent, but has a different collection of properties (user data, cursor location, and click and keyboard behaviors for example). Both the button and the edit control exhibit an “is-a” relationship with the widget because they inherit common properties and have a collection of other properties unique to them.

As an example, consider the following dialogue. It “has-a” Title Bar and it “has-a” collection of Widgets. There are several types of Widgets: the Label Widget “is-a” type of Widget, as is the Button Widget and the Spin Widget. The Spin Widget “has-a” Simple Edit and it “has-a” Simple Button.

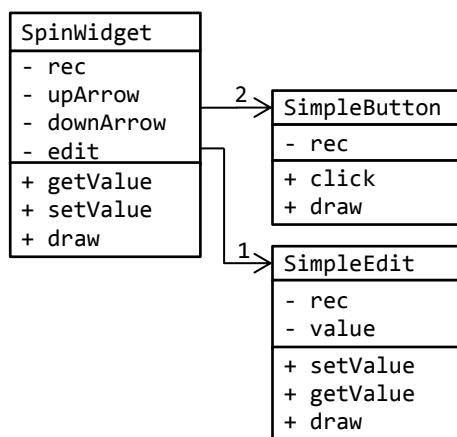


UML class diagrams

Possibly the best way to visualize class relations are through UML class diagrams. Through this tool, we can represent both composition (has-a) and inheritance (is-a) relationships.

Class diagrams for has-a

The composition relationship between classes is demonstrated in UML in a hierarchy by drawing arrows to the side of the child (or newly built class) from the side of the parent (what we are building from) using an open arrow (\rightarrow) with solid lines (—). We also specify the number of instances (called multiplicities) a given parent object is used in the child. This is important because composition relationships can include a large number of children. This could also be a range (1..4) or unlimited (*). Common multiplicities include: 0..1 (no instances or one instance), 1 (exactly one instance), 0..* (zero or more instances), and 1..* (at least one).



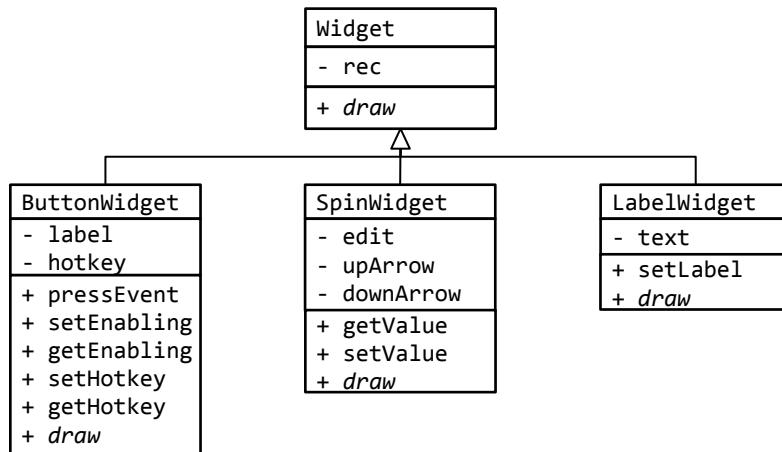
In this example, the `SpinWidget` class has a `SimpleEdit` and two `SimpleButtons`. Observe the three member variables (`edit`, `upArrow`, `downArrow`) typical of “has-a” relations. There are actually four flavors of “has-a” relationships: dependency, association, aggregation, and composition. In the context of this class, we will treat them all the same.

Name	Definition	Example
Dependency	Objects of one class work briefly with objects of another class.	A local variable in a method
Association	Objects of one class work with objects of another class for a prolonged amount of time.	A private member variable
Aggregation	One class owns but shares a reference to objects of another class	A public member variable
Composition	One class contains objects of another class	The <code>SpinWidget</code> example above.

(Miles & Hamilton, 2006, p. 84)

Class diagram for is-a

The inheritance relationship between classes is demonstrated in much the same way except a triangle arrow (Δ) is used and the lines lead from the top of the derived class and into the bottom of the base class. Note that you can't have more than one instance of a parent because, unlike composition relationships, inheritance is essentially the elaboration on a single type.



In this example, there are three manifestations of a `Widget`: the `ButtonWidget`, the `SpinWidget`, and the `LabelWidget`. Every child contains all the elements of the base `Widget` class. In this example, the `rec` property and the `draw()` function are in each of the derived classes.

There are two types of “is-a” relationships. The first is when the child *adds* properties and methods to the parent. This makes the child an elaboration of the parent. In the above example, `ButtonWidget` adds properties (`label` and `hotkey`) and methods (`pressEvent`, `setEnabled`, `getEnabled`, `setHotkey`, and `getHotkey`) to `Widget`. Therefore the `ButtonWidget` has three member variables (`rec`, `label`, and `hotkey`) and seven member functions.

The second type is when the child *redefines* a method from the parent. This is called **polymorphism** or “multiple shapes.” In our example above, `Button`, `Edit`, and `ScrollBar` each have a special version of `draw()`. We signify this type of relationship by italicizing the method `draw()`.

Example 3.0 – GUI

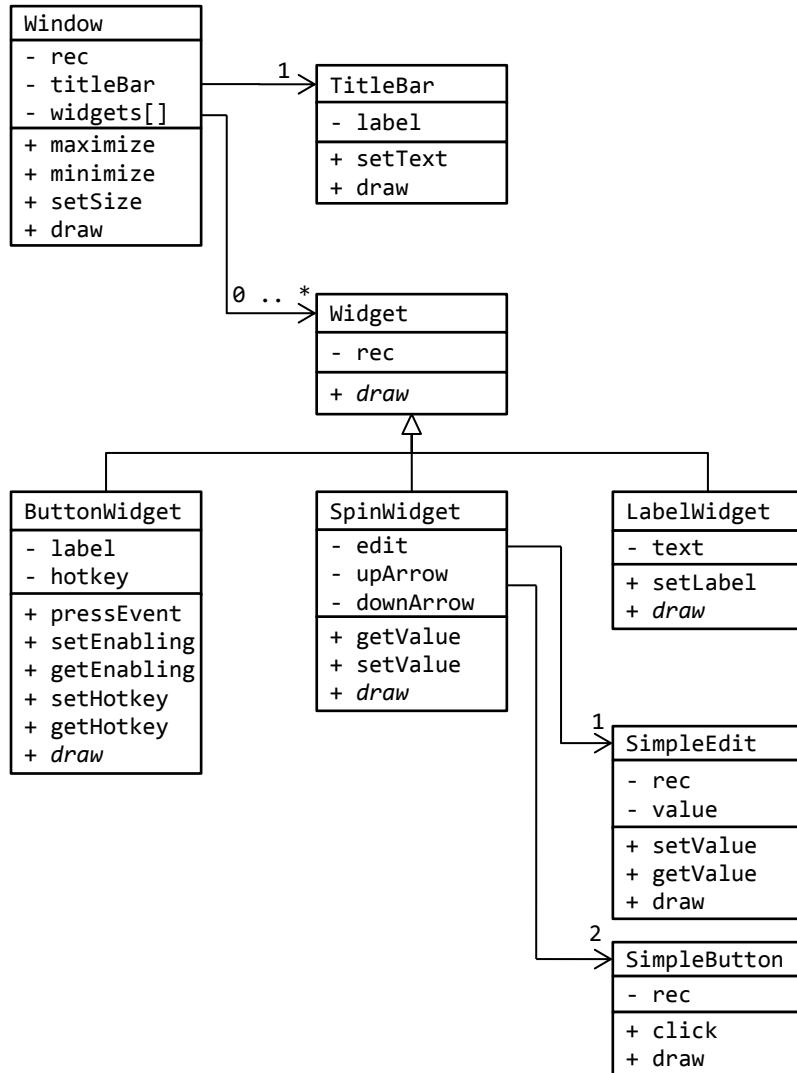
Demo

This example will demonstrate how “is-a” and “has-a” class relations can co-exist in a single problem. Though this problem may seem complex and contrived, it is fairly representative of a C++ program.

Problem

Write a UML class diagram to represent items on a Graphical User Interface (GUI). Start with the window level and drill down to the interactive elements on individual controls.

Solution



In this example, a `Window` has a `TitleBar` and an unbounded collection of `Widgets`. There are three types of widgets: a `ButtonWidget`, a `SpinWidget`, and a `LabelWidget`. While the `ButtonWidget` and `LabelWidget` are not composed of any other objects (ignoring the `Rectangle` type for the sake of simplicity), the `SpinWidget` is composed of three other objects (`upArrow` and `downArrow` `SimpleButtons`, the `SimpleEdit`).

Design considerations

In order to minimize the chance that our class relations design will become unwieldy or overly-complicated, a few design rules are in order: manage variant and invariant attributes carefully, be conscious of layers of abstraction, and prefer abstraction over universalization.

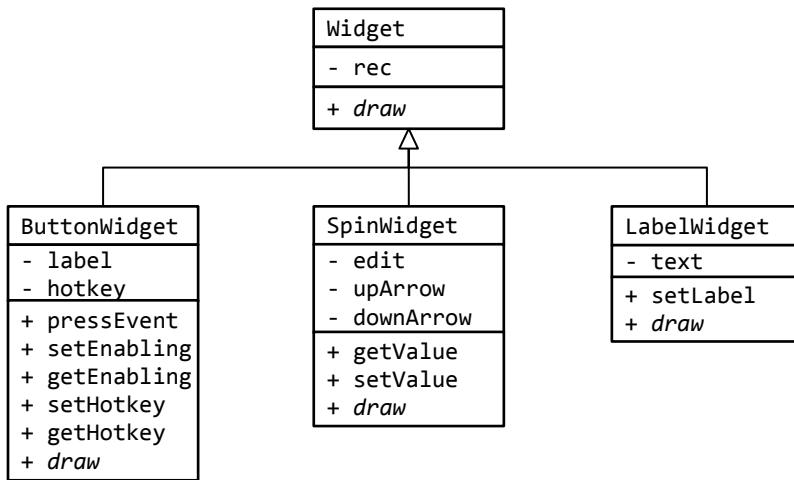
Rule #1: Manage variant and invariant attributes carefully

Use “is-a” relations when two classes share both variant properties (things that are different or distinct between the classes) and invariant properties (things that are shared or common between the classes).

When working with “is-a” relations, it is important to draw the line between the properties invariant between the children (every child and the parent share the same property), and the properties that are designed to be variant (many children are likely to have a different implementation). In one extreme, *every* property is **variant**. This raises the question of whether the children really should be related at all. If the children have nothing in common, why specify a relationship that does not truly exist or is meaningless? On the other extreme, if every property is **invariant**, then there are no differences between the children. If this is the case, why do we specify different children if there truly are no differences?

When designing an “is-a” relationship, the parent or base-class should contain all the invariant properties. This will ensure that all the children inherit the invariant properties. Specify in the child classes all the variant properties. This will represent what is unique about each child.

For example, consider our `Widget` class with its three children:



Each `Widget` will have a position (`Widget::rec`) and a draw method (`Widget::draw()`). Therefore, these are invariant and are therefore rightfully in the base class. The invariant part of the `Button` class includes the label (`ButtonWidget::label`), the hotkey value (`ButtonWidget::hotkey`), and a host of methods.



Sue's Tips

Designing class relations is much easier when you think in terms of variants and invariants. As long as the list of invariants is not empty (there is something in common between the classes being considered) and the list of variants is not empty (there is something unique about each class), then class relations through “is-a” is likely to be a good choice.

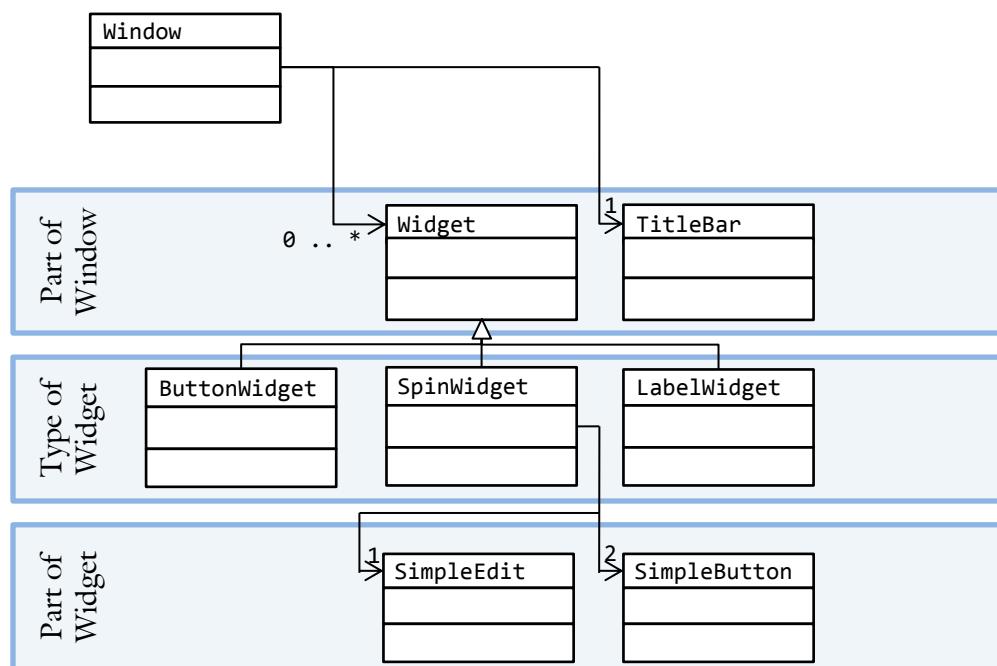
Rule #2: Be conscious of layers of abstraction

When more than one or two “is-a” relations exist, the relations hierarchy can become cumbersome and fragile. You can avoid this problem by making sure that each level of abstraction is consistent and meaningful.

A parent class represents a higher level of abstraction than the child because the child is a more specific version of the parent. When there are several levels of inheritance, the number of levels of abstraction can become quite complex. One design consideration when working with object relationships is to try to make the levels of abstraction consistent and meaningful.

Possibly the best example of this is the major taxonomic levels of living things such as plants and animals. This taxonomy, called the Linnaean taxonomic system (from Carolus Linnaeus 1707-1778 who described it in 1735), consists of seven major divisions: {Kingdom, Phylum, Class, Order, Family, Genus, Species}. There are two aspects of this taxonomy: hierarchy and rank. **Hierarchy** is preserved because the lower you go in the taxonomy the more similar the organism. Members of a Species, for example, are guaranteed to be more similar than members of a Genus. The second, **rank**, relates to the layers of abstraction. The level of specificity for definition of a Species is consistent across the taxonomy. In other words, the requirements for creating a new Species is the same across all Genera of the taxonomy.

Consider the levels of abstraction in our GUI example. If done correctly, each level should mean something. Changes made to the hierarchy contrary to the layers of abstraction are likely to compromise any benefit of the abstraction in the first place.



Pay careful attention to maintaining a consistent abstraction (or rank) level across your design (Wong, Union Design Pattern, 2007). Changes made to the hierarchy contrary to the layers of abstraction are likely to compromise any benefit of the abstraction in the first place

Rule #3: Prefer abstraction over universalization

Though universalization solves the same set of problems as abstraction, the universalization approach yields tighter coupling and looser cohesion. Generally, abstracting is a superior design approach.

There are two basic approaches to representing object relationships: universalizing and abstracting. **Universalizing** is the process of making a single type containing all possible attributes. In our GUI example from earlier, this would entail creating a single object containing a superset of the properties of every widget type. This would mean that the vast majority of the properties would not be meaningful in a given context. The property `cursorPosition` might be very useful in the `Edit` widget, but have no meaning in the `Button` widget. The approach of creating one object representing everything is called universalizing.

Abstracting, on the other hand, is the process of creating an entity capturing the *essence* of relationships and allowing specific implementations to describe their details. In this scenario, only the pertinent properties are available for the object; no irrelevant attributes exist.

Object oriented design principles lean heavily on the abstracting approach over the universalizing approach in an effort to reduce irrelevant data and develop more cohesive data types. Be mindful of temptations to create universal designs; a better abstract design is probably available.

Sam's Corner



Working with universalization is complex and problematic. If there are 100 variations of a given entity in a program, then every function working with entities needs to be aware of all 100 variations. This means that every time an entity is added to the list, every function knowing about entities will need to be modified. Even a single missed function will result in a bug! Programmers used universalization techniques in the past because there were no good abstraction tools. Now, with object oriented languages such as C++, there is seldom a good reason to resort to universalization.

As a general rule, if the predominant line in your function is a massive SWITCH statement, universalization may be rearing its ugly head!

Problem 1 – 5

Given the UML class diagram on the right, what is the relation between:

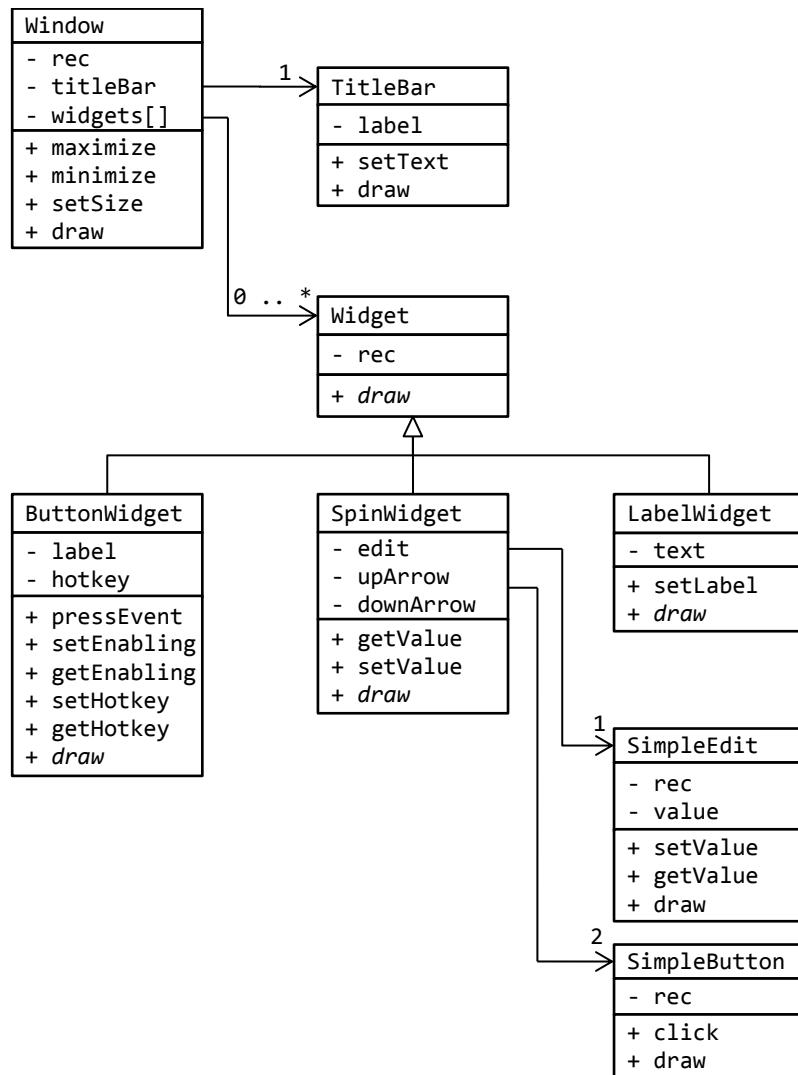
1. SimpleEdit and SpinWidget?

2. ButtonWidget and Widget?

3. SimpleEdit and SimpleButton?

4. Widget and TitleBar?

5. Window and TitleBar?



Please see page 205 and 206 for a hint.

Challenge 6

Create a UML class diagram describing the data-structures in the game “Deal or No Deal.” You can play the game at:

/home/cs165/examples/3-0-dealOrNoDeal.out

Please see page 208 for a hint.

Challenge 7

Create a UML class diagram describing the data-structures for the card game “War.” You can play the game at:

/home/cs165/examples/3-0-war.out

Please see page 209 for a hint.

Unit 3. Inheritance & Polymorphism

3.1 Building Polymorphism

Sam loves the idea of using class relations to solve programming problems. However, he can't see how it can work. Worse! He is afraid that it will incur a huge performance penalty. The only way he can trust a new programming technique is to know how it works. While digging around the internet to get to the bottom of this mystery, he remembers reading about V-Tables earlier in the semester. This should make "is-a" relations quite easy...

Objectives

By the end of this chapter, you will be able to:

- Describe how "is-a" relations can be accomplished with V-Table assignments
- Appreciate how efficient "is-a" can be

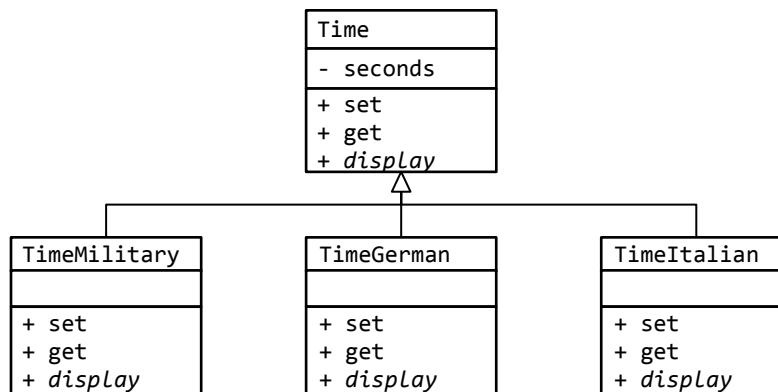
Prerequisites

Before reading this chapter, please make sure you are able to:

- How to define a pointer to a function (Chapter 1.5)
- How to build a class out of a structure and function pointers (Chapter 2.1)
- Create a UML class diagram describing "is-a" relations (Chapter 3.0)

What is polymorphism and why you should care

Polymorphism is the process of one class having more than one variation. Each variation honors the same contract (called "interface") though the behavior details (called "implementation") may be different. Consider, for example, a `Time` class. An object of this class is able to display a time in the traditional US format "3:28pm." We also wish to have variations of this class that use a different time format. We might have the military time format "15:28" as well as the German format "15.28" and the Italian format of "03:28 PM." If we were to have an array of `Time` objects, some of which were traditional, some are military, some German, and some Italian, then polymorphism would be at work. Though each member of the array is of type `Time`, four different flavors are represented in the array.



The purpose of this chapter is to illustrate how the compiler is able to implement polymorphism. This is important for understanding the performance and space implications of making a class polymorphic. Additionally, polymorphism in C++ has a few quirks that are difficult to understand unless you can see the underlying mechanisms. By seeing how the compiler implements polymorphism, these quirks make a lot more sense and polymorphism becomes much easier to use.

Binding

Binding is the process of connecting the name of a function or variable with the location in memory where the code or data resides. If, for example, you instantiate an integer called count (`int count;`), the compiler sets aside four bytes of memory for the integer and binds the memory to the name `count`. The same happens with functions. When a function is defined, the instructions corresponding to the function are placed into memory and the compiler binds the name with that memory address. If the programmer attempts to call a function that is not defined, then the compiler (actually the linker) reports the binding error and compilation is halted.

There are two flavors of binding that occur in a program: early binding and late binding.

Early binding

Early binding is the flavor of binding that the compiler performs. In a traditional program, the function name is “connected” with the code of the function at compile time. In other words, when a function is referenced in some caller code, the compiler knows exactly where the code for the function is. This process is called “early binding” because the binding between the name of the function and the code of the function occurs before the program is executed. Errors in early binding are displayed as a compiler error.

Late binding

Late binding is the flavor of binding that occurs while the program is executing or running. Consider an array of ten integers. If the program initializes one of the members of the array with “`array[index] = 0;`”, then it is impossible for the compiler to tell which location in memory is being changed. The reason is that the value of `index` is not known at compile time; it is resolved at run-time. If the programmer attempts to access an item in an array with an index that is out of bounds (such as `index == -1`), the compiler cannot catch the programmer’s error. Instead we will get a run-time error such as a crash or a segmentation fault.

Segmentation fault (core dumped)

All late binding mechanisms involve pointers, and all uses of pointers indicates late binding is at work. Therefore pointers to variables, arrays, and pointers-to-functions are indications of late binding.

In polymorphic scenarios, the compiler does not know which version of a function will get executed until run time. In this scenario, the function name is actually a pointer. When the object owning the pointer gets instantiated, then the pointer gets initialized. In other words, it is unknown which version of the function will get associated with (or assigned to) the pointer until the object is created. This process an example of late binding because the binding between the name of the function and the code of the function occurs after the program execution has begun.

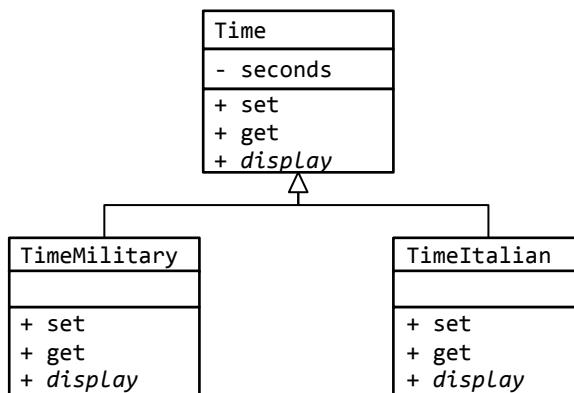
Polymorphism in C++ is accomplished by assigning the v-table to a class when an object is instantiated allowing the programmer to specify the flavor of a class at run-time.

Virtual method tables and polymorphism

Recall from Chapter 2.1 how a class can be built from a structure and function pointers. This could be done by having one function pointer for each method or by creating a special structure called a virtual method table (v-table) containing all the function pointers.

```
struct VTableTime
{
    void (*display)(const Time * pThis);
    void (*set)(Time * pThis, int hour, int minute, int second);
    void (*get)(const Time &* pTime, int & hour, int & minute, int & second);
};
```

In most cases, all the objects from a given class use the same v-table. For example, if there are ten `Time` objects, each object will use the same `display()` function. What happens when you want to have many variations of the `Time` class: one displaying a date using the short format 3:28pm, another using the military version 15:28, and yet another using the Italian format 05:28 PM?



The only differences between these three classes is the different `display()` functions.

```
void displayTime(const Time * pThis);           // 3:28pm
void displayTimeMilitary(const Time * pThis);    // 15:28
void displayTimeItalian(const Time & pThis);     // 03.28 PM
```

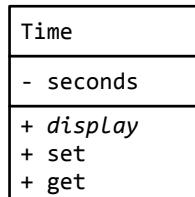
With these functions, we can initialize our three v-tables:

```
const VTableTime V_TABLE_TIME =                  // for the Time variation
{
    &displayTime,                                // for Time::display() with 3:28pm
    &setTime,                                    // for Time::set()
    &getTime                                     // for Time::get()
};

const VTableTime V_TABLE_TIME_MILITARY =          // for the Military Time variation
{
    &displayTimeMilitary,                      // for Time::display() with 15:28
    &setTime,                                    // for Time::set()
    &getTime                                     // for Time::get()
};

const VTableTime V_TABLE_TIME_ITALIAN =            // for the Italian Time variation
{
    &displayTimeItalian,                       // for Time::display() with 03:28 PM
    &setTime,                                    // for Time::set()
    &getTime                                     // for Time::get()
};
```

In Chapter 2.1, we saw how to turn a UML class diagram into a structure:



The class implementation of `Time` using structures is:

```
struct Time                                // the class/structure tag Time
{
    int seconds;                           // the one member variable: seconds
    VTableTime * __vptr;                  // all the methods are in VTableTime
};                                         // using the standard name __vptr
```

As you may recall, using this manual method of instantiating an object requires us to assign the v-table to the object in a separate step:

```
{
    Time time1;                           // no methods currently attached
    time1.__vptr = &V_TABLE_TIME;        // this gets the standard display()
    time1.__vptr->set(&time1, 15, 28, 0);

    Time time2;                           // with the Military version of the
    time2.__vptr = &V_TABLE_MILITARY;   //     v-table, this gets the
    time2.__vptr->set(&time2, 15, 28, 0); //     displayTimeMilitary() function

    Time time3;                           // displayTimeItalian() here
    time3.__vptr = &V_TABLE_ITALIAN;
    time3.__vptr->set(&time3, 15, 28, 0);

    // now display all three Times
    time1.__vptr->display(&time1);      // 3:28pm
    time2.__vptr->display(&time2);      // 15:28
    time3.__vptr->display(&time3);      // 03:29 PM
}
```

There are a couple key take-aways from this:

- The cost of binding a v-table to an object is a single instruction. This is extremely fast.
- Once the appropriate v-table is binded to the object immediately after the object is instantiated, there is no run-time cost to polymorphism.
- Having one derived class off of `Time` and having a thousand derived classes cost the same. Therefore there is zero cost to adding a derived class to a base class.

Example 3.1 – Date

Demo

This example will demonstrate how to create a simple polymorphic `Date` class using structures and function pointers instead of true C++ classes.

Problem

Write a program to represent three variations of dates: a long date in the form of “11th of November, 1888,” a short date in the form of “11/12/1888,” and an empty date displaying the empty string.

The first part of the solution is to make a v-table representing the methods of `Date`:

```
struct VTableDate
{
    bool (*set)(Date * pThis, int year, int month, int day);
    void (*display)(const Date * pThis);
};
```

Next the class definition will include the three member variables and the v-table:

```
struct Date
{
    VTableDate *_vptr;
    int year;
    int month;
    int day;
};
```

Finally it is necessary to bind the appropriate v-table to the newly created `Date` object.

```
void bind(Date * pThis, DateType td)
{
    // these need to be static. If not, when the function is returned, these
    // variables will fall out of scope and be destroyed. That will cause
    // the program to crash...
    static VTableDate vTableDateShort = { &setDate, &displayShort };
    static VTableDate vTableDateLong = { &setDate, &displayLong };
    static VTableDate vTableDateNone = { &setDate, &displayNone };

    switch (td)
    {
        default:
            assert(false); // this should never happen!
            // fall through
        case NONE:
            pThis->_vptr = &vTableDateNone;
            break;
        case SHORT:
            pThis->_vptr = &vTableDateShort;
            break;
        case LONG:
            pThis->_vptr = &vTableDateLong;
            break;
    }
}
```

Solution

The complete solution is available at [3-1-date.html](#) or:

/home/cs165/examples/3-1-date.cpp



See Also

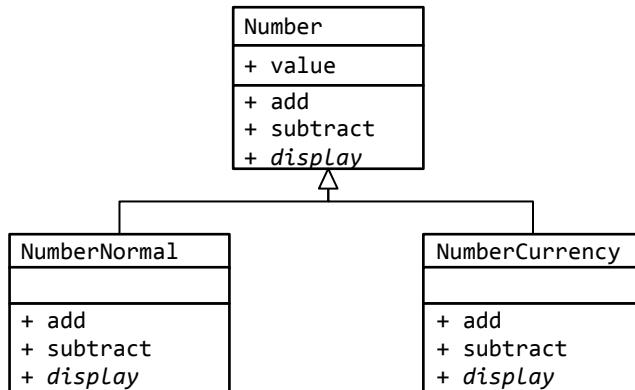
Example 3.1 – Numbers

Demo

This example will demonstrate how to create a simple polymorphic `Number` class using structures and function pointers instead of true C++ classes.

Problem

Write a program to represent two variations of numbers: one displaying the value as an integer and the second as US dollars currency.



The first part of the solution is to make a v-table representing the methods of `Number`:

```
struct VTableNumber
{
    void (*add)(          Number * pThis, const Number & rhs);
    void (*subtract)(     Number * pThis, const Number & rhs);
    void (*display)( const Number * pThis);
};
```

The class definition is extremely simple:

```
struct Number
{
    double value;
    const VTableNumber * __vptr;
```

The binding occurs in `main()`:

```
int main()
{
    ... code removed for brevity ...
    num1.__vptr = &V_TABLE_NUMBER;           // default member functions
    num2.__vptr = &V_TABLE_NUMBER_CURRENCY; // currency member functions
    ... code removed for brevity ...
    num1.__vptr->display(&num1);           // 42
    num2.__vptr->display(&num2);           // $41.99
}
```

See Also

The complete solution is available at [3-1-numbers.html](#) or:

```
/home/cs165/examples/3-1-numbers.cpp
```



Problem 1

Write the UML class diagram for the following class:

```
class Date
{
public:
    void set(int year, int month = 1, int day = 1);
    void display() const;
    bool operator == (const Date & rhs) const;
    bool operator != (const Date & rhs) const;
    const Date & operator = (const Date & rhs);
private:
    bool isLeapYear() const;
    int daysSince1753;
};
```

Please see page 110 for a hint.

Problem 2

Write the UML class diagram for the following class:

```
class Assignment
{
public:
    const Assignment & operator = (const Date & rhs );
    const Assignment & operator = (const string & rhs);
    void setScore( int score);
    void setPoints(int points);
    void display() const;
    const string & getName() const;
    const Date & getDate() const;
    int getScore() const;
    int getPoints() const;
private:
    bool isValid(int score, int points);
    Date date;
    string name;
    int score;
    int points;
};
```

Please see page 205 for a hint.

Problem 3

Write the UML class diagram for the following class:

```
class GradeBook
{
public:
    const GradeBook & operator += (const Assignment & rhs);
    const Assignment & operator [] (int index);
    void display() const;
private:
    vector <Assignment> grades;
};
```

Please see page 205 for a hint.

Problem 4

Write the UML class diagram for the following class:

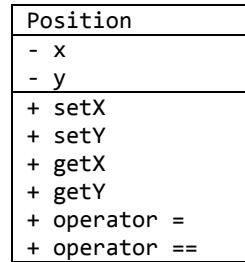
```
class Date
{
public:
    void set(int year, int month = 1, int day = 1);
    void (*display)(const Date *pThis);
    void setLanguage(int code);
    int getYear() const;
    int getMonth() const;
    int getDay() const;
private:
    bool isLeapYear() const;
    int daysSince1753;
};

void displayEnglish(const Date *pThis);
void displayGerman( const Date *pThis);
void displayFrench( const Date *pThis);
```

Please see page 206 for a hint.

Problem 5

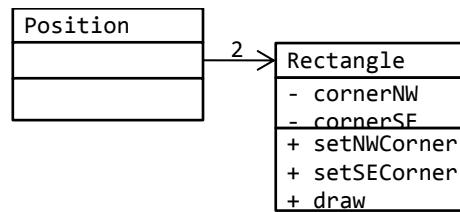
Write the class definition to match the following UML class diagram:



Please see page 127 for a hint.

