C++ Programming for Game Developers Module II

e-Institute Publishing, Inc.

Editor: Susan Nguyen
Cover Design: Adam Hoult

E-INSTITUTE PUBLISHING INC
www.gameinstitute.com

Frank Luna, *C++ Programming for Games II*

Table of Contents

Module II Overview

Module II is the second course in the C++ Programming for Game Developers series. Recall that in Module I we started off by studying fundamental programming concepts like variables, console input and output, arrays, conditional statements, strings, loops, and file input and output. We then pursued higher level programming methodologies such as classes, object oriented programming design, operator overloading, inheritance, and polymorphism. By now you should feel competent with the fundamentals and at least comfortable with the higher level subject matter.

Our aim in Module II is twofold. Our first objective is to finish our study of C++ by examining templates, error handling, the standard template library, and bitwise operations. Templates can be thought of as a class factory, which allows us to generate similar yet unique classes, based on a code template; this allows us to avoid duplicating code that is only slightly different. Error handling is an important topic because things rarely work out as planned, and we will need to be able to detect hardware failures, illegal operations, invalid input, corrupted and missing files, and the like in our code. The standard template library is a set of generic ready to use C++ code that simplifies many day-to-day programming tasks. In the STL chapter you will learn about several useful STL data structures and algorithms, and the ideas behind them. The chapter on bitwise operations provides a deeper understanding of computer memory and how numbers are represented internally. You will also learn how to work in several other numbering systems such as binary and hexadecimal, which are more natural from a computer's point of view.

The second key theme in Module II is Windows programming. Here we will learn how to make familiar Windows applications with resizable windows, mouse input, graphics, menus, dialog boxes, and controls. In addition, we will learn how to implement 2D flicker free animation with double buffering, and how to render 2D sprite images (i.e., graphical representation of game objects such as the main character, landscape, and enemies). Finally, we conclude Module II by walking the reader through the design and analysis of a fully functional 2D Air Hockey game, complete with graphics, physics, artificial intelligence, and input via the mouse. This final project culminates much of the course material.

By the end of this course, you will be well prepared for a first course in 3D game programming, as well as many other interesting computer related fields that require an understanding of computer programming as a qualification.

Chapter 10

Introduction to Templates

Introduction

You should recall from our discussions in Chapter 4 that `std::vector` can be thought of as a "resizable array" (Section 4.6). However, what is interesting about `std::vector` is that we can specify the type of vector to create with the angle bracket syntax:

```
vector<int>    intVec;

vector<float>  floatVec;

vector<bool>   boolVec;

vector<string> stringVec;
```

Thus we can create vectors of different types, just as we can create elementary arrays of different types. But, also recall that a `std::vector` is not some magical entity—it is just a class with methods that handle the internal dynamic memory array resizing. So the question is: how can we create a **generic** class that can work with any type (or at least, some types), like `std::vector` can? The answer to this question leads us to C++ templates and, more generally, **generic programming.**

Before continuing on, we would like to say that the subject of templates is vast, and we can only introduce the basics in this chapter. For advanced/interested readers, we refer you to *C++ Templates: The Complete Guide* by David Vandevoorde and Nicolai M. Josuttis. This book should prove to be an excellent resource for you and is a highly recommended supplement to the material we will study in this chapter.

Chapter Objectives

- Learn how to design and implement generic classes.
- Learn how to define generic functions.

10.1 Class Templates

Consider this small data structure, which represents an interval [*a*, *b*]:

```
struct FloatInterval
{
	FloatInterval();
	FloatInterval(float start, float end);

	float midpoint();

	float a;
	float b;
};

FloatInterval::FloatInterval()
{
	a = 0.0f;
	b = 0.0f;
}

FloatInterval::FloatInterval(float start, float end)
{
	a = start;
	b = end;
}

float FloatInterval::midpoint()
{
	// return the midpoint between a and b.
	return (a + b) * 0.5;
}
```

Although very simple, it is not hard to imagine the need for other types of intervals. For example, we may want to describe integer intervals, character intervals, 3D vector intervals, and color intervals (color values between two colors). The most obvious solution would be just to define these additional interval classes. Here are a few of them:

```
struct IntInterval
{
	IntInterval();
	IntInterval(int start, int end);

	int midpoint();

	int a;
	int b;
};

IntInterval::IntInterval()
{
	a = 0.0f;
	b = 0.0f;
}
```

```cpp
IntInterval::IntInterval(int start, int end)
{
      a = start;
      b = end;
}

int IntInterval::midpoint()
{
      // return the midpoint between a and b.
      return (a + b) * 0.5;
}

struct CharInterval
{
      CharInterval();
      CharInterval(char start, char end);

      char midpoint();
      char a;
      char b;
};

CharInterval::CharInterval()
{
      a = 0.0f;
      b = 0.0f;
}

CharInterval::CharInterval(char start, char end)
{
      a = start;
      b = end;
}

char FloatInterval::midpoint()
{
      // return the midpoint between a and b.
      return (a + b) * 0.5;
}
```

This approach results in a lot of additional code. Moreover, we may need to create new interval types later on, and we would then have to define additional classes. It does not take long to realize that this process can become cumbersome pretty quickly.

Let us instead analyze the interval class to see if we can make any observations that will help us simplify our task. We note that *the only difference* between `FloatInterval` and `IntInterval` is that everywhere we see a `float` in `FloatInterval`, we see an `int` in `IntInterval`—and similarly with `FloatInterval` and `CharInterval`. That is, these classes are essentially the same—only the type they work with is different.

This gives us an idea. We could create a generic `Interval` class using a *variable-type* like so:

```
Struct Interval
{
	Interval();
	Interval(variable-type start, variable-type end);

	variable-type midpoint();

	variable-type a;
	variable-type b;
};

Interval::Interval()
{
	a = 0.0f;
	b = 0.0f;
}

Interval::Interval(variable-type start, variable-type end)
{
	a = start;
	b = end;
}
variable-type Interval::midpoint()
{
	// return the midpoint between a and b.
	return (a + b) * 0.5;
}
```

Then, if we could specify a *type-argument*, we could generate new `Interval` classes that worked the same way, but with different types. All we have to do is substitute the type-argument for the variable-type. For example, if we specified type `float` as the type argument (assume argument types are specified in angle brackets < >), then the following class would be generated:

```
Interval<float> ==
	struct Interval
	{
		Interval();
		Interval(float start, float end);

		float midpoint();

		float a;
		float b;
	};
```

Note: The previous two code boxes do not use actual C++ syntax (though it is similar); pseudo-code was used to illustrate the idea of how template classes work.

This behavior is exactly what template classes allow us to do. Returning to `std::vector`, when we specify the type in the angle brackets, we instruct the compiler to create a vector class based on the specified type by substituting the type-argument (type in angle brackets) into the type-variables of the template vector class.

10.1.1 Class Template Definition

Now that we know what we would use templates for and the basic idea behind how they work, let us examine the *actual* C++ template syntax. Here is how we would define a template `Interval` class in C++:

```
template <typename T>
struct Interval
{
	Interval();
	Interval(T start, T end);

	T midpoint();

	T a;
	T b;
};
```

The first line, `template <typename T>` indicates that the class is a template class. The parameter inside the angle brackets `<typename T>` denotes a variable-type name—in this case, we named the variable-type, `T`. Later, when we want to instantiate an `Interval` of, say, `int`s, the compiler will substitute `int` everywhere there is a `T`.

10.1.2 Class Template Implementation

There is some special syntax required when implementing the methods of a template class. In particular, we must prefix the method with the `template <typename T>` syntax, and refer to the class name as `ClassName<T>`. The following shows how we would implement the methods of `Interval`:

```
template <typename T>
Interval<T>::Interval()
{
	a = T(); // Initialize with default
	b = T(); // constructors of whatever type.
}

template <typename T>
Interval<T>::Interval(T start, T end)
{
	a = start;
	b = end;
}

template <typename T>
T Interval<T>::midpoint()
{
	return (a + b) * 0.5; // return the midpoint between a and b.
}
```

Again, observe how we use `T` to refer to the type, which will eventually be substituted into the template.

Note: The template functionality of C++ is not easy for compiler writers to implement.

10.1.3 Class Template Instantiation

We have already instantiated template classes earlier in Module I of this course, without your even realizing it. For example, the following code generates two classes:

```
vector<int> intVec;
vector<float> floatVec;
```

It generates a `vector` class, where type `int` is substituted into the `typename` parameter, and it generates a second `vector` class, where type `float` is substituted into the `typename` parameter. The compiler generates these classes at compile time—after all, we specify the type-argument at compile time. Once the `int-vector` and `float-vector` classes are generated (remember, generating these classes is merely a matter of substituting the `typename` variable-type with the specified argument-type), we can create instances of them. That is what `intVec` and `floatVec` are—they are instances of the matching `vector` class generated by the compiler.

> **Note:** To further clarify, classes of a particular type are generated only once. That is, if you write:
>
> ```
> vector<float> f1;
> vector<float> f2;
> ```
>
> A `float` version of `vector` is not generated twice—it only needs to be generated once to instantiate any number of `float-vector` object instances.

With our template `Interval` class defined and implemented, we can instantiate objects of various types the same way we do with `std::vector`:

```
Interval<float> floatInterval(0.0f, 10.0f);
Interval<int>   intInterval(2, 4);

float fMidPt = floatInterval.midpoint();
int   iMidPt = intInterval.midpoint();

cout << "fMidPt = " << fMidPt << endl;
cout << "iMidPt = " << iMidPt << endl;
```

> **Note:** A class can contain more than one `typename`. For example, we can define a class like so:
>
> ```
> template <typename T1, typename T2>
> struct Foo
> {
>     T1 a;
>     T2 b;
> };
> ```
>
> When we instantiate a member, we have to specify two types:
>
> ```
> Foo<float, int> foo;
> ```
>
> The above substitutes `float` for `T1`, and `int` for `T2`.

10.2 Example: A `Table` Template Class

For additional template practice, we will create a template `Table` class. A `Table` is sort of like `std::vector`, except that instead of representing a resizable array, it represents a resizable 2D array—or matrix. This table class will be very useful, as there are many datasets in game development that are represented by a table (a game board/grid and 2D image immediately come to mind). We will want tables of many kinds of data types, so naturally we will make a template `Table` class.

10.2.1 Table Data

As we start off the design of our `Table` class, let us first discuss how we shall represent our table. For starters, our table size will not be fixed—it will have *m* rows and *n* columns. Since the number of rows and columns is variable, we must use dynamic memory. In this case, we use a pointer to an array of pointers to arrays.

This probably sounds confusing, and it is definitely a bit tricky at first, but it will be pretty intuitive once you think about it. We first have a pointer to an array, which we allocate dynamically. This array describes the rows of the table. Now each element in this array is also a pointer. These pointers, in turn, each point to another dynamic array, which forms the columns of the table. Figure 10.1 illustrates.

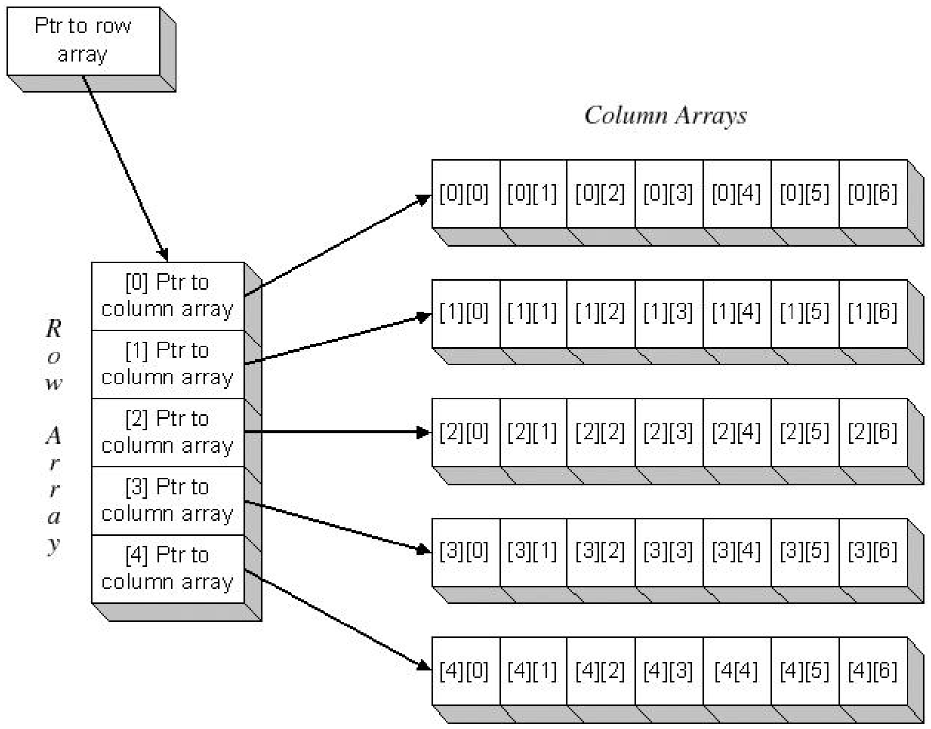


Figure 10.1: A 5x7 table represented with a pointer to an array of pointers to arrays. That is, we first have a pointer to a "row" array. Each element in this row array, in turn, points to a "column," thereby forming a 2D table.

Essentially, to have a variable sized 1D array, we needed one pointer to an array. To have a variable sized 2D array (variable in both rows and columns), we need a pointer to an array of pointers to arrays.

Furthermore, we will want to maintain the number of rows and columns our table has. This yields the following data members:

```
int mNumRows;
int mNumCols;
T** mDataMatrix;
```

The double star notation ** means "pointer to a pointer." This is how we can describe a pointer to an array of pointers to arrays.

10.2.2 Class Interface

As far as methods go, we need to be able to resize a table, construct tables, get the number of rows and columns a table has, provide access to entries in the table, and overload the assignment operator and copy constructor to provide deep copies since our class contains pointer data. The following definition provides these features via the interface:

```
template <typename T>
class Table
{
public:
      Table();
      Table(int m, int n);
      Table(int m, int n, const T& value);
      Table(const Table<T>& rhs);
      ~Table();

      Table<T>& operator=(const Table& rhs);
      T&        operator()(int i, int j);

      int numRows()const;
      int numCols()const;

      void resize(int m, int n);
      void resize(int m, int n, const T& value);

private:
      // Make private because this method should only be used
      // internally by the class.
      void destroy();

private:
      int mNumRows;
      int mNumCols;
      T** mDataMatrix;
};
```

The next three subsections discuss three non-trivial methods of Table. The implementations for the rest of the methods are shown in Section 10.2.6.

10.2.3 The `destroy` Method

The first method we will examine is the `destroy` method. This method is responsible for destroying the dynamic memory allocated by a `Table` object. What makes this method a bit tricky is the pointer to an array of pointers to arrays, which stores our table data. The method is implemented as follows:

```cpp
template <typename T>
void Table<T>::destroy()
{
      // Does the matrix exist?
      if( mDataMatrix )
      {
            // Iterate over each row i.
            for(int i = 0; i < _m; ++i)
            {
                  // Does the ith column array exist?
                  if(mDataMatrix[i] )
                  {
                        // Yes, delete it.
                        delete[]mDataMatrix[i];
                        mDataMatrix[i] = 0;
                  }
            }
            // Now delete the row-array.
            delete[] mDataMatrix;
            mDataMatrix = 0;
      }

      // Table was destroyed, so dimensions are zero.
      mNumRows = 0;
      mNumCols = 0;
}
```

Be sure to read the comments slowly and deliberately. First, we traverse over each row and delete the column array that exists in each element of the row array (See Figure 10.1). Once we have deleted the column arrays, we delete the row array.

10.2.4 The `resize` Method

In this section, we examine the `resize` method, which is relatively more complicated than the other methods of the `Table` class. This method is somewhat tricky because it handles the memory allocation of the pointer to an array of pointers to arrays; the method is presented below:

```cpp
template <typename T>
void Table<T>::resize(int m, int n, const T& value)
{
      // Destroy the previous data.
      destroy();

      // Save dimensions.
      mNumRows = m;
      mNumCols = n;

      // Allocate a row (array) of pointers.
      mDataMatrix = new T*[mNumRows];

      // Now, loop through each pointer in this row array.
      for(int i = 0; i < mNumRows; ++i)
      {
            // And allocate a column (array) for the ith row to build
            // the table.
            mDataMatrix[i] = new T[mNumCols];

            // Now loop through each element in this row[i]
            // and copy 'value' into it.
            for(int j = 0; j < mNumCols; ++j)
                  mDataMatrix[i][j] = value;
      }
}
```

Again, be sure to read the comments slowly and deliberately. The method takes three parameters: the first two specify the dimensions of the table; that is *m* by *n*. The third parameter is a default value to which we initialize all the elements of the table.

The very first thing the method does is call the `destroy` method, as discussed in Section 10.2.3. This makes one thing immediately clear -- we lose the data in the table whenever we call `resize`. If you want to maintain the data, you will need to copy it into a separate table for temporary storage, resize the current table, and then copy the data in the temporary storage back into the newly resized table.

After the resize method, we simply save the new dimensions. The next line:

```cpp
// Allocate a row (array) of pointers.
mDataMatrix = new T*[mNumRows];
```

allocates an array of pointers (see the row in Figure 10.1). What we must do now is iterate over each of these pointers and allocate the column array:

```cpp
// Now, loop through each pointer in this row array.
for(int i = 0; i < mNumRows; ++i)
{
      // And allocate a column (array) for the ith row to build
      // the table.
      mDataMatrix[i] = new T[mNumCols];
```

After we have done this, we iterate over each column in the i-th row, and initialize the table entry with `value`:

```
// Now loop through each element in this row[i]
// and copy 'value' into it.
for(int j = 0; j < mNumCols; ++j)
      mDataMatrix[i][j] = value;
```

On the whole, it is not too complex if you break the method down into parts, and use Figure 10.1 as a guide.

10.2.5 The Overloaded Parenthesis Operator

The final method we wish to discuss is the overloaded parenthesis operator. Although the implementation is straightforward, we draw attention to this method because we have not overloaded the parenthesis operator before.

```
template <typename T>
T& Table<T>::operator()(int i, int j)
{
      return mDataMatrix[i][j];
}
```

Because C++ does not have a double bracket operator [][], which we can overload, we cannot index into a table as we would a 2D array. We instead overload the parenthesis operator to take two arguments, and instruct the method to use these arguments to index into the internal 2D array, in order to return the i-th table entry. This allows us to index into a table "almost" like the double bracket operator would:

```
Table<float> myTable(4, 4);

MyTable(1, 1) = 2.0f;  // Access entry [1][1]
MyTable(3, 0) = -1.0f; // Access entry [3][0]
```

10.2.6 The Table Class

For reference, we provide the entire `Table` definition and implementation together here. Note in particular how the definition and implementation are in the same file—this is necessary for templates. You will be asked to use this class in one of the exercises, so be sure to give it a thorough examination.

```
// Table.h
#ifndef TABLE_H
#define TABLE_H

template <typename T>
class Table
{
```

```
public:
      Table();
      Table(int m, int n);
      Table(int m, int n, const T& value);
      Table(const Table<T>& rhs);
      ~Table();

      Table<T>& operator=(const Table& rhs);
      T&        operator()(int i, int j);

      int numRows()const;
      int numCols()const;

      void resize(int m, int n);
      void resize(int m, int n, const T& value);

private:
      // Make private because this method should only be used
      // internally by the class.
      void destroy();

private:
      int mNumRows;
      int mNumCols;
      T** mDataMatrix;
};

template <typename T>
Table<T>::Table<T>()
{
      mDataMatrix = 0;
      mNumRows    = 0;
      mNumCols    = 0;
}

template <typename T>
Table<T>::Table<T>(int m, int n)
{
      mDataMatrix = 0;
      mNumRows    = 0;
      mNumCols    = 0;
      resize(m, n, T());
}

template <typename T>
Table<T>::Table<T>(int m, int n, const T& value)
{
      mDataMatrix = 0;
      mNumRows    = 0;
      mNumCols    = 0;
      resize(m, n, value);
}

template <typename T>
Table<T>::Table<T>(const Table<T>& rhs)
{
      mDataMatrix = 0;
      mNumRows    = 0;
      mNumCols    = 0;
      *this       = rhs;
}

template <typename T>
Table<T>::~Table<T>()
{
      // Destroy any previous dynamic memory.
      destroy();
}

template <typename T>
Table<T>& Table<T>::operator=(const Table& rhs)
{
      // Check for self assignment.
      if( this == &rhs ) return *this;

      // Reallocate the table based on rhs info.
      resize(rhs.mNumRows, rhs.mNumCols);

      // Copy the entries over element-by-element.
      for(int i = 0; i < mNumRows; ++i)
            for(int j = 0; j < mNumCols; ++j)
                  mDataMatrix[i][j] = rhs.mDataMatrix[i][j];

      // return a reference to *this so we can do chain
      // assignments: x = y = z = w = ...
      return *this;
}

template <typename T>
T& Table<T>::operator()(int i, int j)
{
      return mDataMatrix[i][j]; // return the ijth table entry.
}

template <typename T>
int Table<T>::numRows()const
{
      return mNumRows;          // Return the number of rows.

}

template <typename T>
int Table<T>::numCols()const
{
      return mNumCols;          // Return the number of columns.
}

template <typename T>
void Table<T>::resize(int m, int n)
{
      // Call resize and use default constructor T()
      // as 'value'.
      resize(m, n, T());
}
```

```
template <typename T>
void Table<T>::resize(int m, int n, const T& value)
{
      // Destroy the previous data.
      destroy();

      // Save dimensions.
      mNumRows = m;
      mNumCols = n;

      // Allocate a row (array) of pointers.
      mDataMatrix = new T*[mNumRows];

      // Now, loop through each pointer in this row array.
      for(int i = 0; i < mNumRows; ++i)
      {
            // And allocate a column (array) to build the table.
            mDataMatrix[i] = new T[mNumCols];

            // Now loop through each element in this row[i]
            // and copy 'value' into it.
            for(int j = 0; j < mNumCols; ++j)
                  mDataMatrix[i][j] = value;
      }
}
template <typename T>
void Table<T>::destroy()
{
      // Does the matrix exist?
      if( mDataMatrix )
      {
            for(int i = 0; i < _m; ++i)
            {
                  // Does the ith row exist?
                  if(mDataMatrix[i] )
                  {
                        // Yes, delete it.
                        delete[]mDataMatrix[i];
                        mDataMatrix[i] = 0;
                  }
            }
            // Delete the row-array.
            delete[] mDataMatrix;
            mDataMatrix = 0;
      }

      mNumRows = 0;
      mNumCols = 0;
}

#endif // TABLE_H
```

10.3 Function Templates

Template functions extend the idea of template classes. Sometimes you will have a function which performs an operation that can be performed on a variety of data types. For example, consider a search function; naturally, we will want to search arrays of all kinds of data types. It would be cumbersome to implement a search function for each type, especially since the implementation would essentially be the same—only the types would be different. So a search function is a good candidate for a template design.

The general syntax of a function template is as follows:

```
template <typename T>
return-type FunctionName(parameter-list...)
{
  // Function body
}
```

Next we see two template function examples, one that performs a linear search and a second template function that prints an array.

```
template <typename T>
int LinearSearch(T dataArray[], int arraySize, T searchItem)
{
      // Search through array.
      for(int i = 0; i < arraySize; ++i)
      {
            // Find it?
            if( dataArray[i] == searchItem )
            {
                  // Yes, return the index we found it at.
                  return i;
            }
      }

      // Did not find it, return -1.
      return -1;
}

template <typename T>
void Print(T data[], int arraySize)
{
      for(int i = 0; i < arraySize; ++i)
            cout << data[i] << " ";
      cout << endl;
}
```

The only operator `LinearSearch` uses on the type `T` is the equals == operator. Thus, any type we use with `LinearSearch` (i.e., substitute in for `T`) must have that operator defined (i.e., overloaded). All the built-in types have the equals operator defined, so they pose no problem, and similarly with std::string. But if we wish to use user-defined types (i.e., classes) we must overload the equals operator if we want to be able to use LinearSearch with them.