Problem 6

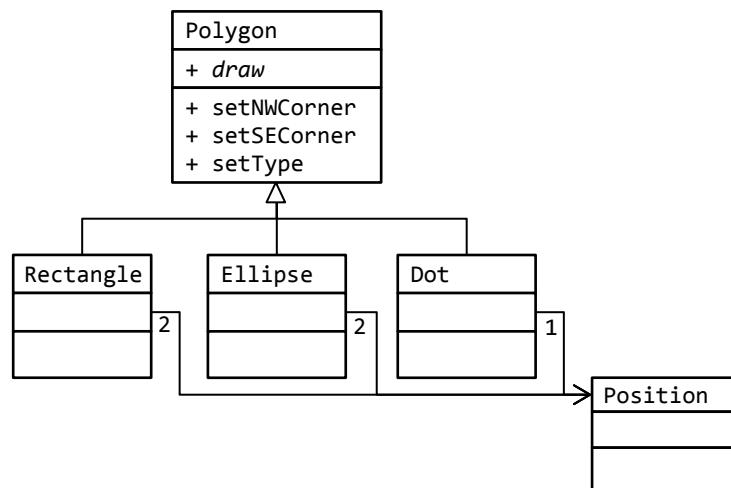
Write the class definition for Rectangle to match the following UML class diagram:



Please see page 127 for a hint.

Problem 7

Write the class definition for Polygon to match the following UML class diagram:



Please see page 216 for a hint.

Unit 3. Inheritance & Polymorphism

Unit 3

3.2 Inheritance

Sam has just finished a `Position` class to represent where a drawn object resides on the screen for his Skeet game. This class has a collection of getters and setters allowing the client to move objects as the game is played. Next Sam is going to write a class to represent the `Bird`. This means that all the getters and setters for the `Position` class needs to be reproduced in the `Bird` class. If only there was a better way...

Objectives

By the end of this chapter, you will be able to:

- Define inheritance and explain when it would be useful
- Define a class that inherits off of a base class

Prerequisites

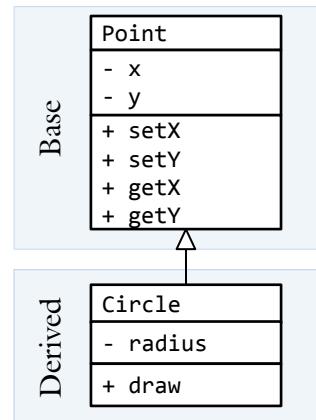
Before reading this chapter, please make sure you are able to:

- Define a class matching a UML class diagram (Chapter 2.0, 3.0)
- Draw a UML class diagram representing an “is-a” relation (Chapter 3.0)
- Design a program with UML class diagrams utilizing “is-a” relations (Chapter 3.0)

What is inheritance and why you should care

Inheritance is the process of building one class off of another. It represents the simplest form of “is-a” object relationships. There are two classes involved with inheritance: the base class and the derived class. The **base class**, also known as the parent class, is the class we start with. Any class can be a base class. The **derived class**, also known as the child class, is the class that is built from the base class. Inheritance occurs when the derived class inherits all the properties and methods from the base class.

Consider a class that stores a `Point`. In this case, `Point` represents the base class because another class will be built from it. Now we will build a derived class called `Circle` that will inherit the `Point` properties and methods. In addition to `Point`’s properties, `Circle` will also have a `radius`. In other words, the `Circle` is “all that and more” to the `Point`.



Inheritance helps us leverage what is common (or invariant) between related classes with zero duplicate code. There is no need to write another version of `Point`’s getters and setters with inheritance, `Circle` simply inherits them from the base class.

Syntax of inheritance

When specifying inheritance relations, all the syntax exists in the base class. C++ does not allow the programmer to indicate inheritance from the standpoint of the base class. If, for example, we wish to inherit `Circle` from `Point`, then the `Point` definition is completely standard:

```
class Point // no indication of inheritance in base class
{
public:
    Point() : x(0), y(0) { }
    Point(int x, int y) { setX(x); setY(y); }
    Point(const Point &pt) { setX(pt.x); setY(pt.y); }
    int getX() const { return x; }
    int getY() const { return y; }
    void setX(int x) { this->x = x; }
    void setY(int y) { this->y = y; }
    friend ostream & operator << (ostream & out, const Point & pt);
    friend istream & operator >> (istream & in, Point & pt);
private:
    int x;
    int y;
};
```

To indicate a derived class is inherited from a base class, it is necessary to put an inheritance indicator in the derived class definition. Consider the `Circle` class:

Here we indicate that `Circle` is a derived class from `Point`. We inherit all that `Point` has, including the member variables and methods.

```
class Circle : public Point // this indicates we inherit off of Point
{
public:
    Circle() : Point(), radius(0) { }
    Circle(int r, int x, int y) : Point(x, y) { setRadius(r); }
    Circle(const Circle & circle);
    int getRadius() { return radius; }
    void setRadius(int r) { this->radius = r; }
    friend ostream & operator << (ostream & out, const Circle & c);
    friend istream & operator >> (istream & in, Circle & c);
private:
    int radius;
};
```

The `circle` class definition begins by inheriting all that `Point` has with the “`: public Point`” part of the class definition. Notice how a circle, in this case, is just a point with a radius. This is usually how class relationships work; the derived class will have everything the base class does, and then some.

```
{
    Circle c(100, 10, 10); // this constructor also initializes x and y
    c.setX(20); // the member functions from Point are
    c.setY(20); // inherited in Circle
    c.setRadius(5); // we also get our own member functions

    cout << c << endl; // since the RHS of << is a Circle, we call
                         // Circle's method instead of Point's
}
```

What is inherited?

In a derived class, all the member variables and all the member functions of the base class are made a part of (or inherited from) the new class. There are three exceptions to this rule: constructors, destructors, and the assignment operator. They are not inherited.

Constructors and destructors are not inherited

Consider the following code:

```
class Base
{
public:
    Base()                      // default constructor
    {
        pBase = new int(0);      // allocate an int and assign to 0
        cout << "Base constructor\n"; // display a message
    }
    ~Base()                     // destructor
    {
        delete pBase;           // free the allocated memory
        cout << "Base destructor\n"; // display a message
    }
private:
    int * pBase;
};
```

```
class Derived : public Base
{
public:
    Derived()                  // default constructor. Note that
    {                          //     Base() gets called
        pDerived = new float(0.0); // allocate a float to 0.0
        cout << "Derived constructor\n"; // display a message
    }
    ~Derived()                 // destructor. Note that
    {                          //     ~Base gets called
        delete pDerived;       // free the allocated memory
        cout << "Derived destructor\n"; // display a message
    }
private:
    float * pDerived;
};
```

Here we have a simple base class and a derived class. Note that when we instantiate a `Derived` object, the base class constructor is not inherited but it is called.

```
int main()
{
    Derived d;
    return 0;
}
```

The output is:

```
Base constructor
Derived constructor
Derived destructor
Base destructor
```

The complete solution is available at [3-2-constructors.html](#) or: /home/cs165/examples/3-2-constructors.cpp.

Assignment operators are not inherited

Assignment operators are also not inherited. This means that we commonly need to define assignment operators for all the derived classes if one is defined for the base class. Consider the following code:

```
class Base
{
public:
    Base() { pBase = new int(0); } // constructor allocates
    ~Base() { delete pBase; } // destructor frees
    Base & operator = (const Base & rhs) // assignment operator will
    { // copy the values (not the
        *pBase = *rhs.pBase; // addresses) of pBase
        cout << "Base assignment operator\n"; // display a message
        return *this;
    }
private:
    int * pBase;
};
```

```
class Derived : public Base
{
public:
    Derived() { pDerived = new float(0.0); } // constructor allocates
    ~Derived() { delete pDerived; } // destructor frees
private:
    float * pDerived;
};
```

Note that there is no assignment operator defined for `Derived`. The compiler will generate a new one for us, one that will copy the member variables (addresses in this case! This will be a bug due to the pointers!) and will call the base-class assignment operator:

```
***** default assignment operator *****
Derived & operator = (const Derived & rhs) // default assignment operator
{
    Base::operator=(rhs); // calls the base-class assignment
    pDerived = rhs.pDerived; // ERROR: this copies the member
    return *this; // variables which is a pointer!
                // We should copy the data instead!
```

Now if we define our own assignment operator, the assignment operator in `Base` is not called!

```
***** assignment operator failing to call the base class *****
Derived & operator = (const Derived & rhs) // our own assignment operator
{
    *pDerived = *rhs.pDerived; // ERROR: base-class = not called
    return *this; // we need to copy the data, not the
                  // pointers
```

Note that `Base`'s assignment operator is not called! We therefore need to call it explicitly. The correct implementation for the assignment operator in the `Derived` class is:

```
***** the correct assignment operator *****
Derived & operator = (const Derived & rhs)
{
    Base::operator=(rhs); // call the base-class assignment
    *pDerived = *rhs.pDerived; // copy the data, not the addresses
    return *this; // return *this
```

The complete solution is available at [3-2-assignment.html](#) or: /home/cs165/examples/3-2-assignment.cpp.

Constructors

Thus far the syntax of inheritance seems straight-forward. The only thing that is necessary to derive off of a base-class is to include the “ : public BaseClass” indication in the class definition.

To illustrate this point, consider the following example:

```
class Base
{
public:
    Base() : value(0) { } // default constructor
    Base(int value) : value(value) { } // non-default constructor
    Base(const Base & rhs) { value = rhs.value; } // copy constructor
    int get() const { return value; }
private:
    int value;
};

class Derived : public Base // inherit Base::getBase() and value
{
public:
    Derived() { } // Base::Base() gets called
    Derived(int value) : Base(value) { } // call Base::Base(int)
    Derived(const Derived & rhs);
};
```

We can explicitly call any version of the base classes’ constructor we choose. If none is explicitly called, the default will be called for us. Otherwise, we can specify the version we wish in the initialization section.

Note that the only place we can call the base classes’ constructor is from the initialization section. If we attempt to do so from the body of the constructor, we will get unexpected results.

```
Derived :: Derived(const derived & rhs)
{
    Base(rhs.get()); // ERROR: we will create a new Base object
                     // and this->value is not set!
```

In this example, we create a new instance of `Base` in the body of the copy constructor. This object never gets assigned to a variable so it is called an “anonymous object.”



Sue’s Tips

It is a good idea to explicitly call the base classes’s constructor from each constructor in the derived class. This makes the intention of the programmer clear.

Redefining

A derived class can add additional methods and variables to those provided by the base class. It is also possible for the derived class to define a method with the same signature as that of the base class. This is called “redefining.” Consider the following example:

```
class Point
public:
    void display() const { cout << '(' << x << ", " << y << ")\\n"; }
... code removed for brevity ...
private:
    int x;
    int y;
};
```

```
class Circle : public Point
public:
    void display() const
    {
        cout << '(' << getX() << ", " << getY() << " r=" << radius << ")\\n"; }
... code removed for brevity ...
private:
    int radius;
};
```

In this example, we have two functions (`circle::display()` and `Point::display()`) with the same signature. When we call `display()` from a `Circle` object, we get the `Circle` version.

```
{
    Point pt;
    pt.display();                                // call Point::display()

    Circle c;
    c.display();                                  // since display() is redefined in Circle, we
                                                //     call Circle::display() here
}
```

Sam's Corner



Just because we re-defined a method in a derived class does not mean that we will never have access to it again. With the scope resolution operator (`::`), we can specify which version of a given redefined class we want to call. Back to our `Point` and `Circle` example from above:

```
{
    Point pt(4, 7);
    pt.display();                                // (4, 7)

    Circle c(10, 3, 8);
    c.display();                                  // (10, 3 r=8)

    c.Point::display();                          // (10, 3)
}
```

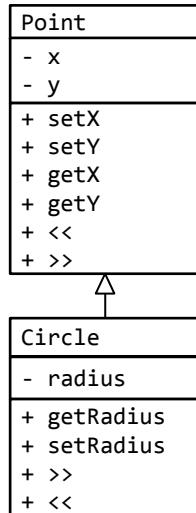
Example 3.2 – Circle

Demo

This example will demonstrate how to do simple inheritance with a single base class and a single derived class.

Problem

Write a class called `Circle` which inherits off of `Point`. Both `Circle` and `Point` implement the usual getters and setters as well as the insertion and extraction operators.



Solution

```
ostream & operator << (ostream & out, const Circle & circle)
{
    out << '(' << circle.getX() // need to use getters here. The member
        << ", " << circle.getY() // variables are not accessible.
        << ", r=" << circle.radius // we do have access to the radius private
        << ')';
    return out;
}
```

```
istream & operator >> (istream & in, Circle & circle)
{
    int x; // we don't have access to Point::x so
    int y; // we need to read into variables

    // read data from in
    in >> x >> y >> circle.radius; // since radius is part of Circle and >>
                                         // is a friend, we use circle.radius

    // now set the values
    circle.setX(x); // only the public Point::setX() and
    circle.setY(y); // Point::setY() is visible to Circle
    return in;
}
```

See Also

The complete solution is available at [3-2-circle.html](#) or:

/home/cs165/examples/3-2-circle.cpp



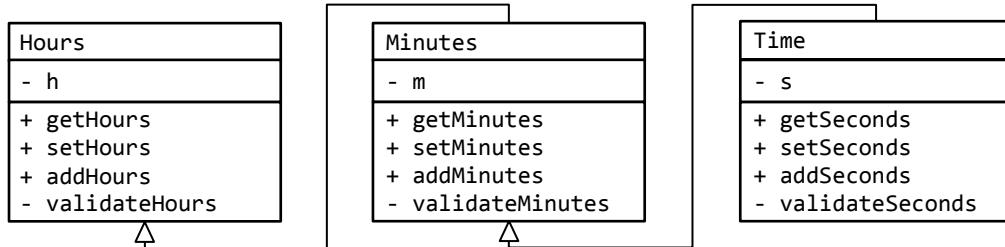
Example 3.2 – Time

Demo

This example will demonstrate how to accomplish multiple levels of inheritance. In other words, a derived class can also serve as a base class.

Problem

Write a class called `Hours`, one called `Minutes` which inherits off of `Hours`, and one called `Time` which adds seconds to `Minutes`.



The `Hours` class is the base class so it has no indications of inheritance:

```
class Hours {
public:
    Hours() : h(12) {}
    Hours(int h) : h(h) {}
    Hours(const Hours & hours) : h(hours.getHours()) {}
    int getHours() const { return h; }
    void setHours(int h) { if (validateHours(h)) this->h = h; }
    void addHours(int h);
private:
    bool validateHours(int h) const { return (0 <= h && h <= 23); }
    int h;
};
```

The `Minutes` class inherits from `Hours`. Notice how we must call our parent's constructor in the initialization section or `Hours::h` will not get set:

```
class Minutes : public Hours {
public:
    Minutes() : m(0), Hours() {}
    ... code removed for brevity ...
private:
    int m;
};
```

Finally, the `Time` class inherits from `Minutes` which, in turn, inherits off of `Hours`.

```
class Time : public Minutes {
public:
    Time() : s(0), Minutes() {}
    ... code removed for brevity ...
private:
    int s;
};
```

The complete solution is available at [3-2-time.html](#) or:

```
/home/cs165/examples/3-2-time.cpp
```

Solution

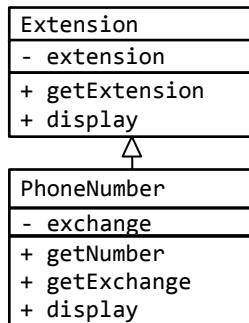
Example 3.2 – PhoneNumber

Demo

This example will demonstrate multiple levels of inheritance as well as redefining member functions.

Problem

Write a class called `PhoneNumber` which inherits from `Extension`.



Solution

The class definition of `PhoneNumber` is the following:

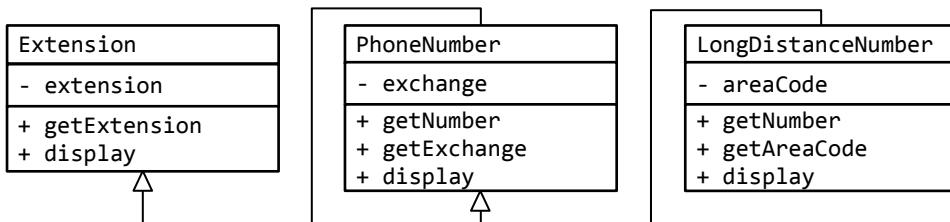
```
class PhoneNumber : public Extension
{
public:
    PhoneNumber(int exchange, int ext) : Extension(ext),
                                              exchange(exchange) {
        int getNumber() const { return exchange * 10000 + getExtension(); }
        int getExchange() const { return exchange; }
        void display() const { cout << exchange << '-' << getExtension(); }
private:
    int exchange;
};
```

Notice how the constructor for `PhoneNumber` calls `Extension`'s constructor in the initialization section.
Notice also how `PhoneNumber::display()` redefines `Extension::display()`.

```
cout << "phoneNumber.display(): ";
phoneNumber.display();                                // redefined from Extension
cout << endl;
cout << "phoneNumber.Extension::display(): ";
phoneNumber.Extension::display();                  // call Extension's version
cout << endl;
```

Challenge

As a challenge, see if you can create a new class called `LongDistanceNumber` which adds the area code to `PhoneNumber`.



See Also

The complete solution is available at [3-2-phoneNumber.html](#) or:

```
/home/cs165/examples/3-2-phoneNumber.cpp
```



Problem 1

Consider the following class:

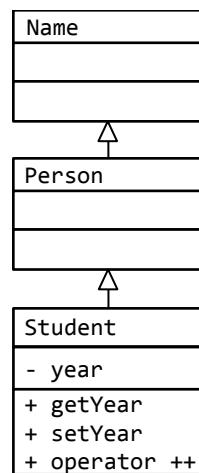
```
class Name
{
public:
    Name();
    Name(const Name & name);
    void setFirst( const string & s);
    void setMiddle(const string & s);
    void setLast( const string & s);
    const string & getFirst() const;
    const string & getMiddle() const;
    const string & getLast() const;
private:
    string first;
    string middle;
    string last;
};
```

Derive a class **Person** that adds gender to **Name**.

Please see page 222 for a hint.

Problem 2

Given the above defined **Name** and **Person** class, define a class called **Student** adding academic year:



Please see page 228 for a hint.

Unit 3. Inheritance & Polymorphism

3.3 Inheritance Qualifiers

Sue has just completed writing a class which is derived off of a base class. While using inheritance proved to be much simpler than duplicating all the functionality from the base class, there is an unfortunate side-effect. In order to make the methods of the derived class as efficient as possible, it is necessary for them to work with the base class' data rather than the publically exposed properties. How can this be done without exposing this data to the client as well?

Objectives

By the end of this chapter, you will be able to:

- Explain why inheritance qualifiers strengthens encapsulation by allowing the programmer to fine-tune access to class data
- Describe `protected` and list scenarios when it would be handy to use it
- For a given base class and derived class, explain why access to a given method or member variable is allowed or not allowed

Prerequisites

Before reading this chapter, please make sure you are able to:

- Explain the differences between `public` and `private` qualifiers (Chapter 2.2)
- Create a UML class diagram representing an “is-a” relation (Chapter 3.0)
- Create a class definition matching a given UML class diagram (Chapter 3.0)

What are inheritance qualifiers and why you should care

Access qualifiers such as `public` and `private` indicate the scope in which a given member variable or method is accessible. As you may recall, the `public` access qualifier indicates that a given member variable or method is accessible to both the client and the member functions within a class. The `private` access qualifier indicates that only a member function has access to a given method or member variable. A third access qualifier, `protected`, is used to indicate that a given member variable or method is accessible to either members of the class or classes which derive off of the base class. We use the # sign to signify this in a UML class diagram:

ClassName
+ publicVariable
protectedVariable
- privateVariable
+ publicMethod
protectedMethod
- privateMethod

Protected

The `protected` qualifier signifies a degree of access somewhere between `public` and `private`. There are three ways in which a given method or member variable might need to be accessed:

- Within: From within the class where the method or member variable is defined
- Derived: From within the class derived from a class where the method or variable is defined
- Object: From an object instantiated from the class where the method or variable is defined

While the `public` access modifier allows access from all three, and `private` restricts access within a class, `protected` is between:

	private	protected	public
Within: from within a class	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Derived: from within a derived class		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Object: from an object instantiated from a class			<input checked="" type="checkbox"/>

When the `protected` keyword works exactly like `private` when there is no derived class:

```
class Base
{
public:
    int basePublic;
    void basePublicMethod();
protected:
    int baseProtected;
private:
    int basePrivate;
};
```

The access an object has to member variables within `Base` is restricted to the `public` member variables.

```
{
    Base base;

    base.basePublic = 1;           // legal. basePublic is public so we have access from
                                  // the client.
    base.baseProtected = 1;        // ERROR! Only methods within Base (there are none) or
                                  // methods in the derived class have access
    base.basePrivate = 1;          // ERROR! No access to privates from the object
}
```

However, the method `Base::basePublicMethod()` has access to all the member variables:

```
void Base :: basePublicMethod()
{
    basePublic = 1;               // public member variables are accessible to methods
    baseProtected = 1;             // protected member variables are also accessible
    basePrivate = 1;               // privates are accessible here only
}
```

When working with a single class, `protected` and `private` work exactly the same. Why, then, would one want to make a member variable or member function `protected`? The answer is: because we would want to give a derived class access to a given member variable or method. In other words, we don't want to reveal details of our class implementation to the client but we might want to reveal those details to a derived class. The `protected` access modifier allows for this.

Consider the following class which derives off of `Base`:

```
class Derived : public Base
{
    public:
        void derivedPublicMethod();
        int derivedPublic;
    protected:
        int derivedProtected;
    private:
        int derivedPrivate;
};
```

The access an object has to member variables within `Base` and `Derived` is restricted to the public member variables.

```
{
    Derived derived;           // six member variables

    derived.basePublic = 1;    // legal because basePublic is public.
    derived.baseProtected = 1; // ERROR: no access to protected items from the object
    derived.basePrivate = 1;   // ERROR: no access to privates from the object

    derived.derivedPublic = 1; // legal because derivedPublic is public
    derived.derivedProtected = 1; // ERROR: no access to protected items
    derived.derivedPrivate = 1; // ERROR: no access to privates
}
```

Therefore, only `public` items are accessible from the object. From a member function, however, we have access to the base classes' `public` and `protected` items.

```
void Derived :: derivedPublicMethod()
{
    derivedPublic = 1;          // all member variables and member functions defined
    derivedProtected = 1;       //     in a class are accessible from a method
    derivedPrivate = 1;         //     belonging to that class.

    basePublic = 1;            // we have access to the base class's publics
    baseProtected = 1;          // we also have access to the protected items
    basePrivate = 1;            // ERROR! no access to privates from the
                               //     derived class
}
```

From this example, we can see that methods within the derived class have more access to the member variables in the base class than does the client. The client (objects created from the class) only has access to `public` methods and variables. Methods in the derived class have access to `public` and `protected` methods and variables. Methods and variables under the `private` designation are only access to methods within the same class.

Sam's Corner



The access qualifiers of `public`, `protected`, and `private` are compile-time checks. They only serve to help the programmer catch errors as the class is compiled rather than provide any run-time checks. Therefore they are early-binding constructs. This means that there is no performance penalty to using `protected` or any other access qualifier.

Inheritance indicators

Recall the syntax for defining a class that inherits off another class:

This is called an inheritance indicator. We specify not only which class from which we inherit, but also how the inheritance occurs

```
class Derived : public Base
{
... code removed for brevity ...
};
```

An **inheritance indicator** is the part of the class definition where the programmer specifies which if any class is the base class. Up until this point, we always inherited as `public`. As you may have guessed, we can also inherit as `protected` or as `private`.

Public inheritance

When a derived class inherits a base class as `public`, all the access qualifiers from the base class remain unchanged. Therefore, if a method or member variable is `public` in the base class, then it will remain `public` in the derived class. This is desirable when the programmer wishes to allow access to the public methods of the base class exactly as if the object was made from the base class. All the examples up to this point used `public` inheritance indicators.

Protected inheritance

When a derived class inherits a base class as `protected`, then all the access qualifiers in the base class that were `public` are treated as if they were `protected`. This serves to allow the derived class methods to access the `public` and `protected` items in the base class, but objects instantiated from the derived class cannot:

```
class DerivedProtected : protected Base // using the same Base class as before
{
    public:
        void derivedProtectedMethod(); // only one method in a class. That is silly!
};
```

Objects instantiated from `DerivedProtected` will not be able to access `Base` member variables:

```
{
    DerivedProtected derived;

    derived.basePublic = 1; // ERROR: was public but now is protected
                           // because Base is inherited as protected
    derived.baseProtected = 1; // ERROR: was protected and now is still
                           // protected so no access is allowed
    derived.basePrivate = 1; // ERROR: no access to privates
}
```

Though there is no access to `Base`'s member variables from an object, there is from `DerivedProtected()`'s methods.

```
void DerivedProtected :: derivedProtectedMethod()
{
    basePublic = 1; // legal! basePublic is now protected
    baseProtected = 1; // legal! was protected and now still is
    basePrivate = 1; // ERROR: basePrivate is still private
}
```

We can only further restrict access with the inheritance indicator, we cannot remove restrictions.

Private inheritance

When a derived class inherits a base class as `private`, then all the access qualifiers in the base class that were `public` or `protected` are treated as if they were `private`. This means that all the member variables and methods in the base class are only visible to the methods in the derived class. The client or other classes deriving off the derived class have no indication that a base class exists.

```
class DerivedPrivate : private Base // using the same Base class as before
{
    public:
        void derivedPrivateMethod(); // only one method in a class. That is silly!
};
```

Objects instantiated from `DerivedPrivate` will not be able to access `Base` member variables:

```
{
    DerivedPrivate derived;

    derived.basePublic = 1; // ERROR: everything from the base class is now
    derived.baseProtected = 1; //     private so there is no access from the
    derived.basePrivate = 1; //     object to any of the methods or variables
}
```

From within the method `derivedPrivateMethod()`, everything from `Base` now appears as `private`:

```
void DerivedPrivate :: derivedPrivateMethod()
{
    basePublic = 1; // legal! basePublic is now private
    baseProtected = 1; // legal! baseProtected is now private
    basePrivate = 1; // ERROR: basePrivate is still private
}
```

Up until this point, there is no difference between inheriting as `protected` and `private`. The only difference appears when another class is derived off it:

```
class DerivedDerived : public DerivedPrivate
{
    public:
        void derivedDerivedMethod()
        {
            basePublic = 1; // ERROR: To DerivedPrivate these are now
            baseProtected = 1; //     private so anyone deriving off of
            basePrivate = 1; //     DerivedPrivate will have no access to them
        }
};
```

One simple rule

This may all seem terribly complex with so many cases. However, there is a simple rule to explain it all:

Inheritance indicators can only add access restrictions, they cannot release them



Sue's Tips

In almost all cases, we use the `public` inheritance indicator when deriving off of a base class. It is extremely rare to need to restrict access to the base class.

Example 3.3 – Inheritance Qualifiers

Demo

This example is completely contrived and does not relate to any real-world problem. The purpose here is to demonstrate all possible inheritance variations. Many lines are commented out. In those cases, the corresponding compile errors are presented below in a comment. Feel free to remove the comment and see the compile errors for yourself.

The base class from which everything is derived is the following:

```
class Base
{
public:
    void pub() {} // Has access to all members of Base
protected:
    void prot() {} // Also has access to all members of Base
private:
    void priv() {} // Also has access to all members of Base
};
```

If we derive off of `Base` using the public inheritance indicator, we get the following compile errors:

```
class DerivedPub : public Base
{
public:
    void derivedPublic()
    {
        pub();      // Was public in the Base, now public from DerivedPub
        prot();     // Was protected in Base, now still protected so no change
        priv();    // Same rules apply here as for derivedPublic() with regards to access
                   // to Base. Of course, we have access to all members of DerivedPub
        void derivedProtected() {}
    private:
        // Same rules apply here as for derivedProtected() and derivedPublic()
        void derivedPrivate() {}
};
```

Now if we try to access these methods from the object, some restrictions apply:

```
int main()
{
    Base base;
    base.pub();           // legal. Base::pub() is public
    base.prot();          // 3-3-qualifiers.cpp:17: error: "void Base::prot()" is protected
    base.priv();          // 3-3-qualifiers.cpp:29: error: "void Base::priv()" is private

    DerivedPub derivedPub;
    derivedPub.pub();     // legal. Base::pub() is still public
    derivedPub.prot();    // 3-3-qualifiers.cpp:17: error: "void Base::prot()" is protected
    derivedPub.priv();    // 3-3-qualifiers.cpp:19: error: "void Base::priv()" is private

    derivedPub.derivedPublic(); // legal. Derived::derivedPublic() is public
    derivedPub.derivedProtected(); // 3-3-qualifiers.cpp:42: error: "derivedProt()" is protected
    derivedPub.derivedPrivate(); // 3-3-qualifiers.cpp:45: error: "derivedPrivate()" is private
}
```

The complete solution is available at [3-3-qualifiers.html](#) or:

```
/home/cs165/examples/3-3-qualifiers.cpp
```

Solution

Unit 3

See Also

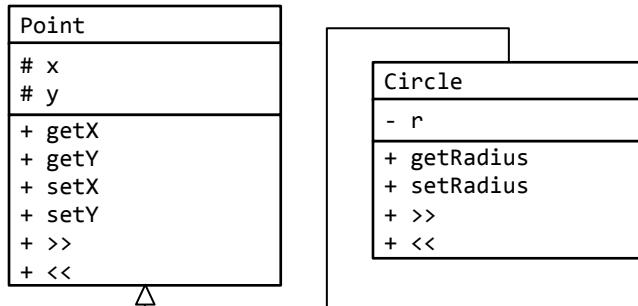
Example 3.3 – Circle

Demo

This next example will demonstrate how to use inheritance to more easily have access to the base class' private member variables without the data being exposed to the client.

Problem

Write a class called `Circle` which inherits off of `Point`. Both `Circle` and `Point` implement the usual getters and setters as well as the insertion and extraction operators.



Solution

With the `protected` inheritance qualifiers used for the `Point` member variables, the class definition is:

```
class Point
{
    ...
    ... code removed for brevity ...
    protected:
        int x;
        int y;
};
```

This allows for `Circle`'s friend functions to have access to `Point`'s member variables:

```
ostream & operator << (ostream & out, const Circle & circle)
{
    out << '(' << circle.x // access to Point's privates
        << ", " << circle.y // access to Circle's privates
        << ", r=" << circle.r
        << ')';
    return out;
}
```

```
istream & operator >> (istream & in, Circle & circle)
{
    in >> circle.x >> circle.y >> circle.r; // access to Point's privates
    return in;
}
```

Challenge

As a challenge, modify the code so `Circle` inherits `Point` as `protected` instead of `public`. Does this change anything? Do the same as a `private`. Why would one want to use the `protected` or `private` inheritance indicator instead of `public`? Hint: what happens when `main()` attempts to access `getX()` or `getY()`? What happens when another class is derived off of `Circle`?

See Also

The complete solution is available at [3-3-circle.html](#) or:

```
/home/cs165/examples/3-3-circle.cpp
```



Problem 1

Given the following class definition:

```
class Base
{
    public:    void pub();
    protected: void prot();
    private:   void priv();
};

class DerivedPub : public Base
{
    public:    void derivedPublic();
    protected: void derivedProtected();
    private:   void derivedPrivate();
};

class DerivedPro : protected Base
{
    public:    void derivedPublic();
    protected: void derivedProtected();
    private:   void derivedPrivate();
};

class DerivedPriv : private Base
{
    public:    void derivedPublic();
    protected: void derivedProtected();
    private:   void derivedPrivate();
};
```

Which of the following lines will yield a compile error?

```
void DerivedPub :: derivedPublic()
{
    pub();                      (a): _____
    prot();                     (b): _____
    priv();                     (c): _____
}

void DerivedPro :: derivedPublic()
{
    pub();                      (d): _____
    prot();                     (e): _____
    priv();                     (f): _____
}

void DerivedPriv :: derivedPublic()
{
    pub();                      (g): _____
    prot();                     (h): _____
    priv();                     (i): _____
}
```

Please see page 236 for a hint.

Problem 2

Given the following class definitions:

```

class Base
{
    public:    void pub() { }
    protected: void prot() { }
    private:   void priv() { }
};

class DerivedPub : public Base
{
    public:    void derivedPublic() { }
    protected: void derivedProtected() { }
    private:   void derivedPrivate() { }
};

class DerivedPro : protected Base
{
    public:    void derivedPublic() { }
    protected: void derivedProtected() { }
    private:   void derivedPrivate() { }
};

class DerivedPriv : private Base
{
    public:    void derivedPublic() { }
    protected: void derivedProtected() { }
    private:   void derivedPrivate() { }
};

```

Which of the following lines will yield a compile error?

```

{
    Base base;
    base.pub();          (a): _____
    base.prot();         (b): _____
    base.priv();         (c): _____

    DerivedPub derivedPub;
    derivedPub.pub();    (d): _____
    derivedPub.prot();   (e): _____
    derivedPub.priv();   (f): _____
    derivedPub.derivedPublic(); (g): _____
    derivedPub.derivedProtected(); (h): _____
    derivedPub.derivedPrivate(); (i): _____

    DerivedPro derivedPro;
    derivedPro.pub();    (j): _____
    derivedPro.prot();   (k): _____
    derivedPro.priv();   (l): _____
    derivedPro.derivedPublic(); (m): _____
    derivedPro.derivedProtected(); (n): _____
    derivedPro.derivedPrivate(); (o): _____

    DerivedPriv derivedPriv;
    derivedPriv.pub();   (p): _____
    derivedPriv.prot();  (q): _____
    derivedPriv.priv();  (r): _____
    derivedPriv.derivedPublic(); (s): _____
    derivedPriv.derivedProtected(); (t): _____
    derivedPriv.derivedPrivate(); (u): _____
}

```

Please see page 236 for a hint.

Challenge 3-5

The game of battleship is played on two 10x10 grids where ships can take anywhere from 2 to 5 squares. The game is played by each player trying to guess the location of the opponent's ships by calling out shots such as "D4."

3. Create a UML class diagram to represent a location on the grid. Call the class `Coordinate`.
 4. From our `Coordinate` class, create a class to represent a ship in the Battleship game. The ship resides on the board and has a length from 2 to 5 points. Also, each ship has a name such as Carrier, Battleship, etc.
 5. There are several derivations of `Ship` in the game Battleship: the Aircraft Carrier (size 5), the Battleship (size 4), the Submarine (size 3), the Cruiser (size 3) and the Destroyer (size 2). Create a class to represent the Submarine.

Please see page 237 for a hint.

Unit 3. Inheritance & Polymorphism

3.4 Virtual Functions

Sam is working on his Chess project but can't quite get it to work the way he wants. While he can create a `Rook` class that derives off of a `Piece`, he can't find a way to make a board of `Pieces`. Every time he tries to put his `Rook` into a `Piece` array, they forget they were `Rooks` and become `Pieces`! He is frustrated! Is there something he is missing?

Objectives

By the end of this chapter, you will be able to:

- Make a function a virtual function through the use of the `virtual` keyword
- Define upcasting, downcasting, and the slicing problem
- Avoid the slicing problem through the use of pointers

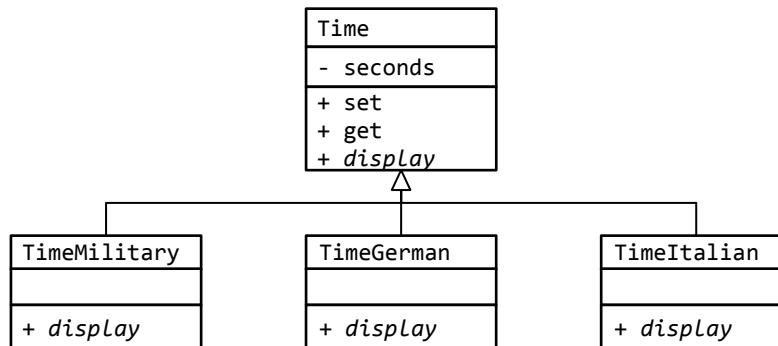
Prerequisites

Before reading this chapter, please make sure you are able to:

- Generate a UML class diagram describing “is-a” class relations (Chapter 3.0)
- Define polymorphism and explain when it might be useful in program (Chapter 3.1)
- Define binding, early binding, and late binding (Chapter 3.1)
- Define a class that inherits off of a base class (Chapter 3.2 and 3.3)

What are virtual functions and why you should care

A **virtual function** is a member function that resides in the v-table associated with a class. This means that an object has a pointer to which version of a method is to be called. Consider, for example, a `Time` base class having three derived classes: `TimeEnglish`, `TimeGerman`, and `TimeItalian`. Each variation of the `Time` class is identical to the parent with one exception: they have different `display()` methods.



Through inheritance, it is possible to create the German derivation of `Time`. However, it is not possible to have an array of `Times`, each one remembering that it is actually English, German, or Italian. The reason is that as soon as the `TimeGerman` object is placed into a `Time` array, it forgets that it was German! Through virtual functions, it becomes possible for `TimeGerman` to remember its German nature and call the appropriate `display()` function.

Binding and polymorphism

Recall the notion of binding from the first two pages of Chapter 3.1 (see the Binding section). Early binding occurs when the compiler has all the information necessary to match a named item (such as a variable or a function) to a location in memory. Inheritance is an example of early binding because, based on the data type of the object, the compiler knows exactly which version of a method belongs to the object. Thus, it turns out that the compiler does not need to implement a v-table for a class exhibiting inheritance. Consider the following code:

```
class Time
{
    public:
        void display(); // displays "3:28pm"
    ... code removed for brevity ...
};

class TimeGerman : public Time
{
    public:
        void display(); // displays "15.28"
    ... code removed for brevity ...
};

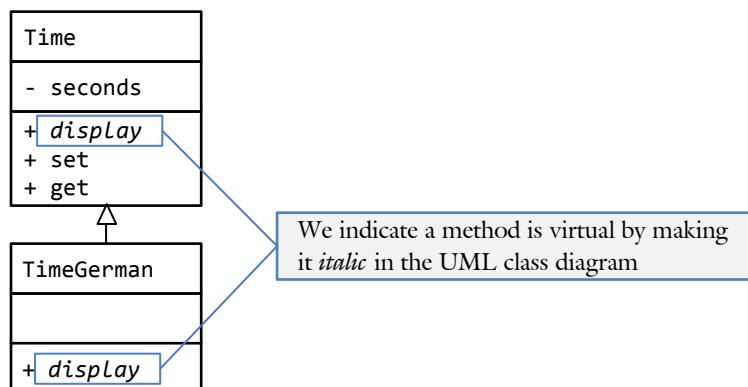
int main()
{
    Time timeNormal; // the compiler knows this is a Time object so
    timeNormal.display(); // it knows which version of display() to
                          // call: Time::display()

    TimeGerman timeGerman; // similarly, the compiler knows this is a
    timeGerman.display(); // TimeGerman object so TimeGerman::display()
                         // is called here
}
```

When early binding is at work, pointers are not necessary. After all, pointers are late-binding tools.

Late binding is needed when the compiler does not have all the information necessary to determine which data-item or function is needed in a given circumstance. Consider an array where the index is provided by the user. The compiler has no way of knowing which item in the array is needed; that decision can only be made at run-time. Since the decision must be made at runtime, late binding is at work. Since late binding is at work, pointers must be involved. Sure enough, arrays are implemented with pointers! Pointers allow the program to determine at run-time which data-item or function is needed. It follows that we will need pointers to implement polymorphism.

In order to make a given class polymorphic, it is necessary to indicate to the compiler that a given function will have more than one variation which will be determined at run-time. We call such methods **virtual functions**. Virtual functions are all those functions which need to be put in a v-table so they can be assigned at run-time. To indicate that a given method is virtual in a UML class diagram, we use *italic*.



Virtual functions in class definitions

To make a method virtual in a class definition, we use the `virtual` keyword before the function declaration:

```
class Time : public Minutes
{
public:
    Time();
    Time(const Time & time);

    virtual void display() const
    {
        cout << getHours() / 12 << ':'
            << getMinutes() << (getHours() > 11 ? "pm" : "am");
    }

    void set(int hours, int minutes = 0; int seconds = 0);
    int getSeconds() const { return seconds; }
private:
    int seconds;
};
```

The `virtual` keyword indicates that this function is virtual and will be put in a v-table.

Even though this function is defined in the class definition, it is not inline. Virtual functions cannot be inline.

The `virtual` keyword has one purpose: to put the associated method in a v-table. This enables the class to be polymorphic. Now if the programmer wishes to create a German variant of the `Time` class, a German version of the `display()` function can be written. To do this, the German version will inherit from `Time`.

```
class TimeGerman : public Time
{
public:
    TimeGerman();
    TimeGerman(const TimeGerman & time);

    virtual void display() const
    {
        cout << getHours() << '.' << getMinutes();
    }
};
```

The `virtual` keyword is not needed; it is inherited off the base-class. It is a good idea to include it though.

Only the constructors and the virtual functions need to be defined in the derived class.

Note that we can only make member functions virtual. Therefore, we can make operators virtual, as long as they are member operators. It is impossible to make the insertion operator (`<<`) virtual because it cannot be a member function.



Sue's Tips

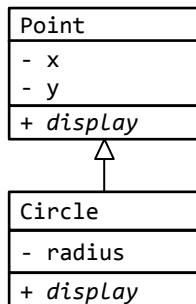
Only make a method `virtual` if it will be re-defined in the derived-class. While the performance penalty of a virtual function is small, it is not zero. The most important reason to minimize the number of virtual functions is that it is misleading to make a function `virtual` if it does not need to be.

The slicing problem

Consider the scenario where we have a `Time` base class and a `TimeGerman` derived class containing the `display()` virtual function. If we create an object of type `Time`, then it is easy to figure out which version of `display()` is needed. The same is true if we create an object of type `TimeGerman`. However, what happens when we assign a `time` object to a `timeGerman` object or vice-versa?

```
{  
    Time time;                      // the data type is "Time"  
    time.set(15 /*h*/, 23 /*m*/);  
    time.display();                 // display "3:23pm" using the Time::display() version  
  
    TimeGerman timeGerman;          // the data type here is "TimeGerman" and the compiler  
    timeGerman.set(15, 23);          // knows exactly what the data type is  
    timeGerman.display();           // "15.23" using TimeGerman::display()  
  
    timeGerman.set(19, 51);  
    time = timeGerman;              // here only the data is copied over. The time object  
    time.display();                // is still a Time so we get "7:51pm"  
  
    time.set(13, 1);  
    timeGerman = time;              // here only the data is copied over. The timeGerman  
    timeGerman.display();           // object is still TimeGerman so we get "13.01"  
}
```

The important thing to get from this example is that assigning an object of one data type to another does not change the data type. We cannot make a `Time` behave like a `TimeGerman` or vice-versa. Sometimes, however, the problem can be quite serious. Consider the following UML class diagram:



In this example, we are creating a `Circle` class from a `Point` class with a `display()` virtual function. Clearly `Circle` has three member variables (`x`, `y`, and `radius`) while `Point` has two (`x`, `y`). What happens when I try to assign a `Circle` into a `Point` or vice-versa? Because there are a different number of member variables, we have a problem.

Downcasting

Downcasting is the process of casting a base class into a derived class. In the above example, it is assigning a `Point` onto a `Circle`:

```
{  
    Point point(3, 5);           // point has the values of (3, 5)  
    Circle circle = point;      // what is the value of circle.radius?  
    circle.display();           // ERROR: circle.radius is undefined!  
}
```

Downcasting is dangerous because the program is required to “make up” information that is not provided. Since a `Point` does not have a `radius` but the `Circle` does, the `circle.radius` member variable is uninitialized.

Upcasting or “slicing”

Upcasting, otherwise known as the **slicing problem**, is the process of casting a derived class into a base class. In this case, we are ignoring what is unique about the derived class and retaining only the base class information. In other words, we are “slicing off” the `Circle`-ness so it can become a `Point`.

```
{
    Circle circle(3, 8, 10);           // circle has the values of (3, 8, r=10)
    Point point = circle;             // we loose the radius here but are keeping rest
    point.display();                 // display "(3, 8)"
}
```

In some rare situations, this may be what you want. It may be needed to slice off the derived values and treat an object like its base class. In situations like these, is better to create a member function returning the base class. We can do it explicitly:

```
Point Circle :: getPoint() const
{
    return (Point) (*this);
}
```

The slicing problem can happen any time an object is cast or when an object is assigned to a variable of the base type. While the assignment operator is one way this could happen, another common way is through passing objects to a function when the parameter is by-value. Recall that by-value parameter passing makes a copy of the variable (whereas by-reference and by-pointer parameters avoid this copying).

In the following example, a new `Time` object “`t`” is created and only the base class portion of the `timeGerman` object is copied into it, thus resulting in the slicing problem.

```
void display(Time t)                      // notice this is passed by-value
{
    t.display();                          // ERROR! Sliced to base Time object
}

int main()
{
    TimeGerman timeGerman;
    display(timeGerman);                // we think we are passing a TimeGerman, but we
                                         // actually are passing a Time
}
```

Avoiding the slicing problem

Sometimes we want to have an array of the base class where each member of the array remembers which derived class it belongs. This is difficult because all the items in an array must be of the same data type:

```
{
    TimeItalian timeItalian;           // "12.00 PM"
    TimeGerman timeGerman;            // "12.00"

    Time array[2];                  // base-type is of type "Time"
    array[0] = timeItalian;          // ERROR! sliced to Time
    array[1] = timeGerman;           // ERROR! sliced to Time also

    array[0].display();              // "12:00pm"
    array[1].display();              // "12:00pm"
}
```

To accomplish this it is necessary to not have an array of items, but rather have an array of pointers. This means that each item in the array gets to retain what is unique about it:

```
{  
    Time * array[2];           // base-type is of type "Time **"  
    array[0] = new TimeItalian; // assigning a "TimeItalian **" to "Time **"  
    array[1] = new TimeGerman;  // assigning a "TimeGerman **" to "Time **"  
  
    array[0]->display();      // "12.00 PM"      (not "12:00pm" because...  
    array[1]->display();      // "12.00"          ... display() is virtual)  
  
    delete array[0];           // since we dynamically allocated our two  
    delete array[1];           //     Times, we need to delete them when done  
}
```

In other words, we avoid the slicing problem by working with pointers rather than objects themselves.

As with arrays, this slicing problem is avoided with functions by making sure that we do not create a separate copy of the parameter, but rather passing a pointer or a reference to the value. If the parameter is passed by pointer, the syntax would look as follows:

```
void display(Time * pTime)           // pass-by-pointer so we avoid making a copy  
{  
    pTime->display();               // Correct use of polymorphism  
}  
  
int main()  
{  
    TimeGerman timeGerman;  
    display(&timeGerman);           // pass a TimeGerman to display()  
}
```

In the above example, because the function is expecting the address of a `Time` object, when it is called, we must use the “address of” operator (`&`) before our object. This is simplified by using the pass-by-reference syntax to the following (which also avoids the slicing problem):

```
void display(Time & t)           // pass-by-reference so we avoid making a copy  
{  
    t.display();                  // Correct use of polymorphism  
}  
  
int main()  
{  
    TimeGerman timeGerman;  
    display(timeGerman);           // pass a TimeGerman to display()  
}
```

In each of the two previous examples, no new copies of the `timeGerman` object are created, but rather the parameter `t` refers to the original variable, and in so doing, the slicing problem is avoided.



Sue's Tips

Every time polymorphism is used, there needs to be a virtual function and a pointer. If the pointer is missing, we have the slicing problem. If the `virtual` keyword is missing, we are lacking the v-table.

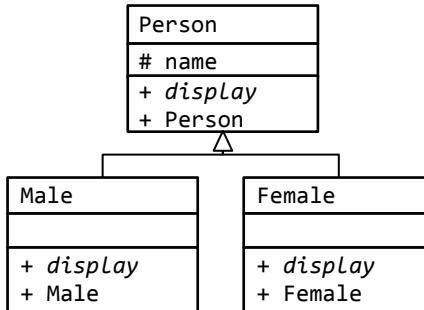
Example 3.4 – Person

Demo

This example will demonstrate the simplest polymorphism example: one derived class and two base classes, each one with a single virtual function.

Problem

Write a class to represent a person. Additionally, create a `Male` and `Female` derived class.



The class definition of the base-class and one derived class is:

```
class Person
{
public:
    Person(const string & s);
    virtual void display();
protected:
    string name;
};
```

```
class Male : public Person
{
public:
    Male(const string & s);
    virtual void display();
};
```

Observe how only the constructor and the virtual functions need to be defined in the derived class. To test the class, we need to create some `Males` and some `Females` and put them in a vector.

Solution

```
{
    // first, we will create a bunch of people.
    Male male1(string("Hurst"));
    Female female1(string("Dewey"));
    Female female2(string("Richards"));

    // A collection of people. Note that each is a pointer to a Person,
    // not a Person. If we make them a Person then we will have the
    // slicing problem and they will lose their gender
    Person * people[3];

    // add them to my collection of people
    people[0] = &male1;
    people[1] = &female1;
    people[2] = &female2;

    // Note how the array only knows about Persons. It knows nothing
    // about males or females. Yet somehow the correct display
    // function is called for each. This is because we are
    // using polymorphism; each person has a vtable which
    // points to the correct function
    for (int i = 0; i < 3; i++)
        people[i]->display();
}
```

See Also

The complete solution is available at [3-4-people.html](#) or:

```
/home/cs165/examples/3-4-people.cpp
```



Example 3.4 – Date

Demo

In this second example, we will see that we don't necessarily have to have pointers to do polymorphism. We can also use pass-by-reference which works basically the same way.

Problem

Write a program that has a `Date` base-class and two derived classes: the `DateShort` form displaying “1/1/2000” and the `DateLong` form displaying “1st of January, 2000”.

Solution

The base-class and each derived class need to implement the `display()` virtual function:

```
ostream & DateShort :: display(ostream & out) const
{
    out << month << "/" << day << "/" << year;
    return out;
}
```

With these polymorphic `display()` functions, we can then the insertion operator for `Date` needs to call the appropriate version of `display()`:

```
ostream & operator << (ostream & out, const Date & date)
{
    return date.display(out);
}
```

To test this, we will create an array of `Date` pointers, assign them to local variables (of type `Date`, `DateShort`, and `DateLong`), and display each with the insertion operator.

```
{
    Date      date;          // each local variable is a Date, DateShort,
    DateShort dateShort;     // or DateLong.  We never slice so each one
    DateLong  dateLong;      // retains its own version of display()
    Date *    dates[3];       // array of pointers to Date so not to slice

    ... code removed for brevity ...
    dates[0] = &date;         // since dates is an array of pointers to Date,
    dates[1] = &dateShort;    // we need to use the address-of operator &
    dates[2] = &dateLong;     // to assign the pointers

    ... code removed for brevity ...
    cout << "Base class with empty display: \\" " << *dates[0] << "\\n";
    cout << "Short version:           \\" " << *dates[1] << "\\n";
    cout << "Long version:            \\" " << *dates[2] << "\\n";
}
```

Challenge

As a challenge, can you add a third derived class that displays just the year?

1888

See Also

The complete solution is available at [3-4-date.html](#) or:

/home/cs165/examples/3-4-date.cpp

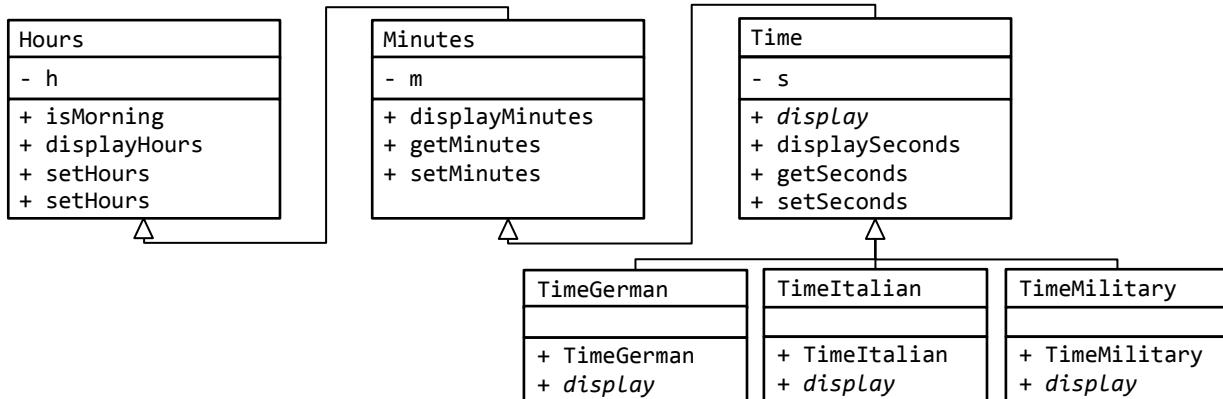


Example 3.4 – Time

Demo This next example will demonstrate how to mix simple inheritance with polymorphism. Here `Minutes` will inherit from `Hours` and `Time` will inherit off of `Minutes`. Additionally, there will be three variations of `Time`: `TimeGerman`, `TimeItalian`, and `TimeMilitary`.

Problem

Write three variations of a `Time` class, each displaying the time using a different format.



Solution

```
class Time : public Minutes
{
public:
    Time() : s(0), Minutes() { }
    Time(int s) : s(s), Minutes() { setSeconds(s); }
    Time(const Time & time);
    int getSeconds() const { return s; }
    void setSeconds(int s) { if (validateSeconds(s)) this->s = s; }
    void addSeconds(int s);
    void displaySeconds() const
    {
        cout << (s < 10 ? "0" : "") << s;
    }
    virtual void display() const
    {
        displayHours(false /*is24*/); // we will display the hours...
        cout << ':'; // ... followed by the colon ...
        displayMinutes(); // ... followed by the minutes ...
        cout << (isMorning() ? "am" : "pm"); // ... and ending with the am/pm
    }
private:
    bool validateSeconds(int s) const { return (0 <= s && s < 60); }
    int s;
};
```

Challenge

As a challenge, can you add another variation of `Time` that displays the seconds as well? Call this new format `TimeLong`:

3:28:19pm

See Also

The complete solution is available at [3-4-time.html](#) or:

/home/cs165/examples/3-4-time.cpp



Demo

Example 3.4 – Slicing problem

In this final example will demonstrate the slicing problem. Specifically, we will create a trivial base class with a single virtual method and derived class. We will then attempt to copy the derived class onto the base class, yielding the slicing problem.

The base-class has a single virtual function called `method()`:

```
class Base
{
public:
    virtual void method() { cout << "Base!\n"; } // trivial virtual method
};
```

The derived class inherits off of `Base` and redefines `method()`:

```
class Derived : public Base
{
public:
    virtual void method() { cout << "Derived!\n"; }
};
```

Our drive program will demonstrate the slicing problem:

```
int main()
{
    // we will create a Derived
    Derived derived;
    // this will display "Derived!"
    derived.method();

    // here we will assign Derived to Base but, due to the slicing problem,
    // we forget that we ever were a Derived. We become, in essence, a Base
    // here. This is the slicing problem.
    Base base = derived;
    // even though we assigned derived to base, we are of type base. We
    // have Base's VTable, not Derived.
    // this will display "Base!"
    base.method();

    // now we will avoid the slicing problem. We will not turn derived
    // into base, we will just copy the pointer. This means that we keep
    // what was unique about it.
    Base *pBase = &derived;
    // this will display "Derived!"
    pBase->method();

    return 0;
}
```

The output of this program is:

```
Derived!
Base!
Derived!
```

See Also

The complete solution is available at [3-4-slicingProblem.html](#) or:

```
/home/cs165/examples/3-4-slicingProblem.cpp
```



Problem 1

What is the output of the following code:

```
class Person
{
    public:
        Person(const string & s) : name(s) { }
        void display() { cout << name << endl; }
    private:
        string name;
};

class Male : public Person
{
    public:
        Male(const string & s) : Person(s) { }
        void display() { cout << "Mr. " << name << endl; }
};

class Female : public Person
{
    public:
        Female(const string & s) : Person(s) { }
        void display() { cout << "Ms. " << name << endl; }
};

int main()
{
    Male smith("Smith");
    Female jones("Jones");
    Person array[2];
    array[0] = smith;
    array[1] = jones;
    array[0].display();
    array[1].display();
}
```

Please see page 250 for a hint.

Challenge 2 - 4

2. Create the UML class diagram to represent the four types of accounts in a bank. Each account will be updated daily based on its interest rate and each account can display the balance.

Checking	Fee for each check Minimum balance
Savings	Interest
Credit	Annual fee Interest
Loan	Term Interest

3. Write the class definition for the `Checking` class from the above UML class diagram.

4. Create an array of `Accounts`, each of which is a `Checking` or `Savings`...

Please see page 247 for a hint.

Unit 3. Inheritance & Polymorphism

Unit 3

3.5 Pure Virtual Functions

Sue has just completed her chess program. She has decided to have a base-class called “Piece” and the following derived classes: Space, King, Queen, Rook, Bishop, Knight, and Pawn. Because she has a Space derived class, there is never a time when she would want to instantiate a Piece object. In fact, any attempt to do so would signify a mistake. Wouldn’t it be great if the compiler could help Sue find such a mistake?

Objectives

By the end of this chapter, you will be able to:

- Define the terms abstract class and pure virtual function
- Describe a situation where a pure virtual function would be the correct tool for the job
- Implement a class containing a pure virtual function

Prerequisites

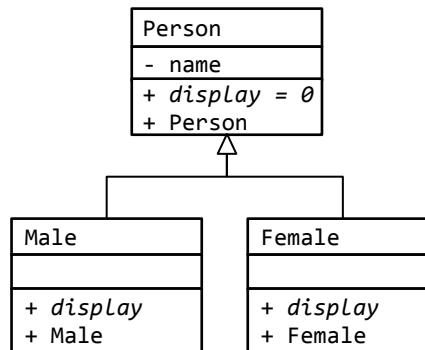
Before reading this chapter, please make sure you are able to:

- Design a class using UML class diagrams illustrating “is-a” relations
- Describe the difference between early-binding and late-binding
- Implement a polymorphic class using virtual functions

What are pure virtual functions and why you should care

A **pure virtual function** is a member function within a base-class that has no implementation. A class derived off of this base class, on the other hand, must have a definition for this function. This means that it is impossible to instantiate a class containing a pure virtual function (because there is no implementation for the pure virtual function) but it is legal to instantiate one of the derived classes. A class containing at least one pure virtual function is called an “**abstract class**.”

The motivation for pure virtual functions is that it is often desirable to create a collection of derived classes when the base-class makes no sense by itself. As a trivial example, consider a Person base-class having two derived classes: Male and Female.



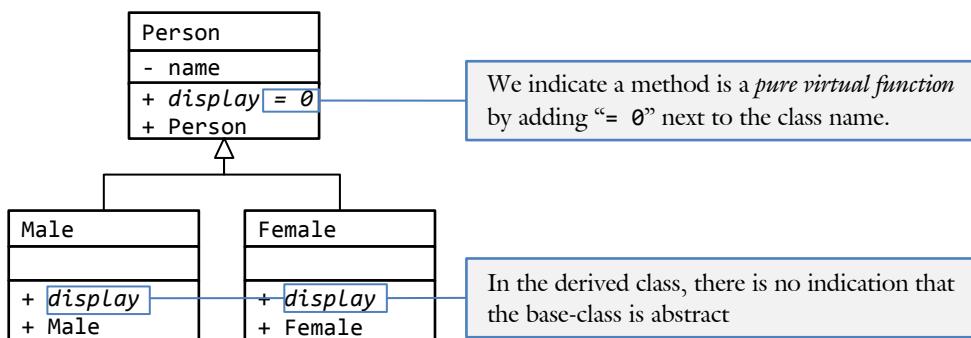
In this example, it is possible to create an object of type Male as well as type Female, but it is impossible to instantiate a Person. A person without gender makes no sense. We would therefore call the display method a pure virtual function and the entire Person class is an abstract class.

Designing with pure virtual functions

Recall from Chapter 3.0 one of the motivations for designing with class relations: to leverage the similarities between classes in an effort to remove redundant or duplicate code. This means that all the invariant (or same) data and methods should reside in the base-class while all the variant (or “unique”) data and methods should reside in the derived classes. It follows that there is nothing to stop us from designing a base class that does not make sense on its own. For example, there are a lot of invariant attributes shared between a `Male` and a `Female`. In fact, aside from their gender (and the odd chromosome), there is no difference. Therefore, the bulk of the functionality and data of the `Male` and `Female` classes should reside in a common base class. We will call this base-class `Person`, the genderless individual. This presents a problem: it makes no sense at all to have a `Person`! What is a person without their gender, an essential characteristic of any individual?

Clearly every `Person` object needs to have a `getIsMale()` method allowing the caller to ascertain the gender of the individual. This method is easy to define for the `Male` and `Female` class, but not so with the `Person` base-class. We call methods like this **pure virtual functions** and a class having such a function is an **abstract class**. When a function is pure, we do not have to provide an implementation for it.

In a UML class diagram, we can specify that a given method is pure by appending “`= 0`” next to the method name:



Note that abstract classes are a special form of the “is-a” relation. Not all virtual functions are pure but all pure functions are virtual! Therefore, we can only use pure virtual functions when the function is already virtual. We can only append the “`= 0`” suffix next to a method that is italic.



Sue's Tips

When considering whether to make a base class abstract, ask yourself if this is the right thing to do: Is there never a reason why the client will want to implement such a class? Is there no method implementation that makes sense? When the answer to these questions is “yes,” then an abstract class might be the right tool for the job. Otherwise, find a suitable default behavior for the methods in question.

Syntax of a pure virtual function

The syntax for indicating that a virtual function is pure is similar to the UML class diagram notation: simply put “`= 0`” next to the function declaration:

```
class Base
{
public:
    //*****
    * BASE :: METHOD
    * This is a pure virtual function (because of the = 0) at the end of the method
    * definition Thus the VTable has a NULL pointer and the compiler will not allow us
    * create an object of type Base
    *****
    virtual void method() = 0;
};
```

When we put “`= 0`” next to the function, we cannot implement the function later. There are two main fallouts for making a virtual function pure: the inability of the client to instantiate objects of the base class type and the elimination of the slicing problem.

No abstract objects

One big impact of making a function pure is that we can no longer instantiate an object of that type. Attempts to do so will yield a compile error. This is a useful tool because it prevents the programmer from accidentally instantiating an object based on a class that has no meaning.

No slicing problem

Probably the best thing about making a base class abstract is that we eliminate the slicing problem. Recall from Chapter 3.4 that the slicing problem originates from the programmer attempting to cast a derived class down to a base class. The result of such an operation is that all that is unique about the derived class is “sliced away” yielding only a base class. In most cases, this is undesirable.

Slicing is impossible with abstract classes because the compiler will not let the programmer instantiate an object based on an abstract class. Any attempt of the programmer to do so will yield a compile error.

This is particularly helpful because slicing problems tend to be subtle. The program compiles and seems to work, but somehow things do not work the way the programmer expects! Trying to find the problem by inspecting the code can be difficult because the statement containing the error looks right. However, when an abstract class is used, the compiler directs the programmer to the exact line of code containing the error. These bugs are much easier to fix.

Sam's Corner



Recall that virtual functions are function pointers residing in a v-table attached to a class. This means that each function pointer is assigned an address at run-time corresponding to which version of the function is to be executed. In the case of pure virtual functions, there is no function implementation for the v-table to point to. Therefore, the function pointer is given the `NULL` or zero address. This is why the “`= 0`” notation is used both in the UML class diagram and in the C++ class definition.

Example 3.5 – Abstract class

Demo

This example demonstrates what happens when the programmer attempts to implement an abstract class. In other words, the expected output is a compile error.

The base class is an abstract class with the a virtual function called `method()`. This means that the v-table has NULL pointer and the compiler will not allow us to create an object of type `Base`:

```
class Base
{
public:
    virtual void method() = 0; // this is a pure virtual function
};
```

The derived class inherits `Base` as public and implements the virtual function `method()`. Note that, though it is defined in the class definition, it is not an inline function. Virtual functions cannot be inline:

```
class Derived: public Base
{
public:
    virtual void method() // implement the virtual function here
    {
        cout << "Derived!\n";
    }
};
```

We then attempt to instantiate a `Base` in `main()`:

```
int main()
{
    Base base; // compile error. Because Base is an Abstract Class due to the
               // pure virtual function method(), we cannot
               // instantiate an object of this type.

    // even if we could create an object called base, we would crash
    // here due to the NULL pointer in the VTable of Base
    base.method();

    return 0;
}
```

The resulting compile error is:

```
3-5-abstractClass.cpp: In function “int main()”:
3-5-abstractClass.cpp:53: error: cannot declare variable “base” to be of abstract
                           type “Base”
3-5-abstractClass.cpp:17: note: because the following virtual functions are pure
                           within “Base”:
3-5-abstractClass.cpp:26: note:         virtual void Base::method()
```

The complete solution is available at [3-5-abstractClass.html](#) or:

```
/home/cs165/examples/3-5-abstractClass.cpp
```

See Also



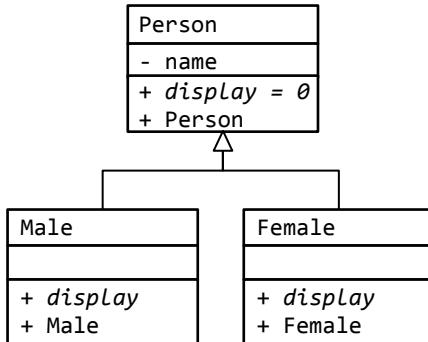
Example 3.5 – People

Demo

This example demonstrates a simple abstract class with two derived classes. In this example there is only one virtual function: the `display()` function.

Problem

Write an abstract class to represent a person. Additionally, create a `Male` and `Female` derived class.



Note how the `display()` function is pure.

Solution

The only difference between this `Person` class and the `Person` class from “Example 3.4 – People” is that the method `display()` is a pure virtual function now.

```
class Person
{
public:
    Person(const string & s) { name = s; }
    virtual void display() const = 0; // pure virtual function
protected:
    string name;
};
```

The functionality of program is the same as the example from the previous chapter. This is because we only attempt to instantiate a `Male` or a `Female` in the driver program:

```
int main()
{
    // first, we will create a bunch of people.
    Male male1(string("Hurst"));           // we can instantiate a number of
    Female female1(string("Richards"));     // Males and Females, but we
    Female female2(string("Dewey"));        // cannot instantiate a Person

    // A collection of people. Note that each is a pointer to a Person,
    // not a Person. Therefore since there is no instantiated Person, we
    // will not get the abstract class compile error
    vector<Person*> people;
    ... code removed for brevity ...
}
```

Challenge

As a challenge, can you add a member variable to `Female` called `isMarried`? Add a parameter to the constructor for `Female` that defaults to `false`. Finally, modify the `display()` function to prepend “Mrs.” if the individual is married and “Miss.” if she is not.

See Also

The complete solution is available at [3-5-people.html](#) or:

```
/home/cs165/examples/3-5-people.cpp
```



Example 3.5 – Date

Demo Recall from “Example 3.4 – Date” the empty `display()` function for the `Date` base class. Without pure virtual functions, we need to do hacks like this when a function in the base class is meaningless. This example will demonstrate how to avoid such hacks.

Problem Modify the `Date` class from “Example 3.4 – Date” to make the `display()` method pure. We will also need to remove the object instantiated from `Date` in the driver program to avoid the compile error that would result otherwise.

The `Date` class from “Example 3.4 – Date” is the following:

```
class Date
{
    ... code removed for brevity ...
    // empty display function
    virtual ostream & display(ostream & out) const { return out; }
    ... code removed for brevity ...
};
```

The same class with a pure `display()` function:

```
class Date
{
    ... code removed for brevity ...
    // NULL display function
    virtual ostream & display(ostream & out) const = 0;
    ... code removed for brevity ...
};
```

For this to compile, the corresponding `Date` objects need to be removed as well:

```
// without pure virtual functions
int main()
{
    Date date;
    DateShort dateShort;
    DateLong dateLong;
    Date * dates[3];

    cin >> date;
    dateLong.copy(date);
    dateShort.copy(date);
    dates[0] = &date;
    dates[1] = &dateShort;
    dates[2] = &dateLong;

    cout << *dates[0] << endl
        << *dates[1] << endl
        << *dates[2] << endl;
    return 0;
}
```

```
// with pure virtual functions
int main()
{
    DateShort dateShort;
    DateLong dateLong;
    Date * dates[2];

    cin >> dateLong;
    dateShort.copy(dateLong);

    dates[0] = &dateShort;
    dates[1] = &dateLong;

    cout << *dates[0] << endl
        << *dates[1] << endl;
    return 0;
}
```

The complete solution is available at [3-5-date.html](#) or:

```
/home/cs165/examples/3-5-date.cpp
```



See Also

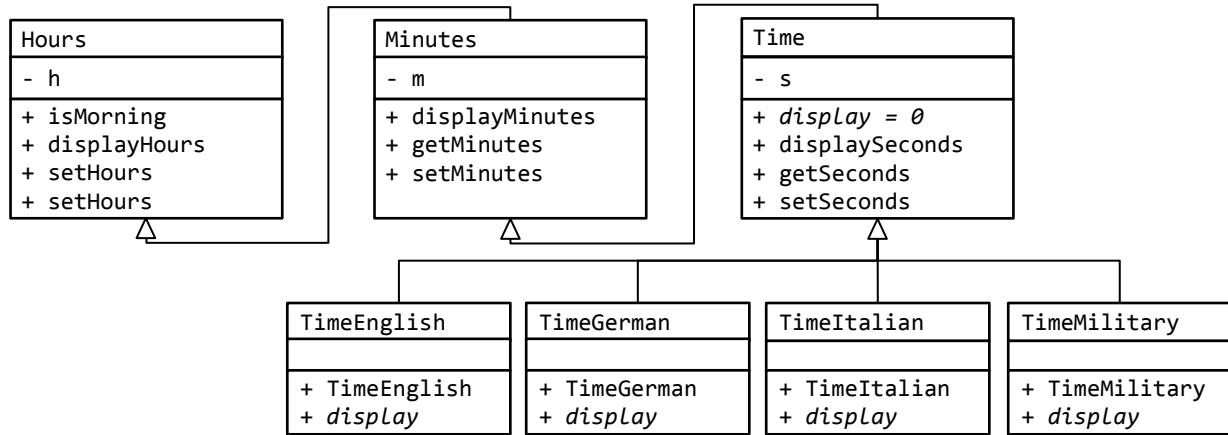
Example 3.5 – Time

Demo

In “3.4 – Time,” the base class displayed the time in the United States – English format. This is a bit presumptuous – why should the default time format be English? Instead, we will make `Time` an abstract base class and have a new `TimeEnglish` derived class displaying the United States – English format.