Similarly, the only operator Print uses on T is the insertion operator. Thus, any type we use with Print (i.e., substitute in for T) must have that operator defined (i.e., overloaded). All the built-in types have the insertion operator defined, so they pose no problem, and similarly with std::string. But if we wish to use user-defined types (i.e., classes) we must overload the insertion operator if we want to be able to use Print with them.

10.3.1 Example Program

What follows is a sample program illustrating our template functions. We set up two arrays, a std::string array, and an int array. We then use the Print function to print both of these arrays, and we allow the user to search these arrays via LinearSearch. The key idea here is that we are using the same template function for different types—that is, these functions are generic and work on any types that implement the stated conditions (overloads the less than operator/insertion operator).

Program 10.1: Template Functions.

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

template <typename T>
int LinearSearch(T dataArray[], int arraySize, T searchItem)
{
      // Search through array.
      for(int i = 0; i < arraySize; ++i)
      {
            // Find it?
            if( dataArray[i] == searchItem )
            {
                  // Yes, return the index we found it at.
                  return i;
            }
      }

      // Did not find it, return -1.
      return -1;
}

template <typename T>
void Print(T data[], int arraySize)
{
      for(int i = 0; i < arraySize; ++i)
            cout << data[i] << " ";
      cout << endl;
}
```

```
int main()
{
      string sArray[8] =
      {
            "delta",
            "lambda",
            "alpha",
            "beta",
            "pi",
            "omega",
            "epsilon",
            "phi"
      };
      int numStrings = 8;

      int iArray[14] = {7,3,32,2,55,34,6,13,29,22,11,9,1,5};
      int numInts = 14;

      int index = -1;

      bool quit = false;
      while( !quit )
      {
            //========================================
            // String search.

            Print(sArray, numStrings);

            string str = "";
            cout << "Enter a string to search for: ";
            cin >> str;

            index = LinearSearch(sArray, numStrings, str);

            if( index != -1 )
                  cout << str << " found at index " << index << endl;
            else
                  cout << str << " not found." << endl;
            cout << endl;

            //========================================
            // Int search.

            Print(iArray, numInts);

            int integer = 0;
            cout << "Enter an integer to search for: ";
            cin >> integer;

            index = LinearSearch(iArray, numInts, integer);

            if( index != -1 )
                cout << integer << " found at index " << index<<endl;
            else
                cout << integer << " not found." << endl;
            cout << endl;
            //========================================
            // Search again?

            char input = '\0';
            cout << "Quit? (y)/(n)";
            cin >> input;

            if( input == 'y' || input == 'Y' )
                  quit = true;
      }
}
```

Program 10.1 Output

```
delta lambda alpha beta pi omega epsilon phi
Enter a string to search for: temp
temp not found.

7 3 32 2 55 34 6 13 29 22 11 9 1 5
Enter an integer to search for: 32
32 found at index 2

Quit? (y)/(n)n
delta lambda alpha beta pi omega epsilon phi
Enter a string to search for: beta
beta found at index 3

7 3 32 2 55 34 6 13 29 22 11 9 1 5
Enter an integer to search for: 22
22 found at index 9

Quit? (y)/(n)n
delta lambda alpha beta pi omega epsilon phi
Enter a string to search for: phi
phi found at index 7

7 3 32 2 55 34 6 13 29 22 11 9 1 5
Enter an integer to search for: 0
0 not found.

Quit? (y)/(n)y
Press any key to continue
```

10.4 Summary

Sometimes we would like to create several versions of a class, where the only difference is the data types involved. Template classes enable us to do this. Specifically, we create a generic class built using generic type-variables, where we can then substitute real concrete types into these type-variables to form new classes. The compiler forms these new classes at compile time. Template classes reduce the number of classes that the programmer must explicitly write, and therefore, they save time and reduce program size/complexity.

1. Template functions are generic and can work on any data type that implements the operations used inside the function body. In this way, we can write one generic function that works for several types, instead of a unique function for each type. Template functions reduce the number of functions that the programmer must explicitly write, and therefore, they save time and reduce program size/complexity.

2. Use templates for classes and functions, which can be generalized to work with more than one type. `std::vector`, `Table` and `LinearSearch` are examples of classes and functions which can be generalized in this fashion. The following exercises illustrate a few more examples, and Chapter 13, on the STL, demonstrates a very sophisticated generic programming design.

10.5 Exercises

10.5.1 Template Array Class

Reconsider the exercise from Section 7.9.2, where you were instructed to implement a resizable `FloatArray` class:

```
// FloatArray.h
#ifndef FLOAT_ARRAY_H
#define FLOAT_ARRAY_H

class FloatArray
{
public:
      // Create a FloatArray with zero elements.
      FloatArray();

      // Create a FloatArray with 'size' elements.
      FloatArray(int size);

      // Create a FloatArray from another FloatArray--
      // be sure to prevent memory leaks!
      FloatArray(const FloatArray& rhs);
      // Free dynamic memory.
      ~FloatArray();

      // Define how a FloatArray shall be assigned to
      // another FloatArray--be sure to prevent memory
      // leaks!
      FloatArray& operator=(const FloatArray& rhs);

      // Resize the FloatArray to a new size.
      void resize(int newSize);

      // Return the number of elements in the array.
      int size();

      // Overload bracket operator so client can index
      // into FloatArray objects and access the elements.
      float& operator[](int i);

private:
      float* mData; // Pointer to array of floats (dynamic memory).
      int    mSize; // The number of elements in the array.
};

#endif // FLOAT_ARRAY_H
```

The problem with this class is that it only works with the `float` type. However, we would naturally want resizable arrays of any type of object. Generalize the class with templates so that it can be used with any type. Test your program by creating a `std::string Array`, and an `int Array`, from the generic template class. Call the template class `Array`.

10.5.2 Template Bubble Sort Function

Reconsider the exercise from Section 3.7.8, where you were instructed to implement a Bubble Sort function like so:

```
      void BubbleSort(int data[], int n);
```

The problem with this function is that it only works with the `int` type; however, we would naturally want to sort arrays of any type of object. Generalize the function with templates so that it can be used with any type that implements the operations (e.g., less than, greater than, etc.) needed to implement the bubble sort algorithm. Test your program by sorting a `std::string` array, and an `int` array, with the template `BubbleSort` function.

10.5.3 Table Driver

Rewrite the exercise from Section 2.7.5, but instead use the `Table` class as discussed in Section 10.2, instead of 2D arrays.

Chapter 11

Errors and Exception Handling

Introduction

Throughout most of the previous chapters, we have assumed that all of our code was designed and implemented correctly and that the results could be anticipated. For example, we assumed that the user entered the expected kind of input, such as a string when a string was expected or a number when a number was expected. Additionally, we assumed that the arguments we passed into function parameters were valid. But this may not always be true. For example, what happens if we pass in a negative integer into a factorial function, which expects an integer greater than or equal to zero? Whenever we allocated memory, we assumed that the memory allocation succeeded, but this is not always true, because memory is finite and can run out. While we copied strings with `strcpy`, we assumed the destination string receiving the copy had enough characters to store a copy, but what would happen if it did not?

It would be desirable if everything worked according to plan; however, in reality things tend to obey Murphy's Law (which paraphrased says "if anything can go wrong, it will"). In this chapter, we spend some time getting familiar with several ways in which we can catch and handle errors. The overall goal is to write code that is easy to debug, can (possibly) recover from errors, and exits gracefully with useful error information if the program encounters a fatal error.

Chapter Objectives

- Understand the method of catching errors via function return codes, and an understanding of the shortcomings of this method.
- Become familiar with the concepts of exception handling, its syntax, and its benefits.
- Learn how to write assumption verification code using asserts.

11.1 Error Codes

The method of using error codes is simple. For every function or method we write, we have it return a value which signifies whether the function/method executed successfully or not. If it succeeded then we return a code that signifies success. If it failed then we return a predefined value that specifies where and why the function/method failed.

Let us take a moment to look at a real world example of an error return code system. In particular, we will look at the system used by DirectX (a code library for adding graphics, sound, and input to your applications). Consider the following DirectX function:

```
HRESULT WINAPI D3DXCreateTextureFromFile(
    LPDIRECT3DDEVICE9 pDevice,
    LPCTSTR pSrcFile,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```

Do not worry about what this function does or the data types this function uses, which you are not familiar with.

This function has a return type `HRESULT`, which is simply a numeric code that identifies the success of the function or an error. For instance, `D3DXCreateTextureFromFile` can return one of the following return codes, which are defined numerically (i.e., the symbolic name represents a number). Which one it returns depends upon what happens inside the function.

- `D3D_OK`: This return code means that the function executed completely successfully.
- `D3DERR_NOTAVAILABLE`: This return code means that the hardware cannot create a texture; that is, texture creation is an "unavailable" feature. This is a failure code.
- `D3DERR_OUTOFVIDEOMEMORY`: This return code means that there is not enough video memory to put the texture in. This is a failure code.
- `D3DERR_INVALIDCALL`: This return code means that the arguments passed into the parameters are invalid. For example, you may have passed in a null pointer when the function expects a valid pointer. This is a failure code.
- `D3DXERR_INVALIDDATA`: This return code means the source data (that is, the texture file) is not valid. This error could occur if the file is corrupt, or we specified a file that is not actually a texture file. This is a failure code.
- `E_OUTOFMEMORY`: This return code means there was not enough available memory to perform the operation. This is a failure code.

By examining the return codes from functions that return error codes, we can figure out if an error occurred, what potentially caused the error, and then respond appropriately. For example, we can write the following code:

```
HRESULT hr = D3DXCreateTextureFromFile([...]);

// Did an error occur?
if( hr != D3D_OK )
{
      // Yes, find our which specific error
      if( hr == D3DERR_NOTAVAILABLE )
      {
            DisplayErrorMsg("D3DERR_NOTAVAILABLE");
            ExitProgram();
      }
      else if( hr == D3DERR_OUTOFVIDEOMEMORY )
      {
            DisplayErrorMsg("D3DERR_OUTOFVIDEOMEMORY");
            ExitProgram();
      }
      else if( hr == D3DERR_INVALIDCALL )
      {
            DisplayErrorMsg("D3DERR_INVALIDCALL");
            ExitProgram();
      }
      else if( hr == D3DXERR_INVALIDDATA )
      {
            DisplayErrorMsg("D3DXERR_INVALIDDATA");
            ExitProgram();
      }
      else if( hr == E_OUTOFMEMORY )
      {
            DisplayErrorMsg("E_OUTOFMEMORY");
            ExitProgram();
      }
}
```

Here we simply display the error code to the user and then exit the program. Note that `DisplayErrorMsg` and `ExitProgram` are functions you would have to implement yourself. They are not part of the standard library.

11.2 Exception Handling Basics

One of the shortcomings of error codes is that for a single function call, we end up writing a lot of *error handling* code, thereby bloating the size of the program. For example, at the end of the previous section we saw that there were many lines of error handling code for a single function call. The problem becomes worse on a larger scale:

FunctionCallA();

// Handle possible error codes

FunctionCallB();

// Handle possible error codes

FunctionCallC();

// Handle possible error codes

...

Such a style of mixing error-handling code in with non-error-handling code becomes so cumbersome that it is seldom religiously followed throughout a program, and therefore, the program becomes unsafe. C++ provides an alternative error-handling solution called **exception handling.**

Exception handling works like this: in a segment of code, if an error or something unexpected occurs, the code **throws** an **exception.** An exception is represented with a class object, and as such, can do anything a normal C++ class object can do. Once an exception has been thrown, the call stack unwinds (a bit like returning from functions) until it finds a **catch** block that handles the exception. Let us look at an example:

Program 11.1: Exception Handling.

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

class DivideByZero
{
public:
      DivideByZero(const string& s);

      void errorMsg();

private:
      string mErrorMsg;
};

DivideByZero::DivideByZero(const string& s)
{
      mErrorMsg = s;
}

void DivideByZero::errorMsg()
{
      cout << mErrorMsg << endl;
}

float Divide(float numerator, float denominator)
{
      if( denominator == 0.0f )
            throw DivideByZero("Divide by zero: result undefined");

      return numerator / denominator;
}

int main()
{
      try
      {
            float quotient = Divide(12.0f, 0.0f);
            cout << "12 / 0 = " << quotient << endl;
      }
      catch(DivideByZero& e)
      {
            e.errorMsg();
      }
}
```

Program 11.1 Output

```
Divide by zero: result undefined
Press any key to continue
```

The very first thing we do is define an exception class called `DivideByZero`. Remember that an exception class is just like an ordinary class, except that we use instances of it to represent exceptions.

The next item of importance is in the `Divide` function. This function tests for a "divide by zero" and if it occurs, we then construct and `throw` a `DivideByZero` object exception. Finally, in the `main` function, in order to *catch* an exception we must use a try-catch block. In particular, we wrap the code that can potentially throw an exception in the `try` block, and we write the exception handling code in the `catch` block. Note that the `catch` block takes an object. This object is the exception we are looking to catch and handle.

It is definitely possible, and quite common, that a function or method will throw more than one kind of exception. We can list catch statements so that we can handle the different kinds of exceptions:

```
	try
	{
		SomeFunction();
	}
	catch(LogicError& logic)
	{
		// Handle logic error exception
	}
	catch(OutOfMemory& outOfMem)
	{
		// Handle out of memory exception
	}
	catch(InvalidData& invalid)
	{
		// Handle invalid data
	}
```

In addition to a chain of catches, we can "catch any exception" by specifying an ellipses argument:

```
	try{
		SomeFunction();
	}
	catch(...){
		// Generic error handling code
	}
```

We can state two immediate benefits of exception handling.

1. The error-handling code (i.e., the catch block) is not intertwined with non-error-handling code; in other words, we move all error handling code into a catch block. This is convenient from an organizational standpoint.
2. We need not handle a thrown exception immediately; rather the stack will unwind until it finds a catch block that handles the exception. This is convenient because, as functions can call other functions, which call other functions, and so on, we do not want to have error handling code after every function/method. Instead, with exceptions we can catch the exception at, say, the top level function call, and any exception thrown in the inner function calls will eventually percolate up to the top function call which can catch the error.

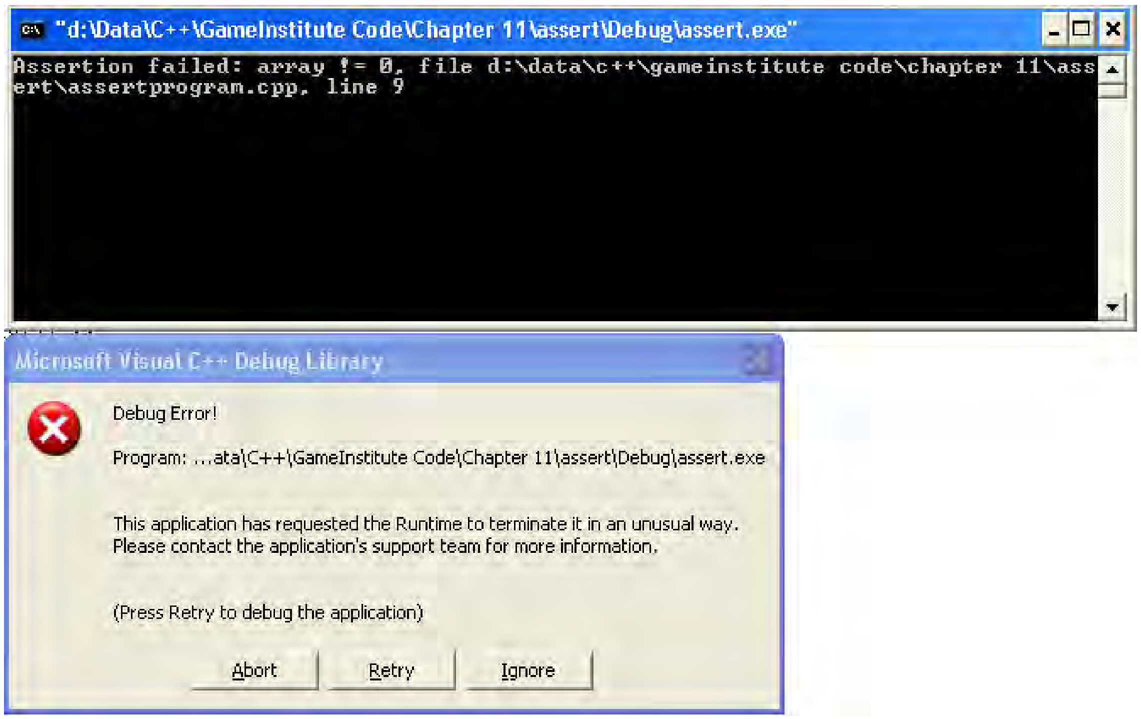
As a final note, be aware that this section merely touched on the basics of exception handling, and there are many more details and special situations that can exist. Also note that the functionality of exception handling is not free, and introduces some (typically minor) performance overhead.

11.3 Assert

In general, your functions and methods make certain assumptions. For example, a "print array" function might assume that the program passes a valid array argument. However, it is possible that you might have forgotten to initialize an array, and consequently, passed in a null pointer to the "print array" function, thus causing an error. We could handle this problem with a traditional error handling system as described in the previous two sections. However, such errors should not be occurring as the program reaches completion. That is, if you are shipping a product that has a null pointer because you forgot to initialize it then you should not be shipping the product to begin with. Error handling should be for handling errors that are generally beyond the control of the program, such as missing or corrupt data resources, incompatible hardware, unavailable memory, flawed input data, and so on.

Still, for debugging purposes, it is very convenient to have self-checks littered throughout the program to ensure certain assumptions are true, such as the validity of a pointer. However, based on the previous argument, we should not need these self-checks once we have agreed that the software is complete. In other words, we want to remove these checks in the final version of the program. This is where `assert` comes in.

To use the `assert` function you must include the standard library <cassert>. The `assert` function takes a single boolean expression as an argument. If the expression is true then the assertion passes; what was asserted is true. Conversely, if the expression evaluates to false then the assertion fails and a dialog box like the one depicted in Figure 11.1 shows up, along with an assertion message in the console window:



"d:\Data\C++\GameInstitute Code\Chapter 11\assert\Debug\assert.exe"

```
Assertion failed: array != 0, file d:\data\c++\gameinstitute code\chapter 11\ass
ert\assertprogram.cpp, line 9
```

Microsoft Visual C++ Debug Library

Debug Error!

Program: ...ata\C++\GameInstitute Code\Chapter 11\assert\Debug\assert.exe

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

(Press Retry to debug the application)

Figure 11.1: Assert message and dialog.

The information the assertion prints to the console is quite useful for debugging; it displays the condition that failed, and it displays the source code file and line number of the condition that failed.

The key fact about `asserts` is that they are only used in the debug version of a program. When you switch the compiler into "release mode" the assert functions are filtered out. This satisfies what we previously sought when we said: "[…] we want to remove these checks [`asserts`] in the final version of the program."

To conclude, let us look at a complete, albeit simple, program that uses `asserts`.

Program 11.2: Using assert.

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

void PrintIntArray(int array[], int size)
{
      assert( array != 0 );   // Check for null array.
      assert( size >= 1 );    // Check for a size >= 0.

      for(int i = 0; i < size; ++i)
            cout << array[i] << " ";

      cout << endl;
}
int main()
{
      int* array = 0;

      PrintIntArray(array, 10);
}
```

The function `PrintIntArray` makes two assumptions:

1) The array argument points to something (i.e., it is not null)
2) The array has a size of at least one element.

Both of these assumptions are asserted in code:

```
// Check for null array.
assert( array != 0 );

// Check for a size >= 0.
assert( size >= 1 );
```

Because we pass a null pointer into `PrintIntArray`, in `main`, the assert fails and the dialog box and assert message as shown in Figure 11.1 appear. As an exercise, correct the problem and verify that the assertion succeeds.

11.4 Summary

1. When using error codes to handle errors for every function or method we write, we have it return a value, which signifies whether the function/method executed successfully or not. If it succeeded then we return a code that signifies success. If it failed then we return a predefined value that specifies where and why the function/method failed. One of the shortcomings of error codes is that for a single function call, we end up writing much more error handling code, thereby bloating the size of the program.

2. Exception handling works like this: in a segment of code, if an error or something unexpected occurs, the code throws an exception. An exception is represented with a class object, and as such, can do anything a normal C++ class object can do. Once an exception has been thrown, the stack unwinds (a bit like returning from functions) until it finds a catch block that handles the exception. One of the benefits of exception handling is that the error-handling code (i.e., the catch block) is not intertwined with non-error-handling code; in other words, we move all error handling code into a catch block. This is convenient from an organizational standpoint. Another benefit of exception handling is that we need not handle a thrown exception immediately; rather the stack will unwind until it finds a catch block that handles the exception. This is convenient because, as functions can call other functions, which call other functions, and so on, we do not want to have error handling code after every function/method. Instead, with exceptions we can catch the exception at the top level function call, and any exception thrown in the inner function calls will eventually percolate up to the top function call which can catch the error. Be aware that the functionality of exception handling is not free, and introduces some performance overhead.

3. To use the `assert` function, you must include the standard library <cassert>. The `assert` function takes a single boolean expression as an argument. If the expression is true then the assertion passes; what was asserted is true. Conversely, if the expression evaluates to false then the assertion fails and a message is displayed along with a dialog box. The key fact about `asserts` is that they are only used in the debug version of a program. When you switch the compiler into "release mode" the assert functions are filtered out.

11.5 Exercises

11.5.1 Exception Handling

This is an open-ended exercise. You are to come up with some situation in which an exception could be thrown. You then are to create an exception class representing that type of exception. Finally, you are to write a program, where such an exception is thrown, and you should catch and handle the exception. It does not have to be fancy. The goal of this exercise is for you to simply go through the process of creating an exception class, throwing an exception, and catching an exception, at least once.

Chapter 12

Number Systems; Data Representation; Bit Operations

Introduction

For the most part, with the closing of the last chapter, we have concluded covering the core C++ topics. As far as C++ is concerned, all we have left is a tour of some additional elements of the standard library, and in particular, the STL (standard template library). But first, we will take a detour and become familiar with data at a lower level; more specifically, instead of looking at `char`s, `int`s, `float`s, etc., we will look at the individual bits that make up these types.

Chapter Objectives

- Learn how to represent numbers with the binary and hexadecimal numbering systems, how to perform basic arithmetic in these numbering systems, and how to convert between these numbering systems as well as the base ten numbering system.

- Gain an understanding of how the computer describes intrinsic C++ types internally.

- Become proficient with the various binary operations.

- Become familiar with the way in which floating-point numbers are represented internally.

12.1 Number Systems

We are all experienced with working in a **base** (or **radix**) ten number system, called the **decimal** number system, where each digit has ten possible values ranging from 0-9. However, after some reflection, it is not difficult to understand that selecting a base ten number system is completely arbitrary. Why ten? Why not two, eight, sixteen, and so on? You might suggest that ten is a convenient number, but it is only convenient because you have presumably worked in it your whole life.

Throughout this chapter, we will become familiar with two other specific number systems, particularly the base two **binary system** and the base sixteen **hexadecimal** systems. We choose these systems because they are convenient when working with computers, as will be made clear in Section 12.4. These new numbering systems may be cumbersome to work with at first, but after a bit of practice you will become very fast at manipulating numbers in these systems.

In order to distinguish between numbers in different systems, we will adopt a subscript notation when the context might not be clear:

- n_{10} : The number n is in the decimal number system.
- n_2 : The number n is in the binary number system.
- n_{16} : The number n is in the hexadecimal number system.

12.1.1 The Windows Calculator

Microsoft Windows ships with a calculator program that can work in binary, octal (a base eight number system we will not discuss), decimal, and hexadecimal. In addition to the arithmetic operations, the program allows you switch (i.e., convert) from one system to the other by simply selecting a radio button. Furthermore, the calculator program can even do logical bit operations AND, OR, NOT, XOR (exclusive or), which we discuss in Section 12.5. Figures 12.1*a*-12.1*e* give a basic overview of the program.

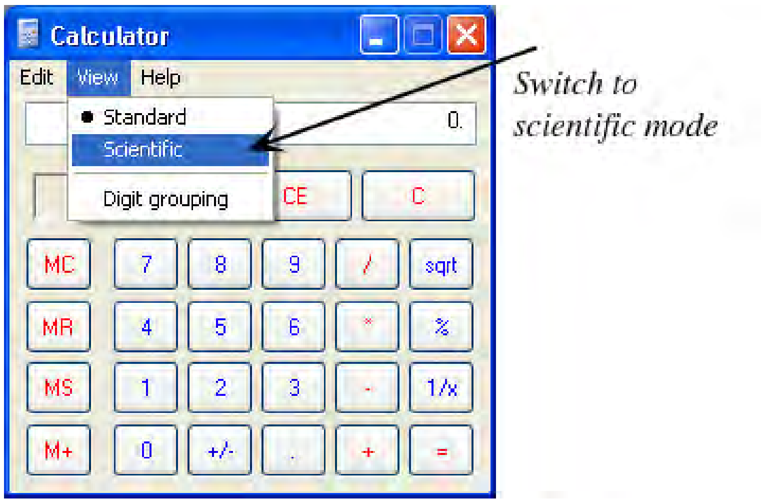


Figure 12.1*a*: The calculator program in "standard" view. To use the other number systems we need to switch to "scientific" view.

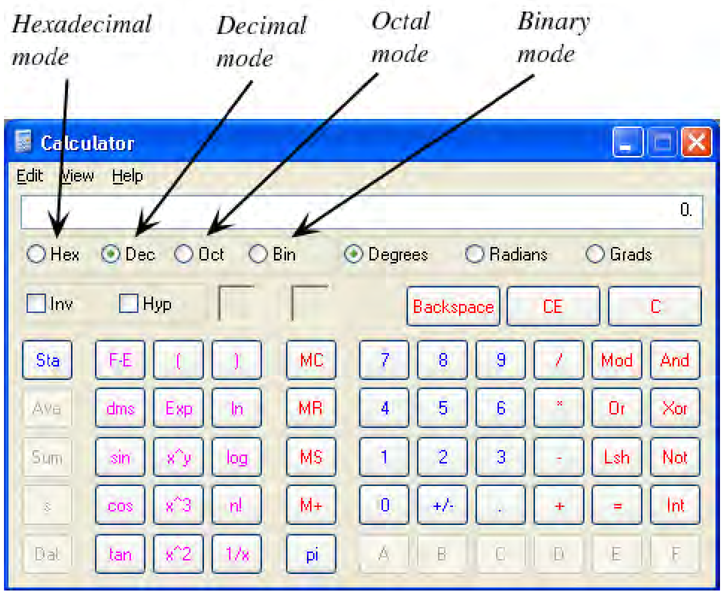


Figure 12.1*b*: The calculator program in "scientific" view. Notice the four radio buttons, which allow us to switch numbering systems any time we want.

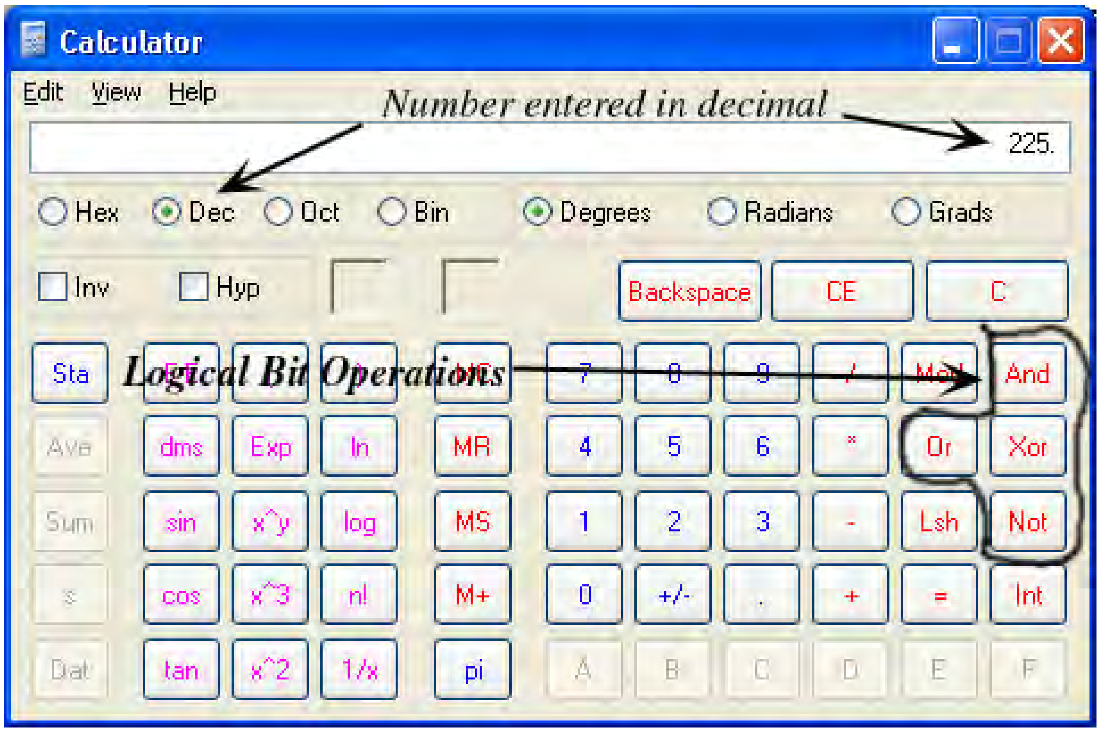

Calculator
Edit View Help
Number entered in decimal
225.
Hex Dec Oct Bin Degrees Radians Grads
Inv Hyp
Backspace CE C
Logical Bit Operations
And Or Xor Not Lsh Mod Int

Figure 12.1*c*: Here we enter a number in the decimal system. We also point out the logical bit operations AND, OR, NOT, and XOR (exclusive or), which are discussed in Section 12.5.

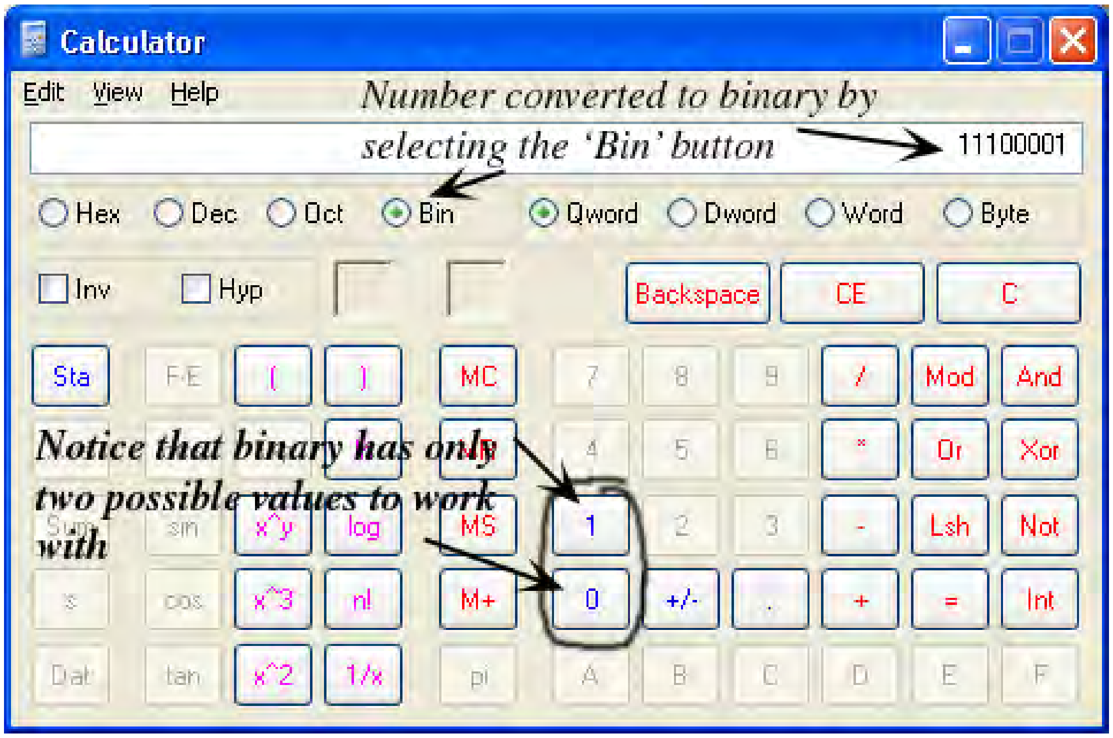

Calculator
Edit View Help
Number converted to binary by selecting the 'Bin' button
11100001
Hex Dec Oct Bin Qword Dword Word Byte
Notice that binary has only two possible values to work with
1
0

Figure 12.1*d*: Here we switched from decimal to binary. That is, "11100001" in binary is "225" in decimal. So we can input a number in any base we want and then convert that number into another base by simply selecting the radio button that corresponds to the base we want to convert to. In addition, we also point out that the program automatically limits you to two possible values in binary.

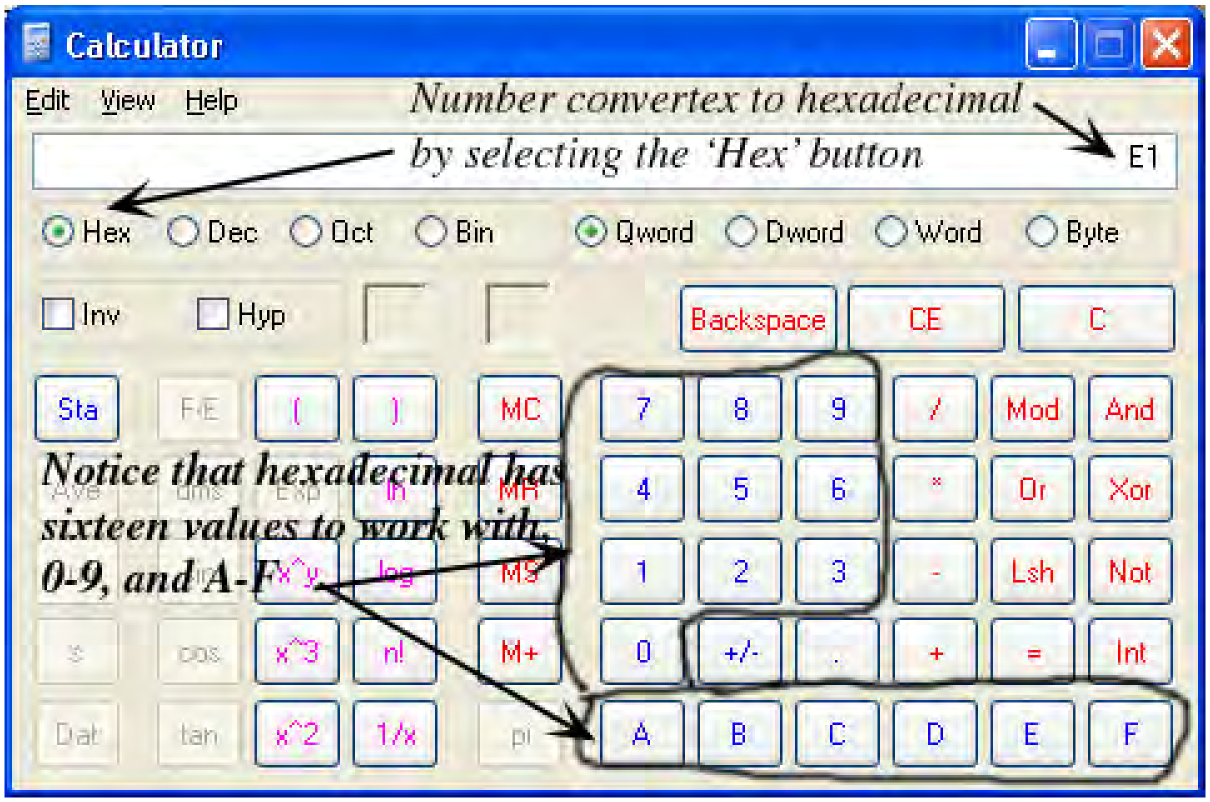

Calculator
Edit View Help
Number convertex to hexadecimal by selecting the 'Hex' button
E1
Hex Dec Oct Bin
Qword Dword Word Byte
Inv Hyp
Backspace CE C
Notice that hexadecimal has sixteen values to work with, 0-9, and A-F
Sta F-E ( ) MC 7 8 9 / Mod And
MR 4 5 6 * Or Xor
x^y log MS 1 2 3 - Lsh Not
x^3 n! M+ 0 +/- . + = Int
Dat tan x^2 1/x pi A B C D E F

Figure 12.1*e*: Here we switched from binary to hexadecimal. That is, "11100001" in binary is "E1" in hexadecimal, which is "225" in decimal. We also point out that there are now sixteen possible hexadecimal numbers to work with. Do not worry too much about this now; we discuss hexadecimal in Section 12.3.

You can use this calculator program to check your work for the exercises.

12.2 The Binary Number System

12.2.1 Counting in Binary

The binary number system is a base two number system. This means that only two possible values per digit exist; namely 0 and 1. In binary, we count similarly to the way we count in base ten. For example, in base ten we count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and then, because we have run out of values, to count ten we reset back to zero and add a new digit which we set to one, to form the number 10.

In binary, the concept is the same except that we only have two possible values per digit. Thus, we end up having to "add" a new digit much sooner than we do in base ten. We count 0, 1, and then we already need to add a new digit. Let us count a few numbers one-by-one in binary to get the idea. For each number, we write the equivalent decimal number next to it.

$0_2 = 0_{10}$
$1_2 = 1_{10}$

There is no ‘2’ in binary—only 0 and 1—so at this point, we must add a new digit:

$10_2 = 2_{10}$
$11_2 = 3_{10}$

Again, we have run out of values in base two, so we must add another new digit to continue:

$100_2 = 4_{10}$
$101_2 = 5_{10}$
$110_2 = 6_{10}$
$111_2 = 7_{10}$
$1000_2 = 8_{10}$
$1001_2 = 9_{10}$
$1010_2 = 10_{10}$
$1011_2 = 11_{10}$
$1100_2 = 12_{10}$
$1101_2 = 13_{10}$
$1110_2 = 14_{10}$
$1111_2 = 15_{10}$
$10000_2 = 16_{10}$

It takes time to become familiar with this system as it is easy to confuse the binary number 1111_2 for the decimal number 1111_{10} until your brain gets used to distinguishing between the two systems.

12.2.2 Binary and Powers of 2

An important observation about the binary number system is that each “digit place” corresponds to a power of 2 in decimal:

$1_2 = 2^0_{10} = 1_{10}$
$10_2 = 2^1_{10} = 2_{10}$
$100_2 = 2^2_{10} = 4_{10}$
$1000_2 = 2^3_{10} = 8_{10}$
$10000_2 = 2^4_{10} = 16_{10}$

$100000_2 = 2^5_{10} = 32_{10}$
$1000000_2 = 2^6_{10} = 64_{10}$
$10000000_2 = 2^7_{10} = 128_{10}$
$100000000_2 = 2^8_{10} = 256_{10}$
$1000000000_2 = 2^9_{10} = 512_{10}$
$10000000000_2 = 2^{10}_{10} = 1024_{10}$
$100000000000_2 = 2^{11}_{10} = 2048_{10}$
$1000000000000_2 = 2^{12}_{10} = 4096_{10}$
…

You should memorize these binary to decimal powers of two up to $2^{12}_{10} = 4096_{10}$, in the same way you memorized your multiplication tables when you were young. You should be able to recollect them quickly. This will allow you to convert binary numbers into decimal numbers and decimal numbers into binary numbers quickly in your head.

As an aside, in base ten we have a similar pattern, but because we are in base ten, the powers are powers of ten instead of powers of two:

$1_{10} = 10^0_{10}$
$10_{10} = 10^1_{10}$
$100_{10} = 10^2_{10}$
$1000_{10} = 10^3_{10}$
$10000_{10} = 10^4_{10}$
…

12.2.3 Binary Arithmetic

We can add numbers in binary just like we can in decimal. Again, the only difference being that binary only has two possible values per digit. Let us work out a few examples.

Addition

<u>Example 1</u>:

```
  10
+  1
-----
   ?
```

We add columns just like we do in decimal. The first column yields 0 + 1 = 1, and the second also 1 + 0 = 1. Thus,

```
  10
+  1
-----
  11
```

<u>Example 2</u>:

```
  101
+  11
------
    ?
```

Adding columns from right to left, we have 1 + 1 = 10 in binary for the first column. So we write zero for this column and carry a one over to the next digit place:

```
   1
  101
+  11
------
    0
```

Adding the second column leads to the previous situation, namely, 1 + 1 = 10. So we write zero for this column and carry a one over to the next digit place:

```
  11
  101
+  11
------
   00
```

Adding the third column again leads to the previous situation, namely, 1 + 1 = 10. So we write zero for this column and carry a one over to the next digit place, yielding our final answer:

```
  11
  101
+  11
------
 1000
```

<u>Example 3</u>:

```
  10110
+   101
--------
      ?
```

Adding the first column (right most column) we have 0 + 1 = 1. So we write 1 for this column:

```
  10110
+   101
--------
      1
```

Adding the second column, we again have 1 + 0 = 1:

```
  10110
+   101
--------
     11
```

The third column yields 1 + 1 = 10, so we write zero for this column and carry over 1 to the next digit place:

```
   1
  10110
+   101
--------
    011
```

Add the fourth and fifth columns both yield 1 + 0 = 1:

```
   1
  10110
+   101
--------
  11011
```

Subtraction

Example 4:

```
  10
-  1
-----
   ?
```

We subtract the columns from right to left as we do in decimal. In this first case we must "borrow" like we do in decimal:

```
   10
  00
-  1
-----
   ?
```

Now 10 – 1 in binary is 1, thereby giving the answer:

```
   10
  00
-  1
-----
   1
```

Example 5:

```
  101
-  11
------
    ?
```

Subtracting the first column we have 1 – 1 = 0. So we write one in the first column:

```
  101
-  11
------
    0
```

In the second column, we must borrow again yielding:

```
  10
  001
-  11
------
   10
```

Example 6:

```
  10110
-   101
--------
      ?
```

Here the first column must borrow, and then 10 – 1 = 1:

```
      10
  10100
-   101
--------
      1
```

In the second column we now have 0 – 0 = 0, in the third column we have 1 – 1 = 0, in the fourth column we have 0 – 0 = 0, and in the fifth column we have 1 – 0 = 1, which gives the answer:

```
      10
  10100
-   101
--------
  10001
```

Multiplication

Multiplying in binary is the same as in decimal; except all of our products have only four possible forms: 0×0, 0×1, 1×0, and 1×1. As such, binary multiplication is pretty easy.

<u>Example 7</u>:

```
  10
x  1
-----
  10
```

<u>Example 8</u>:

```
  101
x  11
------
  101
+1010
------
 1111
```

<u>Example 9</u>:

```
   10110
 x   101
 --------
   10110
  000000
+1011000
---------
 1101110
```

12.2.4 Converting Binary to Decimal

There is a mechanical formula, which can be used to convert from binary to decimal, and it is useful for large numbers. However, in practice, we do not typically need to work with large numbers, and when we do, we use a calculator. So rather than show you the mechanical way, we will develop a way to convert mentally in our head. The mental conversion from binary to decimal relies on the fact that in the binary number system, each “digit place” corresponds to a power of 2 in decimal, as we showed in Section 12.2.2.

Consider the following binary number:

11100001_2

Let us rewrite this number as the following sum:

(1) $11100001_2 = 10000000_2 + 01000000_2 + 00100000_2 + 00000001_2$

From Section 12.2.2 we know each "digit place" corresponds to a power of 2 in decimal. In particular, we know:

$10000000_2 = 2^7_{10} = 128_{10}$
$1000000_2 = 2^6_{10} = 64_{10}$
$100000_2 = 2^5_{10} = 32_{10}$
$1_2 = 2^0_{10} = 1_{10}$

Substituting the decimal equivalents into (1) yields:

$11100001_2 = 128_{10} + 64_{10} + 32_{10} + 1_{10} = 225_{10}$.

12.2.5 Converting Decimal to Binary

Converting from decimal to binary is similar. Consider the following decimal number:

225_{10}

First we ask ourselves, what is the largest power of two that can fit into 225_{10}? From Section 2.2.2 we find that $2^7_{10} = 128_{10}$ is the largest that will fit. We now subtract $2^7_{10} = 128_{10}$ from 225_{10} and get:

$225_{10} - 128_{10} = 97_{10}$

We now ask, what is the largest power of two that can fit into 97_{10}? We find $2^6_{10} = 64_{10}$ is the largest that will fit. Again we subtract $2^6_{10} = 64_{10}$ from 97_{10} and get:

$97_{10} - 64_{10} = 33_{10}$

We ask again, what is the largest power of two that can fit into 33_{10}? We find that $2^5_{10} = 32_{10}$ is the largest that will fit. We subtract $2^5_{10} = 32_{10}$ from 33_{10} and get:

$33_{10} - 32_{10} = 1_{10}$

Finally, we ask, what is the largest power of two that can fit into 1_{10}? We find that $2^0_{10} = 1_{10}$ is the largest that will fit. We subtract 1_{10} from 1_{10} and get zero.

Essentially, we have decomposed 225_{10} into a sum of powers of two; that is:

(2) $$225_{10} = 128_{10} + 64_{10} + 32_{10} + 1_{10} = 2^7_{10} + 2^6_{10} + 2^5_{10} + 2^0_{10}$$

But, we have the following binary to decimal relationships from Section 2.2.2:

$10000000_2 = 2^7_{10} = 128_{10}$
$1000000_2 = 2^6_{10} = 64_{10}$
$100000_2 = 2^5_{10} = 32_{10}$
$1_2 = 2^0_{10} = 1_{10}$

Substituting the binary equivalents into (2) for the powers of twos yields:

$$225_{10} = 10000000_2 + 01000000_2 + 00100000_2 + 00000001_2 = 11100001_2$$

12.3 The Hexadecimal Number System

12.3.1 Counting in Hexadecimal

The hexadecimal (**hex** for short) number system is a base sixteen number system. This means that sixteen possible values per digit exists; namely 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Notice that we must add new symbols to stand for the values above 9 that can be placed in a single digit; that is, A = 10_{10}, B = 11_{10}, C = 12_{10}, D = 13_{10}, E = 14_{10}, F = 15_{10}. As we did in binary, let us count a few numbers one-by-one to get a feel for hex. For each number, we write the equivalent decimal and binary number next to it.