Problem

Modify “3.4 – Time” to make the `Time` class abstract and have a new `TimeEnglish` derived class:



The first step is to make the `Time` class abstract:

```
class Time : public Minutes
{
    ...
    ... code removed for brevity ...
    virtual void display() const = 0; // pure virtual function
    ...
};
```

Next a new `TimeEnglish` class needs to be added:

```
class TimeEnglish : public Time
{
public:
    TimeEnglish() : Time() {}
    TimeEnglish(int s) : Time(s) {}
    TimeEnglish(const TimeEnglish & time) : Time(time) {}
    virtual void display() const // virtual function
    {
        displayHours(false /*is24*/); // we will display the hours...
        cout << ':'; // ... followed by the colon ...
        displayMinutes(); // ... followed by the minutes ...
        cout << (isMorning() ? "am" : "pm"); // ... and ending with the am/pm
    }
};
```

Finally, we need to remember to instantiate a `TimeEnglish` in the driver program rather than a `Time`. Failure to do so will result in a compile error.

See Also

The complete solution is available at [3-5-time.html](#) or:

```
/home/cs165/examples/3-5-time.cpp
```



Review 1

What is the output of the following code?

```
class Base
{
public:
    virtual void method() { cout << "Base!\n"; }
};

class Derived : public Base
{
public:
    virtual void method() { cout << "Derived!\n"; }
};

int main()
{
    Derived d1;
    Base b = d1;
    Derived d2 = b;

    d1.method();
    d2.method();
    b.method();
    return 0;
}
```

Please see page 250 for a hint.

Problem 2

What is the output of the following code?

```
class B
{
public:
    virtual void funky() { cout << "B!\n"; }
};

class A: public B
{
public:
    virtual void funky() { cout << "A!\n"; }
};

int main()
{
    A a;
    B b;

    a.funky();
    b.funky();
}
```

Please see page 222 for a hint.

Problem 3

What is the output of the following code?

```
class B
{
public:
    virtual void funky() = 0;
};

class A: public B
{
public:
    virtual void funky() { cout << "A!\n"; }
};

int main()
{
    B b;

    b.funky();
}
```

Please see page 255 for a hint.

Problem 4

What is the output of the following code?

```
class B
{
public:
    virtual void funky() { cout << "B!\n"; }
};

class A: public B
{
public:
    virtual void funky() { cout << "A!\n"; }
};

int main()
{
    A a;
    a.funky();

    B b = a;
    b.funky();
}
```

Please see page 250 for a hint.

Problem 5

What is the output of the following code?

```
class B
{
public:
    virtual void funky() { cout << "B!\n"; }

};

class A: public B
{
public:
    virtual void funky() { cout << "A!\n"; }

};

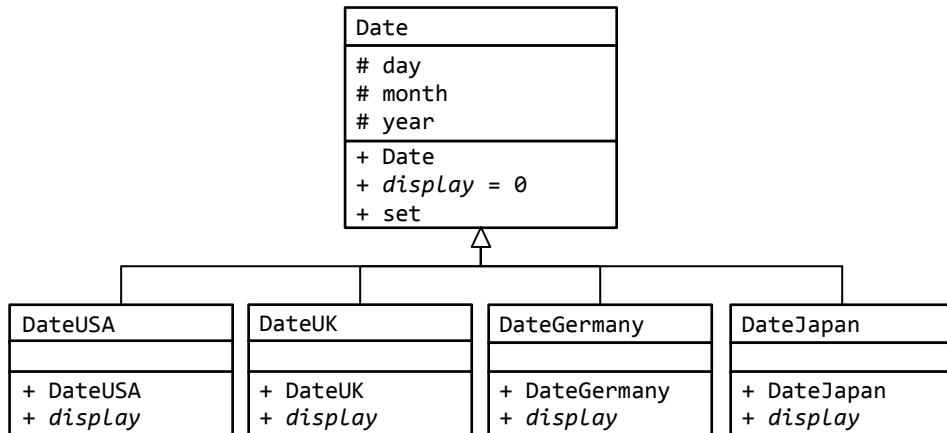
int main()
{
    A a;
    B * pB = &a;

    pB->funky();
}
```

Please see page 245 for a hint.

Problem 6,7

Given the following UML class description:



6. Provide the C++ class definition for the **Date** class and the **DateJapan** class.

7. Implement the **display()** method for each of the derived classes:

DateUSA: 5/21/2010
DateUK: 21/05/2010
DateGermany: 21.05.2010
DateJapan: 2010/05/21

Please see page 259 for a hint.



Unit 4. Abstract Types

4.0 Type-Independent Design	264
4.1 Void Pointers and Callbacks.....	274
4.2 Function Templates	286
4.3 Class Templates	294
4.4 Linked Lists	301
4.5 Iterators.....	314
4.6 Standard Template Library.....	325

4.0 Type-Independent Design

Sam wrote a binary search function for integers last semester. This year he re-purposed that same algorithm to work with the string class. As much as he likes the algorithm, he suspects that he will have to make it work for another data type in the near future. There has got to be a better way!

Objectives

By the end of this chapter, you will be able to:

- Identify situations when generic algorithms and generic data-structures may be used
- Understand the relationship between type-independent design and polymorphism

Prerequisites

Before reading this chapter, please make sure you are able to:

- Describe the principles of modularization and explain why they yield more re-usable code (Procedural Programming in C++, Chapter 2.0)
- Describe the principles of encapsulation and explain why they yield more re-usable code (Chapter 2.0)
- Describe how “is-a” class relations can work with polymorphism (Chapter 3.0, 3.4)

What is type-independent design and why you should care

Type-independent design is the process of designing algorithms and data-structures that can work with a wide variety of data types. Perhaps this is best explained by example. Consider binary search, an algorithm designed to quickly find the location of a given item in a sorted list. This algorithm was developed without a single data type in mind. The binary search algorithm is an example of a type-independent design because it can work with many data types.

Type-independent design is a useful design methodology because it increases the re-usability of code. Just as the principles of modularization lead to more re-usable functions and encapsulation lead to more re-usable classes, type-independent design techniques enable code to be repurposed into a wide variety of previously unforeseen contexts. Consider a tractor-trailer truck. While it certainly exhibits a high degree of modularization (a single well-defined purpose and a simple interface with the highway), the re-usability value comes from the fact that it can work with such a wide variety of payloads (or data types). As long as the payload meets certain requirements, the tractor-trailer can easily be configured to haul it. Type-independent design similarly enables the programmer to develop algorithms and data-structures that work with a wide variety of data types.

Prerequisites for type-independent design

The essence of type-independent design, also called **templates**, is to create a function or class that can operate on a wide variety of data types, many of which do not have to be known by the developer of the function or class. To make this happen, two requirements typically need to be met: the algorithm or data-structure needs to be sufficiently generic so as to be useful for a wide variety of data types, and the data types need to support a set of functions or operators needed by the algorithm. These two requirements will be described in turn.

A template is a mechanism in C++ allowing a programmer to develop a function to work with generic data types

Generic algorithms

Consider the binary search algorithm or the bubble sort algorithm. Both of these algorithms are useful for integers, floating point numbers, text, playing cards, and many other applications. The author of these algorithms certainly did not envision every possible data type that could possibly use a sort or search routine. Instead, the algorithm was designed with all possible data types in mind.

Another class of generic algorithms includes transformation algorithms. For example, data compression or encryption algorithms are inherently data independent. Though HTML files or JPG images certainly have very different data-structures, each could be encrypted using the same algorithm.

Generic data-structures

Data-structures also commonly use templates. A data-structure is an organizational unit of individual data elements. The simplest data-structure is an array, composed of a simple list. Of course the array data-structure can be used for integers, Boolean values, names, and any other data type. Other data-structures include queues (first-in, first-out lists), stacks (first-in, last-out lists), trees, and a host of more advanced and complex structures. In each case, the utility of the data-structure extends beyond a single data type.

Operations

In order to work with a new data type, a minimum number of operations or functions working with the data must be known. This list of operations depends on the context. For example, in order to find a given element in a list, it is necessary to determine if a given instance is the same as the one that is searched for. Thus, the equals (=) operator must be defined for that data type. Another example is a sort algorithm, requiring the data type to support comparisons as well as a swapping operation.

The more operations a given data type supports, the wider variety of generic algorithms the data type can use. Conversely, the more operations a generic algorithm requires of a type, the smaller number of data types are supported by the algorithm. It is therefore desirable for a custom data type to support the widest possible number of operations (typically in the form of operator overloading) and for a generic algorithm to require the smallest possible number of operations.



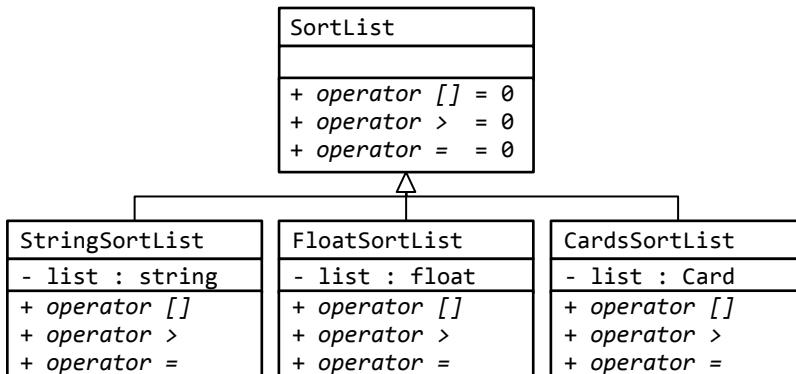
Sue's Tips

The more operations that a data type must support to use a given template, the more difficult it will be for the client to use your template. It is therefore a good idea to carefully manage the list of operations that a given generic data-structure or algorithm uses when designing your template.

Comparison with polymorphism

At this point, you may have noticed that polymorphism addresses many of the same problems that type-independent design is meant to solve. In the case of templates, each data type needs to support a given set of operators to work with a generic algorithm. In the case of polymorphism, the derived classes need to implement the required functions for the generic algorithm to work. They solve the same set of problems.

Consider the bubble sort. To make the bubble sort handle multiple types of lists using polymorphism, it is necessary to define an abstract type called `SortList` with three operations (`[]`, `>`, and `=`). Note that two child classes (`StringSortList` and `FloatSortList`) may implement a given operator (`>`) completely differently.



Through the late-binding feature of polymorphism, the appropriate function will be selected at run-time and the `sort()` function will work with all child data types:

```
void sort(SortList & list)
{
    for (int iOutside = n - 1; iOutside > 0; iOutside--)
        for (int iInside = 0; iInside <= iOutside - 1; iInside++)
            if (list[iInside] > list[iInside + 1])
            {
                SortList tmp = list[iInside];
                list[iInside] = list[iInside + 1];
                list[iInside + 1] = tmp;
            }
}
```

All parameters must be derived from `SortList`

All derived classes must support the `>` operator

All derived classes must support the `=` operator

All derived classes must support the `[]` operator

To do the same thing with a template, a single sort routine will be called using a generic or template data type for the input list. From here, the `[]`, `>`, and `=` operators will be called in the expected way. Then, when a given instance of `sort()` is referenced from the code, the compiler will create a new version of the `sort()` function specific to the passed data type. If the passed data type does not support all the operators referenced in the template function, it will simply fail to compile.

```
template <class T>
void sort(T & list)
{
    for (int iOutside = n - 1; iOutside > 0; iOutside--)
        for (int iInside = 0; iInside <= iOutside - 1; iInside++)
            if (list[iInside] > list[iInside + 1])
            {
                T tmp = list[iInside];
                list[iInside] = list[iInside + 1];
                list[iInside + 1] = tmp;
            }
}
```

The data type `T` is substituted for the caller's passed data type at compile time.

All `T` data types must support the `>` operator

All `T` data types must support the `=` operator

All `T` data types must support the `[]` operator

We will learn the syntax of templates in Chapter 4.2 and 4.3.

How to design with templates

Designing with type-independent design in mind is a multi-step process. It starts with the question of whether templates are the right tool for the job, then moves on to designing the algorithm or data-structure with a simple data type, and finishes with making all references to the data type generic.

Step 1: Is a template the right tool for the job?

Type-independent design all begins with the question “would this algorithm or data-structure work with more than one data type?” For example, a Sudoku solving function will only work with Sudoku boards. It does not make sense to use it for other data types. However, a data-structure designed to contain a set of strings may be a candidate: that same data-structure may be repurposed to contain a set of Accounts.

Many template problems can also be solved using polymorphism as alluded to previously. Which tool should be used? When there is already a shared base-type, polymorphism is the obvious choice. When none exists, templates would probably be a better solution. Consider the following scenarios:

Scenario	Rational	Recommendation
All the clients of a given algorithm share the same base type.	Since there already exists a common base class, there is no additional overhead to using polymorphism rather than building a template.	Polymorphism
The size of the data type is extremely important	Templates duplicate code. Each version of a template function results in another copy of the function as generated by the compiler. Polymorphism does not do this, instead increasing the size of the data type through the use of v-tables.	Templates
Size of the executable is extremely important	Because templates copy code (see above), their use will bloat the size of the executable. It is far cheaper to use polymorphism in this case.	Polymorphism
A simple algorithm is used with a wide variety of data types	Since the data types are so diverse, it does not make sense to create an abstract type to wrap around each individual data type. Templates avoid this necessity.	Templates

Step 2: Use a stand-in data type

The second step is to write the function or data-structure using a simple stand-in data type. Though the function is designed to work with any data type, start with something easy like `floats`. Back to the Bubble Sort example previously, the function should first be written using floating point numbers or some other built-in data type. Care should be taken to test the code thoroughly; it is far easier to find bugs with built-in simple data types than more complex data types that may be used later on.

Step 3: Make the data type generic

The final step in the design process is to replace all references to the stand-in data type (such as `floats`) with the template code. There are several ways to do this: use a void pointer (`void *`) as the data type (see Chapter 11) or use the template mechanism built into C++ (see Chapter 4.2 and 4.3).



Sue's Tips

Some programmers like to use integers as their stand-in data type. This is easy and convenient, but there is one problem. Because integers are often used as counters and as indexes into arrays, it becomes difficult to tell which variables are declared with the stand-in and which need to remain integers. It is therefore a good idea to always use floats as the stand-in data type.

Example 4.0 – Stack

Demo This example will demonstrate the process a programmer may go through when deciding whether a given data-structure should be made into a template.

Problem A stack is a data-structure that contains a list of items. Two operations are supported with a stack: push and pop. Push is the process of adding an item onto the end of the list. Pop is the process of taking an item off the end of the list. Would the stack data-structure be an appropriate application for type-independent design?

The first step is to determine if a template is the right tool for the job. This involves asking a couple questions:

- Would this data-structure work with more than one data type? One can imagine wishing to have a stack of names (people wishing to have access to a professor's office hours for example), a stack of cards (for a variety of card games), and a stack of bills. Yes, a stack can be used with a wide variety of data types.
- Would polymorphism be a better tool for the job? Since there is no obvious base-type connecting all the data types potentially using a stack, templates appear to be a better tool for the job.

The second step is to implement the stack with a stand-in data type: `floats`.

```
class Stack
{
public:
    // create the stack with a zero size
    Stack() : size(0) {}
    // add an item to the stack if there is room. Otherwise throw
    void push(float value) throw(bool)
    {
        if (size < MAX)
            data[size++] = value;    // must support the assignment operator
        else
            throw false;
    }
    // pop an item off the stack if there is one. Otherwise throw
    float pop() throw(bool)
    {
        if (size)
            return data[--size];    // must support the assignment operator
        else
            throw false;
    }
private:
    float data[MAX];           // a stack of FLOATS for now
    int    size;                // number of items currently in the stack
};
```

The third step (make the data-structure work for all data types) will be saved for Chapter 4.1 and 4.3.

Solution The stack implementation using `floats` as the stand-in data type is available at [4-0-stack.html](#) or:

```
/home/cs165/examples/4-0-stack.cpp
```

Example 4.0 – Binary Search

Demo

This example will demonstrate the process a programmer may go through when deciding whether a given algorithm should be made into a template.

Problem

The binary search algorithm is an algorithm useful for finding the location of a given item in a sorted list. Would the binary search algorithm be an appropriate application for type-independent design?

Solution

```
bool binarySearch(const float numbers[], int size, float search)
{
    int iFirst = 0;
    int iLast = size - 1;

    // loop through the list
    while (iLast >= iFirst)
    {
        int iMiddle = (iLast + iFirst) / 2;

        if (numbers[iMiddle] == search)      // must support == operator
            return true;
        if (numbers[iMiddle] > search)      // must support > operator
            iLast = iMiddle - 1;
        else
            iFirst = iMiddle + 1;
    }

    return false;
}
```

The third step (make the algorithm work for all data types) will be saved for Chapter 4.1 and 4.2.

See Also

The stack implementation using `floats` as the stand-in data type is available at [4-0-binarySearch.html](#) or:

```
/home/cs165/examples/4-0-binarySearch.cpp
```

Example 4.0 – Generic Display	
Demo	This example will demonstrate the process a programmer may go through when deciding whether a given algorithm should be made into a template.
Problem	Would a generic display function be an appropriate application for type-independent design?
Solution	<p>The first step is to determine if a template is the right tool for the job. This involves asking a couple questions:</p> <ul style="list-style-type: none"> • Would this algorithm work with more than one data type? On the surface, the answer appears to be “yes.” One might want to create a display function for just about any class. This may work with a graphics application like Asteroids or a Date data type or even a chess board. The problem is that there is basically no shared code between these applications. They do the same types of thing, but in a completely different way. The lack of shared code severely curtails the utility of templates in this application. • Would polymorphism be a better tool for the job? Since the only thing the same between an Asteroids, Date, and Chess display function is the name, then polymorphism does appear to be a candidate for the job. However, there is no obvious base-class representing what is common (or invariant) between these classes. The lack of invariant properties severely curtails the utility of polymorphism in this application. <p>We will not proceed to the second or the third step of the process because templates are not a good tool for the job here.</p>

Problem 1, 2

Consider a function to prompt the user for a `float`, taking error handling into account:

```
float prompt(const char * prompt,
              const char * reprompt = NULL)
{
    bool done = false;
    assert(prompt != NULL);
    float value;

    do
    {
        // instructions
        cout << prompt << ": ";
        cin >> value;

        if (cin.fail())
        {
            if (reprompt != NULL)
                cout << reprompt << endl;
            cin.clear();
            cin.ignore(256, '\n');
        }
        else
            done = true;
    }
    while (!done);

    return value;
}
```

1. Transform the function to work with `Dates`.
2. Back to our `prompt` function, which operators and methods must be supported by our `Date` class to work with `prompt()`?

Please see page 265 for a hint.

Problem 3, 4

Consider a class to store a collection of floats as a stack.

```
class Stack
{
public:
    Stack() : size(0) {}

    void push(float value) throw(bool)
    {
        if (size < MAX)
            data[size++] = value;
        else
            throw false;
    }

    float pop() throw(bool)
    {
        if (size)
            return data[--size];
        else
            throw false;
    }

private:
    float data[MAX];
    int   size;
};
```

3. Transform the class to work with `Dates`.
4. Back to our `Stack` class, which operators and methods must be supported by our `Date` class to work with `Stack`?

Please see page 265 for a hint.

4.1 Void Pointers and Callbacks

Sue was working on her Skeet project (Project 2) and noticed that somehow OpenGL knew to call her `callback()` function every time a new frame in the game is drawn. How did OpenGL know about this function? Why did it take a `void` pointer as a parameter? This is all quite confusing...

Objectives

By the end of this chapter, you will be able to:

- Define “callback” and explain why it is a useful programming pattern
- Describe situations where `void` pointers can help the programmer

Prerequisites

Before reading this chapter, please make sure you are able to:

- Create a variable that is a pointer to a function (Chapter 1.5)
- Pass a pointer to a function as a parameter (Chapter 1.5)
- Cast one data type into another (Procedural Programming in C++, Chapter 1.3)

What are void pointers & callbacks and why you should care

A **callback** is a function pointer passed to another function as a parameter with the expectation that the function pointer will be executed. Perhaps this is best explained by analogy. Consider a general planning for a battle with the enemy. Up on the front lines, he anticipates that the enemy will try some attack. If this happens, he wants his army to respond a certain way. To deal with this contingency, he puts some orders in a sealed envelope for one of his lieutenants. If the lieutenant observes the enemy trying the attack, the lieutenant is to respond with the instructions in the sealed envelope. In this analogy, the general is the caller, the function initiating the exchange and providing the instructions. The lieutenant is the callee, the function receiving the instructions and the function to carry out the instructions at a pre-determined time. The instructions in the sealed envelope are the callback. These are instructions provided by the caller to be executed by the callee. While this analogy with the general, the lieutenant, and the sealed envelope may seem contrived, it is actually quite common in practice. Consider a graphics library (such as OpenGL) enabling a program to display images on the screen and to receive keyboard events. If such a keyboard event were to occur, how would the graphics library notify the client program? The most convenient way for this to happen is for the graphics library (lieutenant) to call a function (sealed envelope) specified by the client (general). This way the program can handle the event as needed.

A **void pointer** is a pointer to an unknown data type. This is the same as saying “I have a pointer to something; I am just not sure what!” This, too, is best explained by an analogy. A mailman delivers letters to a collection of mailboxes. He does not need to know (and frankly he should not look at) what is in each envelope. In essence, the mailman deals with `void` pointers. A `void` pointer cannot be dereferenced; it first must be casted into a known data type. This may seem a bit pointless. Why would one want to carry around a pointer one cannot dereference? The answer is that the mailman (in our analogy) does not know how to dereference the `void` pointer, but the client who receives the letter does.

The combination of `void` pointers and callback functions enable programmers to create algorithms and data-structures that work with any data type. These algorithms and data-structures work with data in the form of `void` pointers, relying on callback functions to know how to work with the data.

Callback functions

There are three players in any callback scenario: the caller, the callee, and the callback function itself:

The **caller** is the function initiating the callback (the general in our analogy). The caller issues a function call to the callee passing the callback function as a parameter.

```
void caller()                                // the function initiating the callback scenario
{
    callee(&callback);                      // the callback function is sent to callee as a
                                                //     parameter in the form of a function pointer
```

The **callee** is the function receiving the call from the caller (the lieutenant). The callee is needed by the caller to carry out some task involving the callback function. To do this the caller needs to accept the callback function as a function pointer. Presumably this callback function will need to be executed at some point. Either it will need to be executed immediately before control is returned to the caller, or it will be saved for some later event.

```
void callee(void (*callback)())      // the callee takes a callback as a parameter
{
    callback();                        // the callee executes the callback
                                                //     on behalf of the caller
```

The **callback** is the function to be executed by the callee on the caller's behalf (the sealed envelope). The callback function is a normal function that gets sent to the callee by the caller as a parameter.

```
void callback()                            // just an ordinary function meant to perform
                                            //     some operation on behalf of the caller
{
                                            //     at a point in time specified by the callee
```

Using callback functions

Typically, the caller does not care *how* the callee responds to the caller's request; the caller just needs to know that the job was done. Under certain situations, however, this simple model is not rich enough to capture what needs to transpire. In these situations, the caller not only specifies what needs to be done, but also how. One of the main ways this is done is through callbacks.

Recall from Chapter 1.5 that a function pointer can be passed as a parameter to another function. Usually this is done with the expectation that the callee will execute the passed function under certain circumstances. Consider the following example of a function receiving input from the user:

```
*****
 * PROMPT
 * Ask the user for a number meeting certain
 * criteria.
 ****
float prompt(bool (*pValidate)(float), const string & message)
{
    float response;
    cout << message << ": ";
    cin >> response;

    while (!pValidate(response))      // call the callback provided by the client
    {
        cout << "Invalid number. Please enter another: ";
        cin >> response;
    }

    return response;
}
```

This function will keep prompting the user for a response, not quitting until a valid number is entered. The question is: under what condition is a response valid? That depends on the callback function `pValidate()` provided by the caller.

The first client of this function is a program asking the user for a GPA. The client (`caller1()`) will need to provide a callback function (`validateGPA()`) to pass to the `prompt()` function:

```
// callback for caller1()
bool validateGPA(float input)
{
    return (input >= 0.0 && input <= 4.0);
}

// first caller, passing validateGPA() as a parameter to the callee prompt()
void caller1()
{
    float gpa = prompt(&validateGPA, "Please enter a GPA");
    cout << "The GPA is: " << gpa << endl;
}
```

The second client is a program asking the user for his salary. This client (`caller2()`) will need to provide a different callback function (`validateIncome()`) for the `prompt()` function:

```
// callback for caller2()
bool validateIncome(float input)
{
    return (input >= 0.00);
}

// second caller, passing validateIncome() as a parameter to the callee prompt()
void caller2()
{
    float income = prompt(&validateIncome, "What is your yearly income");
    cout << "The income is: $" << income << endl;
}
```

By passing a callback function pointer to the `prompt()` function, it has become much more versatile than it would otherwise because it can work with many different types of values.



Sue's Tips

The whole point of a callback function is that the callee can perform an operation on behalf of the caller without knowing anything about that operation. Therefore, it is vital that the callback function be as cohesive as possible and have absolutely no dependencies on the callee. If the callee has to know anything about the inner workings of the callback function, the entire purpose of the callback is defeated!

Example 4.1 – Prompt function

Demo	This example will demonstrate how to pass a callback function to the callee it can perform a specific action according to the caller's specification.
Problem	Write a generic prompt function performing error-recovery and validation according to the specification sent by the caller. The prompt function should also accept zero parameters, one parameter (the prompt string), two parameters (prompt and validation callback), or three parameters (prompt, callback, and error message).
Solution	<p>The first step is to identify the prototype. In order to handle the four ways the <code>prompt()</code> function can be called, default parameters will be specified for all three parameters:</p> <pre>float prompt(const char * sPrompt = "Enter a value", // prompt string bool (*pValidate)(float) = NULL, // validation callback const char * sReprompt = NULL); // reprompt string</pre> <p>Next, the function will be specified. We will borrow heavily from <code>1-1-getIndex.cpp</code> which handles unexpected characters in the input stream. Pay special attention to how the callback function is used.</p> <pre>float prompt(const char *sPrompt, bool (*pValidate)(float), const char *sReprompt) { bool done = false; float value; do { // instructions cout << sPrompt << ": "; cin >> value; if (cin.fail()) // was a non-float entered? { if (sReprompt != NULL) cout << sReprompt << endl; cin.clear(); // clear the error state cin.ignore(256, '\n'); // clear the buffer } else if (NULL != pValidate && !pValidate(value)) // are we valid? { if (sReprompt != NULL) cout << sReprompt << endl; } else // otherwise, we are good! done = true; } while (!done); return value; }</pre>
See Also	<p>The complete solution is available at 4-1-prompt.html or:</p> <pre>/home/cs165/examples/4-1-prompt.cpp</pre> 

Example 4.1 – OpenGL callbacks

Demo Another common use for callback functions occurs when a library needs to notify the client of an event. This is quite common with networking libraries, the operating system, and graphics libraries.

Problem Write a program to accept callback events from OpenGL in the form of keyboard input and drawing requests. Register these callbacks with OpenGL and then write the callback code.

Solution We register callback events with the GLUT interface to OpenGL when the graphics window is first created. This occurs in the `Interface::initialize()` method of `uiInteract.cpp`:

```
void Interface :: initialize(int argc, char ** argv, const char * title)
{
    ... code removed for brevity ...
    // register the callbacks so OpenGL knows how to call us
    glutDisplayFunc( drawCallback );           // register the draw callback
    glutIdleFunc( drawCallback );             // drawCallback gets called twice
    glutKeyboardFunc( keyboardCallback );      // keyboard callback
    glutSpecialFunc( keyDownCallback );        // cursor key callback
    glutSpecialUpFunc( keyUpCallback );         // another cursor key callback
    ... code removed for brevity ...
}
```

Notice how we are passing normal functions to the callback registration functions. We are not passing methods. However, our `Interface` class needs to be made aware of these events. To handle this, we need to instantiate an `Interface` object and call the appropriate method:

```
void keyDownCallback(int key, int x, int y)
{
    // Even though this is a local variable, all the members are static
    // so we are actually getting the same version as in the constructor.
    Interface ui;
    ui.keyEvent(key, true /*fDown*/);
}
```

This means that as soon as the user presses a key on the keyboard, OpenGL will call `keyDownCallback()` so the client can respond to the event. We call this an “asynchronous” callback because it does not occur when the callee is executed but rather at some time in the future.

See Also The complete solution is available at:

```
/home/cs165/prj4/uiInteract.cpp
```

Void pointers

Along with the standard built-in data types (`int`, `float`, `bool`, `char`, etc.), there is a special data type called “`void`.” This is the “type-less” data type, or a data type that has no type. While you can’t declare a variable of type `void`, you often want a function to have the `void` return type.

```
void display(); // return nothing
```

So what is the point of having a data type which can’t be used in a variable declaration? The answer lies with pointers. While we commonly know the type of data an item is pointing to (pointer to a character, for example), this is not always the case. The need frequently arises when a function does not need to know and cannot know the type of data a pointer is pointing to. In these cases, we use a `void` pointer:

```
{  
    void * p; // we point to something, but we don't know what!  
}
```

When we do this, we are stating “I don’t need to know what this thing is pointing to.” Under normal circumstances, we can access the data in a pointer with the dereference operator. For example, consider the case where a pointer refers to the first element in a string. In this case, we can retrieve the data by dereferencing the pointer:

```
{  
    char text[256] = "c-string";  
    char * pChar = text;  
  
    cout << *pChar << endl; // this will display a lowercase 'c'  
}
```

With `void` pointers, we cannot dereference it directly because the resulting data type is `void`. To get around this, we need to cast the pointer to a known data type:

```
{  
    char text[256] = "void pointer";  
    void * pVoid = (void *)text; // cast it to a void * when assigning an address  
                                // to a void pointer  
    cout << *(char *)pVoid << endl; // cast it back to a known data type when we  
                                // want to access the data in the void pointer  
}
```

Therefore it is possible to store data in a `void` pointer as long as someone knows how to cast it back into a known data type. When working with `void` pointers, there are typically two steps:

1. Cast the data from a known data type into a `void *`
2. Cast the data back from `void *` into the known data type

Sam's Corner



When assigning one data type onto another, a cast is usually required:

```
{  
    float floatValue = 3.14;  
    int valueInteger = (int)floatValue; // the compiler might complain without  
                                    // the (int) cast  
}
```

However, we do not have to use a cast when assigning a pointer to a `void` pointer:

```
{  
    void * pVoid = "Text"; // no need to cast a constant char *  
                          // into a void *  
}
```

Example 4.1 – Swap

Demo

This example will demonstrate how to accept void pointers as parameters to a function. The function then does not need to know anything about the data being manipulated.

Problem

Write a swap function that works with any pointer data type. The function should then swap the pointers in memory. Note that it does not swap the data that the pointers refer to.

The purpose of the `swap()` function is to swap pointers. Therefore, the pointers have to be passed by reference. This may look odd at first. Think of it this way: the function needs to be able to change the address of the pointer being passed as a parameter.

```
void swap(void * & pLhs, void * & pRhs)
{
    void * pTemp = pLhs;
    pLhs = pRhs;
    pRhs = pTemp;
}
```

Now to call this function, we need to pass pointers that we can change.

```
int main()
{
    char buffer1[256];           // this is a constant pointer. We can change the
    char buffer2[256];           //     data but not the address of buffer1
    char * p1 = buffer1;         // p1 shares the address of buffer1 but we
    char * p2 = buffer2;         //     have the ability to change the address

    // prompt the user
    cout << "First message: ";
    cin.getline(buffer1, 256);
    cout << "Second message: ";
    cin.getline(buffer2, 256);

    // before we swap
    cout << "Before: \" " << p1 << "\", \" " << p2 << "\"\n";

    // swap them
    swap(p1, p2);               // note that we do not need to cast the char *
                                //          to a void pointer
    // display after the swap
    cout << "After: \" " << p1 << "\", \" " << p2 << "\"\n";

    return 0;
}
```

Unit 4

Challenge

As a challenge, can you change the driver program to swap pointers to `floats`? Note that you will need to allocate the `floats` that are swapped.

See Also

The complete solution is available at [4-1-swap.html](#) or:

```
/home/cs165/examples/4-1-swap.cpp
```



Example 4.1 – Stack

Demo This example will demonstrate how to use `void` pointers in a container (classes designed to hold data). In this case, the container does not need to know what type of data is being stored. Instead, it just needs to give the data back to the client when he asks for it.

Problem Write a class to implement a Stack, a container following the “first-in, last-out” pattern. As with “4.0 Stack,” this class will support the push operation (place an item on the end of the list) and the pop operation (remove an item from the end of the list).

Solution We will start from the stack of `floats` from `4-0-stack.cpp`. The only thing we need to do is replace the `float` stand-ins with `void` pointers.

```
class Stack
{
public:
    // create the stack with a zero size
    Stack() : size(0) {}
    // add an item to the stack if there is room. Otherwise throw
    void push(void * value) throw(bool)
    {
        if (size < MAX)
            data[size++] = value;    // must support the assignment operator
        else
            throw false;
    }
    // pop an item off the stack if there is one. Otherwise throw
    void * pop() throw(bool)
    {
        if (size)
            return data[--size];    // must support the assignment operator
        else
            throw false;
    }
private:
    void * data[MAX];           // a stack of void pointers
    int    size;                // number of items currently in the stack
};
```

When we test the program, we need to cast our data type into a “`void *`” when we push it on and cast it back to something useful when we pop it off.

```
{
    Stack stack;
    stack.push((void *)"text");
    cout << (char *)stack.pop() << endl;
}
```

Challenge As a challenge, modify the driver program so it works with pointers to `strings`. Note that you will need to push on a pointer to a `string` so it will need to be allocated with `new`.

See Also The complete solution is available at [4-1-stack.html](#) or:

```
/home/cs165/examples/4-1-stack.cpp
```



Callbacks with void pointers

Recall that a callback is a function that is passed to a callee so it can be executed on the caller's half. The callee does not need to know anything about the function being passed. Recall that a `void` pointer is a pointer to an unknown data type so, when passed to a function, the function does not need to know anything about the data being passed. It turns out that combining callbacks and `void` pointers give the programmer even more power to work with generic algorithms and data-structures.

Consider the typical `void` pointer scenario: a function taking a `void` pointer as a parameter but is unable to work with the data because nothing is known about the data type of the `void` pointer. However, if the caller also provides a callback function that knows how to work with the `void` pointer, we have the power to implement truly type-independent data-structures and algorithms.

To illustrate this point, recall the bubble sort example from Chapter 4.0. Ideally we should be able to use the bubble sort generic algorithm with just about any data type. It should work with text, numbers, playing cards, and many other data types. As long as the data type supports a few operations, we should be able to use it. Specifically, we need to be able to swap two items and compare two items. This means that if we are to write a generic sort algorithm, we will need to pass several parameters:

- `array`: The list of items that are to be sorted. This will be an array of `void` pointers of course
- `num`: The number of items in the `array`. This will always be an integer
- `compare()`: A callback function knowing how to compare two items in the list. Since `array` contains `void` pointers, the `compare()` function will need to know how to cast the `void` pointer into a known data type
- `swap()`: A callback function knowing how to swap two items in the list.

Recall from Chapter 4.0 the necessity of indentifying the operations required to implement a generic algorithm or generic data-structure. When using callback functions and `void` pointers, each operation gets passed as a callback to the function or class.

Thus the callback function and the `void` pointer work as a pair: the callback is the only function needing to be able to cast the `void` pointer back to the data in which it really represents. In this scenario, the role of the callee is not to interpret the `void` pointer, but rather to just hand it over to the callback.



Sue's Tips

A common objection to the callback/void-pointer pairing is that often more than one parameter needs to be sent to the callback. It turns out, however, than *you only ever need a single parameter*. Since that `void` pointer can refer to a pointer to a structure or to a class, arbitrarily rich data can be sent through that single parameter.

Example 4.1 – Binary search

Demo

This example will demonstrate how to use `void` pointers in conjunction with a callback function in order to implement a generic algorithm. This algorithm should work with any data type as long as there is a `compare()` function that works with it.

Problem

Write a binary search algorithm that can work with any data type. Only four parameters are needed: the array of `void` pointers, the number of items in the array, a callback function to compare two values, and the value being searched for.

Solution

```
bool binarySearch(const void * array[],           // an array of void pointers
                  int size,                      // number of items in array
                  int (*pCompare)(const void *, const void *),
                  const void * search)          // the search value
{
    int iFirst = 0;
    int iLast = size - 1;

    // loop through the list
    while (iLast >= iFirst)
    {
        int iMiddle = (iLast + iFirst) / 2;

        int compare = pCompare(array[iMiddle], search);
        if (compare == 0)                         // 0 means they are the same
            return true;
        if (compare > 0)                          // positive means bigger
            iLast = iMiddle - 1;
        else                                      // negative means smaller
            iFirst = iMiddle + 1;
    }

    return false;
}
```

To test this function, we will need to cast our data into `void` pointers:

```
if (binarySearch((const void **)values,
                 10,
                 (int (*)(const void *, const void *))strcmp,
                 (const void *)search))
    cout << "\tThe name is in the list\n";
else
    cout << "\tThe name is not in the list\n";
```

Challenge

As a challenge, can you change the driver program to search through an array of `string` objects rather than using c-strings. Note that you will need to write your own comparision function and you will need to pass to `binarySearch()` an array of pointers to `string` objects.

See Also

The complete solution is available at [4-1-binarySearch.html](#) or:

/home/cs165/examples/4-1-binarySearch.cpp



Example 4.1 – OpenGL part 2

Demo

This example will demonstrate how the callback mechanism is much more powerful and versatile when paired with a `void` pointer. When a graphics library such as OpenGL notifies the client that it is time to draw, the client needs the program state to complete the operation. This program state comes in as a `void` pointer.

Problem

Write a program to display a rotating polygon on the screen using the OpenGL library. First, create a class representing the state of the program (a polygon in this case), next write a callback function to be executed by OpenGL when it is time to draw, and finally, pass a `void` pointer (with the program state) and a callback function (the draw functionality) to OpenGL.

The first step is to write a class representing the state of the program. For a game, this will be the game class. In this example, it will be the position, angle, and number of sides of the polygon.

```
class Ball
{
public:
    Ball() : sides(3), rotation(0), pt() { }

    // this is just for test purposes. Don't make member variables public!
    Point pt;           // location of the polygon on the screen
    int sides;          // number of sides in the polygon. Initially three
    int rotation;       // the angle or orientation of the polygon
};
```

Next the callback function which will handle any input (from pUI), any physics that need to be computed (rotating the polygon), and the drawing routines.

```
void callBack(const Interface *pUI, void * p)
{
    Ball * pBall = (Ball *)p; // cast the void pointer into a known type
    ... code removed for brevity ...
    if (pUI->isSpace())
        pBall->sides++;           // now use our Ball class as we normally would
        pBall->rotation++;

    // draw
    drawCircle(pBall->pt, /*position*/
               20, /* radius */
               pBall->sides /*segments*/,
               pBall->rotation /*rotation*/);
}
```

Notice how the first thing the callback function does is to cast the `void` pointer into a known data type. The last thing to do is to pass the callback function and the void pointer to OpenGL.

```
int main(int argc, char ** argv)
{
    Interface ui(argc, argv, "Test");      // initialize OpenGL
    Ball ball;                            // initialize the game state
    ui.run(callBack, &ball);              // set everything into action, passing a
                                            // void pointer and callback to OpenGL
    return 0;
}
```

The complete solution is available at:

```
/home/cs165/prj2/uiTest.cpp
```

See Also

Problem 1 - 4

Consider the STL container `vector`. It is an array that grows to accommodate as much data as the client puts in it.

Vector	
-	data
-	capacity
-	size
+	Vector
+	pushBack
+	operator []
+	getSize
-	resize

1. Write the class definition for `vector` where any data type can be stored.
2. Write the code for the array-index operator.
3. Write the code for the `pushBack()` method.
4. Write the code for the `resize()` method.

4.2 Function Templates

Sam has just finished implementing the bubble sort with `void` pointers and callback functions. Unfortunately, he called the bubble sort function with a callback function corresponding to different data than that was in his `void` pointer array. This mixup produced a crash that was difficult to find and debug. If only the compiler could help him catch such bugs...

Objectives

By the end of this chapter, you will be able to:

- Define function templates and explain when they can be used
- Implement a generic function using function templates

Prerequisites

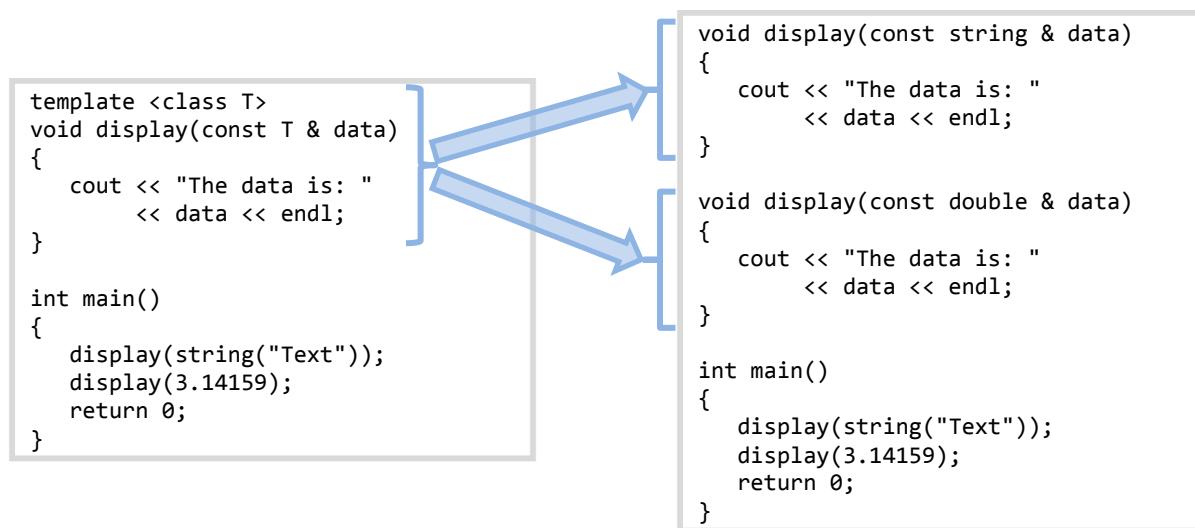
Before reading this chapter, please make sure you are able to:

- Explain circumstances when generic functions can help make code more reusable (Chapter 4.0)
- Enumerate the steps used to define type-independent algorithms (Chapter 4.0)
- Identify the operators required to work with a given generic function (Chapter 4.0)

What are function templates and why you should care

A **function template** is a mechanism in C++ allowing the programmer to implement a generic algorithm that works with any data type. They are an early-binding mechanism allowing the compiler to catch most of the errors introduced by the data abstraction process (unlike the `void` pointer and callback mechanism doing the same thing (see Chapter 4.1)).

Function templates enable us to implement generic algorithms which work with many data types. In effect, the compiler turns our one function template into many individual functions according to the needs of the program:



Function template syntax

There are two aspects to the syntax of a function template: the definition of the template data type (called the **template prefix**) and the use of the template data type in the function (called the **type parameter**).

Template prefix

In order to tell the compiler that a given function uses a template, it is necessary to use a template prefix:

```
template <class T>
```

The template prefix goes immediately before the function definition, stating that this function will have at least one parameter that uses an unknown data type.

Note that we can call our new data type “ T ” though any other valid C++ name will work. By convention we use “ T ” because data types should be TitleCased and because we don’t know anything else about the data type to give it a more descriptive name.

There is one other aspect about the template prefix: it is possible for a function to accept two abstract data types corresponding to potentially dissimilar types. In this case, we can declare two templates in the prefix:

```
template <class T1, class T2>
```

Type parameter

Once a template prefix has been created, the programmer can then use the data type T the same as he would with any other data type in the program. This means that variables and parameters can be declared as type T rather than with a built-in data type (such as `int`, `float`, `bool`, etc.) or with a class (such as `Date`, `Point`, etc.).

```
{
    T variable;    // the variable is of type T
}
```

Template prefix with type parameter

Consider the standard `swap()` function:

```
void swap(float & lhs, float & rhs)
{
    float tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

The same function, using a template prefix and a type parameter, would be:

```
template <class T>
void swap(T & lhs, T & rhs)
{
    T tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

Since it is necessary for both the left-hand-side and the right-hand-side to be the same data type for the `swap()` function to work, only one type parameter is used for both the parameters.

Warning: all the type parameters represented in the template prefix must be represented in the function template parameters. If not, the compiler will not be able to tell which version of the template to use.

How function templates work

Though we introduced the term “template” in conjunction with abstract data types in Chapter 4.0, no explanation was given why we call them “templates.” The answer comes from how templates are used.

When the compiler encounters a standard function definition, it turns the C++ code into machine language which eventually resides in an executable program. This occurs because, at compile time, the compiler knows all it needs to know to turn the C++ function into CPU-ready machine language.

When the compiler encounters a function template, it cannot generate the C++ code. The reason is that it does not know what data type the type parameter “T” refers to. Only when the compiler discovers how the template function is used does the necessary machine language code get generated. To illustrate this point, consider the following code using our newly defined `swap()` template function.

```
{  
    int value1 = 100;  
    int value2 = 200;  
    swap(value1, value2);           // the compiler will now know that an integer  
                                    // version of swap() is required  
}
```

When the compiler encounters this code, it now know that an integer version of `swap()` is required. It will first see if an integer version was created and, if not, generate a new copy of `swap()`:

```
template <class T>  
void swap(T & lhs, T & rhs)  
{  
    T tmp = lhs;  
    lhs = rhs;  
    rhs = tmp;  
}  
  
void swap(int & lhs, int & rhs)  
{  
    int tmp = lhs;  
    lhs = rhs;  
    rhs = tmp;  
}
```

The compiler will literally copy the original version of `swap()` that has the template information and replace all instances of the type parameter “T” with the data type of the parameter: `int`. In other words, it is called a “template” because the compiler uses the declaration of the function as a pattern describing how to generate the version of the function that gets compiled.

A warning...

Consider this very simple (and utterly useless) function template:

```
template <class T1, class T2>  
void displayTwo(T1 & t1, T2 & t2)  
{  
    cout << t1 << t2;  
}
```

If there are 20 available data types in a given program (including built-in data types as well as class definitions), that means that there are $20 \times 20 = 400$ combinations. If we call the `displayTwo()` function with every combination of data types, this means that 400 copies of `displayTwo()` will be generated by the compiler. Therefore the compiler creates a unique copy of the template function for every combination of parameters sent to it. This can result in an unnecessarily large executable.



Sue's Tips

In practice, we seldom encounter more than a couple copies of a given template function in one program. However, the programmer should always be aware of the potential for code-bloat.

Function template quirks

There are a few quirks about function templates that all programmers should be aware of. If we think about how the compiler treats templates (namely generating new copies of the template function), each of the quirks should make sense.

Templates and function signatures

The first quirk is that the compiler needs to figure out which version of the function template is needed based on the function signature. Namely, the function signature implied when the function template is called should unambiguously describe all the type parameters.

To illustrate this point, consider the following function template:

```
template <class T>
T getValue()           // ERROR: the type parameter T is not in the parameter list!
{
    T value;
    cin >> value;
    return value;
}
```

This will result in a compile error:

```
error.cpp: In function “int main()”:
error.cpp:23: error: no matching function for call to “getValue()”
```

All the type parameters needs to be in the parameter list:

```
template <class T>
void getValue(T & value) // no problem because the caller can specify the data type
{                      //      of the type parameter “T”
    cin >> value;
}
```

Supported operators

The second quirk is that the all the operators used in the function template need to be supported by the data type which the type parameter represents. For example, the insertion operator is supported by `int`:

```
{
    int value;
    getValue(value);      // legal because the insertion operator used in getValue()
}                         //      supports the data type “int”
```

However, we will get a compile error if the data type is not supported by the insertion operator:

```
{
    struct Coord           // recall that we can declare a structure in a function
    {
        int row;
        int col;
    } coordinate;          // recall that we can instantiate a variable at declaration
    getValue(coordinate); // ERROR: type Coord does not support the insertion operator
}
```

Every data type that is used by the type parameter must support all the operators used in the template function. If an operator is not supported, we will get a compile error.

Example 4.2 – Swap

Demo

This example will demonstrate the simplest function template, the `swap()` function. Clients of the `swap()` function only need to support the assignment operator.

Problem

Write a generic `swap()` function that can work with any data type. The function will swap the data in the left-hand-side with that of the right-hand-side.

Solution

In Chapter 4.1, we had the following `void` pointer version of the swap function:

```
void swap(void * & pLhs, void * & pRhs)
{
    void * pTemp = pLhs;
    pLhs = pRhs;
    pRhs = pTemp;
}
```

The template version does away with the `void` pointer:

```
template <class T>
void swap(T & lhs, T & rhs) // only one type parameter because both the
                           //   lhs and rhs are the same data type
{
    T tmp = lhs;           // we could also say "T tmp(lhs);;" and it
    lhs = rhs;             //   would work, but it would require T
    rhs = tmp;             //   to support both the copy-constructor
                           //   and the assignment operator
```

Now our driver program will use both the `string` class and floating point numbers:

```
int main()
{
    // swap text
    string text1("First");
    string text2("Second");
    swap(text1, text2);
    cout << "The values swapped: " << text1 << ' ' << text2 << endl;

    // swap numbers
    double value1(3.14159);
    double value2(2.71828);
    swap(value1, value2);
    cout << "The values swapped: " << value1 << ' ' << value2 << endl;

    return 0;
}
```

Notice how two versions of `swap()` will be generated: both the `string` version and the `double`.

Challenge

As a challenge, modify the driver program to swap c-strings. Why does it not work? Hint: how does the assignment operator work for c-strings?

See Also

The complete solution is available at [4-2-swap.html](#) or:

```
/home/cs165/examples/4-2-swap.cpp
```



Example 4.2 – Prompt function

Demo

This example will demonstrate how to use function templates to write generic algorithms. As long as the data type which will eventually be replacing the type parameter supports the extraction operator and can copy itself, the function should work as the client expects.

Problem

Write a generic `prompt()` function that can work with any data type. The `prompt()` function will display the instructions on the screen and handle invalid input.

Solution

```
template <class T>
void prompt(T & t, const char * prompt, const char * reprompt = NULL)
{
    bool done = false;
    assert(prompt != NULL); // the prompt is not optional

    do
    {
        // instructions
        cout << prompt << ": ";
        cin >> t; // T needs to support the insertion

        if (cin.fail())
        {
            if (reprompt != NULL) // the reprompt is optional
                cout << reprompt << endl;
            cin.clear();
            cin.ignore(256, '\n');
        }
        else
            done = true;
    }
    while (!done);
}
```

This function works with any data type supporting the extraction operator (`>>`).

```
int main()
{
    // first with strings
    string text;
    prompt(text, "Please enter text");
    cout << "The text you entered is: " << text << endl;

    // second with double
    double number;
    prompt(number, "Please enter a number", "Invalid character");
    cout << "The number you entered is: " << number << endl;
    return 0;
}
```

See Also

The complete solution is available at [4-2-prompt.html](#) or:

/home/cs165/examples/4-2-prompt.cpp



Example 4.2 – Binary search

Demo This example will demonstrate how use function templates with complex generic algorithms. The type parameter of this function will need to support the greater-than operator (`>`) and the equivalence operator (`==`).

Problem Write a generic `binarySearch()` function that can work with any data type. The prompt function will determine whether a given value is in the passed array.

Solution The easiest way to make a function template for `binarySearch()` is to modify the version with a `float` stand-in from “Example 4.0 – Binary search” and replace “`float`” with “`T`”:

```
template <class T>
bool binarySearch(const T array[],                // an array of values
                  int size,                      // number of items in array
                  const T & search)             // the search value
{
    int iFirst = 0;
    int iLast = size - 1;

    // loop through the list
    while (iLast >= iFirst)
    {
        int iMiddle = (iLast + iFirst) / 2;

        if (array[iMiddle] == search)           // must support == operator
            return true;
        if (array[iMiddle] > search)           // must support > operator
            iLast = iMiddle - 1;
        else
            iFirst = iMiddle + 1;
    }

    return false;
}
```

Observe how the template version of this function is nearly identical to the `float` stand-in; it is only necessary to replace all instances of “`float`” with “`T`”.

Challenge As a challenge, can you modify the driver program to search through an array of `string` objects? Why would it work with `string` objects but not with c-strings?

See Also The complete solution is available at [4-2-binarySearch.html](#) or:

/home/cs165/examples/4-2-binarySearch.cpp



Problem 1

Write a function to return the maximum of two values. The two values passed as parameters can be of any data type that supports the greater-than operator (`>`). Example of how the function can be used:

```
{
    cout << max(5, 2) << endl;
    cout << max(string("first"), string("last"));
}
```

Please see page 290 for a hint.

Problem 2

Write a function to display the contents of an array. Each item in the array will have a space separating them on the screen. The data type of the array, of course, can be anything. Example:

```
{
    int data1[] = { 4, 2, 7, 3};
    display(data1, 4);           // displays "4 2 7 3"

    char data2[] = "template";
    display(data2, 8);          // displays "t e m p l a t e"
}
```

Please see page 292 for a hint.

Problem 3

Write a function sum the contents of an array. Return the sum. The data type of the array can be anything of course. Example:

```
{
    int data1[] = { 4, 2, 7, 3};
    cout << sum(data1, 4) << endl;      // displays "16"

    string data2[3] = { string("Thomas "), string("E. "), string("Ricks") };
    cout << sum(data2, 3) << endl;      // displays "Thomas E. Ricks"
}
```

Please see page 292 for a hint.

4.3 Class Templates

Sue understands how to make a template out of a function, but that does not help her make an abstract data-structure. She would like to make an array-like data-structure that works with any data type. If only there was a way to apply the template mechanism to a class...

Objectives

By the end of this chapter, you will be able to:

- Design a class with an abstract data type
- Create a class template
- Explain situations when class templates would be a useful tool

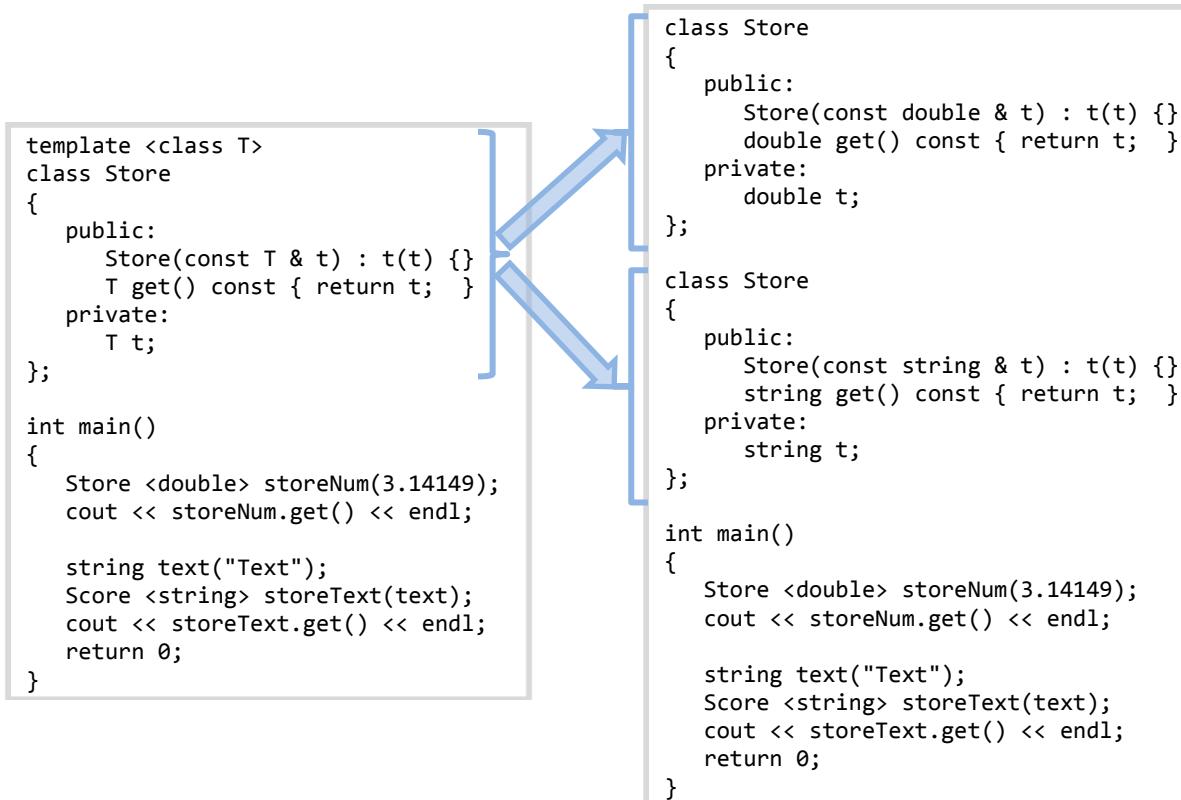
Prerequisites

Before reading this chapter, please make sure you are able to:

- Write a class definition using built-in data types
- Create a function template

What are class templates and why you should care

A **class template** is similar to a function template with one exception: one or more member variable (rather than a parameter) uses an abstract data type. While function templates were useful for implementing generic algorithms (like the bubble sort or the swap function), class templates are useful for implementing generic data-structures (like the stack or an array).



Defining a class template

Defining a class template is exactly the same as defining any other class with the exception of the template prefix and the type parameter. There are, however, two additional quirks. First, the template prefix needs to go before every method definition. This can get tedious, but the C++ language demands it. Second, the full name of a template class includes the template designation. In other words, a template class `Store` is actually “`Store <T>`” because the template designation is an integral part of the class identity.

Template prefix

Just as a template prefix needed to go before the function definition of a template function, the template prefix needs to go before the class definition for a class template. Consider, for example, a class designed to store a value:

```
template <class T>
class Store
{
public:
    Store(const T & t) : t(t) {}
    T get() const { return t; } // notice that methods defined inside
private:                      // a class definition do not need
    T t;                      // the template prefix
};
```

The template prefix signifies that the data type `T` can be used anywhere in the class definition. This includes as a parameter to a member function, a local variable within a member function, and as a member variable.

Of course it is possible to define more than one data type in a template prefix:

```
template <class T1, class T2>
class StoreTwo
{
public:
    Store(const T1 & t1, const T2 & t2) : t1(t1), t2(t2) {}
    T1 getOne() const { return t1; }
    T2 getTwo() const { return t2; }
private:
    T1 t1;
    T2 t2;
};
```

Notice how the `getOne()` method returns a `t1` but does not take a `t1` as a parameter. This was impossible with function templates because the client cannot specify which version of the function to use. We do not have this constraint with class templates. The client specifies the version of the class at declaration time. See the section “Declaring an object from a class template” in two pages for details of how this works.

Type parameter

Once the template prefix has been defined, it can be used anywhere a standard data type can be used. In the case of class templates, this almost always means it is used to define member variables. However, it is also commonly used as parameters, return data types, and local variables.

```
template <class T>
class Store
{
public:
    Store(const T & t) : t(t) {} // type parameter used as a parameter
    T get() const { return t; } // type parameter used as a return type
private:
    T t;                      // type parameter used as a member variable
};
```

Class name

When a template is used with a class, the complete class name includes the template designation. Back to our previous example, there is no such thing as the “`Store`” class! Instead there is the “`Store<T>`” class.

To illustrate this point, consider the copy constructor. As you may recall from Chapter 2.4, the copy constructor takes a constant parameter of the same data type as the class. Since the data type of the class is `Store<T>` (not `Store!`), the copy-constructor must be:

```
template <class T>
class Store
{
public:
    Store(const T & t) : t(t) {}
    Store(const Store<T> & t)           // notice the data type of the parameter
    {
        this->t = t.get();
    }
    T get() const { return t; }
    void set(const T & t);
private:
    T t;
};
```

A common mistake is to forget the template designation in the class name.

Method definitions

The final aspect of defining a class template pertains to defining a method outside the class definition. Recall that methods defined inside a class definition are inline. Inline functions and methods should be trivial functions, consisting of just a few lines. Unlike normal non-template classes, template classes must be defined entirely in the header (.h) file. This includes the template class definition, inline method definitions, and non-inline definitions.

Every method definition needs to be preceded by the template prefix. Additionally, since method definitions include the class name, we need to include the template designation as well. Consider a simple `set()` method for our `Store<T>` class:

```
*****
* STORE<T> :: SET
*****
template <class T>
void Store<T> :: set(const T & t)
{
    this->t = t;
}
```

Notice the template prefix immediately before the method definition. Also notice that the full class name of “`Store<T>`” must be used.

Sam's Corner



You do not need to use `T` for your template parameter name. Any other designation can be used. However, we almost always use `T` so it is clearly recognized as a template parameter. If you feel particularly creative, you could use `T` for the template parameter name in the class definition and `Q` in the method name. There is nothing to keep the programmer from doing this. That being said, it is generally considered to be a very bad idea! Always use `T` (or `T1` and `T2`) for your template parameter names.

Declaring an object from a class template

The final aspect of using a class template is that, at object declaration time, the flavor of the class template needs to be specified. With a function template, the parameters passed to the function tell the compiler which version of the function is needed. With a class template, we need to be explicit. For example, if the user wishes to instantiate a `double` version of the `Store<T>` class, the following declaration is used:

```
int main()
{
    Store<double> s(3.14159);           // use the "double" version of Store<T>
    cout << s.get() << endl;
    return 0;
}
```

This syntax should be a bit familiar. Recall the syntax for declaring a `vector` of integers:

```
#include <vector>

int main()
{
    vector<int> numbers;           // use the "int" version of vector<T>
    return 0;
}
```

If more than one type parameter is used in a class template, we include all the data types in the `<>`:

```
int main()
{
    StoreTwo<double,int> s(3.14159, 42); // use the "double, int" version of
    cout << s.getOne() << endl;           //   StoreTwo<T1, T2>
    return 0;
}
```

Designing with class templates

The most common use for class templates is to define a custom data-structure. A data-structure is a mechanism for storing or organizing data in a program. Thus far we have discussed data-structures only sparingly. An array is a data-structure, as is a `vector`. We will learn about a new data-structure called a linked list next chapter. In each of these cases, the way that the data-structure works with the individual entities should be completely independent of the nature of the entity. In other words, arrays all work the same, be they `float` arrays or `string` arrays. This is why data-structures are commonly defined with class templates.



Sue's Tips

Data-structures represent a rich topic of discussion, far too rich for this text. Fortunately we have an entire class devoted to studying data-structures: CS 235.

Example 4.3 – Store

Demo

This example will demonstrate the use of a class template with the simplest data-structure imaginable: one to store a single value for the user.

Problem

Write a class to store a single value for the user. The value can be any data type specified by the user, as long as the data type supports the copy constructor and the assignment operator. An example of the use includes:

```
{  
    Store<double> s(3.14159);           // use the "double" version of Store<T>  
    cout << s << endl;  
}
```

Solution

```
template <class T>  
class Store  
{  
public:  
    Store(const T & t) : t(t) { }  
    Store(const Store<T> & store) { *this = store; }  
    T get() const { return t; }  
    void set(const T & t);  
    Store<T> & operator = (const Store<T> & store)  
    {  
        this->t = store.get();  
        return *this;  
    }  
    friend ostream & operator << (ostream & out, const Store<T> & store)  
    {  
        out << store.get();  
        return out;  
    }  
private:  
    T t;  
};
```

Additionally one method is not defined inline (though it should!), `Store<T>::set()`:

```
template <class T>  
void Store<T> :: set(const T & t)  
{  
    this->t = t;  
}
```

Challenge

As a challenge, can you overload the extraction operator (`>>`), the assignment operator with a `T` as the right-hand-side (as opposed to a `Store<T>`), and the `Store<T>::set()` method taking a `Store<T>` on the right-hand-side?

See Also

The complete solution is available at [4-3-Store.html](#) or:

```
/home/cs165/examples/4-3-Store.cpp
```



Example 4.3 – Container

Demo	This example will demonstrate a more complex and worthwhile use of a class template: an array data type that can store any type of data.
Problem	Write a class to emulate the behavior of the <code>vector<T></code> class in the standard template library. This class will support the copy constructor to copy one array onto another, the get method for read/write access, insert method to push an item onto the array, and a variety of other convenient methods and operators.
Solution	<p>The most important part of the class definition includes:</p> <pre>template <class T> template <class T> class Container { public: // constructors Container() : numItems(0), capacity(0), data(0x00000000) {} Container(const Container & rhs) throw (const char *); Container(int capacity) throw (const char *); // destructor : free everything ~Container() { if (capacity) delete [] data; } // is the container currently empty bool empty() const { return numItems == 0; } // remove all the items from the container void clear() { numItems = 0; } // how many items are currently in the container? int size() const { return numItems; } // add an item to the container void insert(const T & t) throw (const char *); T & get(int index) throw (const char *); private: T * data; // dynamically allocated array of T int numItems; // how many items are currently in the Container? int capacity; // how many items can I put on the Container before full? };</pre>
Challenge	As a challenge, can you modify the insert method so the container grows to accommodate any amount of items? See example “4.1 Expanding Arrays” from “Procedural Programming in C++” for details of how to make the buffer grow.
See Also	<p>The complete solution is available at 4-3-container.html or:</p> <div style="border: 1px solid #ccc; padding: 5px; display: inline-block;">/home/cs165/examples/4-3-container.cpp</div>

Problem 1 - 3

Consider a class representing a pair of values. Examples could be a pair of integers (x, y for example), a pair of names (first and last for example) or a `Date` paired with a `string`.

1. Write a class definition for a pair of values. Include three constructors, a getter, a setter, and two member variables.
 2. Write the function definitions for the copy constructor, the getter, and the setter.
 3. Given our new class, declare two objects that are: an integer paired with a float, a string paired with a string. Declare the objects and set them to a value.

Please see page 298 for a hint.

4.4 Linked Lists

Arrays are a great data-structure for storing a collection of values. It is easy to add items to the end of an array and to access any item in the middle of the array provided the index is known. However it is difficult to remove items from the middle of an array. Doing so requires us to shift all the items in the array. Sam is confronted with this while working on his Asteroids project. While it is easy to add a bullet to the collection of game entities, it is difficult to remove a dead bullet without shifting the entire list. If only there was a more convenient data-structure!

Objectives

By the end of this chapter, you will be able to:

- Define and describe the following terms: linked list, node
- Describe the UML class diagram of a `Node` class or structure
- Create a structure definition matching a linked list description
- Explain the situations when a linked list would be preferable to an array and vice-versa

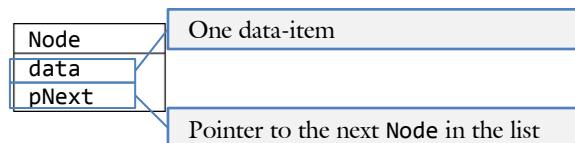
Prerequisites

Before reading this chapter, please make sure you are able to:

- Define the NULL pointer (Procedural Programming in C++, Chapter 4.1)
- Define a structure matching a given UML class diagram (Chapter 1.3)
- Define a generic data-structure as it pertains to abstract types (Chapter 4.0)
- Define a class template (Chapter 4.3)

What are linked lists and why you should care

A **linked list** is a type of data-structure where each item in the list is connected to the rest of the list through pointers. Each item in a linked list is called a **node**. A node is a structure or class consisting of some data-item and a pointer to the next node in the list.



When a collection of nodes point to each other, we have a linked list.



Because the nodes in a linked list do not need to reside next to each other in memory as they do with arrays, it is possible to add items to the middle of a linked list without shifting the rest of the list. This makes them more efficient than arrays in many situations.

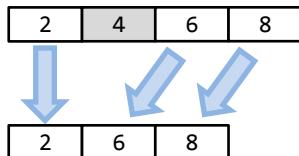


Sue's Tips

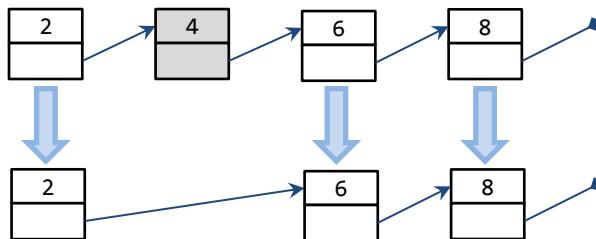
When working with linked lists, it is almost always helpful to draw the block diagram like the one represented above. This helps you keep your pointers straight.

Linked lists and arrays

Arrays were the first “aggregate” or “bucket-of” data-structures we learned about last semester. Arrays have the property that each i^{th} item in the array is guaranteed to be in the memory location beside the $(i+1)^{\text{th}}$ item. This is convenient because you can always jump directly to a given item in the list (we call this “**random access**”). Unfortunately, this makes it difficult to add or remove items from the middle of the list. In order to accomplish this, we typically need to shift all the items after the added or removed item, which can be expensive. Consider an array of four items: {2, 4, 6, 8}. If the second item (4) were to be removed, all the items after it will need to be shifted over by one spot.



Unlike arrays, linked lists are free from proximity constraints. In other words, the i^{th} and $(i+1)^{\text{th}}$ item do not have to be next to each other in memory. While this design feature precludes random access (you need to walk through the list one item at a time to find a given element), it greatly simplifies modification to the list. If the second item in a linked list is to be removed, it is only necessary to redirect the pointer from the first item; there is no need to adjust the rest of the linked list.