$0_{16} = 0_{10} = 0_2$	$10_{16} = 16_{10} = 10000_2$	$20_{16} = 32_{10} = 100000_2$
$1_{16} = 1_{10} = 1_2$	$11_{16} = 17_{10} = 10001_2$	$21_{16} = 33_{10} = 100001_2$
$2_{16} = 2_{10} = 10_2$	$12_{16} = 18_{10} = 10010_2$	$22_{16} = 34_{10} = 100010_2$
$3_{16} = 3_{10} = 11_2$	$13_{16} = 19_{10} = 10011_2$	$23_{16} = 35_{10} = 100011_2$
$4_{16} = 4_{10} = 100_2$	$14_{16} = 20_{10} = 10100_2$	$24_{16} = 36_{10} = 100100_2$
$5_{16} = 5_{10} = 101_2$	$15_{16} = 21_{10} = 10101_2$	$25_{16} = 37_{10} = 100101_2$
$6_{16} = 6_{10} = 110_2$	$16_{16} = 22_{10} = 10110_2$	$26_{16} = 38_{10} = 100110_2$
$7_{16} = 7_{10} = 111_2$	$17_{16} = 23_{10} = 10111_2$	$27_{16} = 39_{10} = 100111_2$

$8_{16} = 8_{10} = 1000_2$	$18_{16} = 24_{10} = 11000_2$	$28_{16} = 40_{10} = 101000_2$
$9_{16} = 9_{10} = 1001_2$	$19_{16} = 25_{10} = 11001_2$	$29_{16} = 41_{10} = 101001_2$
$A_{16} = 10_{10} = 1010_2$	$1A_{16} = 26_{10} = 11010_2$	$2A_{16} = 42_{10} = 101010_2$
$B_{16} = 11_{10} = 1011_2$	$1B_{16} = 27_{10} = 11011_2$	$2B_{16} = 43_{10} = 101011_2$
$C_{16} = 12_{10} = 1100_2$	$1C_{16} = 28_{10} = 11100_2$	$2C_{16} = 44_{10} = 101100_2$
$D_{16} = 13_{10} = 1101_2$	$1D_{16} = 29_{10} = 11101_2$	$2D_{16} = 45_{10} = 101101_2$
$E_{16} = 14_{10} = 1110_2$	$1E_{16} = 30_{10} = 11110_2$	$2E_{16} = 46_{10} = 101110_2$
$F_{16} = 15_{10} = 1111_2$	$1F_{16} = 31_{10} = 11111_2$	$2F_{16} = 47_{10} = 101111_2$

12.3.2 Hexadecimal Arithmetic

Again, as with binary, hex arithmetic is the same as decimal arithmetic in concept. The only difference being the number of digits we are working with.

Addition

<u>Example 1</u>:

```
  A
+ 5
----
  ?
```

We know A is 10 in decimal, so 10 + 5 is 15, but 15 decimal corresponds to F in hex, thus:

```
  A
+ 5
----
  F
```

<u>Example 2</u>:

```
  53B
+  2F
------
    ?
```

Again, it is probably easiest to convert from hex to decimal and then back again as you perform the sum, column-by-column. B + F correspond to 11 + 15 in decimal, which gives 26 decimal. Converting that to hex gives 1A. So we write an A down for the first column and carry a one over:

```
   1
  53B
+  2F
------
    A
```

The second column gives 2 + 3 + 1, which is the number 6. And the third column gives 5. Thus we have:

```
   1
  53B
+  2F
------
  56A
```

Subtraction

<u>Example 3</u>:

```
  53B
-  2F
------
    ?
```

Starting the subtraction at the rightmost column, we have B – F. But, because B < F, we must borrow. Borrowing yields:

```
    10
  52B
-  2F
------
    ?
```

Note that the borrowed "10" is in hex, and corresponds to 16 in decimal. So we have 10 + B – F, or in decimal 16 + 11 – 15 = 12, which C in hex:

```
    10
  52B
-  2F
------
    C
```

Subtracting the second column we get 2 – 2 = 0, and the third 5 – 0 = 5. Thus the answer is:

```
    10
  52B
-  2F
------
  50C
```

Multiplication

Example 4:

```
  20BA
X   12
------
     ?
```

Performing the multiplication yields:

```
   11
  20BA
X   12
------
   1
  4174
+20BA0
-------
 24D14
```

12.3.3 Converting Hexadecimal to Binary

In converting hex to binary, the key idea is that each hex digit can be described by exactly four binary digits. This is because it takes four binary digits to describe a decimal number in the range [0, 15], which is the decimal range a hex digit takes on ([0, F]). Consequently, to convert from hex to binary we write four place holders underneath each hex digit, as Figure 12.2 shows:

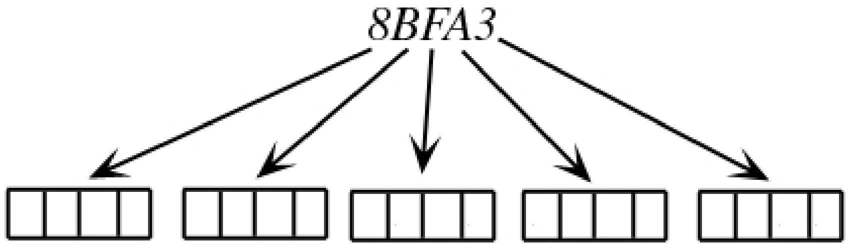


Figure 12.2: Setting up binary placeholders for a conversion from hex to binary.

We then convert each hex digit to its binary form, which is quite easy to do mentally since we only need to look at four binary digits at a time. Figure 12.3 shows the conversion of each hex digit to binary:

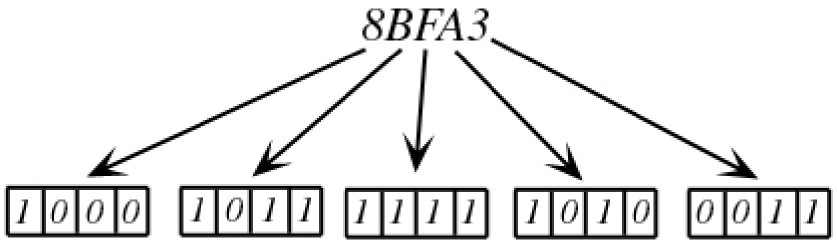


Figure 12.3: Converting each hex digit to binary.

As we can see, we made the following conversions:

$8_{16} = 1000_2 = 8_{10}$
$B_{16} = 1011_2 = 11_{10}$
$F_{16} = 1111_2 = 15_{10}$
$A_{16} = 1010_2 = 10_{10}$
$3_{16} = 0011_2 = 3_{10}$

Thus, we have $8BFA3_{16} = 10001011111110100011_2$.

> **Note:** You will find it is quite easy to convert between binary and hex because both number systems are powers of 2. Converting between these bases to and from octal (base 8) is equally easy, again, because base 8 is also a power of two. One of the reasons we like to work in hex is because we can mentally "visualize" the binary digit layout in hex just as clearly as seeing the actual binary number. In addition, hex has the nice property of being able to describe the same number with a smaller number of digits. That is, big numbers in binary take a lot of digits to describe, so from that standpoint, hex is much more compact.

12.3.4 Converting Binary to Hexadecimal

Converting from binary to hex is equally easy. Consider the number 111010111110111000_2. We group the binary digits into sets of four starting from the right to the left:

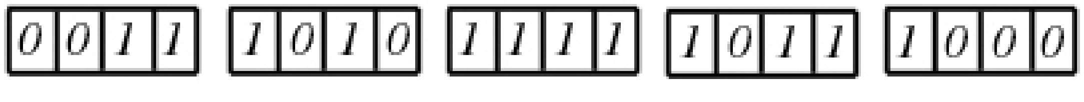


Figure 12.4: Dividing a binary number into groups of four, from right to left.

Note that if the last (left-most) binary digits do not form a set of four, we then pad the set with zeros to make it four. For each set of four binary digits we equate that to a single hex digit (0-F). If you memorized the binary to decimal power of two conversions from Section 2.2.2 and how hex digits relate to decimal, then this is easy to do mentally.

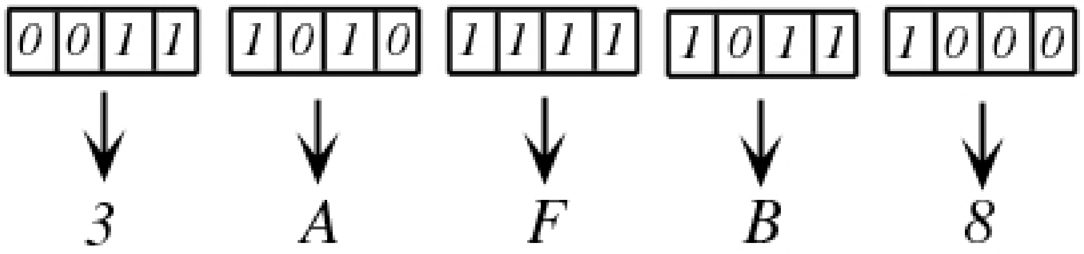


Figure 12.5: Converting each binary group of four to hex.

Thus, we have $111010111110111000_2 = 3AFB8_{16}$.

> **Note:** We do not discuss conversions between hex and decimal because we can convert either one to binary first, and then from binary we convert to either hex or decimal.

12.4 Bits and Memory

It is typically common computer knowledge, even among laypersons, that a computer (at least most machines) works with 0's and 1's internally—the so-called "on-off" switches. Indeed, if we recall from Chapter 1 in Module I, the compiler is responsible for translating our C++ code into machine code, which is all 0's and 1's. For this reason, a computer's natural number system is binary.

The term **bit** comes from **b**inary dig**it.** So when we talk about an 8-bit byte, we are talking about a piece of memory that is represented by 8-bits; that is, 8 binary digits are used to express a byte sized piece of information. Similarly, when we talk about a 32-bit integer, we are talking about a piece of memory that is represented by 32-bits. Recall the following abridged table from Chapter 1, which are intrinsic type sizes on a 32-bit Windows system:

Type	Range	Bytes Required
`unsigned char`	[0, 255]	1
`unsigned short`	[0, 65535]	2
`unsigned int`	[0, 4294967295]	4

Now that we know something about the binary number system we can see where these type ranges stem from. Given 8-bits to work with, we can express binary numbers in the range $[00000000_2,\ 11111111_2]$. However, that range is equivalent to $[0_{10},\ 255_{10}]$ in decimal. Given 16-bits to work with, we can express binary numbers in the range $[0000000000000000_2,\ 1111111111111111_2]$. However, that range is equivalent to $[0_{10},\ 65535_{10}]$ in decimal. The same logic can be made for 32-bit numbers, 64-bit numbers, and so on. We now see where those type ranges stem from. Clearly, the more bits we use, the more kinds of numbers we can represent.

Note: We do not discuss negative number representations here, as we are just trying to get a flavor for the underlying data representations. A more detailed look at internal computer data representation is best left to a course in computer architecture.

12.5 Bit Operations

As stated, a bit (binary digit) is a 0 or 1, which can be thought of as "on-off" or even as a "true-false" value. As such, it would make sense to be able to apply logical operations to bits. In our vocabulary, we say a bit is **set** if it is on; that is, it has a value of 1. Conversely, a bit is **not set** if it is off; that is, it has a value of 0.

12.5.1 AND

First, it should be noted that a bit is not directly addressable memory. The smallest addressable memory block is a byte, by definition. So when we do bit operations, we are doing them on the set of bits that make up larger integer types like `chars`, `ints`, and so on.

That said, the bitwise AND examines the one-to-one corresponding bits in two segments of addressable memory. For illustration purposes we will work with 8-bit `chars`. For each corresponding bit, the AND operation asks if both bits are set (i.e., are both "true"), and if so, the AND operation returns true (1) for that bit, otherwise it returns false (0) for that bit. The bitwise AND is performed with a single & symbol. Here is an example:

```
unsigned char A = 0xB9;
unsigned char B = 0x91;
unsigned char C = A & B; // AND the bits and store the result in C.
```

To see what is happening, let us look at the binary:

```
A = 1011 1001
B = 1001 0001

    1011 1001
  & 1001 0001
  -----------
C = 1001 0001
```

So, what we do is match each corresponding bit in `A` and `B` together, and if both bits are set, then the corresponding bit in `C` is set. If both bits are not set, then the corresponding bit in `C` is *not* set. As you can see, only the first bit, fifth bit, and eighth bit are both on in `A` and `B`, and thus those are the only bits set in `C`.

Note: The "0x" syntax means we are specifying a number in hexadecimal.

12.5.2 Inclusive OR

The bitwise OR examines the one-to-one corresponding bits in two segments of addressable memory. For illustration purposes we will work with 8-bit `chars`. For each corresponding bit, the OR operation asks if at least one of the two bits is set (i.e., is at least one "true"), and if so, the OR operation returns true (1) for that bit, otherwise it returns false (0) for that bit. The bitwise OR is performed with a single vertical bar | symbol. Here is an example:

```
unsigned char A = 0xB9;
unsigned char B = 0x91;
unsigned char C = A | B; // OR the bits and store the result in C.
```

To see what is happening, let us look at the binary:

```
A = 1011 1001
B = 1001 0001

    1011 1001
  | 1001 0001
  -----------
C = 1011 1001
```

We match each corresponding bit in `A` and `B` together, and if at least one of the two bits is set, then the corresponding bit in `C` is set. If neither bit is set, then the corresponding bit in `C` is *not* set.

12.5.3 NOT

The bitwise NOT operator, or complement operator, simply negates each bit. That is, an on bit (1) becomes on off bit (0), and conversely, an off bit (0) becomes an on bit (1). The bitwise NOT operator is a unary operator and its symbol is the tilde (~). Here is an example:

```
unsigned char A = 0xB9;
unsigned char C = ~A; // NOT the bits in A and store the result in C.
```

To see what is happening, let us look at the binary:

```
A = 1011 1001

  ~ 1011 1001
  -----------
C = 0100 0110
```

As you can see, the bitwise NOT operator, or complement operator, simply negates each bit. That is, an on bit (1) becomes on off bit (0), and conversely, an off bit (0) becomes an on bit (1).

12.5.4 Exclusive OR

The bitwise exclusive or (XOR) examines the one-to-one corresponding bits in two segments of addressable memory. For illustration purposes we will work with 8-bit `chars`. For each corresponding bit, the XOR operation asks if one of the two bits is set, but not both (i.e., one is “true” but not both), and if this condition is so, the XOR operation returns true (1) for that bit, otherwise it returns false (0) for that bit. The bitwise XOR is performed with a caret ^ symbol. Here is an example:

```
unsigned char A = 0xB9;
unsigned char B = 0x91;
unsigned char C = A ^ B; // XOR the bits and store the result in C.
```

To see what is happening, let us look at the binary:

```
A = 1011 1001
B = 1001 0001

    1011 1001
  ^ 1001 0001
  -----------
C = 0010 1000
```

We match each corresponding bit in `A` and `B` together, and if one of the two bits is set but not both, then the corresponding bit in `C` is set. If either bit is set or both bits are set, then the corresponding bit in `C` is *not* set.

12.5.5 Shifting

The bit-shifting operation is not a logical operation. The bit-shifting operator allows you to shift the bits to the left or right. Let us look at an example:

```
unsigned char A = 0xB9;
unsigned char B = 0x91;
unsigned char C = A << 3; // shift bits in A three bits to the left.
unsigned char D = B >> 2; // shift bits in B two bits to the right.
```

The symbol << means “left shift” and the symbol >> means “right shift.” The right hand operand specifies how many bits to shift. So in the above example, `C = A << 3` means shift the bits in `A` three bits to the left, and store the result in `C`; and `D = B >> 2` means shift the bits in `B` two bits to the right, and store the result in `D`.

Let us look at the binary to see what is happening more closely:

```
A = 1011 1001
B = 1001 0001
```

```
C = 1011 1001 << 3 = 1100 1000
D = 1001 0001 >> 2 = 0010 0100
```

Can you see how the bits “shifted over” by the specified number? Note that when you shift bits “off” the memory region you are working with, they are lost—they do not “loop” back around.

12.5.6 Compound Bit Operators

C++ provides compound bit operators as well. The following table summarizes:

Compound Operator	**Meaning**
`A &= B`	`A = A & B`
`A |= B`	`A = A | B`
`A ^= B`	`A = A ^ B`
`A <<= B`	`A = A << B`
`A >>= B`	`A = A >> B`

12.6 Floating-Point Numbers

Before concluding this chapter, we want to spend a little time giving you an overview of how floating-point numbers are represented. The floating-point representation your common PC uses is the IEEE 754 (Institute of Electrical and Electronics Engineers) standard, which was a standard developed in order to increase software portability among computers.

A floating-point number is represented in a **normalized** scientific notation such as $\pm 5.1234 \times 10^{12}$, and has three components:

1) a sign bit
2) an exponent
3) a mantissa (non-exponent part)

Normalized means that there is only one non-zero digit before the **radix point** (we say *radix point* to be general since we may be working in number systems other than decimal, so we do not call it a *decimal point*). A normalized form is used so that the same number cannot be described in two different forms. For example, the following are equivalent:

$$314.0 \times 10^{-2} = 31.4 \times 10^{-1} = 3.14 \times 10^{0}$$

Having two different forms of the same number can make implementation of floating-point numbers more complicated. Therefore, we always use the normalized form (3.14×10^{0} in the above example) so that there is only one form of a given number.

As some background vocabulary, the exponent part of a floating-point number determines the **range** of the number (how big and small it can get) and the mantissa part of a floating-point number determines the **precision** of the number (the "step" size between two adjacent numbers on the number line). The more bits we devote to the exponent part, the larger the range becomes, and the more bits we devote to the mantissa part, the more precise the number becomes.

Because the radix point always follows the left-most *non-zero* digit in normalized form (e.g., 3.14×10^0), in binary the left most non-zero binary digit will always be 1 in normalized form. Here are some examples of normalized binary numbers:

1.0101×2^3
1.011101×2^{-5}
1.0100001×2^8

Because the left-most non-zero digit is always a 1 in normalized binary form, we do not need to store that 1, as we can assume it is always there, and thus get an extra bit of precision. This understood bit is called the **hidden bit.**

The following figure shows the layout of a 32-bit floating-point number (i.e., a `float`) and a 64-bit float-point number (i.e., a `double`).

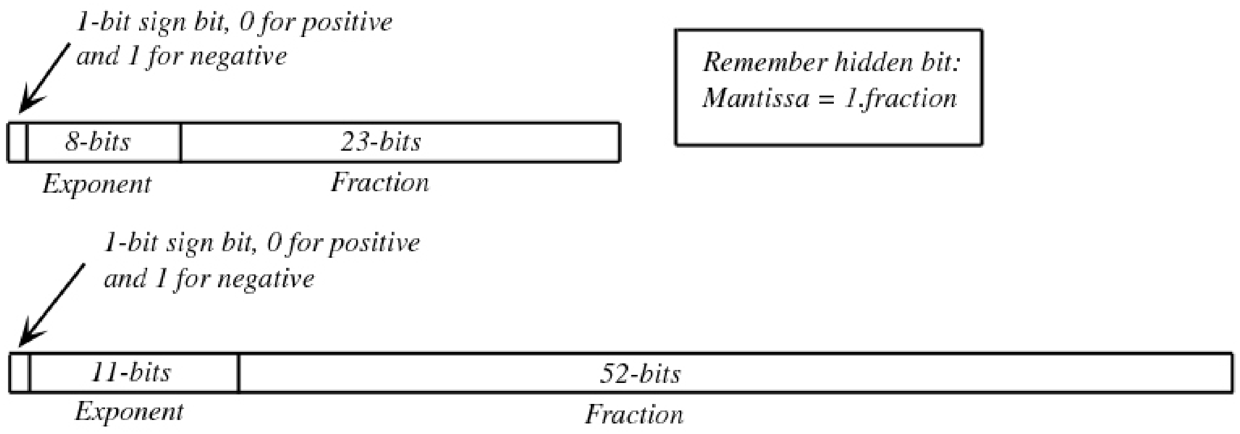

1-bit sign bit, 0 for positive and 1 for negative
8-bits
23-bits
Exponent
Fraction
Remember hidden bit: Mantissa = 1.fraction
1-bit sign bit, 0 for positive and 1 for negative
11-bits
52-bits
Exponent
Fraction

Figure 12.6: 32-bit float layout and 64-bit double layout.

There are some special reserved bit patterns that take on special meaning:

- An exponent part of all zeros and a mantissa of all zeros (in binary) are interpreted as zero.
- An exponent part of all ones and a mantissa of all zeros (in binary) are interpreted as infinity.
- An exponent part of all ones and a non-zero mantissa (in binary) are interpreted as "not a number" (NaN), which is used to denote the result of undefined operations such as division by zero, division of infinity over infinity, and the like.

This is all we want to say about floating-point numbers in an introductory C++ course. A more detailed look at internal computer data representation is best left to a course in computer architecture.

12.7 Summary

1. The binary number system is a base two number system. This means that two possible values per digit exists; namely 0 and 1. The hexadecimal (hex for short) number system is a base sixteen number system. There are exists sixteen possible values per digit; namely 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Notice that we must add new symbols to stand for the values above 9 that can be placed in a single digit; that is, A = 10_{10} , B = 11_{10} , C = 12_{10} , D = 13_{10} , E = 14_{10} , F = 15_{10} .

2. The term "bit" comes from "*b*inary dig*it*." When we talk about an 8-bit byte, we are talking about a piece of memory that is represented by 8-bits; that is, 8 binary digits are used to express a byte sized piece of information. Similarly, when we talk about a 32-bit integer, we are talking about a piece of memory that is represented by 32-bits. Given 8-bits to work with, we can express binary numbers in the range $[00000000_2,\ 11111111_2]$. However, that range is equivalent to $[0_{10},\ 255_{10}]$ in decimal. Given 16-bits to work with, we can express binary numbers in the range $[0000000000000000_2,\ 1111111111111111_2]$. However, that range is equivalent to $[0_{10},\ 65535_{10}]$ in decimal. The same logic can be made for 32-bit numbers, 64-bit numbers, and so on. We now see where those type ranges come from. Clearly, the more bits we use, the more kinds of numbers we can represent.

3. Five bit operations exist:

    a. A bitwise AND written as A & B, which for each corresponding bit in A and B, returns a "true" bit if the bits are both "true", otherwise it returns a "false" bit.
    b. A bitwise OR written as A | B, which for each corresponding bit in A and B, returns a "true" bit if at least one of the bits is "true," otherwise it returns a "false" bit.
    c. A bitwise NOT written as ~A, which negates each bit in A ("true" bits become "false" and conversely).
    d. A bitwise XOR written as A ^ B, which for each corresponding bit in A and B, returns a "true" bit if one of the bits is "true" but not both, otherwise it returns a "false" bit.
    e. A bit shift operation written as A << *m* for a shift in the left direction of *m* bit spaces, or as A >> *n* for a shift in the right direction of *n* bit spaces.

4. A floating-point number is represented in a normalized scientific notation such as $\pm 5.1234 \times 10^{12}$, and has three components: 1) a sign bit, 2) an exponent, and a mantissa (non-exponent part). Normalized means that there is only one non-zero digit before the radix point (we say radix point to be general since we may be working in number systems other than decimal; we do not call it a decimal point).

5. The exponent part of a floating-point number determines the range of the number (how big and small it can get) and the mantissa part of a floating-point number determines the precision of the number (the "step" size between two adjacent numbers on the number line).

6. Because the radix point always follows the left-most *nonzero* digit in normalized form (e.g., 3.14×10^0), in binary the left most nonzero binary digit will always be 1 in normalized form. Because the left-most nonzero digit is always a 1 in normalized binary form, we do not really need to store that 1, as we can assume it is always there, and thus get an extra bit of precision. This understood bit is called the "hidden bit."

12.8 Exercises

12.8.1 Binary Arithmetic

Perform the following binary arithmetic operations. Use a calculator, such as the Windows calculator program, to check your work.

```
  110               1111          010101          11001101
+ 101             + 1111        + 101010        + 11110101
------            -------       ---------       ----------
    ?                 ?               ?                  ?



  110               1111         1010101          1111001101
- 101             - 1111        - 101010        -   11110101
------            -------       ---------       ------------
    ?                 ?               ?                    ?



  110               1111          10101          11001101
x  10             x   11        x   110        +        1
------            -------       --------       ----------
    ?                 ?              ?                  ?
```

12.8.2 Hex Arithmetic

Perform the following hex arithmetic operations. Use a calculator, such as the Windows calculator program, to check your work.

```
  AAA           BB          FEDCA          12FE5BA
+ 555         + A4        + 1234         + 45AFCD
------        -----       -------        ---------
    ?            ?             ?                ?
  AAA           BB           FEDCA            12FE5BA
- 555         - A4         - 1234           - 45AFCD
------        -----        -------          ---------
    ?             ?              ?                  ?


  AAA           BB           FEDCA            12FE5BA
x   5         x A4         x     A          x      CF
------        ------       -------          ---------
    ?             ?              ?                  ?
```

12.8.3 Base Conversions

For the following conversions, use a calculator, such as the Windows calculator program, to check your work.