Notice that a linked list and an array solve the same class of problems: they hold a collection of data. We call these collections “containers”. In a sense, a linked list is an array of objects. However, linked lists have some fundamental differences from arrays: allocation, indexing, and manipulation.

In an array, we can leave data unused and uninitialized. In other words, items in the array can be left empty. Each item in a linked list, on the other hand, must be allocated. This is typically done with the `new` operator. Therefore, it is much more expensive to create a linked list than an array of corresponding size.

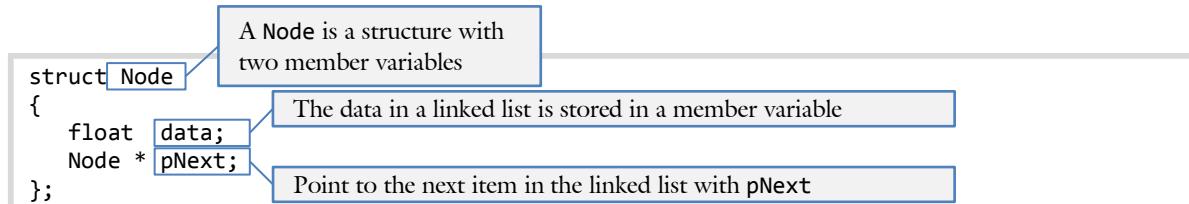
Allocation Indexing Changing

Because the nodes in a linked list do not need to be in a pre-specified order in memory, the only way to find the next item in the list is through following the pointer from the previous node. This means that linked lists do not support random access. Unlike arrays, the only way to find the i^{th} item in the list is to loop through the previous items. It is impossible to jump to that location using pointer arithmetic as we did with arrays.

If you wanted to delete or insert data in an array, the rest of the items in the array need to be shifted. This can be expensive. With linked lists, however, insertion and deletion can be done without affecting any of the other items in the list. This makes these operations much less expensive. Whenever a programming problem calls for a container that can handle efficient modification of data in the middle of a list, a linked list should be considered.

Syntax of linked lists

As previously defined, every linked list is a collection of nodes. A node can be any structure or class as long as it has at least two member variables: the data comprising what is stored in the linked list (commonly called `data`), and a pointer to the next node in the list (commonly called `pNext`).



By convention, we signify a `Node` is at the end a linked list by giving `pNext` the `NULL` address:

```
{
    Node linkedList;           // each linked list is started with a head
                               // node; the rest can be reached through pNext
    linkedList.data = 3.14159; // each node contains data
    linkedList.pNext = NULL;   // when pNext equals NULL, there are no nodes
}                           // after it in the list
```

If three items are to be added to a linked list, then three `Nodes` will need to be allocated.

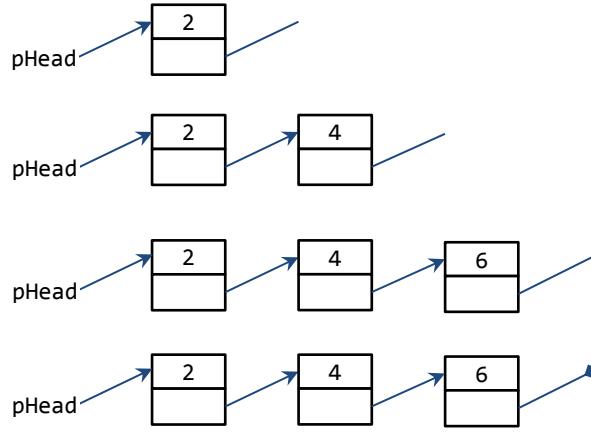
```
{
    Node * pHead;

    // add the first node
    pHead = new Node;
    pHead->data = 2.0;

    // add the second node
    pHead->pNext = new Node;
    pHead->pNext->data = 4.0;

    // add the third node
    pHead->pNext->pNext = new Node;
    pHead->pNext->pNext->data = 6.0;

    // mark the end of the list
    pHead->pNext->pNext->pNext = NULL;
}
```



Notice how we can access successive items on the linked list by using the `pNext->pNext` chain. The final item in the chain points to `NULL`. We signify this with the symbol.

Sam's Corner



Though the examples in this chapter work exclusively with linked lists of `floats`, it is possible to make a linked list using any data type. You can either substitute `float` for any other data type, or make the structure a template. The latter is probably your best bet; once you get your linked list code working, why would you ever want to change it or rewrite it?

Looping through a linked list

Up to this point, we have two standard FOR loops: the loop through an array using an index and the loop through a c-string using a pointer. The introduction of linked lists requires a third standard FOR loop.

Loop 1: arrays

When writing a FOR loop to traverse an array, we start at the 0 index and move onto the end:

```
for (int i = 0; i < size; i++)  
    cout << array[i] << endl;  
// must know the size  
// use [] to access members
```

When working with a vector, the loop is nearly identical:

```
for (int i = 0; i < array.size(); i++)  
    cout << array[i] << endl;  
// get size with array.size()  
// use [] operator
```

Loop 2: c-strings

With c-strings, we typically do not know the length of the string; we need to search for the null character ('\0'). Since the null character is zero or false, the second standard FOR loop is:

```
for (char * p = text; *p; p++)  
    cout << *p << endl;  
// stop when *p is false  
// access data with *p
```

Loop 3: linked lists

In order to loop though a linked list, we have a new standard FOR loop:

The beginning of the loop starts with a pointer, *p*, which will begin by pointing to the first node in the list, defined as *pHead*. Like the c-string loop, we use a pointer to walk through a linked list.

Recall that we signify the end of a linked list with the **NULL** address for *pNext*. Therefore, whenever our current node (*p*) is the **NULL** address, we stop the loop. Recall that the **NULL** address evaluates to **false**.

The loop will increment by updating *p* to point to the next item in the list. The next item in the list is reached with *p->pNext*. Thus, to advance *p* in the list use the following statement:
p = p->pNext

```
for (Node * p = pHead; p; p = p->pNext)  
    cout << p->data << endl;
```

Notice we also use the dereferencing arrow (->) when we output the value. This is because the *data* member variable holds the data of the list.



Sue's Tips

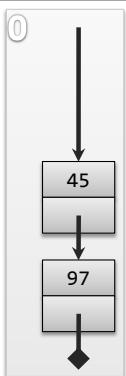
The linked list FOR loop is exactly the same regardless of data type of the *data* member variable. It is used so often that it is worth memorizing.

Example 4.4 – Loop through a linked list

Demo	This example will demonstrate how to loop through a linked list using the third standard FOR loop. Each Node will need to be both allocated and deleted.
Problem	Write a program to prompt the user for a value and put the value on the end of a linked list. Next display the values and empty the list when finished.
Solution	<p>We will use the standard <code>Node</code> definition for our linked list:</p> <pre style="border: 1px solid black; padding: 5px;">struct Node { float data; Node * pNext; };</pre> <p>To fill the linked list, it is necessary to allocate the nodes one-by-one. We will need a variable to keep track of the head of the list (<code>pHead</code>) as well as one to keep track of our current position in the linked list (<code>pCurrent</code>). Note how we need to carefully manage our <code>pNext</code> pointer on each <code>Node</code>.</p> <pre style="border: 1px solid black; padding: 5px;">void fill(Node * &pHead) // because pHead will receive a new value { // add the first node // when we allocate that node, we pHead = new Node; // need to pass pHead by-reference cout << "> "; cin >> pHead->data; // use the arrow operator -> to reach // the member variable from a pointer // add the second node pHead->pNext = new Node; cout << "> "; cin >> pHead->pNext->data; ... code removed for brevity ... }</pre> <p>The loop to display the contents of the linked list is the third standard FOR loop:</p> <pre style="border: 1px solid black; padding: 5px;">ostream & operator << (ostream & out, const Node * pHead) { for (const Node * p = pHead; p; p = p->pNext) out << p->data << ' '; return out; }</pre> <p>Note that we will need to delete the nodes when finished. This requires us to loop through the nodes, one by one, and delete each one. Again the third standard FOR loop is needed:</p> <pre style="border: 1px solid black; padding: 5px;">void empty(Node * &pHead) { for (Node * p = pHead; p; p = p->pNext) delete p; }</pre> <p>The complete solution is available at 4-4-loop.html or:</p> <p style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-right: 20px;">/home/cs165/examples/4-4-loop.cpp</p> 
See Also	

Adding an item to the head

To add a node to the head of a linked list, it is necessary to do three things: allocating a new `Node`, initializing the next `pNext` pointer in the newly allocated `Node`, and resetting `pHead` to point to the new `Node`.

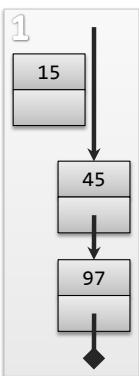


We start this process with an existing list. This list will have two nodes containing the values 45 and 97.

The function to change this list will take a pointer to the head node (`pHead`) as a parameter. However, since we are going to change what the head node points to, it must be pass by-reference:

```
void addToHead(Node * & pNode, float data)
{
    // insertion code will go here
}
```

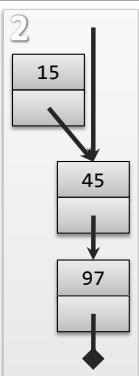
Thus we pass a pointer by-reference because we need to change `pHead`.



The first step of this process is to allocate a new `Node`. We will call this node `pNew`, keeping with the naming convention we have for all pointers. At allocation time, we will also initialize the value of our node (to 15 in this case):

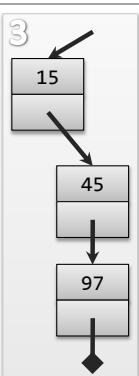
```
// 1. allocate and initialize create a new node
Node * pNew = new Node;           // allocate the node
pNew->data = data;               // set the data member variable
```

Recall that, since we are working with pointers to a structure, we need to use the arrow operator (`->`) to access the member variables of `pNew`.



The next step is to attach the new node to the existing list. We will do this by making the `pNext` member variable of `pNew` (containing the value 15) refer to the old head of the list (`pHead` containing 45). When we are done, both `pHead` and `pNew->pNext` will refer to the same node (node 45):

```
// 2. put pHead in line after the new node
pNew->pNext = pHead;           // set pNext, the old head
```



The final step is to make sure the linked list starts with our new node, not with the old head. We accomplish this by assigning `pHead` to `pNew` (15):

```
// 3. finally, pNew is the new head
pHead = pNew;                  // pNew is the new head. Note that
                                // pHead is a pointer by-reference
```

Recall that `pHead` is a parameter variable, not a local variable. This line of code changes the `pHead`. If `pHead` was pass-by-pointer alone, we would be only changing our copy of the head of the list, not the variable passed by the caller. Therefore, in order for the client's `pHead` to be changed, the pointer must be pass-by-reference.

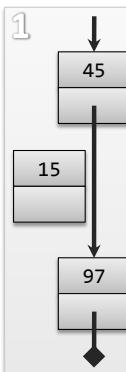
Adding an item to the middle

Adding an item to the middle of a linked list is very similar to adding to the start. Again, with linked lists, we don't have to worry about shifting all of the elements in the list when adding a new item. There will be three steps in this process: allocating a new node, initializing the next `pNext` pointer in the newly allocated node, and resetting `pHead` to point to the new node



We start this process with a list containing two items. We will wish to add our new node between item 45 and 97. The function to change this list will take a pointer to the node before the one we will add the new number (`pNode`) as a parameter as well as the data to be added (`data`):

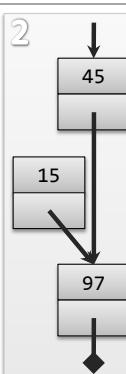
```
void add(Node * pNode, float data)
{
    // 0. if the list is empty, use addToHead()
    if (pNode == NULL)
        return;
}
```



In this process, the first step is to allocate the node using the `new` operator. Just like before, we will call this node `pNew`. At allocation time, we will also initialize the value of our node (to 15 in this case):

```
// 1. create a new node
Node * pNew = new Node;           // allocate the node
pNew->data = data;               // set the data member variable
```

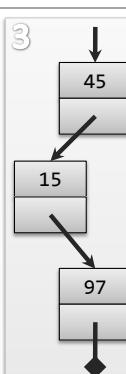
Again, since we are working with pointers to a structure, we need to use the arrow operator (`->`) to access the member variables of `pNew`.



The next step is to attach the new node to the existing list. Recall that we received from the caller as a parameter a pointer to the node before the one where we will add the new node (`pNode`). This node currently has the value 45.

Now we update the new node pointer (`pNew->pNext`) to the where `pNode->pNext` used to point. This means `pNew->pNext` and `pNode->pNext` both point the same place (97):

```
// 2. fix the pNext pointer from pNew to the spot after it in the list
pNew->pNext = pNode->pNext;
```



The final step is to make sure that the head now points to our new node. We do this by assigning `pNode->pNext` to `pNew`:

```
// 3. finally get pInsert->pNext to point to pNew
pNode->pNext = pNew;
```

Now that we are finished, notice how our linked list correctly connects the three nodes. The node containing 97 was never changed and nothing before 45 was ever changed.

Example 4.4 – Add to a linked list

Demo

This example will demonstrate how to add a node to a linked list. Both the cases where the number is at the head of the list and where the number is in the middle of the linked list.

Problem

Write a program to add a node to a linked list. The user should be able to specify where the item will go in the list (user input is underline):

```
Enter four numbers to put in the linked list
Data: 90
      90
Data: 100
Index after which 100 will be inserted: 0
      100 90
Data: 95
Index after which 95 will be inserted: 1
      100 95 90
Data: 85
Index after which 85 will be inserted: 3
      100 95 90 85
```

Solution

First we need a function to add a number to the beginning of a linked list:

```
void addToHead(Node * & pHead, float data)
{
    // 1. allocate and initialize create a new node
    Node * pNew = new Node;           // allocate the node
    pNew->data = data;              // set the data member variable

    // 2. put pHead in line after the new node
    pNew->pNext = pHead;           // set pNext, the old head of the list

    // 3. finally, pNew is the new head
    pHead = pNew;                  // pNew is the new head. Note that
                                    // pHead is a pointer by-reference
}
```

Next we need a function to add node containing data to the linked list after pNode:

```
void add(Node * pNode, float data)
{
    // 0. if the list is empty, use addToHead()
    if (pNode == NULL)
        return;

    // 1. create a new node
    Node * pNew = new Node;           // allocate the node
    pNew->data = data;              // set the data member variable

    // 2. fix the pNext pointer from pNew to the spot after it in the list
    pNew->pNext = pNode->pNext;

    // 3. finally get pNode->pNext to point to pNew instead of the old node
    pNode->pNext = pNew;
}
```

See Also

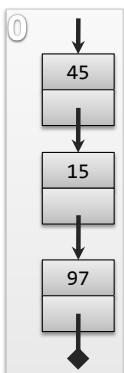
The complete solution is available at [4-4-add.html](#) or:

```
/home/cs165/examples/4-4-add.cpp
```



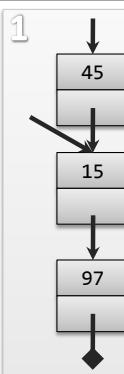
Remove from the head of the list

Just like adding an item to a list, the process of removing an item can be described in three steps; saving the next item, deleting the old item, and making the head point to the next item. Again, we will use the structure `Node`.



In this example, we start with a list that contains three nodes, containing values 45, 15, and 97. We will remove the first item, 45. This function will take the pointer to the head node (`pHead`) and, because our goal is to change `pHead`, we will pass the pointer by-reference:

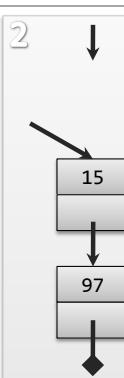
```
void removeFromHead(Node * & pHead)
{
    // 0. do nothing if there is nothing to do
    if (pHead == NULL)
        return;
}
```



In this process, the first step is to save the location of the next item in the list. This is done by creating a new pointer, `pSave`, and initialize it as the value of `pHead->pNext`:

```
// 1. remember the second item
Node * pSave = pHead->pNext;
```

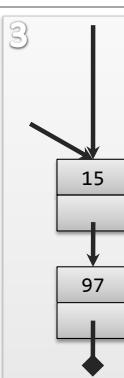
This step is important because we will need to know where the new `pHead` is going to point to once we've deleted the `pHead` node.



Now we need to delete the first item in the list (`pHead`), using the `delete` operator:

```
// 2. delete the first item
delete pHead;
```

Note that at this point, our linked list is broken. There is nothing connecting `pHead` (which currently points to a deleted node) to the next item in the list (containing 15). We must fix this pointer to complete the linked list.



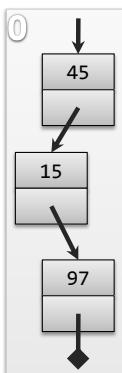
The final step is to make sure that the head now points to the next node. We saved the location of this node using the pointer `pSave`, so all we have to do is assign `pHead` to `pSave` and the next node becomes the head of the list. The only way this can work is if we pass `pHead` to our function by-reference:

```
// 3. make head point to the second item
pHead = pSave;
```

When the function exits, `pSave` will fall out of scope leaving our properly formed linked list starting with `pHead`.

Remove from the middle of the list

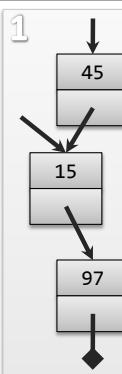
Removing an item from the middle of a list is convenient because no items in the list need to be shifted in any way. This process can also be reduced into three steps; saving the location of the item, updating the head pointer, and deleting the node.



We will start with a list that contains three nodes. We will remove one node from the list, as specified by a passed parameter.

One parameters will be passed to our `remove()` function: a pointer to the node immediately before the one we wish to remove (`pPrevious`) with the value 15:

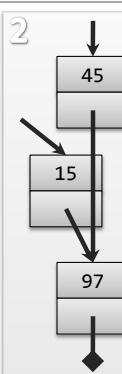
```
void remove(Node * pPrevious)
{
    // if the list is empty or there is just one item, use removeFromHead
    if (pPrevious == NULL || pPrevious->pNext == NULL)
        return;
}
```



We need to save the location of item we plan to delete. We will create a pointer called `pSave` and assign it to the location of `pHead->pNext`:

```
// 1. save the location of the item before the one we are deleting
Node * pSave = pPrevious->pNext;
```

Note that two pointers will be pointing to the node containing 15: `pDelete` and `pPrevious->pNext`.



The next step is to update the pointer of the head (`pPrevious->pNext`) to the item following the saved node (`pSave->pNext`):

```
// 2. update the pointer of pPrevious->pNext
pPrevious->pNext = pDelete->pNext;
```

At this point, the linked list has forgotten about the node we are deleting. If we start from `pHead`, we will never encounter `pDelete` (with the value 15); it is no longer part of the chain. However, we still need to delete the node.



Finally, we will delete the saved node using the `delete` operator:

```
// 3. delete the pointer at pDelete
delete pDelete;
```

Notice with this and all of the previous examples we didn't have to shift any of the nodes to add or delete items. We simply needed to change the pointers.

Example 4.4 – Remove from a linked list

Demo

This example will demonstrate how to remove items from a linked list. It will handle both the special case of removing the head of a list as well as the general case of removing from the middle of a list.

Problem

Write a program to remove a node from a linked list. The user should be able to specify the index of the number to be removed from the list (user input is underline):

```
Enter five numbers to put in the linked list
#1: 101
#2: 102
#3: 103
#4: 104
#5: 105
before: 101 102 103 104 105
Which item do you want to remove? 3
after: 101 102 104 105
```

Solution

The special case is to remove from the beginning of the list. To handle this, it is necessary for the `pHead` pointer to be passed by-reference.

```
void removeFromHead(Node * & pHead)
{
    // 0. do nothing if there is nothing to do
    if (pHead == NULL)
        return;

    // 1. remember the second item
    Node * pSave = pHead->pNext;

    // 2. delete the head
    delete pHead;

    // 3. make the head point to the second item
    pHead = pSave;
}
```

The general case is to remove from the middle of the list. We must first detect if the passed node is `NULL` or if the next node is `NULL`. In either of these cases, we must call `removeFromHead()`.

```
void remove(Node * pPrevious)
{
    // if the list is empty or there is just one item, use removeFromHead
    if (pPrevious == NULL || pPrevious->pNext == NULL)
        return;

    // 1. save the location of the item before the one we are deleting
    Node * pDelete = pPrevious->pNext;

    // 2. update the pointer of pPrevious->pNext
    pPrevious->pNext = pDelete->pNext;

    // 3. delete the pointer at pDelete
    delete pDelete;
}
```

See Also

The complete solution is available at [4-4-remove.html](#) or:

```
/home/cs165/examples/4-4-remove.cpp
```



Review 1

Recall from Unit 3 that only member functions can be polymorphic. Recall from Unit 2 that the insertion operator (`<<`) cannot be a member function. The combination of these makes it impossible to create a polymorphic insertion operator.

Define a template version of the insertion operator that works with any data type defining a public `display()` method.

```
class Date
{
public:
    void display(ostream & out) const;
... code removed for brevity ...
};
```

Note that, if the `display()` function in the passed object is virtual, we can have the equivalent of a polymorphic insertion operator.

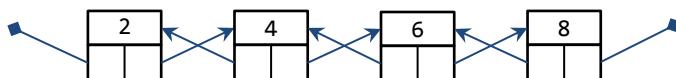
Please see page 167 and 287 for a hint.

Problem 2-5

A singly linked list is a data-structure where each node is connected to the next node in the list through a `pNext` pointer:



A doubly linked list is a data-structure where each node is connected to the next and the previous node through a pointer:



2. Write the UML for a doubly linked list `Node` structure.
3. Create a structure definition for `Node`. Make sure the structure is a template.
4. Using the `addToHead()` function as your guide, write the code to add onto the beginning of a doubly linked list.
5. Using the `remove()` function as your guide, write the code to remove from the middle of a doubly linked list. Hint: you will pass as a parameter the node to be removed.

Please see page 311 for a hint.

4.5 Iterators

Sue gets confused by the many different versions of the FOR loop: for arrays, for c-strings, and linked lists. Even though both FOR loops do essentially the same thing, the specifics of the loops are completely different. If only there was just one FOR loop that she could use for all types of containers.

Objectives

By the end of this chapter, you will be able to:

- Provide a definition of an iterator and explain why they are useful programming constructs
- Recite the syntax for an iterator through a string object, a vector, and a linked list
- Use an iterator to solve programming problems involving loops

Prerequisites

Before reading this chapter, please make sure you are able to:

- Instantiate a `vector` of `floats` and loop through all the items in the `vector` (Review chapter)
- Be able to overload the common operators as both members and non-members (Chapter 2.6 – 2.8)
- Recite the syntax for the three standard FOR loops: the loop through an array, through a c-string, and the loop through a linked list (Chapter 4.4)

What are iterators and why you should care

An **iterator** is a tool used to provide a standard way to iterate or traverse through a collection of values. They are designed to work the same regardless of the data type stored in the collection and regardless of how the collection is stored in memory. For example, an iterator used to loop through the items in a vector should work the same as an iterator used to loop through the items in a linked list.

Iterators are a key component to encapsulation. Recall from Unit 2 that one of the purposes of encapsulation is to shield the user of a class from the specific implementation details used to build the class. In other words, a programmer instantiating an object should not need to know how data is stored within the class from which the object was defined. Iterators are designed to help with this. Since linked lists are stored in memory so differently than arrays, it is necessary to use a completely different loop to iterate through the items in a linked list than it is to loop through the items in an array.

Arrays	C-Strings	Linked Lists
<pre>for (int i = 0; i < num; i++) cout << array[i];</pre>	<pre>for (char * p = text; *p; p++) cout << *p;</pre>	<pre>for (Node * p = pHead; p; p = p->pNext) cout << p->data;</pre>

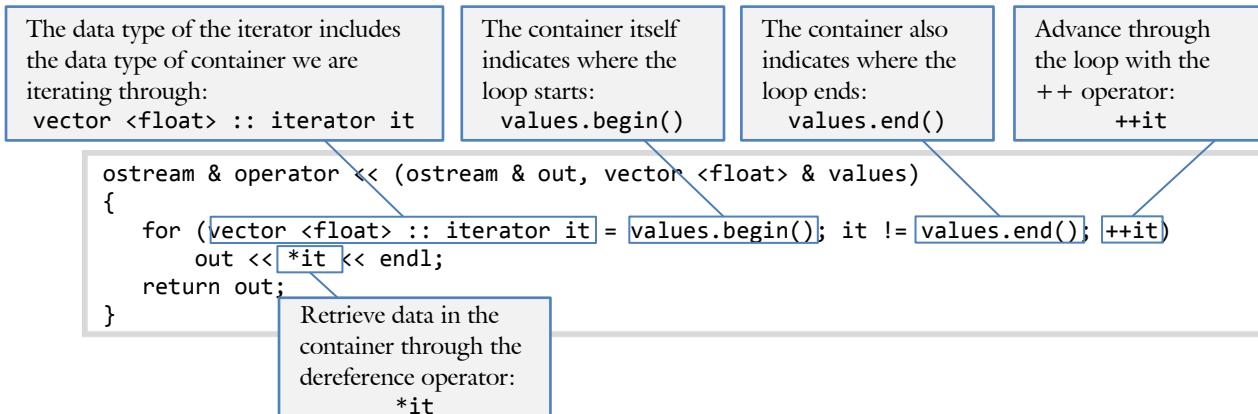
Iterators avoid this necessity by providing a standard way to iterate through all collections of values:

```
for (string :: iterator it = text.begin(); it != text.end(); ++it)
    cout << *it;
```

Syntax for iterators

For every container class, there is an associated iterator. In other words, we use a different iterator for a `string` object, a `vector`, and a `list` (the vector and the list will be discussed in Chapter 4.6).

Consider, for example, a `vector` of floating point numbers containing the values {1.2, 3.4, 5.6}. A function displaying the values of the `vector` using an iterator is the following:



There are several things to observe about the syntax for iterators: the iterator data type, the process of initializing an iterator, the process of verifying the iterator is not beyond the end of the data, the process of advancing the iterator, and finally the process of retrieving data from the container.

Declaring an iterator

The syntax for an iterator, by convention, is the syntax of the data type through which the iterator is to loop followed by “`:: iterator`”. Consider the following iterator declarations:

Container	Iterator declaration
string object	<code>string :: iterator it;</code>
vector of integer	<code>vector <int> :: iterator it;</code>
vector of Date	<code>vector <Date> :: iterator it;</code>
list of double	<code>list <double> :: iterator it;</code>

Initializing an iterator

The container tells the iterator where to start the loop. The container does this, by convention, with a method called `begin()`. The iterator then provides the assignment operator to accept the return value from the `begin()` method.

End condition

Every FOR loop has a Boolean expression indicating when to leave the loop. The container indicates the end of the data through a method called `end()`. The iterator then provides the absolute comparison operators (`==` and `!=`) and often the relative comparison operators (`<`, `<=`, `>`, `>=`) as well. Each of these operators compares the return value from `end()` with the iterator’s notion of the progress through the loop.

Advancing the iterator

By convention, the iterator advances through the container with the `++` operator. Both the prefix (`++it`) and the postfix (`it++`) versions of the `++` operator are defined in the iterator.

Note that advancing through an array (`i++`), through a c-string (`p++`), and through a linked list (`p = p->pNext`) are quite different. The iterator needs to know these differences in order to advance properly. Thus the iterator needs to be aware of the internal data representation of the container it is designed to iterate through. This is why there is a special iterator designed for each container.

Accessing data from the container

By convention, we access data from the container by using the dereference operator (`*`) on the iterator. Recall from Chapter 2.8 that the dereference operator is among the operators that can be overloaded as a member function.

Reverse iterators

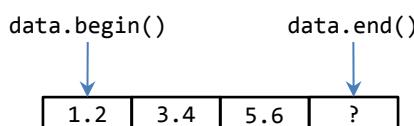
It turns out that traversing a list backwards requires a different iterator than traversing a list forwards. In order to understand why this is the case, consider a `vector` with the following values {1.2, 3.4, 5.6}:

```
{  
    vector <float> data;  
    data.push_back(1.2);  
    data.push_back(3.4);  
    data.push_back(5.6);  
}
```

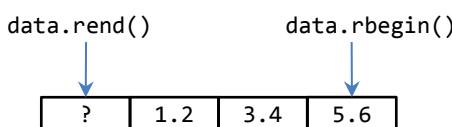
The iterator to display these values would be:

```
{  
    for (vector <float> :: iterator it = data.begin(); it != data.end(); it++)  
        cout << *it << endl;  
}
```

Notice how `data.begin()` points to the beginning of the list but `data.end()` must point to one item off the end of the list:



For a reverse loop to work, a different set of `begin()` and `end()` methods will be needed from the container. These are called the `rbegin()` and `rend()`, respectively.



The loop to walk through a vector backwards is then:

```
{  
    for (vector <float> :: reverse_iterator it = data.rbegin(); it != data.rend(); ++it)  
        cout << *it << endl;  
}
```

Observe how the name of the iterator is “`reverse_iterator`” rather than “`iterator`”.

Example 4.5 – Vector iterator

Demo	This example will demonstrate how to use an iterator to traverse a <code>vector</code> . Both forward and reverse iterators will be demonstrated.
Problem	Write a program to prompt the user for an unspecified number of words. Continue prompting until the user selects “Done!” Next display the list both forward and backwards.
Solution	<pre>{\n vector <string> words;\n\n // fill the vector of words;\n cout << "Please enter some words. Specify \"Done!\" when done\\n";\n string text;\n cin >> text;\n while (text != "Done!")\n {\n words.push_back(text);\n cin >> text;\n }\n\n // loop through the list\n cout << "The list forwards:\\n";\n for (vector <string> :: iterator it = words.begin();\n it < words.end();\n it++)\n cout << "\\t" << *it << endl;\n\n // loop through the list backwards\n cout << "The list backwards:\\n";\n for (vector <string> :: reverse_iterator it = words.rbegin();\n it != words.rend();\n ++it)\n cout << "\\t" << *it << endl;\n}</pre>
Challenge	As a challenge, can you change the above program so it works with a <code>vector</code> of <code>Date</code> objects from the <code>Date</code> class built in Unit 2?
See Also	The complete solution is available at 4-5-vectorIt.html or: <code>/home/cs165/examples/4-5-vectorIt.cpp</code>

Example 4.5 – List iterator

Demo

This example will demonstrate how to use an iterator to traverse a list. A list is a linked list container that we will learn about more in Chapter 4.6.

Problem

Write a program to prompt the user for five numbers and put them in a list. Next, remove the even numbers so only the odd ones remain. Finally, display the resulting list.

Solution

```
int main()
{
    // declare a list object
    list <int> data;

    // prompt the user for five numbers
    cout << "Please enter five numbers\n";
    for (int i = 0; i < 5; i++)
    {
        int value;
        cout << " > ";
        cin >> value;
        data.push_back(value);
    }

    // delete the even numbers
    for (list <int> :: iterator it = data.begin(); it != data.end(); /* blank */ )
        if (*it % 2 == 0)                      // determine if the number is even
            it = data.erase(it);              // update it when we erase the number
        else
            ++it; // we must increment it only if we don't erase an element

    // display the numbers
    cout << "The list with even numbers removed\n";
    for (list <int> :: iterator it = data.begin(); it != data.end(); ++it)
        cout << *it << ' ';
    cout << endl;

    return 0;
}
```

The trickiest part here is where the even numbers are removed. Notice how we remove an item from the list with the `erase()` method. The `erase()` method takes an iterator as a parameter (indicating which node of the linked list to remove) and also returns an iterator. Returning an iterator is an important step because, after the node is erased, the iterator will point to an invalid location in memory. By updating the `it` iterator, we are compensating for the removed node.

Challenge

As a challenge, can you change the above program so it works with a `list of string` objects? Next, rather than removing the even numbers, instead remove the words beginning with a capital letter.

See Also

The complete solution is available at [4-5-listIterator.html](#) or:

/home/cs165/examples/4-5-listIterator.cpp



Building an Iterator

It is not difficult to build an iterator for a given container; one only needs to abstract the components of the non-iterator version of the loop. First, consider the loop to iterate through a string object:

```
for (char * p = text.buffer; p != text.buffer + text.size; p++)
    cout << *p << endl;
```

We would like this loop to work with an iterator:

```
for (StringIterator it = text.begin(); it != text.end(); it++)
    cout << *it << endl;
```

The first question is: what will be the member variable for the `StringIterator` class? The obvious answer is: a pointer to a character, the same data type used for the non-iterator version of the FOR loop.

Start the loop

With our storage strategy found, we can define the `begin()` function in our `String` class:

```
class String
{
    ... code removed for brevity ...
    char * begin() { return buffer; }
    ... code removed for brevity ...
private:
    char * buffer;
    int size;
};
```

Now we can define our `StringIterator` class to accept this data:

```
class StringIterator
{
public:
    StringIterator(char * p) : p(p) { }
    StringIterator & operator = (char * p) { this->p = p; }
private:
    char * p;
};
```

End the loop

How do we know we are at the end of the loop? The answer is when the pointer is referring to the null character. Therefore, our `String::end()` method should return a pointer to the null character:

```
class String
{
    ... code removed for brevity ...
    char * end() { return buffer + size; }
    ... code removed for brevity ...
};
```

Now we can define the not-equals operator in our `StringIterator` class to test this condition:

```
class StringIterator
{
    ... code removed for brevity ...
    bool operator != (const char * p) const { return this->p != p; }
    ... code removed for brevity ...
};
```

Advance

With the start of the loop defined and the end of the loop defined, we next need to define how advance our iterator. We do not need to make any modifications to the String class to do this, StringIterator should be able to do it itself.

```
class StringIterator
{
... code removed for brevity ...
    StringIterator & operator ++ () { p++; return *this; }
... code removed for brevity ...
};
```

Retrieve data

The final step in the iterator definition process is to fetch the data referred to by the iterator. We use the dereference operator to do that. Since our member variable is a pointer to the data, the function is trivial:

```
class StringIterator
{
... code removed for brevity ...
    char & operator * () { return *p; }
... code removed for brevity ...
};
```

Observe how we return the character by-reference. This is important because the client of the iterator class needs to not only retrieve data from the string, but also set data to the string:

```
{
    StringIterator it = text.begin();
    *it = 'z';                                // use * to set data in text
    cout << *it << endl;                      // use * to retrieve data from text
}
```

Using StringIterator

Non-Iterator Loop

A FOR loop for a string class:

```
{
    char * p;

    for (p = text.buffer;
        p != text.buffer + text.size;
        ++p)
        cout << *p;
}
```

Notice how the loop here is different than the standard FOR loop for c-strings. Here we are taking advantage of the fact that we know the buffer size. Thus with a `size` member variable, we do not need to check for the null character.

Iterator Loop

The standard iterator FOR loop:

```
{
    StringIterator it;

    for (it = text.begin();
        it != text.end();
        ++it)
        cout << *it;
}
```

Since `text.begin()` returns `text.buffer`, the first part of the loop is identical to the non-iterator version. The same can be said for the `text.end()` part, the increment part, and the dereference part. In other words, the two loops are the same.