Convert the following decimal numbers to binary:

$22_{10} = ?_2$

$63_{10} = ?_2$

$95_{10} = ?_2$

$133_{10} = ?_2$

Convert the following binary numbers to decimal:

$10001_2 = ?_{10}$

$101101_2 = ?_{10}$

$10101010_2 = ?_{10}$

$11111011_2 = ?_{10}$

Convert the following binary numbers to hex:

$1101_2 = ?_{10}$

$11000101_2 = ?_{10}$

$1101101010111011_2 = ?_{10}$

$1111001101011101010101011_2 = ?_{10}$

Convert the following hex numbers to binary:

$5B_{16} = ?_2$

$D9B_{16} = ?_2$
$4BC3DF_{16} = ?_2$
$1A2B3C4D5E6F7_{16} = ?_2$

12.8.4 Bit Operations

For the following operations, use a calculator, such as the Windows calculator program, to check your work. Note that some operations may not be available on a calculator. If that is the case, write a C++ program to do the operation in order to check your work.

Given:

```
unsigned char A = 0xF8;
unsigned char B = 0x5A;
```

Evaluate the following expressions for C; give your answer in binary:

1. `unsigned char C = A & B;`

2. `unsigned char C = A | B;`

3. `unsigned char C = ~A;`

4. `unsigned char C = ~B;`

5. `unsigned char C = A ^ B;`

6. `unsigned char C = A << 2;`

7. `unsigned char C = A >> 3;`

8. `unsigned char C = B << 1;`

9. `unsigned char C = B >> 4;`

12.8.5 Binary to Decimal

Write a program that asks the user to input a binary string (e.g., “11100010101”). Note that the input is a *string*, not a number. The string specifies a binary number. The program then is to convert the binary number which the string specifies into a decimal number and output the result. Your output should look like this:

```
Enter a binary string: 101
101 = 5 in decimal.
Press any key to continue.
```

12.8.6 Decimal to Binary

Write the reverse conversion of Exercise 12.8.5. That is, write a program that asks the user to input a decimal number (this time it is *not* a string but a number). The program then is to convert the decimal number which the user entered into a binary *string* (e.g., “11100010101”) and output the result. Your output should look like this:

```
Enter a decimal number: 14
14 = 1110 in binary.
Press any key to continue.
```

12.8.7 Bit Operation Calculator

Write a program that allows the user to enter in two integer values, A and B, in radix ten, and store them in `chars`, `shorts`, or `ints` (your choice depending on the size of the numbers you want to support). Convert these decimal numbers to binary string representations, as you did in Exercise 12.8.6, and output the binary form to the user. Then have the program compute and output (in binary) A & B, A | B, ~A, ~B, A ^ B, A << 2, and B >> 3. For example, in the program you will write (assuming we are working with `chars`):

```
unsigned char C = A & B;
```

We now want to output the result of A & B (that is C) as a binary string, so we pump C through our decimal to binary string converter for output:

```
cout << “A & B = “ << ToBinaryString( C ) << endl;
```

Your output should look like this:

```
Enter a number A in the range [0, 255]: 5
Enter a number B in the range [0, 255]: 8
```

```
A = 5 in binary is 00000101
B = 8 in binary is 00001000

A & B = 00000000
A | B = 00001101
~A = 11111010
~B = 11110111
A ^ B = 00001101
A << 2 = 00010100
B >> 3 = 00000001
```

12.9 References

Murdocca, Miles J., and Vincent P. Heuring. *Principles of Computer Architecture*. Prentice Hall, 2000.

Chapter 13

STL Primer

Introduction

In this chapter, we take a tour of the standard template library and examine several data structures, algorithms, and programming paradigms.

Essentially, data structures are objects for storing collections of data. An array is a data structure at the most primitive level, and although arrays have some advantages (fast indexing of elements), they also have some pitfalls (slow arbitrary insertion), which motivate the employment of better suited data structures for the task at hand. Algorithms act on data structures. Sorting, searching, generating, removing, and counting are all examples of operations that we would apply on data structures in this chapter using the pre-built standard template library algorithm functions.

Chapter Objectives

- Discover how lists, stacks, queues, deques, and maps work internally, and in which situations they should be used.
- Become familiar with a handful of the generic algorithms the standard library provides and how to apply these algorithms on a variety of data structures.
- Learn how to create objects that act like functions, called functors, and learn how to create and use predicates with the standard library.

13.1 Problems with Arrays

New students often wonder at first why we even need other data structures. Why not simply use arrays (or the resizable `std::vector`) for everything? To motivate our discussion of various data structures, let us first discuss some problems with arrays, and also with `std::vector`, as `std::vector` is implemented internally using arrays. Suppose that we have an array of ten integers, where the first five array elements are used and the second five are free:

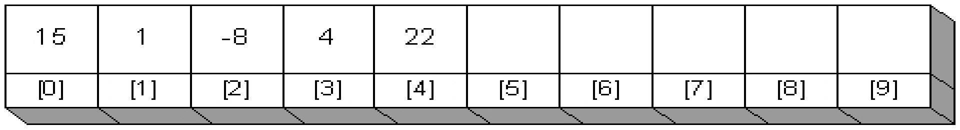


Figure 13.1: An array with five empty elements.

Assume that we want to insert a new integer 7 at element [2]. Because an integer already exists at [2], and assuming we do not want to overwrite that value, we will shift the elements starting at [2] over one element to the right, in order to make room for the new integer 7:

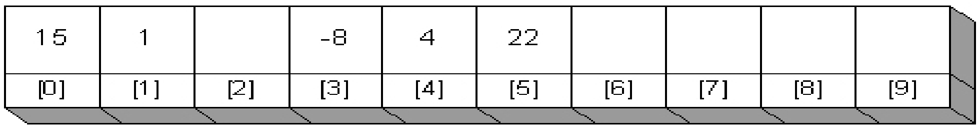


Figure 13.2: Shifting some array elements to the right to make room for an insertion into element [2].

We can now write 7 to element [2]:

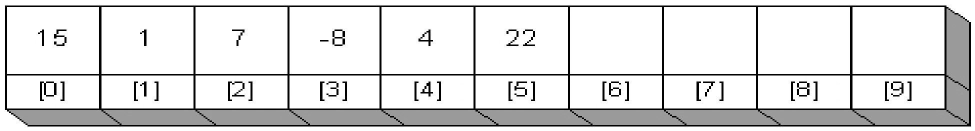


Figure 13.3: The array after insert 7 into element [2].

The point of the previous exercise is to demonstrate that inserting values into arbitrary array positions requires some work. In particular, the work required is shifting the elements around to make room for the value to be inserted. (Note: a similar argument can be made for deletion of an arbitrary element.) With large arrays, this can require a lot of shifts and these memory shifts can get expensive. The key point to understand is that arrays are not inherently suited for arbitrary data insertions. However, inserting at the end of an array requires no shifting.

Still, there is another problem with arrays. Specifically, arrays are slow to resize. Suppose that we have filled up the ten elements from the previous array and need more space. We must then resize the array. Recall from Chapter 4 in Module I the following function:

```
int* ResizeArray(int* array, int oldSize, int newSize)
{
      // Create an array with the new size.
      int* newArray = new int[newSize];

      // New array is a greater size than old array.
      if( newSize >= oldSize )
      {
            // Copy old elements to new array.
            for(int i = 0; i < oldSize; ++i)
                  newArray[i] = array[i];
      }
      // New array is a lesser size than old array.
      else // newSize < oldSize
      {
            // Copy as many old elements to new array as can fit.
            for(int i = 0; i < newSize; ++i)
                  newArray[i] = array[i];
      }

      // Delete the old array.
      delete[] array;

      // Return a pointer to the new array.
      return newArray;
}
```

We must allocate new memory for the newly resized array and then copy the elements from the old array into the new array. Again, this is slow for large arrays. We have an expensive memory allocation and, potentially, we have a lot of memory copying to do. The key point is that arrays are not inherently suited for growing in size.

Third, in some of the previous exercises you did, you implemented a linear search and binary search function. We learned that the linear search is quite slow, and that the binary search is quite fast. However, the binary search required the array to be previously sorted. The key point is that arrays are not inherently optimized for fast searching.

To summarize, we have three problems with arrays:

1. Slow arbitrary insertion.
2. Slow resizing.
3. Not optimized for searching.

Note: Despite these shortcomings, arrays have some benefits, and it is your job to look at the particular application and weigh the cost to benefit ratios. If you do not need arbitrary insertions (you only need to insert at the end of the array) then problem (1) goes away. If you are not resizing the array very frequently then problem (2) becomes negligible. As already stated, array elements can be accessed very fast. In fact, this is the primary advantage of arrays—fast arbitrary element access. By simply providing an index, we can offset very quickly to an element. This is not true for some of the other data structures we will encounter.

Again, you will need to use judgment in selecting a data structure for a task. Each has advantages and disadvantages, and you want to pick the one that is best suited for the task.

13.2 Linked Lists

13.2.1 Theory

A **linked list** can be thought of as a chain of data elements, which are called **nodes** in the context of linked lists. Figure 13.4 gives a typical graphical representation of a linked list.

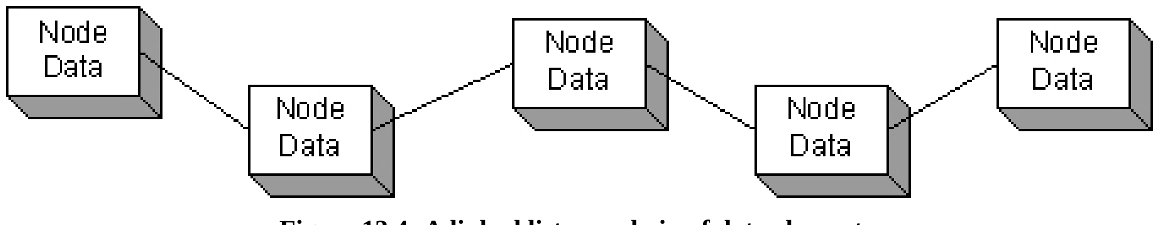


Figure 13.4: A linked list as a chain of data elements.

Unlike arrays, the nodes of a linked list are not in contiguous memory, as Figure 13.4 alludes to. Therefore, we need a way of connecting nodes together to build the "chain." This is accomplished via pointers. Each node has a pointer to the next node and a pointer to the previous node. This is called a **double-ended linked list** or **doubly-linked list.** Figure 13.5 illustrates:

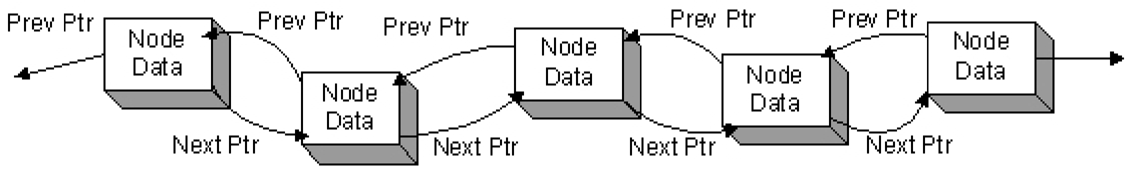


Figure 13.5: We connect the nodes together with pointers, thereby forming the chain of connectivity.

Notice that the first node's previous pointer is null, and the last node's next pointer is null. This follows from the fact that the first node, by definition, has no previous node, and the last node, by definition, has no next node.

Therefore, in addition to having data stored at a node, it must additionally store two pointers. Thus a typical linked list node looks like this in code:

```
template<typename T>
struct LinkedNode
{
	T data;
	LinkedNode* next;
	LinkedNode* prev;
};
```

We use templates because we would probably like to have linked lists that can support many different types of data.

We use a linked list class to store the first and last node. This class also provides methods for inserting/deleting, and traversing nodes in the linked list, as well as an overloaded copy constructor, and an overloaded assignment operator.

```
Template<typename T>
class LinkedList
{
public:
	LinkedList();
	~LinkedList();
	LinkedList(const LinkedList& rhs);
	LinkedList& operator=(const LinkedList& rhs);

	bool isEmpty();
	LinkedNode* getFirst();
	LinkedNode* getLast();

	void insertFirst(T data);
	void insertLast(T data);
	void insertAfter(T tKey, T tData);
	void removeFirst();
	void removeLast();
	void remove(T removalCandidate);
	void destroy();
private:

	LinkedNode* mFirst;
	LinkedNode* mLast;
};
```

The pointer connection between nodes makes it easy to insert and delete elements at any position. Figure 13.6 graphically represents insertion of a node after another node.

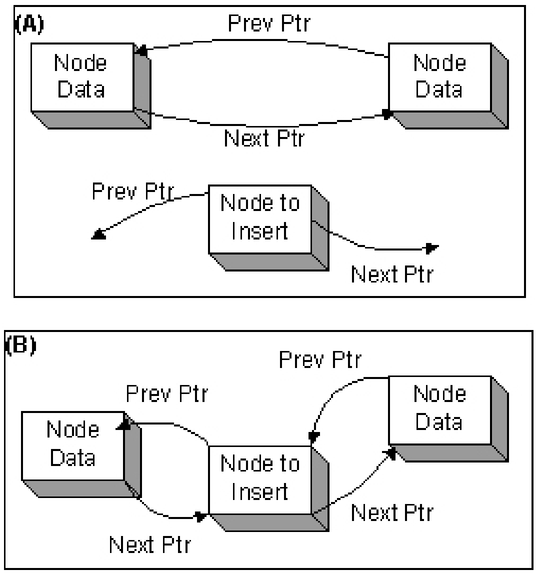


Figure 13.6: To insert a new node between two nodes we simply need to reassign some pointers.

To insert a newly allocated node after a node, we must adjust the next and previous pointers around the nodes where we are inserting. A sample implementation of this insertion is as follows:

```
bool LinkedList::insertAfter(T tKey, T tData)
{
	if(IsEmpty()) return false;

	// Get a ptr to the front of the list
	LinkedNode<T>* current = mFirst;

	// Loop until we find tKey (the value of the node to
	// insert after)
	while(current->data != tKey)
	{
		// Hop to the next node
		current = current->next;

		// Test if we reached the end, if we did we didn't
		// find the node to insert after (tKey).
		if(current == 0)
			return false;
	}
	// Allocate memory for the new node to insert.
	LinkedNode<T>* newNode = new LinkedNode<T>(tData);

	// Special case: Are we inserting after the last node?
	if(current == mLast)
	{
		newNode->next = 0;
		mLast = newNode;
	}
	// No, link in newNode after the current node.
	else
	{
		newNode->next = current->next;
		current->next->prev = newNode;
	}

	newNode->prev = current;
	current->next = newNode;

	return true;
}
```

Deletion is essentially the reverse operation. That is, find the node, delete it, and readjust the pointers so they do not connect to the deleted node.

Inserting a node to the front or to the back of the list is the same idea, but simpler, since we are not inserting between two nodes. Here is a sample implementation of the insertFirst method:

```
void LinkedList::insertFirst(T tData)
{
	LinkedNode<T>* newNode = new LinkedNode<T>(tData);

	// If the list is empty, this is the first and the last node
	// and doesn't have a previous ptr.
	if(IsEmpty())
		mLast = newNode;

	// If the list is not empty, the new node becomes the previous
	// node of the current first node.
	else
		mFirst->prev = newNode;

	// The new node's next ptr is the old first ptr.  May be null
	// if this is the first node being added to the list.
	newNode->next = mFirst;

	// The new node becomes the first ptr.
	mFirst = newNode;
}
```

Because all of the nodes of a linked list are allocated dynamically, we must traverse the list node-by-node and delete each element when we want to destroy the entire linked list. Here is a sample implementation:

```
void LinkedList::destroy()
{
      // Is there at least one node in the list?
      if(mFirst != 0)
      {
            // Get a pointer to the first node.
            LinkedNode<T>* current = mFirst;

            // Loop until the end of the list.
            while(current != 0)
            {
                  // Save the current node.
                  LinkedNode<T>* oldNode = current;

                  // Move to the next node.
                  current = current->next;

                  // Delete the old node.
                  delete oldNode;
                  oldNode = 0;
            }
      }
}
```

Before we can delete the current node, we must get a pointer to the next node. If we did not first save a pointer to the next node, then we would lose it when we deleted the node:

```
delete current;
current = current->next; // ERROR, current was deleted—can't
                         // access next.
```

Unfortunately, because a linked list is not contiguous in memory, we cannot access a node by supplying an index, which gives us a direct offset. Instead we must traverse the linked list one node at a time until we reach the desired node. Thus, we have the following linked list performance properties:

1. Linked lists have fast arbitrary insertions and deletions.
2. Linked lists have slow arbitrary node access.
3. Linked lists, like arrays, have no inherent searching optimization properties.

In the standard library, a doubly-ended linked list is modeled with the `std::list` class (#include <list>). The subsequent sections examine how to use this class.

13.2.2 Traversing

For general purposes, the C++ containers work with **iterators.** Iterators are objects that refer to a particular element in a container. Iterators provide operators for navigating the container and for accessing the actual element to which is being referred. Iterators overload pointer related operations for this functionality. In this way, pointers can be thought of as iterators for raw C++ arrays. For example, the data to which an iterator refers is accessed with the dereference operator (*). We can move the iterator to the next and previous element using pointer arithmetic operators such as ++ and --. (We can use these operators because they have been overloaded for the iterator.) Note, however, that although these operators behave similarly, the underlying implementation will be different for different containers.

For example, moving up and down the nodes of a linked list is different than moving up and down the elements of an array. However, by using iterators and the same symbols to move up and down, a very beautiful generic framework is constructed. Because every container has the same iterators using the same symbols that know how to iterate over the container, a client of the code does not necessarily need to know the type of container. It can work generally with the iterators, and the iterators will "know" how to iterate over the respective container in a polymorphic fashion.

> **Note:** The obvious implementation of an iterator is as a pointer. However, this need not be the case, and you are best not thinking that it necessarily is. The implementations of iterators vary depending on the container they are used for.

Getting back to `std::list`, we can iterate over all the elements in a list as follows:

```
list<int>::iterator iter = 0;
for( iter = myIntList.begin(); iter != myIntList.end(); ++iter )
{
      cout << *iter << " ";
}
cout << endl;
```

First we declare a list iterator called `iter`. We then initialize this iterator with `myIntList.begin()`, which returns an iterator to the first node in the list. Then we loop as long as `iter` does not reach the end of the list. The method call `myIntList.end()` returns an iterator to the node after the last node—the terminating null node. We increment from one node to the next node by calling the increment operator on the iterator: `++iter`. This is sort of like pointer arithmetic, moving from one array element to the next; however, linked lists are not arrays in contiguous memory.

Internally, the ++ operator for `list` iterators does whatever is necessary to move from one node to the next. Also, you should note that we can move from one node to the previous node using the decrement operator --. Again, iterators were designed to be pointer-like for generic purposes. Finally, to get the data value to which an iterator refers we use the dereference operator: `*iter`. Again, this is like dereferencing a pointer, at least syntax-wise. Remember, we will view iterators as pointer-like, but not necessarily as pointers.

13.2.3 Insertion

There are three methods we are concerned with for inserting elements into a `list`.

1. `push_front`: Inserts a node at the front of the list.
2. `push_back`: Inserts a node at the end of the list.
3. `insert`: Inserts a node at some position identified by an iterator.

The following program illustrates:

Program 13.1: Sample of std::list.

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

int main()
{
      list<int> myIntList;

      // Insert to the front of the list.
      myIntList.push_front(4);
      myIntList.push_front(3);
      myIntList.push_front(2);
      myIntList.push_front(1);

      // Insert to the back of the list.
      myIntList.push_back(5);
      myIntList.push_back(7);
      myIntList.push_back(8);
      myIntList.push_back(9);

      // Forgot to add 6 to the list, insert before 7.  But first
      // we must get an iterator that refers to the position
      // we want to insert 6 at.  So do a quick linear search
      // of the list to find that position.
      list<int>::iterator i = 0;
      for( i = myIntList.begin(); i != myIntList.end(); ++i )
            if( *i == 7 ) break;

      // Insert 6 were 7 is (the iterator I refers to the position
      // that 7 is located.  This does not overwrite 7; rather it
      // inserts 6 between 5 and 7.
      myIntList.insert(i, 6);

      // Print the list to the console window.
      for( i = myIntList.begin(); i != myIntList.end(); ++i )
            cout << *i << " ";

      cout << endl;
}
```

Program 13.1 Output

```
1 2 3 4 5 6 7 8 9
Press any key to continue
```

13.2.4 Deletion

There are three methods of concern for deleting elements from a `list`.

4. `pop_front`: Deletes the node at the front of the list.
5. `pop_back`: Deletes the node at the end of the list.
6. `remove(x)`: Searches the list and deletes the node with the value `x`.

The following program illustrates (the new deletion code is bolded):

Program 13.2: Sample of std::list deletion methods.

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

int main()
{
      list<int> myIntList;

      // Insert to the front of the list.
      myIntList.push_front(4);
      myIntList.push_front(3);
      myIntList.push_front(2);
      myIntList.push_front(1);

      // Insert to the back of the list.
      myIntList.push_back(5);
      myIntList.push_back(7);
      myIntList.push_back(8);
      myIntList.push_back(9);

      // Forgot to add 6 to the list, insert before 7.  But first
      // we must get an iterator that refers to the position
      // we want to insert 6.
      cout << "Before deletion..." << endl;
      list<int>::iterator i = 0;
      for( i = myIntList.begin(); i != myIntList.end(); ++i )
            if( *i == 7 ) break;

      myIntList.insert(i, 6);

      // Print the list to the console window.
      for( i = myIntList.begin(); i != myIntList.end(); ++i )
            cout << *i << " ";
      cout << endl;

      // Remove the first node in the list.
      myIntList.pop_front();

      // Remove the last node in the list.
      myIntList.pop_back();

      // Remove the node that has a value of 5
      myIntList.remove( 5 );

      // Print the list to the console window.
      cout << "After deletion..." << endl;
      for( i = myIntList.begin(); i != myIntList.end(); ++i )
      {
            cout << *i << " ";
      }
      cout << endl;
}
```

Program 13.2 Output

```
Before deletion...
1 2 3 4 5 6 7 8 9
After deletion...
2 3 4 6 7 8
Press any key to continue
```

13.3 Stacks

13.3.1 Theory

A **stack** is a **LIFO** (last in first out) data structure. A stack does what it sounds like—it manages a stack of data. Consider a stack of objects; say a stack of dinner plates. You stack the plates on top of each other and then to remove a plate, you remove it from the top. Thus, the last plate you added to the stack is the first one you would remove: hence the name LIFO (last in first out). Figure 13.7 shows a graphical representation of a stack:

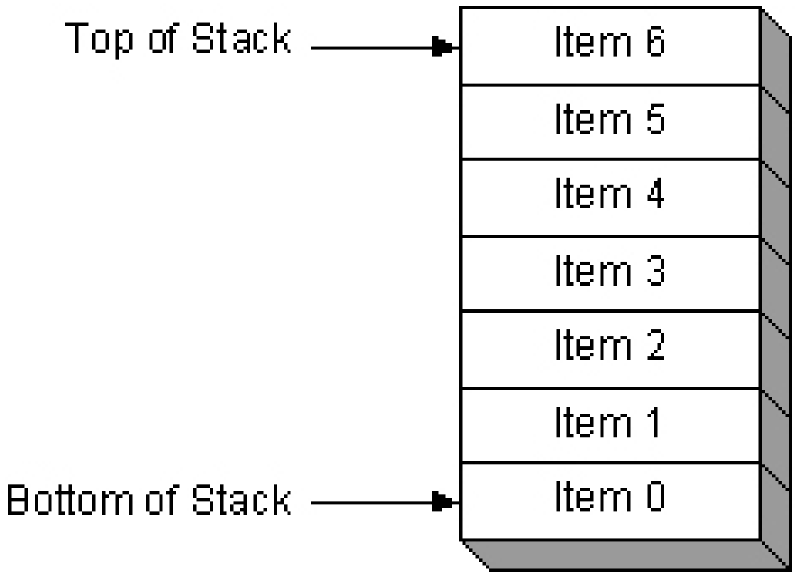


Figure 13.7: In a stack, data items are "stacked," conceptually, on top of each other.

As far as vocabulary is concerned, placing items on the stack is called **pushing** and taking items off the stack is called **popping.**

A stack container is useful for when you need to model a set of data that is naturally modeled as a stack. For example, consider how you might implement an "undo" feature in a program. As the user makes changes you push the changes onto the stack:

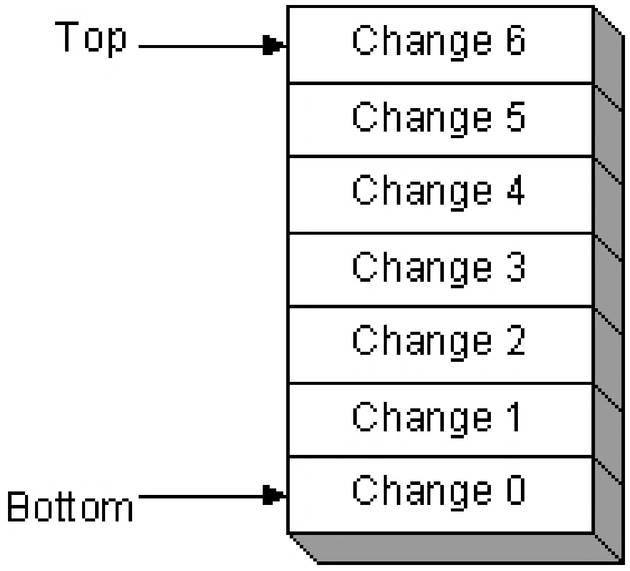


Figure 13.8 Pushing program changes onto a stack.

As you know, an "undo" feature erases the last change you made, so to "undo" the last modification you would simply pop the last change off the stack to erase it:

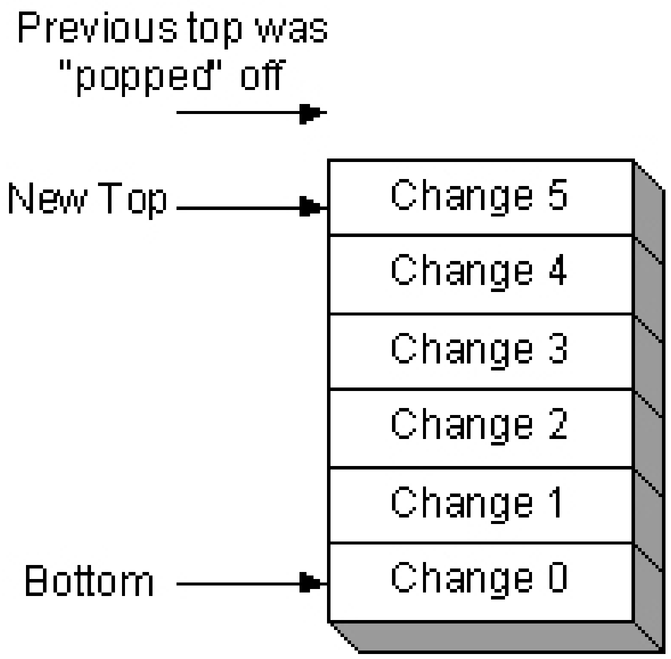


Figure 13.9: Popping a program change off of the stack.

You can implement a stack using an array, but a linked list works well also. Essentially, every time the client pushes an item on the stack, you will append that item to the back of the linked list. When the user wishes to pop an item off the stack, you will simply delete the last item from the linked list. The last linked list node represents the top of the stack and the first linked list node represents the bottom of the stack.

Besides pushing and popping, the client often wants to access the top item of the stack. We could implement this method by returning a reference to the top item on the stack, which, in our linked list implementation, would be the last node.

13.3.2 Stack Operations

In the standard library, a stack is represented with `std::stack` (#include <stack>). `std::stack` contains four methods of interest.

1. `empty`: Returns `true` if the stack is empty (contains no items) or `false` otherwise.
2. `push`: Pushes an item onto the stack.
3. `pop`: Pops the top item from the stack.
4. `top`: Returns a reference to the top item on the stack.

Let us look at a way to write a word reverse program using a stack.

Program 13.3: String reverse using a stack.

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

int main()
{
	stack<char> charStack;

	cout << "Enter a string: ";
	string input = "";
	getline(cin, input);

	for(int i = 0; i < input.size(); ++i)
		charStack.push(input[i]);

	cout << "The reverse string is: ";
	for(int j = 0; j < input.size(); ++j)
	{
		cout << charStack.top();
		charStack.pop();
	}
	cout << endl;
}
```

Program 13.3 Output

```
Enter a string: Hello World
The reverse string is: dlroW olleH
Press any key to continue
```

The program instructs the user to enter a string. We then loop through each character from beginning to end and push the character onto the stack—Figure (13.10*a*). To output the string in reverse order we simply output and pop each character in the stack one by one—Figure (13.10*b*). As you can see, by the nature of the stack, the characters will be popped in reverse order.

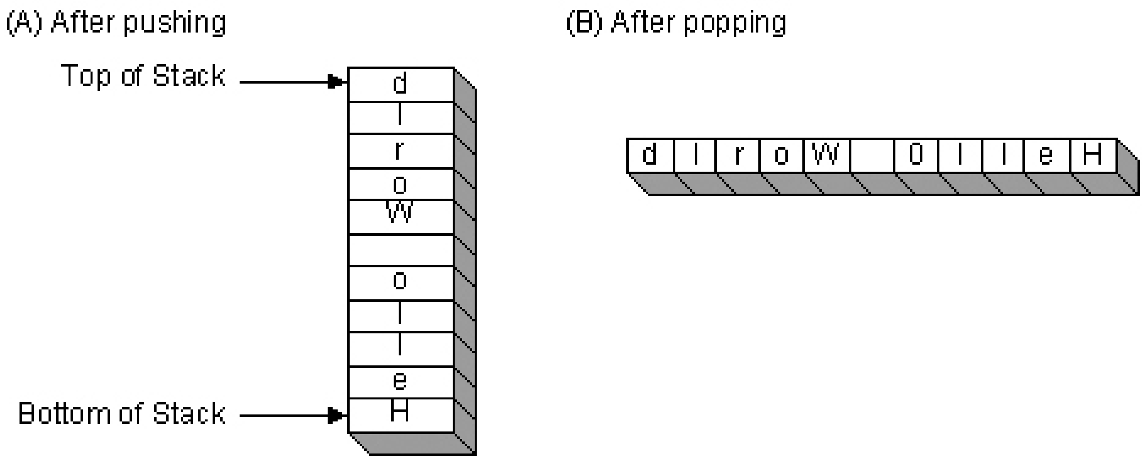


Figure 13.10: As we push a string onto a stack it gets stored backwards by the nature of the stack.

13.4 Queues

13.4.1 Theory

A **queue** is a **FIFO** (first in first out) data structure. A queue does what it sounds like—it manages a queue of data. Consider a line of customers in a store. The first customer in line is the first to be processed, the second customer is the second to be processed, and so on.

A queue container is useful for when you need to model a set of data that is naturally modeled in a first come, first serve situation. For example, when we get into Windows programming later in the course, we will find that our application responds to events. An event may be a mouse click, a key press, a window resize, etc. Generally, events should be processed in the order that they occur (a first come, first serve situation). As events occur, Windows (the OS) adds these events to an application "event queue." The application then processes these events one-by-one as fast as it can. Obviously, when the system is idle, no events occur and the event queue is empty.

In some situations, some Windows events are considered more important than others, and they should have "priority" over the other events. A queue that moves items ahead in the line is called a **priority queue** (`std::priority_queue`, also in <queue>). This happens in other situations as well; VIP clients may be allowed to "cut in line" in some business institutions. We will talk more about the Windows event system in the following chapters. For now we simply wanted to provide a real world utility of a queue.

As with a stack, a queue can be implemented with an array or linked list as the underlying container type.

13.4.2 Queue Operations

In the standard library, a queue is represented with `std::queue` (#include <queue>). `std::queue` contains five methods of interest.

1. `empty`: Returns `true` if the queue is empty (contains no items) or `false` otherwise.
2. `push`: Adds an item to the end of the queue.
3. `pop`: Removes the item from the front of the queue (the item first in line).
4. `front`: Returns a reference to the first item in the queue.
5. `back`: Returns a reference to the last item in the queue.

We now show a way to write a palindrome-testing program using a stack and queue.

Program 13.4: Using a stack and queue to determine if a string is a palindrome.

```
#include <queue>
#include <stack>
#include <iostream>
#include <string>
using namespace std;

int main()
{
      // Input a string.
      string input;
      cout << "Enter a string: ";
      getline(cin, input);

      // Add the strings characters to a stack and a queue.
      stack<char> wordStack;
      queue<char> wordQueue;
      for(int i = 0; i < input.size(); ++i)
      {
            wordStack.push(input[i]);
            wordQueue.push(input[i]);
      }

      // For each character in the stack and queue test to see if the
      // front and top characters match, if they don't then we can
      // conclude that we do not have a palindrome.
      bool isPalindrome = true;
      for(int i = 0; i < input.size(); ++i)
      {
            if( wordStack.top() != wordQueue.front() )
            {
                  isPalindrome = false;
                  break;
            }
            // Pop and compare next characters.
            wordStack.pop();
            wordQueue.pop();
      }

      if( isPalindrome )
            cout << input << " is a palindrome." << endl;
      else
            cout << input << " is _not_ a palindrome." << endl;
}
```

Program 13.4 Output 1

```
Enter a string: Hello World
Hello World is _not_ a palindrome.
Press any key to continue
```

Program 13.4 Output 2

```
Enter a string: abcdcba
abcdcba is a palindrome.
                     Press any key to continue
```

First, we input the string and place the characters into the respective data structure. Figure 13.11 shows an example of how the queue and stack look after this. One way of looking at a palindrome is as a string that is the same when read front-to-back or back-to-front. Due to the nature of the data structures, the queue stores the string in front-to-back order and the stack stores the string in back-to-front order. So, by comparing the front character of the queue with the top character of the stack, one-by-one, we can test if the string is the same when read front-to-back or back-to-front.

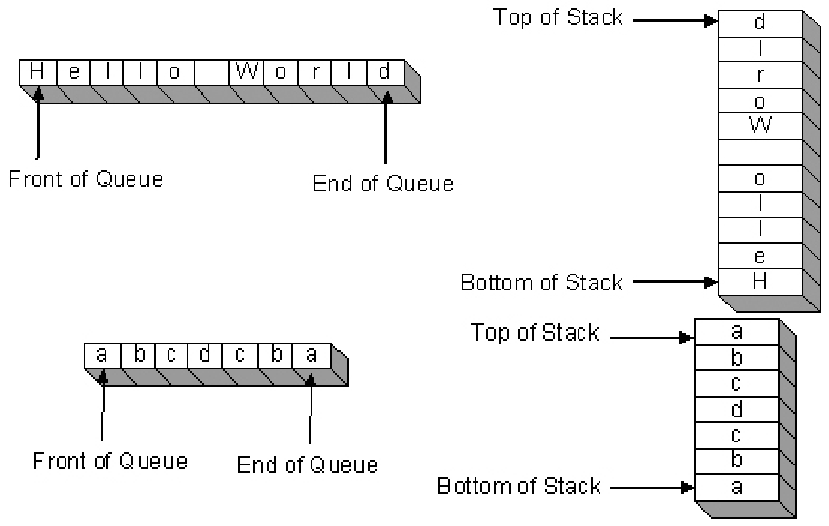


Figure 13.11: The stack and queue after pushing some characters into them.

13.5 Deques

13.5.1 Theory

A deque (pronounced "deck") is a double-ended queue. That is, we can insert and pop from both the front end and the back end. Clearly, this kind of behavior can be accomplished with a double-ended linked list. However, what makes the deque novel is that it uses an array internally. Thus, accessing the elements is still fast. Stroustrup's *The C++ Programming Language: Special Edition* gives a clear and concise description:

"*A deque is a sequence optimized so that operations at both ends are about as efficient as for a list, whereas subscripting approaches the efficiency of a vector. […] Insertion and deletion of elements "in the middle" have vector like (in)efficiencies rather than list like efficiencies. Consequently, a deque is used where additions and deletions take place 'at the ends.'*" (474).

13.5.2 Deque Operations

In the standard library, a deque is represented with `std::deque` (#include <deque>). `std::deque` contains five methods of interest.

1. `empty`: Returns `true` if the deque is empty (contains no items) or `false` otherwise.
2. `push_front`: Adds an item to the front of the deque.
3. `push_back`: Adds an item to the end of the deque.
4. `pop_front`: Removes an item from the front of the deque.
5. `pop_back`: Removes an item from the back of the deque.
6. `front`: Returns a reference to the first item in the deque.
7. `back`: Returns a reference to the last item in the deque.

In addition to the above-mentioned methods, you can get the size of a deque with the `size` method and you can access an element anywhere in the deque with the overloaded bracket operator [].

13.6 Maps

13.6.1 Theory

Often we want the ability to associate some search key with some value. For example, in writing a dictionary program and given a word (the key), we would like to be able to quickly find and extract the definition (the value). This is an example of a map; that is, the word maps to the definition. So far, a map does not seem like anything special, as we could just use an array or list and do a search for the specified item. However, what makes the C++ map interesting is that the searching is done very quickly because the internal map data structure is typically a red-black binary search tree, which is inherently optimized for fast searching—it is essentially a data structure that can always be searched in a binary search method by its very nature. We will not get into tree data structures in this course (see the *3D Graphics Programming Module II* course here at the Game Institute for a thorough explanation of tree data structures). Rather, we will just learn how to use the standard library map class.

Note: An important fact about maps is that the keys should be unique.

13.6.2 Insertion

A map is represented with the standard library `std::map` class (#include <map>). Unlike the other STL containers discussed so far, `std::map` takes two type parameters:

`map`<*key type*, *value type*>

For example, here we instantiate a map where the search key is a string, and the value type is an integer:

```
map<string, int> myMap;
```

To insert an item into the map, you use the overloaded bracket operator, where the argument inside the brackets denotes the key, and the right hand operand of the assignment operator specifies the value you wish to associate with that key:

```
myMap[key] = value; // Inserts value with associated key
```

If a value is already associated with that key (i.e., a value with that key has already been inserted into the map) then the preceding syntax overwrites the value at that key with the new value:

```
myMap[key] = value1;
myMap[key] = value2; // Overwrite the previous value at key with value2
```

Using our (`string`, `int`) map, we could insert values like so:

```
map<string, int> myStringIntMap;

myStringIntMap["Tim"]      = 22;
myStringIntMap["Vanessa"]  = 18;
myStringIntMap["Adam"]     = 25;
myStringIntMap["Jennifer"] = 27;
```

We will assume the integer represents the person's age. Realistically, we would probably associate the person's name with more than just an age—probably an entire `Person` object.

13.6.3 Deletion

To delete an item from the map, you use the `erase` method, where we specify the key of the item to erase:

```
// Remove adam from the map.
myStringIntMap.erase("Adam");
```

13.6.4 Traversal

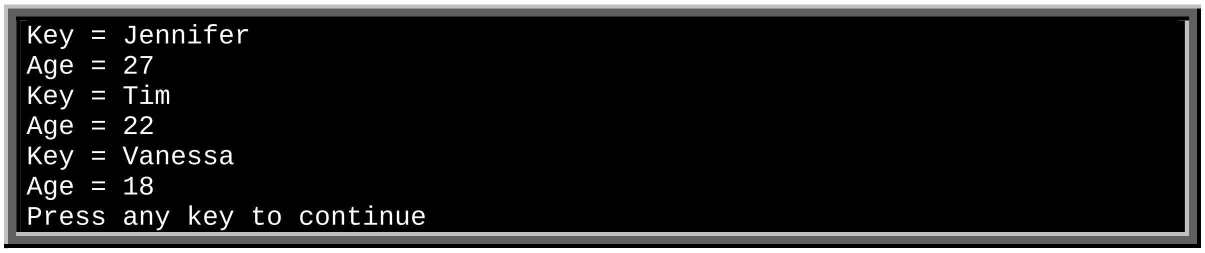
Traversals of the map are done with iterators. We use the `begin` method to get an iterator to the first element and we increment the iterator until we reach the end:

Program 13.5: Map traversal.

```
#include <map>
#include <string>
```

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

int main()
{
	map<string, int> myStringIntMap;

	myStringIntMap["Tim"]      = 22;
	myStringIntMap["Vanessa"]  = 18;
	myStringIntMap["Adam"]     = 25;
	myStringIntMap["Jennifer"] = 27;

	// Remove adam from the map.
	myStringIntMap.erase("Adam");

	map<string, int>::iterator i = 0;
	for(i = myStringIntMap.begin(); i != myStringIntMap.end(); ++i)
	{
		cout << "Key = " << i->first << endl;
		cout << "Age = " << i->second << endl;
	}
}
```

Program 13.5 Output



```
Key = Jennifer
Age = 27
Key = Tim
Age = 22
Key = Vanessa
Age = 18
Press any key to continue
```

What is interesting here is that, since a `map` has two associated values, the `map` iterator provides access to those two values via the `first` member of the map iterator (`i->first`) and via the `second` member of the map iterator (`i->second`).

13.6.5 Searching

We started our discussion with maps saying that, given a search key we can find the associated value very quickly with the `map` data structure. To find an item given the key, we use the `find` method:

```
map<string, int>::iterator i = 0;

i = myStringIntMap.find( "Vanessa" );

cout << i->first << " is " << i->second << " years old." << endl;
```

Assuming the data was inserted as before, this would output: `Vanessa is 18 years old.`

Note: All the containers we have studied in this chapter have been templates. This is because we want to use containers for many different *types* of objects. The template related code of the standard library is termed the **standard template library (STL)**.

13.7 Some Algorithms

The C++ standard library includes some pre-built functional algorithms (#include <algorithm>) which operate on data sets. These functions are template functions and can thus work on a variety of data types. To act on data sets, the algorithms typically need to traverse the data; to traverse a container the functions rely on iterators. For example, the `find` algorithm takes two iterator arguments marking the range of a container to search for a value:

```
vector<int> intVec(10);

// [...] Fill intVec

// Search for '5'
std::vector<int>::iterator i = 0;
i = find(intVec.begin(), intVec.end(), 5);
```

We could just as well use `find` with a `list`, `deque`, or `map`. Note that we use vectors in the examples, but they can work for all containers that can be navigated with iterators.

The *C++ Programming Language: Special Edition* states that there are 60 standard library algorithms. We will only present a few in this course to give you a general idea. The important thing to realize is that such an algorithmic library exists in C++, and information about it can be pursued at your leisure.

Note: It is recommended at this point that you spend some time perusing the complete reference documentation of the standard template library here:

http://www.sgi.com/tech/stl/table_of_contents.html

You do not need to read it all, but you should become familiar with the layout so that you can look things up as needed for reference, as we cannot cover the entire library here in one chapter.

13.7.1 Functors

To understand many of the standard library algorithms, we need to understand the idea of functors first. Basically, a **functor** is a "function object;" that is, an object that acts like a function. How do we do that? Well, when a function is called we have the syntax of the function name followed by parentheses: *functionname*(). We could simulate that syntax by creating a class and overloading the parenthesis operator:

```
class Print
{
public:
      void operator()(int t)
      {
            cout << t << " ";
      }
};
```

Then we can write:

```
Print p;
p( 5 ); // Print 5, notice how this looks like function syntax.
```

This may seem obscure; why not just create a `print` function? We give two reasons:

1. Functors are classes and thus can also contain member variables. This can lead to more flexible kinds of functions since we can store extra data in the member variables.

2. We can pass "functions" disguised as functors into other functions. These other functions, which receive a functor as an argument, can then invoke the function the functor represents as necessary and where necessary. You cannot do this with a traditional function.

Perhaps an example will clear things up. Let us examine the `generate` algorithm. The `generate` algorithm iterates over a container and calls a function for each element to "generate" a value for that element. The `generate` function cannot know what kind of value to generate—it is task-specific. Therefore, we must supply the generation function to `generate`, and `generate` will invoke that function for each element. However, we cannot pass a function as a parameter to a function. Syntax such as:

```
Foo( Bar() );
```

will simply pass the value which `Bar` evaluates into `Foo`—it does not pass in the function itself, and we cannot call it inside `Foo`. Therefore, to get around this, we pass a functor, which is an object that acts like a function. To clarify, here is a sample implementation of how `generate` might be implemented:

```
template<typename Iter, typename Function>
Function generate(Iter begin, Iter end, Function f)
{
    while( begin != end )
    {
        *begin = f(); // Invoke functor f on element
        ++begin; // Next
    }
    return f;
}
```

As you can see, `generate` needs to invoke *some* functor `f`. Notice the function is a template, so that it can work with *any* functor. In our example, we will generate a random number for each element in a container. We define the following functor:

```
class Random
{
public:
	Random(int low, int high)
		: mLow(low), mHigh(high)
	{
	}

	int operator()()
	{
		return mLow + rand() % (mHigh - mLow + 1);
	}

private:
	int mLow;
	int mHigh;
};
```

Then to generate a random number for each element we write:

```
int main()
{
	srand(time(0));

	// Allocate 15 integers
	vector<int> intVec;
	intVec.resize(15);

	Random r(2, 7);
	generate(intVec.begin(), intVec.end(), r);
}
```

> **Note**: By passing different functors into `generate`, you can have the `generate` function generate values in different ways, which thereby makes the generate function (and the other STL algorithms) extremely generic, since the client supplies the custom function in the form of a functor.

At this point we would like to print our randomly generated vector. Just as there is an algorithm that generates a value for each element, there is also an algorithm that "does something" for each element, where we define the "something" as a functor. This algorithm is called `for_each`. Let us now restate the printing functor we showed earlier:

```
class Print
{
public:
	void operator()(int t)
	{
		cout << t << " ";
	}
};
```

To print each element we would write:

```
Print p;
for_each(intVec.begin(), intVec.end(), p);
```

Let us now put everything together into one compilable program and examine the results.

Program 13.6: `generate` and `for_each` demonstration.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;

class Random
{
public:
      Random(int low, int high)
            : mLow(low), mHigh(high)
      {
      }

      int operator()()
      {
            return mLow + rand() % (mHigh - mLow + 1);
      }

private:
      int mLow;
      int mHigh;
};

class Print
{
public:
      void operator()(int t)
      {
            cout << t << " ";
      }
};

int main()
{
      srand(time(0));

      // Allocate 15 integers
      vector<int> intVec;
      intVec.resize(15);

      Random r(1, 20);
      generate(intVec.begin(), intVec.end(), r);
```

```
	Print p;
	for_each(intVec.begin(), intVec.end(), p);

	cout << endl;
}
```

Program 13.6 Output

```
14 19 20 15 2 10 12 5 2 9 3 15 8 6 17
Press any key to continue
```

Note that in practice we would make `Print` a template functor so that it could print many different kinds of types.

13.7.2 Some More Algorithms

Sometimes we may want to count the number of times a particular value appears in a sequence. We can do this with the `count` algorithm. Here is some sample code:

Program 13.7: `count` demonstration.