Example 4.5 – Container class

Demo This example will demonstrate how to create an iterator for a class designed to hold data. We call this a “container.” There are several containers in the Standard Template Library (STL). This particular container is generic: it can handle any data type.

Problem Write a class similar to the `vector` we created for Chapter 4.3 (`/home/cs165/examples/4-3-vector.cpp`). Additionally, create an iterator called `ContainerIterator`.

We will need to add two methods to `vector` to start and end the iterator:

```
class Container
... code removed for brevity ...
    ContainerIterator<T> begin() { return ContainerIterator<T> (data); }
    ContainerIterator<T> end()   { return ContainerIterator<T> (data + size); }
... code removed for brevity ...
};
```

Additionally, a new class called `ContainerIterator<T>` will need to be created:

```
template <class T>
class ContainerIterator
{
public:
    ContainerIterator()      : p(0x00000000) {}
    ContainerIterator(T * p) : p(p)           {}

    ContainerIterator <T> & operator = (const ContainerIterator & rhs)
    {
        this->p = rhs.p;
        return *this;
    }

    bool operator != (const ContainerIterator & rhs) const {return rhs.p!=this->p;}
    bool operator == (const ContainerIterator & rhs) const {return rhs.p==this->p;}

    T & operator * () { return *p; }

    ContainerIterator <T> & operator ++ () { p++; return *this; }

private:
    T * p;
};
```

Note how the iterator works much like that for the `String` class described earlier in the chapter.

Challenge As a challenge, can you also create another class representing a constant iterator? The only differences will be that it will have a “`const T * p;`” as a member variable and the dereference operator will return a `T` by-value instead of by-reference.

Another challenge is to create a reverse iterator. In this case, create a `rbegin()` and `rend()` method in the `Container` class, and define the `--` operator in `ContainerIterator`.

See Also The complete solution is available at [4-5-container.html](#) or:

```
/home/cs165/examples/4-5-container.cpp
```

Problem 1

Write the code to return the sum of values in a `vector` of integers. The function (`computeSum()`) will take a `vector` of integers as a by-reference parameter and return an integer. Please use iterators to solve this problem.

Please see page 317 for a hint.

Problem 2

Write the code to return the sum of values in a `list` of integers. The function (`computeSum()`) will take a `list` of integers as a by-reference parameter and return an integer. Please use iterators to solve this problem.

Please see page 318 for a hint.

Challenge 3

Consider the following `String` class:

```
class String
{
    public:
        String();
        char operator [] (int index);
        char *begin() { return text; }
        char *end()   { return text + length; }

    ... code removed for brevity ...
    private:
        char text[256];
        int length;
};
```

Define an iterator for the above `String` class that behaves as one would expect:

```
{
    String text;
    ... code remove for brevity ...

    for (StrIter it = text.begin(); it != text.end(); ++it)
        cout << *it;
}
```

Note that the `StrIter` class will need to support the following operators: not equal (`!=`), prefix increment (`++`), assignment (`=`), and dereference (`*`).

Please define the `StrIter` class.

Please see page 321 for a hint.

Challenge 4

Consider the following `Node` class:

```
class Node
{
public:
    Node();
    ~Node();
    Node * getNext();
    int   getValue() const { return value; }
    Node * begin();
    Node * end();
private:
    int   value;
    Node * pNext;
};
```

Define an iterator for the above `Node` class that behaves as one would expect:

```
{
    Node node;
    ... code remove for brevity ...

    for (NodeIter it = node.begin(); it != node.end(); ++it)
        cout << *it;
}
```

Note that the `NodeIter` class will need to support the following operators: not equal (`!=`), prefix increment (`++`), assignment (`=`), and dereference (`*`).

Define the `NodeIter` class.

4.6 Standard Template Library

Sam is working with a program that manipulates a large amount of data. In anticipation of this program, Sam wrote a special array class that can grow to accomodate data-sets of any size. However, as he starts the program, Sam realizes that a linked list would be a more appropriate data-structure. He is about to start writing a linked list class when Sue stops by the lab and introduces him to the standard template library.

Objectives

By the end of this chapter, you will be able to:

- Explain the purpose of the standard template library
- Use the `vector`, `list`, and `map` classes in a program
- Describe situations where the vector, list , and map classes would be useful tools to solving a problem

Prerequisites

Before reading this chapter, please make sure you are able to:

- Define and use a class template (Chapter 4.3)
- Define and use a linked list (Chapter 4.4)
- Define and use an iterator to the vector class (Chapter 4.5)

What is the STL and why you should care

The **Standard Template Library** (STL) is a collection of tools designed to make common programming tasks easier. The STL, in combination with the stream classes (`istream` and `ostream`) and the `string` class, are instrumental in most programming tasks. This is particularly relevant because C++ has no built-in way to interact with the console (which is why `cin` and `cout` are so important) and has only primitive data-structures.

The name “Standard Template Library” does a good job of explaining its purpose. First, it is “standard,” meaning it is available on all installations of C++. You can always depend on the STL regardless of the platform you are developing on. Second, it is based on “templates.” All the data-structures in the STL are implemented using templates so they can work with any data type. Finally, they are a “library.” Not being built into the C++ language, we must explicitly include them in our code through the `#include` mechanism.

There are many **containers** (data-structures designed to hold or contain data) in the STL. The most commonly used are the `vector` (an array-like data-structure), `list` (a linked list), and the `map` (a data-structure enabling a simple database).



Sue's Tips

It is wise for every programmer to become acquainted with the STL and similar libraries which are part of other programming languages. They are like “power-ups” on classic video games; they make you a more powerful programmer. By leveraging the work done by others, it is possible to implement more complex programs, quicker, and with fewer bugs than it would be to implement these data-structures on your own. As Sir Isaac Newton once said “If I have seen further it is by standing on ye sholders *sic* of giants.”

Vector

The `vector` container is a data-structure similar to an array. However, there are three fundamental differences between a vector and an array. First, a `vector` object can grow to accommodate any amount of data whereas the size of an array is fixed. Second, whereas arrays can be either a local variable or created with the `new` statement, `vector` data is always dynamically allocated. Finally, the `vector` class supports iterators whereas arrays do not.

When to use

A `vector` should be considered for a given problem when:

- **Unknown size:** When the size of the data-set is unknown at compile time, a `vector` should be considered. If the size of the data-set is constant and known, use an array
- **Random access:** When it is necessary to access items in the middle of the array, consider a `vector`. Items can be accessed quickly because `vectors` ensure items reside in a continuous block of memory
- **Add/Remove from end:** When data is commonly added or removed only from the end of the data-set, consider using a `vector`. Otherwise, consider using a `list`

Syntax

In order to declare a `vector` object, it is first necessary to include the `vector` library:

```
#include <vector>
```

Note that the `vector` class is in the standard namespace so either include “`using namespace std;`” or refer to a `vector` by its full name “`std::vector`”.

To declare a `vector` object, it is necessary to indicate the data type of the data being contained. Since `vector` is a class template, the syntax should be familiar:

```
{  
    vector <double> data;           // a vector of doubles  
}
```

Methods

The most commonly used methods supported by the `vector` class are the following:

Method	Use
<code>operator []</code>	Random access is achieved through the square-bracket operator just like an array. This will return the item by-reference so it can be used both as a getter and as a setter
<code>push_back()</code>	Items are added onto the end of a <code>vector</code> through the <code>push_back()</code> method. If the size exceeds the capacity of the <code>vector</code> , the capacity will be increased
<code>size()</code>	Returns the number of items currently in the <code>vector</code>
<code>capacity()</code>	Returns the size of allocated storage capacity currently in the <code>vector</code>
<code>clear()</code>	Removes all items currently in the <code>vector</code>

Discussion

One interesting thing about `vectors` (and the `string` class) is how the capacity grows to accommodate user data. When a `vector` is first declared, the capacity is zero. After the initial allocation, every time the client attempts to put more data into the `vector` than is allowed by the current capacity, the capacity is doubled.

Example 4.6 – Vector of floats

Demo	This example will demonstrate how to do the common manipulations with a <code>vector</code> .
Problem	<p>Write a program to prompt the user for five numbers, put the numbers in a <code>vector</code>, and then display the results on the screen.</p> <pre>Please enter 5 values: # 1: 124 # 2: 165 # 3: 235 # 4: 246 # 5: 364 The values are: 124, 165, 235, 246, 364 size: 0 capacity: 8</pre>
Solution	<pre>{ vector <float> data; // declaration // fill the vector cout << "Please enter 5 values:\n"; for (int i = 0; i < 5; i++) { float value; cout << "\t# " << i + 1 << ": "; cin >> value; data.push_back(value); // add with push-back } // display the list cout << "The values are: "; for (int i = 0; i < data.size(); i++) // get size with size() cout << data[i] // random access with [] << (i == 4 ? "\n" : ", "); // the destructor usually erases the list data.clear(); // erase everything cout << "size: " // retrieve the size with size() << data.size() << endl; cout << "capacity: " // retrieve the capacity of the << data.capacity() << endl; // vector with capacity() }</pre>
Challenge	As a challenge, can you change the FOR loop to use an iterator rather than an index?
See Also	The complete solution is available at 4-6-vector.html or: <pre>/home/cs165/examples/4-6-vector.cpp</pre>

Example 4.6 – Vector growth

Demo

This example will demonstrate how the `vector` capacity grows as more data is entered. Observe how the capacity is doubled once the size reaches the current capacity.

Problem

Write a program to display how the capacity increases as data is placed on a `vector`:

```
When first created, there are 0 items in the vector
Size    Capacity
 1      1
 2      2      Reallocation from 1 to 2
 3      4      Reallocation from 2 to 4
 4      4
 5      8      Reallocation from 4 to 8
 6      8
 7      8
 8      8
 9      16     Reallocation from 8 to 16
10     16
11     16
12     16
13     16
14     16
15     16
16     16
17     32     Reallocation from 16 to 32
18     32
```

Solution

This demo will first instantiate a `vector` object then fill the container with numbers. With each number entered, the size and capacity will be displayed.

```
{
    vector <int> list;

    // The initial size of a vector
    cout << "When first created, there are "
        << list.capacity()
        << " items in the vector\n";

    // Now we will add items to the list, one at a time
    cout << "Size\tCapacity\n";
    for (int i = 0; i < 18; i++)
    {
        list.push_back(i);
        cout << " " << list.size()
            << "\t " << list.capacity() << endl;
    }
}
```

See Also

The complete solution is available at [4-6-vectorGrowth.html](#) or:

```
/home/cs165/examples/4-6-vectorGrowth.cpp
```



List

The `list` container is a linked list data-structure. It is therefore useful in all situations where adding to the middle of the list or removing from the middle of the list is common.

Use

A `list` should be considered for a given problem when:

- **Unknown size:** When the size of the data-set is unknown at compile time, a `list` should be considered. If the size of the data-set is constant and known, use an array
- **Sequential access:** When it is only necessary to access members of the data-set sequentially (either forwards or backwards), consider using the `list`. If random access is required, then consider using the `vector` class or an array
- **Add/Remove from middle:** When data is commonly added or removed from the middle of the data-set, consider using a `list`. Since no shifting is required, this operation is very efficient with the `list` class. If adding and removing only occurs at the end of the data-set, consider using a `vector`

Syntax

The library containing the `list` class is called `list`:

```
#include <list>
```

To declare an object of type `list`, it is first necessary to specify the data type of each item in the data-set:

```
{
    list <double> data;           // a linked list of doubles
}
```

Methods

The most commonly used methods supported by the `list` class are the following:

Method	Use
<code>push_back()</code>	Items are added onto the end of a <code>list</code> through the <code>push_back()</code> method. It is also common to put items at the head of the list with <code>push_front()</code>
<code>insert()</code>	Insert an item in the middle of a <code>list</code> . The <code>insert()</code> method takes an iterator parameter indicating the location of the inserted node. It returns an iterator so the loop can be continued from the current location
<code>erase()</code>	Remove an item from the middle of a <code>list</code> . The <code>erase()</code> method takes an iterator parameter indicating the location of the node to be removed. It returns an iterator so the loop can be continued from the current location
<code>size()</code>	Returns the number of items currently in the <code>list</code>
<code>clear()</code>	Removes all items currently in the <code>list</code>

Variations

Another variant of the `list` container is the `slist`. The `list` container is a doubly linked list, meaning it is possible to use both forward and reverse iterators. Thus the nodes in the `list` container have both a `pNext` and `pPrev` pointers. The `slist` container can only be traversed in the forward direction. Its nodes only have the `pNext` pointer.

Example 4.6 – List of words

Demo

This example will demonstrate how to create a `list` object, add items to the `list`, and display the results both forwards and backwards.

Problem

Write a program to add two words to the end of a list, two words to the beginning, and two words to the middle. Then display the list forwards and backwards:

```
The list displayed forwards
    one    two    three    four    five    six
The list displayed backwards
    six    five    four    three    two    one
```

Solution

```
{
    // declare a list of strings
    list <string> data;

    // add items on to the end of the list with push_back
    data.push_back(string("five"));
    data.push_back(string("six"));

    // add items to the head of the list
    data.push_front(string("two"));
    data.push_front(string("one"));

    // add items to the middle
    list <string> :: iterator it = data.begin();
    it++; // skip the first
    it++; // skip the second
    it = data.insert(it, string("four"));
    it = data.insert(it, string("three"));

    // display the list forwards
    cout << "The list displayed forwards\n";
    for (it = data.begin(); it != data.end(); it++)
        cout << '\t' << *it << ' ';
    cout << endl;

    // display the list backwards
    cout << "The list displayed backwards\n";
    list <string> :: reverse_iterator rit;
    for (rit = data.rbegin(); rit != data.rend(); rit++)
        cout << '\t' << *rit << ' ';
    cout << endl;
}
```

Challenge

As a challenge, can you add the code necessary to remove items from the `list`? Specifically, can you remove the word “three” from the `list` after it is displayed forwards?

See Also

The complete solution is available at [4-6-listWords.html](#) or:

```
/home/cs165/examples/4-6-listWords.cpp
```

Map

The `map` is a data-structure designed to store and retrieve items in a data-set based on a key. Therefore the client provides two pieces of information when adding an item to a `map`: the data to be stored and the key with which the data will be retrieved. Thus in many ways the `map` data-structure works like a simple database.

When to use

A `map` should be considered for a given problem when:

- **Non-integer key:** The key of an array or a `vector` is always an integer: the index. The key of a `map` can be any data type. If a non-integer key is used, consider using a `map`
- **Non-continuous key:** Even if the key is an integer, an array or `vector` may not be practical. Consider an application where a user's name is associated with the telephone number. Since there are ten million combinations in a seven digit telephone number, one would need an array of ten million to associate phone numbers with user names. With a `map`, it is possible to associate names with numbers without allocating ten million values

Syntax

In order to declare a `map` object, it is first necessary to include the `map` library:

```
#include <map>
```

To declare a `map` object, it is necessary to indicate the data type of the data being contained as well as the data type of the key. To associate phone numbers (integers) with names (`string` objects), we would need:

```
{
    map <int, string> data;           // a map with an integer key and string data
}
```

Methods

The most commonly used methods supported by the `map` class are the following:

Method	Use
<code>operator []</code>	Random access is achieved with a <code>vector</code> through the square-bracket operator. Note that the parameter inside the square brackets <code>[]</code> is the key type, not necessarily an integer. Note that items can either be retrieved or added with the square bracket operator.
<code>erase()</code>	Removes an item at a given location. That location can be provided either by an iterator or by a key.
<code>clear()</code>	Removes all items currently in the <code>map</code> .

Example 4.6 – Phone numbers

Demo

This example will demonstrate how to declare, fill, and retrieve items out of a `map`.

Problem

Write a program to prompt the user for several phone numbers and names. The program will then store the phone number in memory using the user name as a key. Finally, the program will retrieve the phone number based on the user name. Use input is underline:

```
Please enter five phone numbers and names:  
> 496-7608 Br. Helfrich  
> 496-7606 Br. Ercanbrack  
> x7601 Sister Price  
> 911 Emergency  
> (208)496-1111 President Gilbert  
Enter someone's name: President Gilbert  
That phone number of President Gilbert is (208)496-1111
```

Solution

```
{  
    // declare a map between phone numbers and names  
    map <string, string> phoneNumbers;           // a map with a string index  
    string number;                                //      and a string data-item  
    string name;  
  
    // fill the list  
    cout << "Please enter five phone numbers and names:\n";  
    for (int i = 0; i < 5; i++)  
    {  
        cout << "> ";  
        cin  >> number;  
        cin.ignore();  
        getline(cin, name);  
        phoneNumbers[name] = number;                // add items to the map with []  
    }  
  
    // now retrieve the name  
    cout << "Enter someone's name: ";  
    getline(cin, name);  
    number = phoneNumbers[name];                  // retrieve items with []  
    if (number.size() == 0)                      // if the key does not exist, a  
        cout << "No number from that name\n";       //      default object is returned  
    else  
        cout << "That phone number of "  
            << name << " is "  
            << number << endl;  
}
```

As a challenge, can you modify this program to store the user's birthday? Therefore the data-item (second parameter in the map declaration) will be a `Date` rather than a `string`. You will need to use your `Date` class from Unit 2.

See Also

The complete solution is available at [4-6-mapPhone.html](#) or:

```
/home/cs165/examples/4-6-mapPhone.cpp
```



Example 4.6 – Phonetic alphabet

Demo

This example will demonstrate how to declare, fill, and retrieve items out of a map.

Problem

Write a program to convert a letter to the phonetic alphabet equivalent. The first six letters of the phonetic alphabet corresponding to A, B, C, D, E, and F are “Alfa,” “Bravo,” “Charlie,” “Delta,” “Echo”, and “Foxtrot.” Note that we will only convert uppercase letters.

```
What letter? F  
The letter 'F' corresponds to Foxtrot
```

The alphabet is stored in the file /home/cs165/examples/4-6-mapAlphabet.txt.

Solution

The function to read the phonetic data from the 4-6-mapAlphabet.txt file and place it in the map is:

```
void readAlphabet(const char * fileName, map <char, string> & alphabet)  
{  
    // open the file  
    ifstream fin(fileName);  
    if (fin.fail())  
    {  
        cout << "ERROR: Unable to open the alphabet file " << fileName << endl;  
        return;  
    }  
  
    // read the file. The first item is the letter  
    char letter;  
    string word;  
    while (fin >> letter >> word)  
        alphabet[letter] = word; // add data to the map with  
                                // the [] operator  
  
    // close the file and bail  
    fin.close();  
    return;  
}
```

The code to retrieve the data from the user user is:

```
{  
    map <char, string> alphabet;  
  
    // read the alphabet  
    readAlphabet("/home/cs165/examples/4-6-mapAlphabet.txt", alphabet);  
  
    // prompt the user for a letter  
    char letter;  
    cout << "What letter? ";  
    cin >> letter;  
  
    // display the results  
    cout << "The letter '" << letter  
        << "' corresponds to "  
        << alphabet[letter] << endl; // retrieve the data with the  
                                // [] operator. Note that  
                                // this is case-sensetive  
}
```

See Also

The complete solution is available at [4-6-mapAlphabet.html](#) or:

```
/home/cs165/examples/4-6-mapAlphabet.cpp
```



Problem 1

Of the following tools: {string, vector, stream, map, list}, identify which would be useful in the following scenario:

- A tool useful for most I/O applications
- Like an array except a non-integer key can be used
- Useful only with text
- Like an array except adding and deleting are fast, no random access
- Like an array except it will grow as the size increases

Please see page 326, 329, 331 for a hint.

Problem 2-5

Define a variable for each of the following problems:

2. A collection of student names
3. A data-structure having three components: the assignment name, the points possible, and the student's score
4. A collection of assignments (from #3)
5. A program translating English words to French

Please see page 326, 329, 331 for a hint.

Problem 6

Consider the following code:

```
{  
    vector <string> names;  
    fill(names);  
}
```

Write the function `fill()` to fill the parameter with student names until the user enters the text "done"

Please see page 326 for a hint.

Challenge 7 - 8

Consider the following code:

```
{  
    map <string, string> dict;  
    fill(dict);  
    translate(dict);  
}
```

7. Write the function `fill()` to fill the dictionary with English / French word pairs. Stop when user enters "!"
8. Write the function `translate()` to prompt the user for an English word and display the French response. Stop when the user types "!"

Please see page 333 for a hint.

Assignment 4.6: Reverse file

Write a program to read the data from a file line-by-line. You will need to store the contents of the file in one of the STL containers. Next, display the file backwards.

Example

Consider the following familiar file (/home/cs165/assign46.txt):

```
#include <iostream>

int main()
{
    std::cout << "Hello world!" << std::endl;
}
```

Execution of the program will be (input in underline):

```
Enter the name of the file: /home/cs165/assign46.txt
}
    std::cout << "Hello world!" << std::endl;
{
int main()

#include <iostream>
```

Instructions

Please verify your solution against test-bed with:

```
testBed cs165/assign46 assignment46.cpp
```

Please see page 326 for a hint.



Appendix

A. Elements of Style.....	338
B. C++ Syntax Reference Guide	343
C. Glossary	347
D. Index	362

A. Elements of Style

While the ultimate test of a program is how well it performs for the user, the value of the program is greatly limited if it is difficult to understand or update. For this reason, it is very important for programmers to write their code in the most clear and understandable way possible. We call this “programming style.”

Elements of Style

Perhaps the easiest way to explain coding style is this: give the bugs no place to hide. When our variable names are clearly and precisely named, we are leaving little room for confusion or misinterpretation. When things are always used the same way, then readers of the code have less difficulty understanding what they mean.

There are four components to our style guidelines: variable and function names, spacing, function and program headers, general comments, and other standards.

Variable and function names

The definitions of terms and acronyms of a software program typically consist of variable declarations. While variables are declared in more than one location, the format should be the same. Using descriptive identifiers reduces or eliminates the need for explanatory comments. For example, `sum` tells us we are adding something; `sumOfSquares` tells us specifically what we are adding. Use of descriptive identifiers also reduces the need for comments in the body of the source code. For example, `sum += x * x;` requires explanation. On the other hand, `sumOfSquares += userInput * userInput;` not only tells us where the item we are squaring came from, but also that we are creating a sum of the squares of those items. If identifiers are chosen carefully, it is possible to write understandable code with very few, if any, comments. The following are the University conventions for variable and function names:

Identifier	Example	Explanation
Variable	<code>sumOfSquares</code>	Variables are nouns so it follows that variable names should be nouns also. All the words are TitleCased except the first word. We call this style camelCase.
Function	<code>displayDate()</code>	Functions are verbs so it follows that function names should also be verbs. Like variables, we camelCase functions.
Constant	<code>PI</code>	Constants, include <code>#defines</code> , are ALL_CAPS with an underscore between the words.
Data types	<code>Date</code>	Classes, enumeration types, type-defs, and structures are TitleCased with the first letter of each word capitalized. These are CS 165 constructs.

Spacing

During the lexing process, the compiler removes all the spaces between keywords (such as `int`, `for`, or `if`) and operators (such as `+` or `>=`). To make the code human-readable, it is necessary to introduce spaces in a consistent way. The following are the University conventions for spaces:

Rule	Example	Explanation
Operators	<code>tempC = 5.0 / 9.5 (tempF - 32.0)</code>	There needs to be one space between all operators, including arithmetic (+ and %), assignment (= and +=) and comparison (>= and !=)
Indentation	<code>int answer = 42; if (answer > 100) cout << "Wrong answer!"; }</code>	With every level of indentation, add three white spaces. Do not use the tab character to indent.
Functions		Put one blank line between functions. More than one results in unnecessary scrolling, less feels cramped
Related code	<code>// get the data float income; cout << "Enter income: "; cin >> income;</code>	Much like an essay is sub-divided into paragraphs, a function can be sub-divided into related statements. Each statement should have a blank line separating them.

Function and program headers

It takes quite a bit of work to figure out what a program or function is trying to do when all the reader has is the source code. We can simplify this process immensely by providing brief summaries. The two most common places to do this are in function and program headers.

A function header appears immediately before every function. The purpose is to describe what the program does, any assumptions made about the input parameters, and describe the output. Ideally, a programmer should need no more information than is provided in the header before using a function. An example of a function header is the following:

```
*****
* GET YEAR
* Prompt the user for the current year. Error checking
* will be performed to ensure the year is valid
*   INPUT: None (provided by the user)
*   OUTPUT: year
*****
```

A program header appears at the beginning of every file. This identifies what the program does, who wrote it, and a brief description of what it was written for. Our submission program reads this program header to determine how it is to be turned in. For this reason, it is important to start every program with the template provided at `/home/cs165/template.cpp`. The header for Assignment 1.0 is:

```
*****
* Program:
*   Assignment 10, Hello World
*   Brother Helfrich, CS165
* Author:
*   Sam Student
* Summary:
*   Display the text "Hello world" on the screen.
*   Estimated: 0.7 hrs
*   Actual:     0.5 hrs
*   I had a hard time using emacs.
*****
```

General comments

We put comments in our code for several reasons:

- To describe what the next few lines of code do
- To explain to the reader of the code why code was written a certain way
- To write a note to ourselves reminding us of things that still need to be done
- To make the code easier to read and understand

Since a comment can be easily read by a programmer and source code, in many cases, must be decoded, one purpose of comments is to clarify complicated code. Comments can be used to convey information to those who will maintain the code. For example, a comment might provide warning that a certain value cannot be changed without impacting other portions of the program. Comments can provide documentation of the logic used in a program. Above all else, comments should add *value* to the code and should not simply restate what is obvious from the source code. The following are meaningless comments and add no value to the source code:

```
int i; // declare i to be an integer  
i = 2; // set i to 2
```

On the other hand, the following comments add value:

```
int i; // indexing variable for loops  
i = 2; // skip cases 0 and 1 in the loop since they were processed earlier
```

With few exceptions, we use line comments (//) rather than block comments /* ... */ inside functions. Please add just enough comments to make your code more readable, but not so many that it is overly verbose. There is no hard-and-fast rule here.

“Commenting out” portions of the source code can be an effective debugging technique. However, these sections can be confusing to those who read the source code. The final version of the program should not contain segments of code that have been commented out.

Other standards

Because of the way printers and video displays handle text, readability is improved by keeping each line of code less than 80 characters long.

Subroutines and classes should be ordered in a program such that they are easy to locate by the reader of the source code. For larger programs, an alphabetical arrangement works well. For shorter programs, following the order in which the subroutines are invoked or objects are instantiated is useful.

Each curly brace should be on its own line; this makes them easier to match up.

Just as one would expect correct spelling in a technical paper, one should also find correct spelling within a program

Style checklist

Comments

- program introductory comment block
- identify program
- identify instructor and class
- identify author
- brief explanation of the program
- brief explanation of each class
- brief explanation of each subroutine

Variable declarations

- declared on separate lines
- comments (if necessary)

Identifiers

- descriptive
- correct use of case
- correct use of underscores

White space

- white space around operators
- white space between subroutines
- white space after key words
- each curly brace on its own line

Indentation

- statements consistently indented
- block of code within another block of code further indented

General

- code appropriately commented
- each line less than 80 characters long
- correct spelling
- no unused (e.g. commented out) code

Example 1: Method

The following is an example of a class method with excellent style.

```
*****
 * Extraction      cin >> x;
 * RETURN:          istream by reference (so we can say (cin >> x) >> y;)
 * PARAMETER:       istream by reference (we do not want a copy of cin)
 *                  by reference        (no copies but we do want to change this)
 ****
istream & operator >> (istream & in, Card & card)
{
    // input comes in the form of a string
    string input;
    in >> input;

    // do the actual work
    card.parse(input);
    assert(card.validate());

    // return the input stream
    return in;
}
```

Example 2: Header

The following example is a header file with excellent style.

```
*****  
 * This file describes the WRITE interface, a class  
 *      designed to write data to a file  
*****/  
#ifndef _WRITE_H_  
#define _WRITE_H_  
  
#include <fstream>          // necessary for the ofstream object  
#include <string>           // necessary for the string parameters  
  
*****  
 * WRITE  
 * This class will write text to a file without having  
 * the client have to worry about opening or closing it  
*****/  
class Write  
{  
    public:  
        Write() : isOpen(false) { }  
        Write(const std::string & fileName);  
        ~Write();  
        void writeToFile(const std::string & text);  
  
    private:  
        bool         isOpen;           // did we successfully open the file?  
        std::ofstream fout;           // the file stream object  
};  
#endif // _WRITE_H_
```

Example 3: Makefile

The following example is a makefile with excellent style.

```
#####  
# Program:  
#      Example 1.5, Complex Numbers demo  
#      Brother Helfrich, CS165  
# Author:  
#      Br. Helfrich  
# Summary:  
#      This program will demonstrate how to break a large program into several files  
#####  
  
#####  
# The main rule  
#####  
a.out: complexTest.o complex.o  
        g++ -o a.out complexTest.o complex.o  
        tar -cf example14.tar *.h *.cpp makefile  
  
#####  
# The individual components  
#      complex.o      : Compile only if complex.cpp or complex.h changed  
#      complexTest.o  : Compile only if complexTest.cpp or complex.h changed  
#####  
complexTest.o: complexTest.cpp complex.h  
        g++ -c complexTest.cpp  
  
complex.o: complex.cpp complex.h  
        g++ -c complex.cpp
```

B. C++ Syntax Reference Guide

Name	Syntax	Example
Pre-processor directives	#include <libraryName> #define <MACRO_NAME> <expansion>	<pre>#include <iostream> // for CIN & COUT #include <iomanip> // for setw() #include <fstream> // for IFSTREAM #include <string> // for STRING #include <cctype> // for ISUPPER #include <cstring> // for STRLEN #include <cstdlib> // for ATOF #define PI 3.14159 #define LANGUAGE "C++"</pre>
Function	<ReturnType> <functionName>(<params>) { <statements> return <value>; }	<pre>int main() { cout << "Hello world\n"; return 0; }</pre>
Function parameters	<DataType> <passByValueVariable>, <DataType> & <passByReferenceVariable>, const <DataType> <CONSTANT_VARIABLE>, <BaseType> <arrayVariable>[]	<pre>void function(int value, int &reference, const int CONSTANT, int array[]) { }</pre>
COUT	cout << <expression> << ... ;	<pre>cout << "Text in quotes" << 6 * 7 << getNumber() << endl;</pre>
Formatting output for money	cout.setf(ios::fixed); cout.setf(ios::showpoint); cout.precision(<integerExpression>);	<pre>cout.setf(ios::fixed); cout.setf(ios::showpoint); cout.precision(2);</pre>
Declaring variables	<DataType> <variableName>; <DataType> <variableName> = <init>; const <DataType> <VARIABLE_NAME>;	<pre>int integerValue; float realNumber = 3.14159; const char LETTER_GRADE = 'A';</pre>
CIN	cin >> <variableName>;	<pre>cin >> variableName;</pre>
IF statement	if (<Boolean-expression>) { <statements> } else { <statements> }	<pre>if (grade >= 70.0) cout << "Great job!\n"; else { cout << "Try again.\n"; pass = false; }</pre>
Asserts	assert(<Boolean-expression>);	<pre>#include <cassert> // at top of file { assert(gpa >= 0.0); }</pre>

Name	Syntax	Example
FOR loop	<pre>for (<initialization statement>; <Boolean-expression>; <increment statement>) { <statements> }</pre>	<pre>for (int iList = 0; iList < sizeList; iList++) cout << list[iList];</pre>
WHILE loop	<pre>while (<Boolean-expression>) { <statements> }</pre>	<pre>while (input <= 0) cin >> input;</pre>
DO-WHILE Loop	<pre>do { <statements> } while (<Boolean-expression>);</pre>	<pre>do cin >> input; while (input <= 0);</pre>
Read from File	<pre>ifstream <streamVar>(<fileName>); if (<streamVar>.fail()) { <statements> } <streamVar> >> <variableName>; <streamVar>.close();</pre>	<pre>#include <fstream> // at top of file { ifstream fin("data.txt"); if (fin.fail()) return false; fin >> value; fin.close(); }</pre>
Write to File	<pre>ofstream <streamVar>(<fileName>); if (<streamVar>.fail()) { <statements>; } <streamVar> << <expression>; <streamVar>.close();</pre>	<pre>#include <fstream> // at top of file { ofstream fout("data.txt"); if (fout.fail()) return false; fout << value << endl; fout.close(); }</pre>
Fill an array	<pre><BaseType> <arrayName>[<size>]; <BaseType> <arrayName>[] = { <CONST_1>, <CONST_2>, ... }; for (int i = 0; i < <size>; i++) <arrayName>[i] = <expression>;</pre>	<pre>int grades[10]; for (int i = 0; i < 10; i++) { cout << "Grade " << i + 1 << ": "; cin >> grades[i]; }</pre>
C-Strings	<pre>char <stringName>[<size>]; cin >> <stringName>; for (char *<ptrName> = <stringName>; *<ptrName>; <ptrName>++) cout << *<ptrName>;</pre>	<pre>char firstName[256]; cin >> firstName; for (char *p = firstName; *p; p++) cout << *p;</pre>
String Class	<pre>string <stringName>; cin >> <stringName>; cout << <stringName>; getline(<streamName>, <stringName>); if (<stringName1> == <stringName2>) <statement>; <stringName1> += <stringName2>; <stringName1> = <stringName2>;</pre>	<pre>string string1; // declare string string2 = "124"; // initialize cin >> string1; // input getline(cin, string2); // getline if (string1 == string2) // compare string1 += string2; // append string2 = string1; // copy</pre>

Name	Syntax	Example
Switch	<pre>switch (<integer-expression>) { case <integer-constant>: <statements> break; // optional ... default: // optional <statements> }</pre>	<pre>switch (value) { case 3: cout << "Three"; break; case 2: cout << "Two"; break; case 1: cout << "One"; break; default: cout << "None!"; }</pre>
Conditional Expression	<Boolean-expression> ? <expression> : <expression>	<pre>cout << "Hello, " << (isMale ? "Mr. " : "Mrs. ") << lastName;</pre>
Multi-dimensional array	<pre><BaseType> <arrayName>[<SIZE>][<SIZE>]; <BaseType> <arrayName>[] [<SIZE>] = { { <CONST_0_0>, <CONST_0_1>, ... }, { <CONST_1_0>, <CONST_1_1>, ... }, ... }; <arrayName>[<index>][<index>] = <expression>;</pre>	<pre>int board[3][3]; for (int row = 0; row < 3; row++) for (int col = 0; col < 3; col++) board[row][col] = 10;</pre>
Allocate memory	<pre><ptr> = new(nothrow) <DataType>; <ptr> = new(nothrow) <DataType>(<init>); <ptr> = new(nothrow) <BaseType>[<SIZE>];</pre>	<pre>float *pNum1 = new(nothrow) float; int *pNum2 = new(nothrow) int(42); char *text = new(nothrow) char[256];</pre>
Free memory	<pre>delete <pointer>; // one value delete [] <arrayPointer>; // an array</pre>	<pre>delete pNumber; delete [] text;</pre>
Command line parameters	<pre>int main(int <countVariable>, char **<arrayVariable>) { }</pre>	<pre>int main(int argc, char **argv) { }</pre>
Input errors	<pre><iostream>.fail(); <iostream>.clear(); <iostream>.ignore(<numChars>,<token>)</pre>	<pre>if (cin.fail()) { cin.clear(); cin.ignore(256, '\n'); }</pre>
Exceptions	<pre>try { <statements> throw <expression> } catch (<declaration>) { <statements> }</pre>	<pre>try { cin >> data; if (data < 0) throw data; } catch (int error) { cout << "An error occurred!"; }</pre>