```
// [...] Functors snipped.
int main()
{
	srand(time(0));

	// Allocate 15 integers
	vector<int> intVec;
	intVec.resize(15);

	Random r(1, 10);
	generate(intVec.begin(), intVec.end(), r);

	Print p;
	for_each(intVec.begin(), intVec.end(), p);

	cout << endl;

	// count how many times '5' occurs.
	int result = count(intVec.begin(), intVec.end(), 5);

	cout << "5 appears " << result << " times." << endl;
}
```

Program 13.7 Output

```
5 8 1 7 1 10 10 5 9 10 7 5 2 2 1
5 appears 3 times.
Press any key to continue
```

Another useful algorithm is the `replace` algorithm. This algorithm will replace every occurrence of a specified value in a container with a new value. Sample snippet:

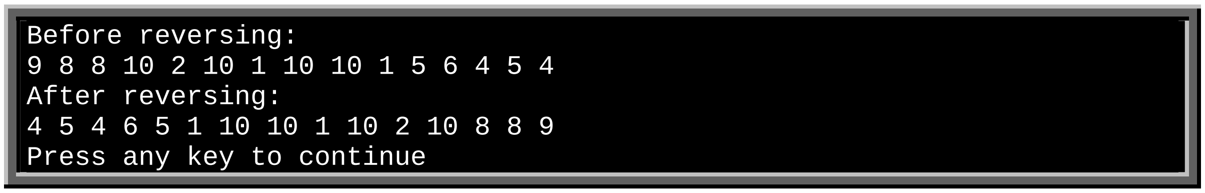
```
// replace 5 with 99.
replace(intVec.begin(), intVec.end(), 5, 99);
```

The `reverse` algorithm will reverse the order of a container. The following sample illustrates:

Program 13.8: `reverse` demonstration.

```
// [...] Functors snipped.
int main()
{
	srand(time(0));

	// Allocate 15 integers
	vector<int> intVec;
	intVec.resize(15);

	Random r(1, 10);
	generate(intVec.begin(), intVec.end(), r);

	cout << "Before reversing: " << endl;
	Print p;
	for_each(intVec.begin(), intVec.end(), p);
	cout << endl;

	cout << "After reversing: " << endl;
	reverse(intVec.begin(), intVec.end());

	for_each(intVec.begin(), intVec.end(), p);
	cout << endl;
}
```

Program 13.8 Output



```
Before reversing:
9 8 8 10 2 10 1 10 10 1 5 6 4 5 4
After reversing:
4 5 4 6 5 1 10 10 1 10 2 10 8 8 9
Press any key to continue
```

> **Note**: To emphasize, these algorithms will work for any containers that can be iterated over, such as a list, deque, map, or array. Also remember that there are 60 algorithms and we have only mentioned a handful. The goal here was to give you an idea of what exists and what the standard library algorithms can do for you. Be sure to check the documentation contained at:
>
> http://www.sgi.com/tech/stl/table_of_contents.html
>
> or at the MSDN library (http://msdn.microsoft.com/).

13.7.3 Predicates

A predicate is a special functor that returns a `bool` value. For example, to implement a generic sorting algorithm we need to know how to compare two objects. For simple types, using the comparison operators (<, >) may suffice. However, for more complex objects, a more complex comparison operation may be needed. Predicates allow us to define and supply the comparison "function" (the predicate functor) that should be used with the algorithm. For example, in one program the staff recently needed to sort a list of 3D objects based on their distance from the virtual camera in a 3D scene. To facilitate this, we made the following predicate:

```
class TriPatchPredicate
{
public:
	TriPatchPredicate(const D3DXVECTOR3& eyePosL)
		: mEyePosL(eyePosL){}

	bool operator()(TriPatch* lhs, TriPatch* rhs)
	{
		// Sort front to back the list based on distance from
		// the eye.
		D3DXVECTOR3 a = lhs->aabb.center() - mEyePosL;
		D3DXVECTOR3 b = rhs->aabb.center() - mEyePosL;

		float d0 = D3DXVec3Length( &a );
		float d1 = D3DXVec3Length( &b );

		// Is the distance d0 less than d1.
		return d0 < d1;
	}
private:
	D3DXVECTOR3 mEyePosL;
};
```

This code has some processes occurring which you are not familiar with, but what is important is the predicate, which as you can see, is a functor where the overloaded parenthesis operator returns a `bool`.

To use this predicate in a sorting algorithm, we wrote:

```
TriPatchPredicate tpp(gMainCamera->getPos());
mVisibilityList.sort( tpp );
```

Where `mVisibilityList` is a `std::list` object.

Despite predicates giving increased flexibility in terms of how to define a logical operation, sometimes you may just want to use the basic logical operations with the standard library. To facilitate this, the standard library provides some predefined predicates that just do what their corresponding operator does:

Predicate	Corresponding Operator
`equal_to`	`==`
`Not_equal_to`	`!=`
`Greater`	`>`
`Less`	`<`
`greater_equal`	`>=`
`less_equal`	`<=`
`logical_and`	`&&`
`logical_or`	`||`
`logical_not`	`!`

To use these pre-built predicates you must #include <functional>.

13.8 Summary

1. Arrays have three general problems that motivate us to look at other data structures. First, inserting or deleting a value in the middle of the array is slow because it requires the data elements to be shifted about. Second, resizing an array can be slow because a memory allocation and deallocation must be performed and the array elements must be copied from the old array into the newly resized array. Third, arrays are not optimized for searching inherently. Still, these problems are only significant in some cases: If you are only adding or deleting elements at the end of an array then problem one disappears; if you do not need to resize the array frequently then problem two becomes negligible; and if you do not need to search the array then problem three becomes insignificant. The primary advantage of arrays is that they we can access random elements in the array very efficiently with only an offset index into the array.

2. A linked list can be thought of as a chain of data elements, which are called nodes. A linked list that allows forward and backward traversals is called a double-ended linked list. The main benefit of linked lists is that random insertions and deletions are efficient because no shifting must occur—only some pointer reassignments occur. The main disadvantage of a linked list is that we cannot randomly access nodes quickly (as we can with arrays); that is, the list must be traversed one by one until we find the node we seek. Linked lists can grow pretty efficiently, as there is no necessary copy from the old container to the newly resized container. Although, for each item we add/delete we must do a memory allocation/deallocation.

3. A stack is a LIFO (last in first out) data structure. A stack does what it sounds like—it manages a stack of data (i.e., data stacked on top of one another). A stack container is useful for when you need to model a set of data that is naturally modeled in a stack-like fashion. As far as vocabulary is concerned, placing items on the stack is called pushing and taking items off the stack is called popping.

4. A queue is a FIFO (first in first out) data structure. A queue does what it sounds like—it manages a queue of data. Consider a line of customers in a store. The first customer in line is the first to be processed; the second customer is the second to be processed, and so on. Essentially, a queue can model a "first come, first serve" system.

5. A deque (pronounced "deck") is a double-ended queue. That is, we can insert and pop from both the front end and the back end. This kind of behavior can be accomplished with a double-ended linked list. However, what makes the deque novel is that it uses an array internally. Thus, accessing the elements is still fast.

6. A map associates a search key with a value. Given the search key, the map can find the associated value very quickly. Maps are inherently optimized for searching due to their underlying implementation (usually a red-black binary search tree). Maps should be preferred when you need to search a container often.

7. The standard library provides a plethora of algorithms that can operator on the various STL containers. Such algorithms include, but are not limited to, searches, reversals, counting, generating, and sorting. To use some of the standard library algorithms you must supply a functor, which is a function object (an object that behaves like a function by overloading the parenthesis operator). In addition, to use other standard library algorithms you must supply a predicate, which is a particular kind of functor that returns a `bool`. Predicates are used to define complex comparison functions in the case that the built-in comparison operations (e.g., <, >) are unsatisfactory.

13.9 Exercises

13.9.1 Linked List

Write your own linked list class. More specifically, implement the class definition of the linked list discussed in this chapter:

```
template<typename T>
class LinkedList
{
public:
	LinkedList();
	~LinkedList();
	LinkedList(const LinkedList& rhs);
	LinkedList& operator=(const LinkedList& rhs);

	bool isEmpty();
	LinkedNode* getFirst();
	LinkedNode* getLast();

	void insertFirst(T data);
	void insertLast(T data);
	void insertAfter(T tKey, T tData);
	void removeFirst();
	void removeLast();
	void remove(T removalCandidate);
	void destroy();
private:
	LinkedNode* mFirst;
	LinkedNode* mLast;
};
```

13.9.2 Stack

Write your own stack class. That is, implement the following class definition:

```
template<typename T>
class Stack
{
public:
	Stack();
	~Stack();

	T& getTopItem();

	bool isEmpty(void);
	void push(T newElement);
	void pop();
private:
      LinkedList<T> mList;
};
```

13.9.3 Queue

Write your own queue class. That is, implement the following class definition:

```
template<class T>
class Queue
{
public:
      Queue();
      ~Queue();

      T& getFirst();
      bool isEmpty();

      void push(T newElement);
      void pop();
private:
      LinkedList<T> mList;
};
```

13.9.4 Algorithms

Design and implement your own `for_each`, `count`, `reverse`, and `sort` algorithms for an array.

Chapter 14

Introduction to Windows Programming

Introduction

With the core C++ language behind us, we now begin the second major theme of this course: Windows (Win32 for short) programming. What we mean by Win32 programming is that instead of writing console applications, we will learn to write programs that look like traditional Windows programs. You are, no doubt, used to seeing and working with these programs. Figure 14.1 shows a basic Win32 program and outlines some of its characteristics, which you should already be familiar with.

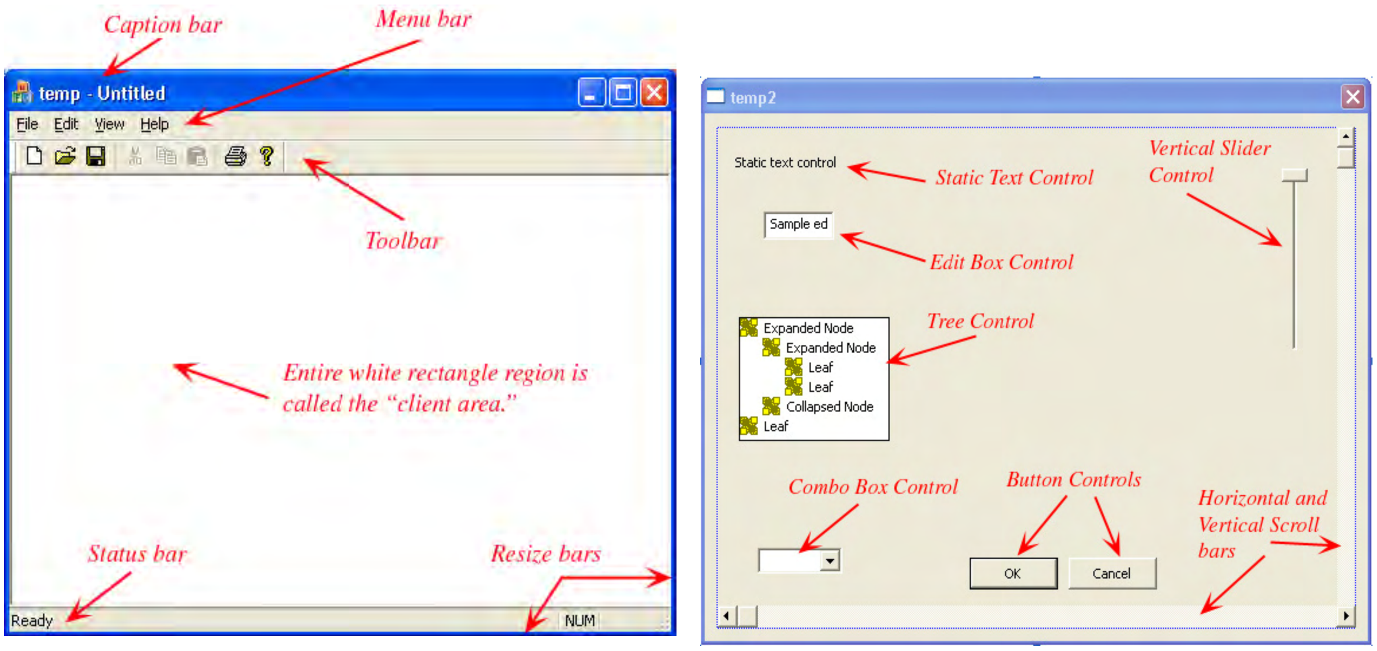


Figure 14.1: GUI features labeled.

The primary characteristic of a Win32 application is the **GUI (Graphical User Interface).** GUI refers to the graphics with which the user interacts—the menus, buttons, scroll bars, etc. Instead of interacting with a console, users now interact with a GUI.

Students often have difficulty making the transition from pure C++ to Windows programming. The primary reason is that to program Windows, we must use a set of code which Microsoft developed that is known as the **Win32 API (Application Programming Interface).** The Win32 library is a large low-level set of functions, structures, and types that allow you to create and customize windows.

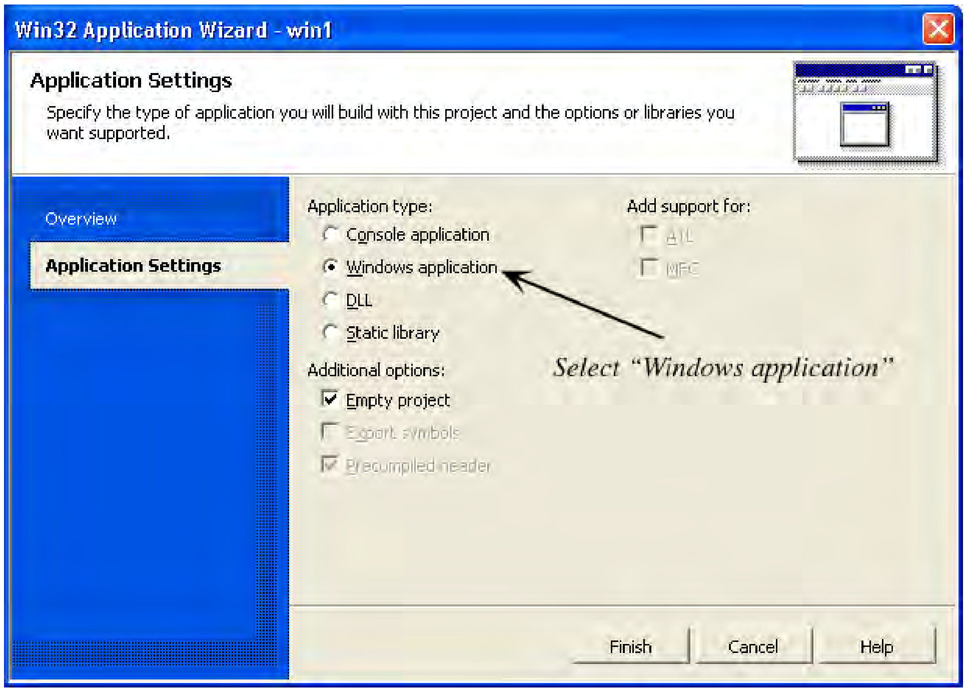
At this point, you are already familiar with using library code with the STL. However, unlike the STL where you can pick and choose the pieces of code you want to use, in Win32 programming you will have to use quite a bit of library code from the start. As all of the new structures, types, and functions, must be introduced fairly rapidly, and in combination with the new concepts, this can be daunting at first. Furthermore, the Win32 library code will do some things which you will probably not understand (at least, not understand what the code is doing internally). So you should take it slow, and do not get frustrated if there is something you do not fully understand—the theory will come in time. For now just concentrate on getting the programs working rather than on what the Win32 library is doing behind the scenes.

Chapter Objectives

- Learn how to create a basic Win32 application.
- Gain an understanding of the event driven programming model.

14.1 Your First Windows Program

The first thing you need be aware of when creating Win32 applications is that you must create your Visual C++ projects a little differently than you do with a console application. In particular, instead of selecting "Console application" from the "Application Settings" dialog box, you must select "Windows application," as Figure 14.2 shows.


Win32 Application Wizard - win1

Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:
- ( ) Console application
- (•) Windows application
- ( ) DLL
- ( ) Static library

Add support for:
- [ ] ATL
- [ ] MFC

Additional options:
- [✓] Empty project
- [ ] Export symbols
- [✓] Precompiled header

Select "Windows application"

Figure 14.2: Window Application Settings for a "Windows application."

By selecting "Windows application," Visual C++ will include all the necessary Win32 library files you need in order to write Win32 programs. That is the only necessary project change. Let us start by examining the following simple Win32 program:

> **Note:** When we use the Win32 API and write Win32 programs, you must be aware that the code is Windows specific; that is, it is no longer portable code.

Program 14.1: Your First Windows Program.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   PSTR cmdLine,
                   int showCmd)
{
      MessageBox(0, "First Win32 Program.", "Window Title", MB_OK);
}
```

This program displays the following message box as output:

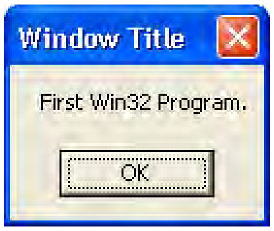


Figure 14.3: Output to Program 14.1.

Program 14.1 is useful because it allows us to introduce three Win32 concepts concisely.

First, in order to write Win32 programs, you need to include the Windows header file (#include <windows.h>); this header file provides the prototypes and definitions of various functions and types you will need to program the Win32 API. Second, Win32 programs no longer start with the `main` function. Instead, they start with the `WinMain` function. `WinMain` returns an integer, which specifies the code that is returned to the Windows operating system. The prefix `WINAPI` modifies the calling convention of `WinMain`; that is, the way in which it places items on the memory stack. Such low-level details are best left to a course in assembly language, and so we will not discuss it further here.

The first parameter, `hInstance` of type `HINSTANCE`, is essentially a value that identifies the application for Windows—an application ID, if you will, that Windows OS passes into your application when it begins. It is necessary to ID various applications because Windows can be running several different applications at once. Note that the Win32 API defines the type `HINSTANCE`.

The second parameter `hPrevInstance` is no longer used in 32-bit Windows programming—it is a legacy of 16-bit Windows.

The third parameter `cmdLine` is a string of command line arguments. (`PSTR` is essentially a `typedef` for a `char*`). Command line arguments are string arguments a user can pass into an application before it starts. Command line arguments typically give the application special instructions on how it should execute.

Finally, the fourth parameter, `showCmd`, is an integer that specifies how the main application window should be initially shown. For example, should it be maximized, minimized or normal? The Windows OS makes the decision based on a variety of factors. For instance, if you try to launch an application while the system is busy, Windows will pass in a value for `showCmd` indicating that it, perhaps, be minimized. In summary, the parameters to `WinMain` are parameters the OS passes into the application when the application starts.

The third key idea Program 14.1 illustrates is the `MessageBox` function. Again, this is a function the Win32 API defines and implements. MSDN gives the following declaration:

```
int MessageBox(
  HWND hWnd,          // handle to owner window
  LPCTSTR lpText,     // text in message box
  LPCTSTR lpCaption,  // message box title
  UINT uType          // message box style
);
```

Before getting to the specifics of this function, there are a few fundamental Win32 types we need to briefly discuss. First, an `HWND` is an ID for a particular window, just like an `HINSTANCE` is an ID for a particular application. The 'H' in both of these stands for **handle**: `HWND` means "windows handle" and `HINSTANCE` means "application instance handle." Again we need these IDs or handles, so that we can refer to individual Windows objects, such as applications and windows. The second new type is `LPCTSTR`; this is nothing more than a `typedef` to a `const char*` (if Unicode[1] is not defined). Finally, `UINT` is a `typedef` for `unsigned int`.

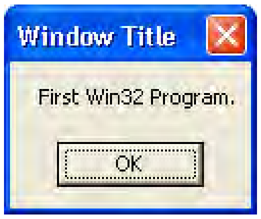
As the comment says, the first parameter (`hWnd`) is a handle to the window that will "own" the message box. That is, the handle of the window with which you want to associate the message box. You can pass null (0) for this parameter if you do not want to specify any association.

The second parameter (`lpText`) is the string of text you want to be displayed in the message box's body. In Program 14.1 we specify "First Win32 Program." As the output shows, this is the string that is displayed in the message box's body.

The third parameter (`lpCaption`) is the string that is to be displayed in the message box's caption bar. In Program 14.1 we specify "Window Title" for this parameter, and sure enough, that is the string displayed in the message box's caption bar.

The fourth parameter (`uType`) is an unsigned integer value that denotes a style flag. Here is an abridged list of possible style flags (these are predefined values of type `unsigned int`):

`MB_OK`: Instructs the message box to display an OK button.


Window Title

First Win32 Program.

Figure 14.4: The MB_OK message box style.

[1] Unicode uses 16-bit characters, which allows for a far greater range of characters we can represent. This is useful for supporting international character sets. We do not use Unicode in this book.

`MB_OKCANCEL`: Instructs the message box to display an OK and CANCEL button.

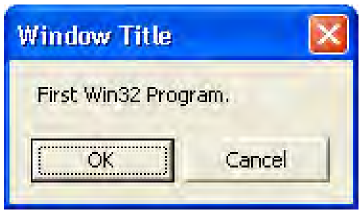


Figure 14.5: The MB_OKCANCEL message box style.

`MB_YESNO`: Instructs the message box to display a YES and NO button.

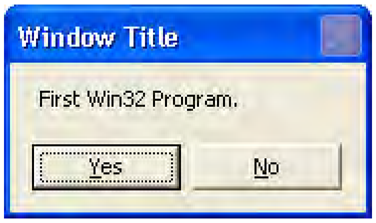


Figure 14.6: The MB_YESNO message box style.

`MB_YESNOCANCEL`: Instructs the message box to display a YES, NO, and CANCEL button.

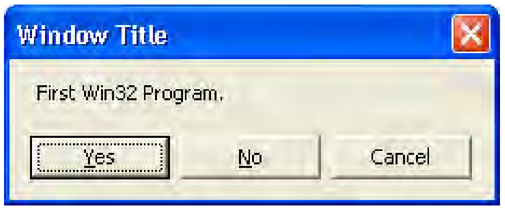


Figure 14.7: The YESNOCANCEL message box style.

Finally, the message box's return value depends on which button the user pressed; here is an abridged list of return values (see the Win32 documentation for more details):

`IDOK`: The user pressed the OK button.
`IDCANCEL`: The user pressed the CANCEL button.
`IDYES`: The user pressed the YES button.
`IDNO`: The user pressed the NO button.

You can test which value was returned using an "if" statement and thus determine which button the user selected and then take appropriate action.

Note: Many of the Win32 functions will have different style flags or values that enable you to customize various things. However, because there are so many different flags for the different API functions, we cannot cover all of them in this text. Therefore, it is important that you learn to use the Win32 documentation to obtain further info on a Win32 function or type. The documentation is typically included in the help file of Visual C++. For example, in Visual C++ .NET, you would go to the *Menu->Help->Index* (Figure 14.8).

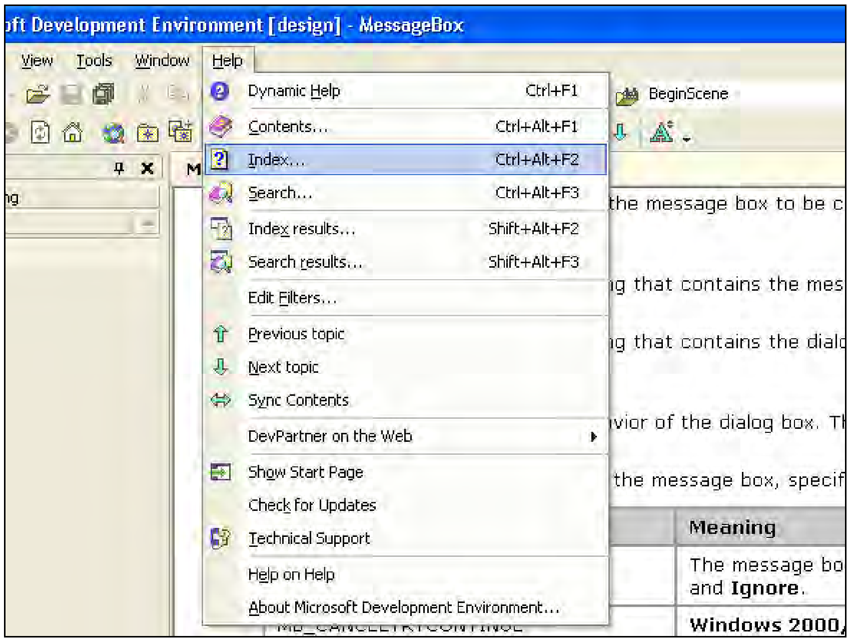


Figure 14.8: Launching the documentation index.