Structures	<pre>struct <DataType> { <member variable declarations> };</pre>	<pre>struct Point { int row; int col; };</pre>
Header files	<pre>#ifndef <FILE_TAG> #define <FILE_TAG> <body of the header> #endif // FILE_TAG</pre>	<pre>#ifndef _POINT_H_ #define _POINT_H_ struct Point { int row; int col; }; #endif // _POINT_H_</pre>
makefile	<pre><target> : <Dependency List> <Recipe></pre>	<pre>a.out: file.o interface.o g++ file.o interface.o tar -cf file.tar *.h *.cpp</pre>
Default parameters	<pre><Return> <funcName>(<DataType> <variable> = <value>);</pre>	<pre>void func(int value = 0);</pre>
Function pointer	<pre><Return> (*<pointer>">(<param List>);</pre>	<pre>void (*ptrFunction)(int value);</pre>
Class syntax	<pre>class <ClassTag> { <public/private>: <variable declarations> <method definitions> };</pre>	<pre>class Point { public: void set(int r, int c); private: int row; int col; };</pre>
Method definition	<pre><Return> <ClassName>::<methodName>() { <statements> }</pre>	<pre>void Point :: set (int r, int c) { row = r; col = c; }</pre>

C. Glossary

#define	A #define (pronounced “pound define”) is a mechanism to expand macros in a program. This macro expansion occurs before the program is compiled. The following example expands the macro PI into 3.1415	Chapter 1.4
#ifdef	The #ifdef macro (pronounced “if-deaf”) is a mechanism to conditionally include code in a program. If the condition is met (the referenced macro is defined), then the code is included.	Chapter 1.4
#ifndef	The #ifndef macro works exactly the same as #ifdef except code is included only when the condition is <u>not</u> met.	Chapter 1.4
abstract class	An abstract class is a class containing at least one pure virtual function. This means that it is impossible to instantiate an object from an abstract class.	Chapter 3.5
abstracting	The process of creating an entity capturing the <i>essence</i> of relationships and allowing specific implementations to describe their details. Abstracing is the Object-Oriented way to addressing a problem involving class relations whereas universalizing is more of a procedural approach.	Chapte 3.0
access modifier	A part of the notation of the UML class diagram that allows the designer of the class to specify whether a given member variable or method is accessible to clients of the class or just to methods of the class itself. The three access modifiers are + for public, - for private, and # for protected.	Chapter 2.0 Chapter 2.2
accessor	Otherwise known as a “getter,” an accessor is a method providing the client access to data from one or more member variable.	Chapter 2.3
address-of operator	The address-of operator (&) yields the address of a variable in memory. It is possible to use the address-of operator in front of any variable.	Review

aggregate data type	A data type built from one or more built-in data types. An array is an aggregate data type because it consists of more than one instance of the base-type. Structures and classes are also aggregate data types	Chapter 1.2
algorithms	The part of a design document describing how certain functions are to be implemented. This is commonly expressed with pseudocode	Chapter 1.0
array	An array is a data-structure containing multiple instances of the same item. In other words, it is a “bucket of variables.” Arrays have two properties: all instances in the collection are the same data type and each instance can be referenced by index (not by name).	Review
	<pre>{ int array[4]; // a list of four integers array[2] = 42; // the 3rd member of the list }</pre>	
arrow operator	The arrow operator (->) enables the programmer to more easily access member variables and member functions from a pointer to a class or a pointer to a structure. The following are equivalent:	Chapter 1.3
	<pre>void display(const Point * pt) { cout << pt->x << (*pt).x; }</pre>	
assert	An assert is a function that tests to see if a particular assumption is met. If the assumption is met, then no action is taken. If the assumption is not met, then an error message is thrown and the program is terminated. Asserts are designed to only throw in debug code. To turn off asserts for shipping code, compile with the -DNDEBUG switch.	Chapter 1.1
base class	A base class, also known as a parent class, is the class from which a derived class is built.	Chapter 3.2
binding	Binding is the process of connecting the name of a function or variable with the location in memory where the code or data resides. This may happen at compile time (called “early binding”) or at run-time (called “late binding”)	Chapter 3.1
bitwise operator	A bitwise operator is an operator that works on the individual bits of a value or a variable.	Review Chapter 2.6
bool	A <code>bool</code> is a built-in datatype used to describe logical data. The name “Bool” came from the father of logical data, George Boole.	Review
	<pre>bool isMarried = true;</pre>	
Boolean operator	A Boolean operator is an operator that evaluates to a Bool (<code>true</code> or <code>false</code>). For example, consider the expression (<code>value1 == value2</code>). Regardless of the data type or value of <code>value1</code> and <code>value2</code> , the expression will always evaluate to <code>true</code> or <code>false</code> .	Review
buffer	An array of data to be filled with user input	Chapter 1.1

callback	A callback is a function pointer passed to another function as a parameter with the expectation that the function pointer will be executed.	Chapter 4.1				
callee	A function that is called from another function. The function initiating the call is called the caller. The function receiving the call is the callee.	Chapter 4.1				
caller	A function initiating a call to another function. The function receiving the call is the callee.	Chapter 4.1				
casting	The process of converting one data type (such as a <code>float</code>) into another (such as an <code>int</code>). For example, <code>(float)3</code> equals <code>3.0</code> .	Review				
catch	A part of the exception handling mechanism, the <code>catch</code> block receives an exception thrown by another part of the program	Chapter 1.2				
	<pre>catch (int rocks) { cout << "Ouch!\n"; }</pre>					
char	A <code>char</code> is a built-in datatype used to describe a character or glyph. The name “Char” came from “Character,” being the most common use.	Review				
	<pre>char letterGrade = 'B';</pre>					
comments	Comments are notes placed in a program not read by the compiler.	Review				
cohesion	The measure of the internal strength of a module. In other words, how much a function does one thing and one thing only. The seven levels of cohesion are: Functional, Sequential, Communicational, Procedural, Temporal, Logical, and Coincidental.	Review				
conditional expression	A conditional expression is a decision mechanism built into C++ allowing the programmer to choose between two expression, rather than two statements.	Review				
	<pre>cout << (grade >= 60.0 ? "pass" : "fail");</pre>					
coupling	Coupling is the measure of information interchange between functions. The four levels of coupling are: Data, Stamp, Control, and External.	Review				
class	A data-structure consisting of member variables (like a structure) and member functions. A class is not a variable, but rather a template or plan by which a variable can be made. A variable built from a class is called an object.	Chapter 2.0				
class diagram	One tool in the UML suite of tools, the class diagram helps the programmer design with classes. The following three row table is a class diagram for a simple class:	Chapter 1.3 Chapter 2.0 Chapter 3.0				
	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>Name</td></tr> <tr><td>- first</td></tr> <tr><td>- last</td></tr> <tr><td>+ display</td></tr> </table>	Name	- first	- last	+ display	
Name						
- first						
- last						
+ display						
class template	A class template is similar to a function template with one exception: one or more member variable (rather than a parameter) uses an abstract data type. While function templates were useful for implementing generic	Chapter 4.3				

algorithms (like the bubble sort or the swap function), class templates are useful for implementing generic data-structures (like the stack or array).

client	A function that is using code provided elsewhere in the program. If one function calls another function, the first is a client of the second. If one function instantiates an object from a class described elsewhere, the function is a client of the class.	Chapter 2.0
composition	A form of “has-a” where one class consists of a collection of subordinate classes.	Chapter 3.0
constructor	A constructor is a method in a class that is guaranteed to be called when an object is instantiated. There are three variations of constructors: default constructor (taking no parameters), a copy constructor (taking an object of the same type as the class itself as a constant by-reference parameter), and a non-default constructor (any other constructor that is not a default or copy constructor).	Chapter 2.4
	<pre>Point::Point() { x = 0; y = 0; };</pre>	
container	A container is data-structures designed to hold or contain data. Examples of containers are arrays, linked lists, the <code>vector</code> class, the stack, and the queue.	Chapter 4.6
<code>cout</code>	<code>CO</code> stands for <u>Console</u> <u>OUT</u> put. Technically speaking, <code>cout</code> is the destination or output stream. In other words, it is the following example, it is the destination where the insertion operator (<code><<</code>) is sending data to. In this case, that destination is the screen.	Review
	<pre>cout << "Hello world!";</pre>	
c-string	A c-string is how strings are stored in C++: An array of characters terminated with a null (' <code>\0</code> ') character.	Review
data type	A way of representing something in a computer program. Built-in data types include integers (<code>int</code>), real numbers (<code>float</code> , <code>double</code> , etc.), characters (<code>char</code>), etc. The programmer can create his own data type with structures (<code>struct</code>), enumerations (<code>enum</code>), typedefs (<code>typedef</code>), and more.	Chapter 1.3
data-structures	The part of a design document describing how data is stored in memory while the program is running. For an object-oriented program, the UML class diagram is commonly used here	Chapter 1.0
default parameter	Default parameters allow the author of a function to specify the value of a passed parameter if the client chose not provide one.	Chapter 1.5
	<pre>void getline(char * text, // required int bufferSize = 256); // default</pre>	

delete	The delete operator serves to free memory previously allocated with new . When a variable is declared on the stack such as a local variable, this is unnecessary; the operating system deletes the memory for the user. However, when data is allocated with new , it is the programmer's responsibility to delete his memory.	Review
dereference operator	The dereference operator '*' will retrieve the data referred to by a pointer.	Review
derivation	A form of "is-a" class relations where one class is a manifestation or type of a parent class.	Chapter 3.0
derived class	A derived class, also known as the child class, is the class that is built from a base class.	Chapter 3.2
design document	A document containing a collection of tools used to help draft out the solution to a complex programming problem. Design documents usually consist of the following components: problem definition, design overview, interface design, structure chart, algorithms, data-structures, file format, and error handling.	Chapter 1.0
design overview	The part of a design document describing how the problem is to be solved. This is an overview of the big design decisions. Typically it consists of a high-level summary of the data-structures, algorithms, or function organization	Chapter 1.0
destructor	A method in a class that is guaranteed to be called when an object is destroyed such as when it falls out of scope. There is only one destructor; destructors cannot take parameters.	Chapter 2.4
do ... while	One of the three types of loops, a DO-WHILE loop continues to execute as long as the condition in the Boolean expression evaluates to true. This is the same as a WHILE loop except the body of the loop is guaranteed to be executed at least once.	Review
double	A double is a built-in datatype used to describe large real numbers. The word "Double" comes from "Double-precision floating point number," indicating it is just like a float except it can represent a larger number more precisely.	Review

downcasting	Downcasting is the process of casting a base class into a derived class. This is dangerous because the program is required to “make up” information that is not provided.	Chapter 3.4
dynamically-allocated array	A dynamically-allocated array is an array that is created at run-time rather than at compile time. Stack arrays have a size known at compile time. Dynamically-allocated arrays, otherwise known as heap arrays, can be specified at run-time. <pre>{ int *array = new int[size]; }</pre>	Review
early binding	Early binding is the flavor of binding that the compiler performs. In a traditional program, the function name is “connected” with the code of the function at compile time. Errors in early binding are displayed as a compiler error.	Chapter 3.1
encapsulation	Encapsulation is the process of separating the use of an item from its implementation. This is one of the fundamental characteristics of OO programming: encapsulation, inheritance, polymorphism, and templates	Chapter 2.0
eof	When reading data from a file, one can detect if the end of the file is reached with the <code>eof()</code> function. Note that this will only return true if the end of file marker was reached in the last read. <pre>if (fin.eof()) cout << "The end of the file was reached\n";</pre>	Review
error flags	An error handling technique where a program reports an error has occurred by returning <code>false</code> .	Chapter 1.2
error handling	The part of a design document describing the types of errors it may be encountered in the program, how to detect the errors, and what to do when the error condition arises. There are three types of errors: user errors introduced by user input, file errors originating from unexpected data in a file, and internal errors resulting from bugs in the program	Chapter 1.0
error ID	An error ID (EID) is an error handling technique where all possible errors are enumerated. A function reporting the error would then return the code corresponding to the encountered error	Chapter 1.2
escape sequences	Escape sequences are simply indications to <code>cout</code> that the next character in the output stream is special. Some of the most common escape sequences are the newline (<code>\n</code>), the tab (<code>\t</code>), and the double-quote (<code>\"</code>)	Review
exceptions	A special error-handling mechanism built into the C++ language consisting of the <code>try</code> , <code>throw</code> , and <code>catch</code> elements	Chapter 1.2
expression	A collection of values and operations that, when evaluated, result in a single value. For example, <code>3 * value</code> is an expression. If <code>value</code> is defined as <code>float value = 1.5;</code> , then the expression evaluates to 4.5.	Review

extraction operator The extraction operator (`>>`) is the operator that goes between `cin` and the variable receiving the user input. In the following example, the extraction operator is after the `cin`.

```
cin >> data;
```

file format The part of a design document describing how to persist that data between executions of the program Chapter 1.0

for One of the three types of loops, the FOR loop is designed for counting. It contains fields for the three components of most counting problems: where to start (the Initialization section), where the end (the Boolean expression), and what to change with every count (the Increment section). Review

```
for (int i = 0; i < num; i++)
    cout << array[i] << endl;
```

friend Friends are special functions in C++ that have access to the private member variables and private methods of a class. Chapter 2.7

fstream The `fstream` library contains tools enabling the programmer to read and write data to a file. The most important components of the `fstream` library are the `ifstream` and `ofstream` classes. Review

```
#include <fstream>
```

function pointer A pointer to a function rather than a pointer to data. The following is a pointer to a void function taking an integer as a parameter: Chapter 1.5

```
{
    void (*pointer)(int parameter);
```

function signature A function signature is the combination of the name of the function with the data type of the parameters it takes. This is used to uniquely identify a function. Chapter 1.5

function template A function template is a mechanism in C++ allowing the programmer to implement a generic algorithm that works with any data type. Chapter 4.2

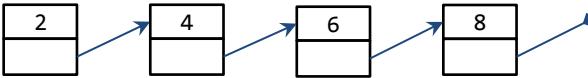
```
template <class T>
void function(T & t)
{
}
```

getline The `getline()` method works with `cin` to get a whole line of user input. Review

```
char text[256];           // getline needs a string
cin.getline(text, 256);   // the size is a parameter
```

header file A header file is a file containing function prototypes and type definitions. It is compiled through the `#include` mechanism. Chapter 1.4

ifstream	The <code>ifstream</code> class is part of the <code>fstream</code> library, enabling the programmer to write data to a file. IFSTREAM is short for “ <u>I</u> nput <u>F</u> ile <u>S</u> TREAM.”	Review
	<pre>#include <fstream> { ifstream fin("file.txt"); ... }</pre>	
implementation file	Another name for a source file	Chapter 1.4
inheritance	A form of “is-a” class relations where one class is a manifestation or type of a parent class.	Chapter 3.0
inheritance indicator	An inheritance indicator is the part of the class definition where the programmer specifies which if any class is the base class. In the following example, <code>Base</code> is the base class and “ <code>: public Base</code> ” is the inheritance indicator.	
	<pre>class Derived : public Base { code removed for brevity ... };</pre>	
inline	The <code>inline</code> keyword suggests to the compiler that it would be more efficient to copy-paste the code of the function body into the locations from which the function was called rather than call the function in the traditional manner.	Chapter 1.5 Chapter 2.3
	<pre>inline void displayError() { cout << "An error occurred\n"; }</pre>	
insertion operator	The insertion operator (<code><<</code>) is the operator that goes between <code>cout</code> and the data to be displayed. As we will learn in CS 165, the insertion operator is actually the function and <code>cout</code> is the destination of data. In the following example, the insertion operator is after the <code>cout</code> .	Review Chapter 2.6
	<pre>cout << "Hello world!";</pre>	
interface design	The part of a design document describing the output, input, and various error messages of a program	Chapter 1.0
int	An <code>int</code> is a built-in datatype used to describe integral data. The word “Int” comes from “Integer” meaning “all whole numbers and their opposites.”	Review
	<pre>int age = 19;</pre>	
invariant	Once design consideration when working with “is-a” class relations, invariant attributes refer to those attributes that are the same between two classes.	Chapter 3.0

iomanip	The IOMANIP library contains the <code>setw()</code> method, enabling a C++ program to right-align numbers. The programmer can request the IOMANIP library by putting the following code in the program:	Review
iostream	The IOSTREAM library contains <code>cin</code> and <code>cout</code> , enabling a simple C++ program to display text on the screen and gather input from the keyboard. The programmer can request IOSTREAM by putting the following code in the program:	Review
iterator	An iterator is a tool used to provide a standard way to iterate or traverse through a collection of values. They are designed to work the same regardless of the data type stored in the collection and regardless of how the collection is stored in memory.	Chapter 4.5
late binding	Late binding is the flavor of binding that occurs while the program is executing or running. Late binding errors are displayed as a run-time error such as a crash or a segmentation fault.	Chapter 3.1
	<code>Segmentation fault (core dumped)</code>	
	All late binding mechanisms involve pointers, and all uses of pointers indicates late binding is at work. Therefore pointers to variables, arrays, and pointers-to-functions are indications of late binding.	
linked list	A linked list is a type of data-structure where each item in the list is connected to the rest of the list through pointers. Each item in a linked list is called a node.	Chapter 4.4
		
local variable	A local variable is a variable defined in a function. The scope of the variable is limited to the bounds of the function.	Review
makefile	A <code>makefile</code> is a file describing how multiple files can be compiled together into a single program. It contains the dependencies for a given project. In the <code>makefile</code> , the programmer indicates how to tell if a given file needs to be rebuilt, and how.	Chapter 1.4
member variable	The list of variables comprising a structure or a class. In the following example, <code>row</code> and <code>col</code> are member variables	Chapter 1.3
	<pre>struct Position { int row; int col; };</pre>	
modulus	The remainder from division. Consider $14 \div 3$. The answer is 4 with a remainder of 2. Thus fourteen modulus 3 equals 2: $14 \% 3 == 2$	Review

multi-dimensional array	A multi-dimensional array is an array of arrays. Instead of accessing each member with a single index, more than one index is required. The following is a multi-dimensional array representing a tic-tac-toe board:	Review				
	<pre>{ char board[3][3]; }</pre>					
mutator	Otherwise known as a “setter,” a mutator is a method whose purpose is to modify or change the data stored in one or more member variable.	Chapter 2.3				
	<pre>void Date :: setYear(int year) { this->year = year; }</pre>					
new	It is possible to allocate a block of memory with the <code>new</code> operator. This serves to issue a request to the operating system for more memory. It works with single items as well as arrays.	Review				
	<pre>{ int *pValue = new int; // one integer int *array = new int[10]; // ten integers }</pre>					
node	A node is a structure or class consisting of some data-item and a pointer to the next node in a linked list.	Chapter 4.4				
	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>Node</td></tr> <tr><td> </td></tr> <tr><td>data</td></tr> <tr><td>pNext</td></tr> </table>	Node		data	pNext	
Node						
data						
pNext						
null	The null character, also known as a null terminator, is a special character marking the end of a c-string. The null character is represented as ' <code>\0</code> ', which is always defined as zero.	Review				
	<pre>{ char nullCharacter = '\0'; // 0x00 }</pre>					
NULL	The <code>NULL</code> address corresponds to the zero address <code>0x00000000</code> . This address is guaranteed to be invalid, making it a convenient address to assign to a pointer when the pointer variable does not point to anything.	Review				
	<pre>{ int *pValue = NULL; // points to nothing }</pre>					
object	A variable created from a class. In some situations, the term “object” also refers to a variable created from a structure	Chapter 2.2				
object file	An object file is a partially compiled file. Rather than compiling a file directly into an executable (such as <code>a.out</code>), an object file is compiled into assembly which must be combined (or linked) with other object files before an executable is created.	Chapter 1.4				

<code>ofstream</code>	The <code>ofstream</code> class is part of the <code>fstream</code> library, enabling the programmer to write data to a file. OFSTREAM is short for “ <u>Output File STREAM</u> .”	Review
	<pre>#include <fstream> { ofstream fout("file.txt"); ... }</pre>	
operator overloading	Operator overloading is the process of using a more convenient and human-readable notation for calling a function (such as “ <code>4 + 3</code> ”), than the functional notation common in programming languages (such as “ <code>add(4, 3)</code> ”).	Chapter 2.6
overloading	Overloading is the process of having more than one function with the same name. The only differences between the functions are the parameters they take.	Chapter 1.5
	<pre>int add(int value1, int value2); float add(float value1, float value2);</pre>	
pass-by-reference	Pass-by-reference, also known as “call-by-reference,” is the process of sending a parameter to a function where the caller and the callee share the same variable. This means that changes made to the parameter in the callee will be reflected in the caller. You specify a pass-by-reference parameter with the ampersand &.	Review
	<pre>void passByReference(int &parameter);</pre>	
pass-by-pointer	Pass-by-pointer, more accurately called “passing a pointer by reference,” is the process of passing an address as a parameter to a function. This has much the same effect as pass-by-reference.	Review
	<pre>void passByPointer(int *pParameter);</pre>	
pass-by-value	Pass-by-value, also known as “call-by-value,” is the process of sending a parameter to a function where the caller and the callee have different versions of a variable. Data is sent one-way from the caller to the callee; no data is sent back to the caller through this mechanism. This is the default parameter passing convention in C++.	Review
	<pre>void passByValue(int parameter);</pre>	
pointer	A pointer is a variable holding an address rather than data. A data variable, for example, may hold the value <code>3.14159</code> . A pointer variable, on the other hand, will contain the address of some place in memory.	Review
polymorphism	Polymorphism is the process of one class having more than one variation. Each variation honors the same contract (called “interface”) though the behavior details (called “implementation”) may be different.	Chapter 3.1
<code>private</code>	An access modifier indicating that only member functions can have access to the corresponding member variable or method	Chapter 2.0 Chapter 2.2

problem description	The part of a design document describing what problem is to be solved. This commonly includes a high-level description of the functionality and behavior of the program	Chapter 1.0
protected	An access modifier indicating that only member functions and member functions of derived classes can have access to the corresponding member variable or method.	Chapter 3.3
prototype	A prototype is the name, parameter list, and return value of a function to be defined later in a file. The purpose of the prototype is to give the compiler “heads-up” as to which functions will be defined later in the file.	Review
pseudocode	Pseudocode is a high-level programming language designed to help people design programs. Though it has most of the elements of a language like C++, pseudocode cannot be compiled. An example of pseudocode is:	Chapter 1.0
	<pre>computeTithe(income) RETURN income ÷ 10 END</pre>	
public	An access modifier indicating that the client can have access to the corresponding member variable or method.	Chapter 2.0 Chapter 2.2
pure virtual function	A pure virtual function is a member function within a base-class that has no implementation. A class derived off of this base class, on the other hand, must have a definition for this function. This means that it is impossible to instantiate a class containing a pure virtual function (because there is no implementation for the pure virtual function) but it is legal to instantiate one of the derived classes.	Chapter 3.5
scope	Scope is the context in which a given variable is available for use. This extends from the point where the variable is defined to the next closing braces }.	Review
scope resolution operator	The scope resolution operator (::) is used to indicate to the compiler which class the method is associated with:	Chapter 2.2
	<pre>int Card :: getSuit() { return value / 13; }</pre>	
separate compilation	The process of designing a program with multiple files, each of which is a manageable size.	Chapter 1.4
sizeof	The sizeof function returns the number of bytes that a given datatype or variable requires in memory. This function is unique because it is evaluated at compile-time where all other functions are evaluated at run-time.	Review
	<pre>{ int integerVariable; cout << sizeof(integerVariable) << endl; // 4 cout << sizeof(int) << endl; // 4 }</pre>	

slicing problem	The slicing problem is the process of casting a derived class into a base class. In this case, we are ignoring what is unique about the derived class and retaining only the base class information.	Chapter 3.4
source file	Also known as an implementation file, a source file is a file containing the source code for a program.	Chapter 1.4
stack variable	A stack variable, otherwise known as a local variable, is a variable that is created by the compiler when it falls into scope and destroyed when it falls out of scope. The compiler manages the creation and destruction of stack variables whereas the programmer manages the creation and destruction of dynamically allocated (heap) variables.	Review
STL	The Standard Template Library (STL) is a collection of tools designed to make common programming tasks easier.	Chapter 4.6
string	A “string” is a computer representation of text. The term “string” is short for “an alpha-numeric string of characters.” This implies one of the most important characteristics of a string: is a sequence of characters. In C++, a string is defined as an array of characters terminated with a null character.	Review
	<pre>{ char text[256]; // a string of 255 characters }</pre>	
structure	A mechanism to create a custom data type based on previously defined data types	Chapter 1.3
	<pre>struct Position { int row; int col; };</pre>	
structure chart	A structure chart is a design tool representing the way functions call each other. It consists of three components: the name of the functions of a program, a line connecting functions indicating one function calls another, and the parameters that are passed between functions.	Chapter 1.0
structure tag	The part of a structure declaration indicating the name of the newly created data type. In the following example, Position is the structure tag	Chapter 1.3
	<pre>struct Position { int row; int col; };</pre>	
static	The static keyword is a modifier attached to a variable to indicate that only one copy of the variable will exist in a program	Chapter 2.5
	<pre>{ static int value; }</pre>	

styleChecker	styleChecker is a program that performs a first-pass check on a student's program to see if it conforms to the University style guide. The styleChecker should be run before every assignment submission.	Appendix A
tabs	The tab key on a traditional typewriter was invented to facilitate creating tabular data (hence the name). The tab character ('\t') serves to move the cursor to the next tab stop. By default, that is the next 8 character increment. <pre>cout << "\tTab";</pre>	Review
TAR	TAR (<u>Tape ARchiving utility</u> , originally designed for creating an archive on the old-style reel-to-reel tape storage devices), is a program conceptually similar to ZIP: it combines multiple files into one and compresses them. We use TAR to combine our separate source files for the purpose of submission.	Chapter 1.4
template	A template is a function or class that can operate on a wide variety of data types, many of which do not have to be known by the developer of the function or class.	Chapter 4.0
template prefix	A statement immediately preceding a function template declaration indicating the type of type parameters used in the function. <pre>template <class T></pre>	Chapter 4.2
this	A pointer to the class itself from within the body of a method. This is useful to distinguish a member variable from a parameter of the same name.	Chapter 2.2 Chapter 2.8
throw	A part of the exception handling mechanism, the throw statement allows the programmer to signify that an error has occurred <pre>throw "rocks";</pre>	Chapter 1.2
throw list	A list of all the possible un-caught exceptions that a given function might throw. <pre>void display() throw (bool);</pre>	Chapter 1.2
try	A part of the exception handling mechanism, the try statement signifies that an exception may be thrown in the associated block of code <pre>try { if (windows == true) throw "rocks"; }</pre>	Chapter 1.2
type parameter	A data type associated with a template allowing a function or a class to be used with any data type. <pre>{ T variable; }</pre>	Chapter 4.2

UML	Unified Markup Language, UML is a suite of tools helping programmers design with classes and structures. The most commonly used UML tool and perhaps the most useful is the class diagram	Chapter 1.3 Chapter 2.0
universalizing	The process of making a single type containing all possible attributes. Universalizing is a more procedural approach to addressing a problem involving class relations whereas Abstracting is the Object-Oriented approach. Generally we should avoid universalizing.	Chapter 3.0
upcasting	Another name for the slicing problem, upcasting is the process of casting a derived class into a base class.	Chapter 3.4
variable	A variable is a named location where you store data. The name must be a legal C++ identifier (comprising of digits, letters, and the underscore _ but not starting with a digit) and conform to the University style guide (camelCase, descriptive, and usually a noun). The location is determined by the compiler, residing somewhere in memory.	Review
variant	Once design consideration when working with “is-a” class relations, variant attributes refer to those attributes that are different between two classes.	Chapter 3.0
virtual function	A virtual function is a member function that resides in the v-table associated with a class. This means that an object has a pointer to which version of a method is to be called.	Chapter 3.4
virtual function table	A virtual function table, also known as a v-table, is a structure containing function pointers to all the methods in a class. By adding a v-table as a member variable to a structure, the structure becomes a class	Chapter 2.1 Chapter 3.1
v-table	Another name for a virtual function table	Chapter 2.1
void pointer	A void pointer is a pointer to an unknown data type.	Chapter 4.1
while	One of the three types of loops, a WHILE-loop continues to execute as long as the condition inside the Boolean expression is true.	Review

```
while (grade < 70)
    grade = takeClassAgain();
```

D. Index

#define 365
 #ifdef 365
 #ifndef 365
 #include 87
 :: *See* scope resolution operator
 -> 77
 abstract class 269, 365
 abstracting 225
 access modifier 117, 136, 365
 accessor 144
 address-of operator 365
 aggregate data-type 71
 aggregation 219
 argc 362
 argv 362
 array 17, 360, 366
 arrow operator 366
 assert 55, 359, 366
 association 219
 base class 237, 366
 binding 229, 258, 366
 bitwise operator 366
 bool 4, 366
 Boolean operators 366
 callback 289, 290, 367
 callee 290, 367
 caller 290, 367
 cast 5
 casting 367
 catch 65
 catch-all 65
 cctype 22
 char 4, 367
 child class 237
 chmod 90
 cin 10, 359
 cin.clear() 54
 cin.fail() 53
 cin.ignore() 54
 class 115, 135, 367
 class template 310, 368
 client 98, 368
 cohesion 72, 86, 115, 367
 functions 14
 coincidental 15
 comments 355, 356, 367
 communicational 14

composition 218, 219, 368
 conditional expression 7, 362, 367
 const 4, 146
 const_cast 5
 constructor 151, 152, 241, 368
 copy 154
 default 152
 non-default 153
 container 341, 368
 continuous data 121
 coupling 86, 115, 367
 cout 9, 359, 368
 cstdlib 22
 cstring 22
 data 120
 data abstraction 115
 default parameters 103, 368
 delete 7, 362, 370
 dependency 219
 dereference operator 370
 derivation 218, 370
 derived class 237, 370
 design document 370
 Design Document
 Algorithms 33, 40
 Data Structure 42
 Data Structures 33
 Design Overview 33, 36
 Error Handling 33, 44, 48
 File Format 33, 43
 Interface Design 33, 37
 Problem Description 33, 35
 Structure Chart 33, 38
 destructor 151, 156, 370
 discrete data 121
 double 4, 370
 downcasting 261, 371
 dynamic_cast 5
 early binding 229, 371
 EID 63, 371
 encapsulation 114
 eof 371
 error flags 371
 error handling 371
 escape sequences 371
 expression 371
 extraction operator 7, 176, 373

file i/o	43, 50
flags	62
float	4
friend	192, 193, 194, 373
fstream	10, 373
function pointer	107, 128, 230, 373
function signature	99, 107, 305, 373
function template	302, 303, 373
functional	14
g++	89, 90
generic algorithms	281, 302
generic data-structures	281, 310
getline	10, 373
getter	122, <i>See</i> accessor
has-a	218, 219
header file	105, 137, 145, 373
if	359
ifstream	360, 374
implementation	227
implementation file	87
infix	174
information hiding	115
inheritance	218, 374
inheritance indicators	250
inline	105, 145, 374
insertion operator	7, 175, 374
instantiate	136
int	4, 374
interface	227
interval data	121
invariant	222, 374
iomanip	9, 376
iostream	9, 376
is-a	218, 220
iterator	331, 376
iterators	330
internator	374
late binding	229, 376
linked list	317, 376
list	345
local variable	376
logical	15
long	4
long double	4
loop	
do-while	13, 360, 370
for	13, 360, 373
while	12, 360, 384
makefile	91, 137, 376
map	347, 348, 349
member function	115, 117
member variable	115, 376
method	115, 117
modularization	115
modulus	8, 376
multi-dimensional arrays	17, 362, 377
mutator	144, 377
new	7, 362, 371, 377
node	317, 377
null	377
NULL	377
object	115, 377
object file	90, 377
ofstream	360, 378
operator	99, 173, 201, 378
-- decrement	181, 182, 186, 196, 202, 212
- negative	183, 186, 188, 196, 202
- subtraction	177, 186, 188, 196, 202, 212
! not	183, 186, 188, 196, 202
!= not equals	184, 186, 188, 196, 202, 212
% modulus	177, 186, 188, 196, 202
&& and	207, 213
() function call	206, 212
* multiplication	177, 186, 188, 196, 202
*= multiply by	179, 186, 188, 196, 202
/ division	177, 186, 188, 196, 202
/= divide by	179, 186, 188, 196, 202
[] square bracket	205, 213
or	207, 213
+ addition	177, 186, 188, 196, 202, 208- 212
++ increment	181-188, 196, 202, 209, 211-212
+= add onto	179, 186-196, 202, 208, 211-213
< less than	184, 186, 188, 196, 202, 212
<< insertion	175, 186-196, 202, 212, 243, 253
<= less than or equal	184-188, 196, 202, 212
= assignment	204, 212, 213
-= subtract from	179, 186, 188, 196, 202, 212
== equivalence	184-196, 202, 209, 212
> greater than	184, 186, 188, 196, 202, 212
>= greater than or equal	184-188, 196, 202, 212
>> extraction	176-188, 196, 202, 212, 243, 253
parent class	237
pass-by-pointer	378
pass-by-reference	16, 378
pass-by-value	16, 378
pointer	378
arithmetic	20
declare	20
polymorphism	378
private	117, 136, 378
procedural	14
properties	120
protected	248, 380
prototype	103, 380
pseudocode	40, 380
public	117, 136, 380
pure virtual function	269, 270
random access	318

redefining	242	styleChecker	383
reinterpret_cast	5	switch	362
scope	380	tabs	383
scope resolution operator	138, 380	TAR	92, 383
separate compilation	380	template	281
sequential	14	template prefix	303, 311
setf()	359	temporal	14
setter	122, <i>See</i> mutator	this	140, 383
setw()	9	throw	64, 383
short	4	throw list	383
sizeof()	7, 380	try	64, 383
slicing problem	261, 262, 381	type parameter	303, 311
source file	87, 137	UML	384
standard template library	341	UML class diagram	42, 72-74, -128-165, 219-257
static	164, 165, 381	universalizing	225, 384
static_cast	5	unsigned	4
STL	381, <i>See</i> standard template library	upcasting	262, 384
strings	23, 360, 381	variable	384
c-strings	368	variant	222, 384
struct	74	vector	23, 342, 343, 344
structure	381	virtual function	258, 384
structure chart	38, 381	void pointer	289, 295, 384
structure tag	381	v-table	127, 129, 230, 384

This project would not have been possible without the keen eye, perspective, and attention to detail of Adam Harris. He produced many of the illustrations, drafted many of the chapters, and proofed every single word.

I would also like to thank Scott Burton and Scott Ercanbrack for their contribution to this effort.