The index help should then open to the right of the interface. Enter the function of type you would like more information on, as Figure 14.9 shows we search for `MessageBox`.

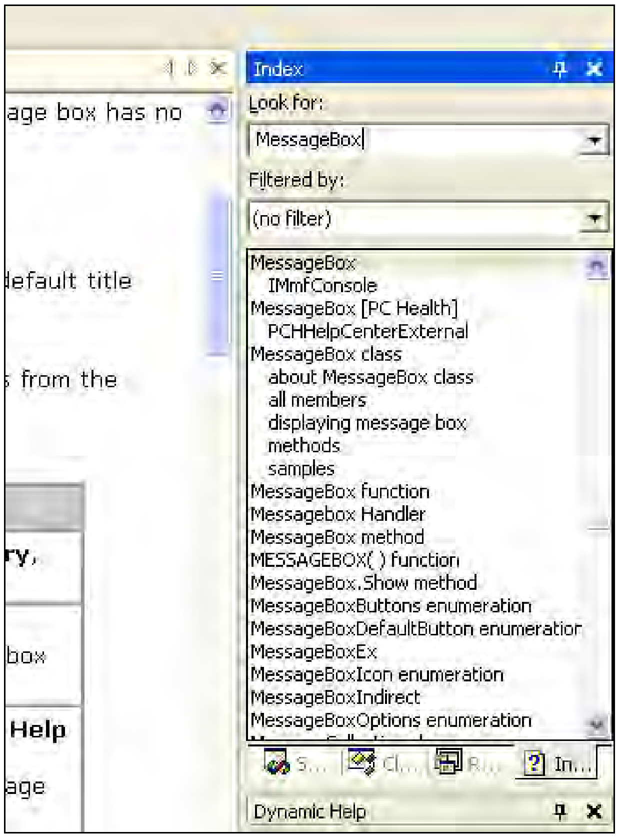
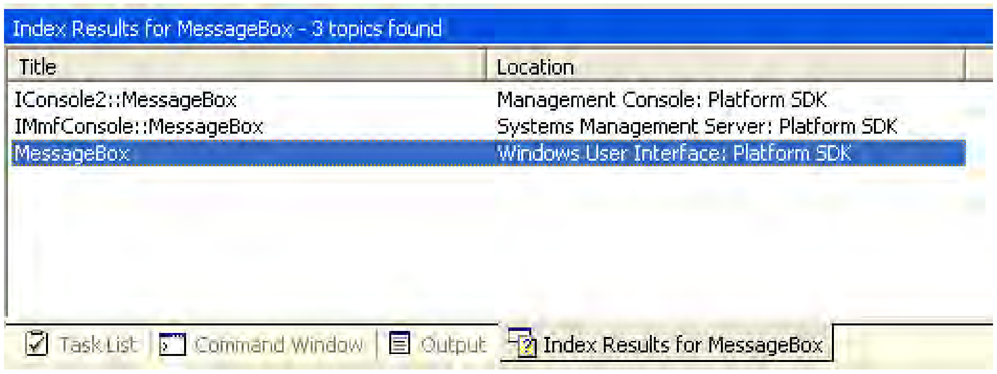


Figure 14.9: Searching for the MessageBox documentation.

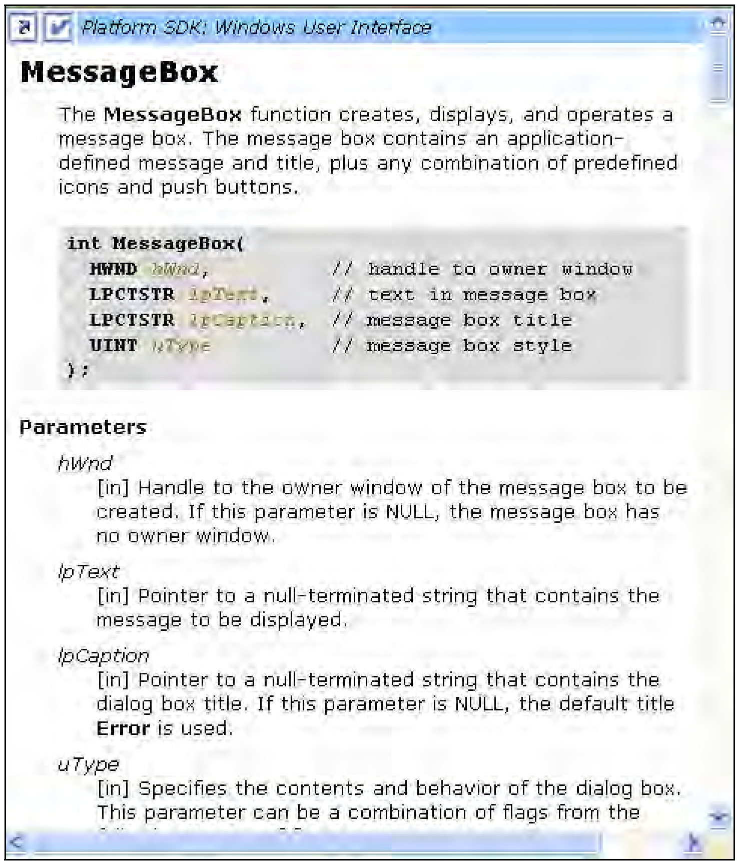
Finally, selecting (double click) on the found `MessageBox` entry gives us several subcategories more information can be found:


Index Results for MessageBox - 3 topics found

Title	Location
IConsole2::MessageBox	Management Console: Platform SDK
IMmfConsole::MessageBox	Systems Management Server: Platform SDK
MessageBox	Windows User Interface: Platform SDK

Figure 14.10: Selecting the MessageBox documentation for the Win32 API.

Here, we want to select the "Windows User Interface: Platform SDK;" this is essentially the Win32 API documentation guide. So selecting that yields the complete documentation for the `MessageBox` function:


Platform SDK: Windows User Interface

MessageBox

The **MessageBox** function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

```
int MessageBox(
  HWND hWnd,          // handle to owner window
  LPCTSTR lpText,     // text in message box
  LPCTSTR lpCaption,  // message box title
  UINT uType          // message box style
);
```

Parameters

hWnd
[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText
[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption
[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title **Error** is used.

uType
[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags from the

Figure 14.11: The MessageBox documentation.

14.2 The Event Driven Programming Model

14.2.1 Theory

One of the key differences between the console programming we have been doing since Module I and Windows programming is the **event driven programming model.** In console programming, your code begins at `main` and then executes line-by-line, while looping, branching, or jumping to function calls along the way. Windows programming is different. Instead, a Windows program typically sits and waits for something to happen—an **event.** An event can be a mouse click, a button press, a menu item selection, a key press, and so on. Once Windows recognizes an event, it adds a message to the application's **priority message queue** for which the event was targeted (remember Windows can be running several applications concurrently). A Windows application constantly scans the message queue for messages and when one is received it is forwarded to the window it was intended for (a single Windows application can consists of multiple windows itself—a main window and child windows, for example). More specifically, a message in the application message queue is forwarded to the **window procedure** of the window it was intended for.

The window procedure (also called a message handler) is a special function each window has (though several windows can share the same message procedure), which contains the code necessary to handle the specified event the message originated from. For example, if a button is pressed (a button is a child window to the parent window it lies on) then the button's window procedure will contain the code that gets executed when that button is pressed.

14.2.2 The `MSG` Structure

A message in Windows is represented with the following `MSG` structure:

```
struct MSG {
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
};
```

`hwnd`: This member is the handle to the window for which the message is designated.

`message`: This member is a predefined unique unsigned integer symbol that denotes the specific type of message.

`wParam`: A 32-bit value that contains extra information about the message. The exact information is specific to the particular message.

`lParam`: Another 32-bit value that contains extra information about the message. The exact information is specific to the particular message.

`time`: The time stamp at which time the message was generated.

`pt`: The (x, y) coordinates, in screen space, of the mouse cursor at the time the message was generated. The `POINT` structure is defined by the Win32 API and looks like this:

```
struct POINT {
  LONG x; // x-coordinate
  LONG y; // y-coordinate
};
```

Here are some example message types, which would be placed in `message`:

`WM_QUIT`: This message is sent when the user has indicated their desire to quit the application (by pressing the close 'X' button, for example).

`WM_COMMAND`: This message is sent to a window when the user selects an item from the window's menu. Some child windows, such as button controls, also send this message when they are pressed.

`WM_LBUTTONDBLCLK`: This message is sent to a window when the user double-clicks with the left mouse button over the window's client area.

`WM_LBUTTONDOWN`: This message is sent to a window when the user presses the left mouse button over the window's client area.

`WM_KEYDOWN`: This message is sent to the window with keyboard focus when the user presses a key. The `wParam` of the message denotes the specific key that was pressed.

`WM_SIZE`: This message is sent to a window when the user resizes the window.

14.3 Overview of Creating a Windows Application

The creation of even a simple Windows application is quite lengthy in terms of lines of code, but not too lengthy when we consider the amount of functionality we will get in return. We will have actually drawn a window (we have not done any drawing in this course so far), and moreover, the window can be resized, minimized, maximized, and other such things. The key steps for creating a basic Windows application are outlined below:

1. Define the window procedure for the main application window. Recall that a window procedure is a special function each window has (though several windows can share the same message procedure), which contains the code necessary to handle the specified event the message originated from.

2. Fill out a `WNDCLASS` instance. By filling out this structure you are able to define some core properties that your window will have.

3. Register the `WNDCLASS` instance. Before you can create a window based on the `WNDCLASS` instance you have filled out, you must register it with Windows.

4. Create the window. Now that you have registered a `WNDCLASS` instance with Windows, you can create a window. Creating a window is done with a single function call, which again allows you to customize some of the features of the window.

5. Show and update the window. In this step, you need to actually instruct Windows to display (show) your window (by default it will not be visible). In addition, you must update the window for the first time.

6. Finally, enter the message loop. After you have created the main window of your application, you are ready to enter the message loop. The message loop will constantly check for and handle messages as the message queue gets filled. The message loop will not exit until a quit message (`WM_QUIT`) is received.

With the preceding basic roadmap in place, we can now discuss the details of each step.

14.3.1 Defining the Window Procedure

As already stated, a window procedure is a special function each window has (though several windows can share the same message procedure), which contains the code necessary to handle the specified event from which the message originated. However, the window procedure must follow some Win32 API guidelines. In particular, all window procedures must have a certain declaration:

```
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

You can name the window procedure whatever you want; here we call it `WndProc`. Additionally, you can name the parameters whatever you want as well (although the names used above are very commonly used); however, all four parameters of the shown types must be present. The return type of the window procedure must be of type `LRESULT`, which is simply `typedefed` as a `long`—an error code will be returned via this return value. Observe that the function name is prefixed with the symbol `CALLBACK`. This denotes that the window procedure is a **callback function.** A callback function is a function that we do not directly call ourselves. Rather, the Win32 API will call this function automatically. In particular, the Win32 API will call the window procedure function when a message from the message loop is dispatched to it.

As you can see, the window procedure takes four parameters. Together, these parameters provide you with enough information to handle the message.

`hWnd`: The handle to the window the message is aimed for. This parameter corresponds with the `MSG::hwnd` member.

`msg`: A predefined unique unsigned integer symbol that denotes the specific type of message. This parameter corresponds with the `MSG::message` member.

`wParam`: A 32-bit value that contains extra information about the message. The exact information is specific to the particular message. This parameter corresponds with the `MSG::wParam` member.

`lParam`: Another 32-bit value that contains extra information about the message. The exact information is specific to the particular message. This parameter corresponds with the `MSG::lParam` member.

And again, just to reiterate, the Win32 API will call the window procedure passing the appropriate arguments into the window procedure's parameters.

Now that we know how the window procedure must be declared, how would we go about implementing it? A window procedure is typically implemented as one large switch statement. The switch statement is used to determine which block of code should be executed based on the specific message. For example, if the left mouse button was pressed, then the code to handle a `WM_LBUTTONDOWN` message should be executed. Likewise, if a key was pressed then the code to handle a `WM_KEYDOWN` message should be executed. Here is an example:

```
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      switch( msg )
      {
      case WM_LBUTTONDOWN:
            MessageBox(0, "WM_LBUTTONDOWN message.", "Msg", MB_OK);
            return 0;

      case WM_KEYDOWN:
            if( wParam == VK_ESCAPE )
                  DestroyWindow(hWnd);
            return 0;

      case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
      }

      return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

The switch statement grows as you add different kinds of messages to handle. In the above code we only directly handle three types of messages. Let us now examine the components of this function definition.

```
case WM_LBUTTONDOWN:
      ::MessageBox(0, "Hello, World", "Hello", MB_OK);
      return 0;
```

If the sent message is of type `WM_LBUTTONDOWN`, then our particular window (remember window procedures are window specific—you define one, unless you share them, for each window) displays a message box. Once we handle the message, we are done and, therefore, can return out of the function (return 0).

```
case WM_KEYDOWN:
      if( wParam == VK_ESCAPE )
            ::DestroyWindow(hWnd);
      return 0;
```

If the sent message is of type `WM_KEYDOWN` and the key pressed was the escape key (symbolically defined in code as `VK_ESCAPE`) then we have specified that the window should be destroyed (`DestroyWindow(hWnd)`). The only parameter for `DestroyWindow` is a handle to the window that is to be destroyed. This function sends a `WM_DESTROY` message to the window identified by `hWnd`. Observe that for the `WM_KEYDOWN` message, the `wParam` contains the key code for the key that was pressed. Again, `wParam` and `lParam` provide extra message specific information—some messages do not need extra information and these values are zeroed out.

```
case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
```

The last type of message we specifically handle is the `WM_DESTROY` message. This message would be sent if the user presses the escape key (we defined the window to be destroyed if the escape key was pressed in the previous message) or if the user pressed the 'X' button on the window to close it. In response to this message, we use the `PostQuitMessage` API function to add a `WM_QUIT` message to the application message queue. This will effectively terminate the loop so that the program can end. The only parameter to `PostQuitMessage` is an exit code and this is almost always zero.

There is also some functionality that is common to almost every window. For example, just about every window can be resized, minimized and maximized. It seems redundant to define this behavior repeatedly for each window. Consequently, the Win32 API provides a default window procedure that implements this common generic functionality. So for any message we do not specifically handle, we can just forward the message off the default window procedure:

```
return DefWindowProc(hWnd, msg, wParam, lParam);
```

This buys us some extra, albeit generic, functionality for free. If you do not like the default behavior, then you simply handle the message yourself in the window procedure so that it never gets forwarded to the default window procedure.

> **Note:** While small Windows programs typically only have one window procedure, large windows programs will usually have much more. Games usually only have one, because games do not typically work much with the Win32 API; rather they use a lower level API such as DirectX.

14.3.2 The WNDCLASS Structure

An instance of the WNDCLASS structure is used to define the properties of your window, such as styles, the background color, the icon image, the cursor image, and the window procedure associated with any window you create based on this WNDCLASS instance. Here is the definition:

```
typedef struct _WNDCLASS {
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCTSTR     lpszMenuName;
    LPCTSTR     lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

style: A combination of style flags for customization. There exists a myriad of bit flags that can be combined to create various styles; see the Win32 documentation for details. Generally, the flag combination CS_HREDRAW and CS_VREDRAW are used as the style (combined via a bitwise OR), which means the window will repaint itself when either the horizontal or vertical window size changes.

lpfnWndProc: Pointer to the window procedure you want to associate with the windows that are built based on this WNDCLASS instance.

cbClsExtra: Extra 32-bit memory slot to reserve custom information. We do not use this value in this course.

cbWndExtra: Extra 32-bit memory slot to reserve custom information. We do not use this value in this course.

hInstance: A handle to the application with which you want the windows you create to be associated. Recall that WinMain passes in the application instance handle through its first parameter.

hIcon: A handle to an icon which will be used for the window. You can get a handle to an icon via the API function LoadIcon. To load the default application icon, for example, you would write:

```
LoadIcon(0, IDI_APPLICATION); // returns an HICON
```

Some other intrinsic icons are:

- IDI_WINLOGO – Windows logo icon
- IDI_QUESTION – Question mark icon
- IDI_INFORMATION – Information icon
- IDI_EXCLAMATION – Exclamation icon

`hCursor`: A handle to a cursor which will be used for the window. You can get a handle to a cursor with the API function `LoadCursor`. To load the default arrow cursor, for example, you would write:

```
LoadCursor(0, IDC_ARROW); // returns an HCURSOR
```

Some other intrinsic cursors are:

- `IDC_CROSS` – Crosshair cursor
- `IDC_WAIT` – Hourglass cursor

`hbrBackground`: A handle to a brush which specifies the windows background color. You can get a handle to a brush with the API function `GetStockObject`. To get a handle to a white brush, for example, you would write:

```
(HBRUSH)GetStockObject(WHITE_BRUSH); // returns a HBRUSH
```

Note that we have to cast the return value to an `HBRUSH`. Some other intrinsic brush types are:

- `BLACK_BRUSH` – Black brush
- `DKGRAY_BRUSH` – Dark gray brush
- `GRAY_BRUSH` – Gray brush
- `LTGRAY_BRUSH` – Light gray brush

`lpszMenuName`: The name of the window menu. We will be creating and enabling menus via another method, so we will be setting this value to zero.

`lpszClassName`: A unique string name (identifier) we want to give the `WNDCLASS` instance, so that we can refer to it later. This can be any name you want.

A typical `WNDCLASS` instance would be created and filled out like so:

```
WNDCLASS wc;

wc.style         = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc   = WndProc;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;
wc.hInstance     = hInstance;
wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
wc.hbrBackground = (HBRUSH)::GetStockObject(WHITE_BRUSH);
wc.lpszMenuName  = 0;
wc.lpszClassName = "MyWndClassName";
```

14.3.3 WNDCLASS Registration

Before you can create a window based on the `WNDCLASS` instance you have filled out, you must register it with Windows. To register a `WNDCLASS` instance, you use the `RegisterClass` function:

```
RegisterClass( &wc );
```

This is how we pass in a pointer to the `WNDCLASS` instance which we want to register.

14.3.4 CreateWindow

After we have registered a `WNDCLASS` instance with Windows, we can create a window. To create a window, we use the `CreateWindow` function:

```
HWND CreateWindow(
  LPCTSTR lpClassName,
  LPCTSTR lpWindowName,
  DWORD dwStyle,
  int x,
  int y,
  int nWidth,
  int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
  LPVOID lpParam
);
```

`lpClassName`: The name of the `WNDCLASS` instance to use in order to create the window (i.e., the name we specified for `wc.lpszClassName`).

`lpWindowName`: A unique string name to give the window we are creating. This is the name that will appear in the window's title/caption bar.

`dwStyle`: A combination of style flags specifying how the window should look. Typically, this is set to `WS_OVERLAPPEDWINDOW`, which is a combination of styles `WS_OVERLAPPED`, `WS_CAPTION`, `WS_SYSMENU`, `WS_THICKFRAME`, `WS_MINIMIZEBOX`, and `WS_MAXIMIZEBOX`. See the Win32 API documentation for complete details on window styles.

`x`: The x-coordinate position of the upper-left corner of the window, relative to the screen, and measured in pixels.

`y`: The y-coordinate position of the upper-left corner of the window, relative to the screen, and measured in pixels.

`nWidth`: The width of the window, measured in pixels.

`nHeight`: The height of the window, measured in pixels.

`hWndParent`: Handle to a parent window. Windows can be arranged in a hierarchical fashion. For example, controls such as buttons are child windows, and the window they lie on is the parent window. If you wish to create a window with no parent (e.g., the main application window) then specify null for this value.

`hMenu`: Handle to a menu which would be attached to the window. We will examine menus in later chapters. For now we set this to null.

`hInstance`: Handle to the application instance the window is associated with.

`lpParam`: A pointer to optional user-defined data; this is optional and can be set to null.

Note: Windows uses a different coordinate system than those which you may be familiar with. Typical mathematics uses a coordinate system where +y goes "up" and –y goes "down." However, Windows uses a coordinate system where +y goes "down" and –y goes "up." Moreover, the upper-left corner of the screen corresponds to the origin. This system is referred to as **screen space.** Figure 14.12 illustrates the differences:

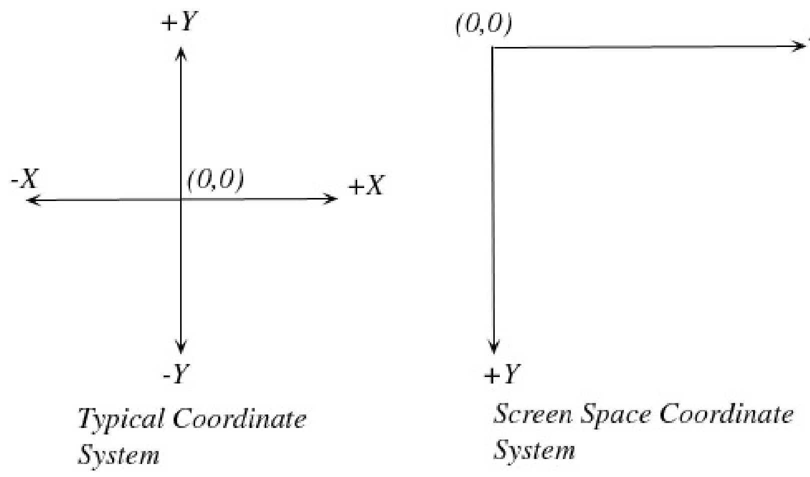


Figure 14.12: A typical coordinate system on the left, and the Windows coordinate system on the right.

Another simple but important structure in the Win32 API is the RECT structure. It is defined like so:

```
typedef struct _RECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT, *PRECT;
```

The members describe a rectangle in screen space. Figure 14.13 illustrates:

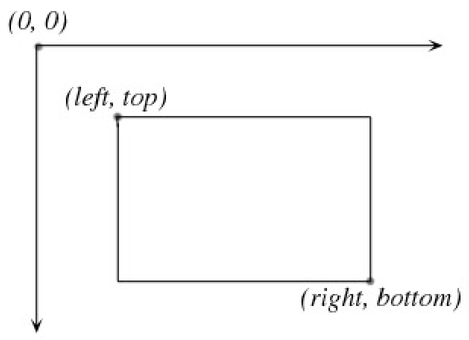


Figure 14.13: A rectangle in screen space coordinates.

> As Figure 14.13 shows, the point (left, type) defines the upper-left vertex of the rectangle and the point (right, bottom) defines the lower-right vertex of the rectangle.

This function returns a window handle (`HWND`) to the newly created window if the function is successful. If the function fails then it returns null (0). Here is a typical example call:

```
hWnd = ::CreateWindow("MyWndClassName", "MyWindow",
     WS_OVERLAPPEDWINDOW, 0, 0, 500, 500, 0, 0, hInstance, 0);

if(hWnd == 0)
{
      ::MessageBox(0, "CreateWindow - Failed", 0, 0);
      return false;
}
```

14.3.5 Showing and Updating the Window

A window is not shown by default, so after we create it we must show it and, in addition, update it for the first time:

```
ShowWindow(hWnd, showCmd);
UpdateWindow(hWnd);
```

Both of these functions require an `HWND` argument that identifies the window that should be shown and updated. Additionally, `ShowWindow` requires a second argument that specifies how the window should be shown. Some valid values are:

- `SW_SHOW` – Shows the window in the position and dimensions specified in `CreateWindow`
- `SW_MAXIMIZE` – Shows the window maximized
- `SW_MINIMIZE` – Shows the window minimized.

It is actually good form to show the window as Windows instructs; that is, using the value the `showCmd` parameter, from `WinMain`, contains.

14.3.6 The Message Loop

We said that a Windows application constantly checks the message queue for messages; this is done with the **message loop.** A typical message loop looks like this:

```
MSG msg;
ZeroMemory(&msg, sizeof(MSG));

while(GetMessage(&msg, 0, 0, 0) )
{
	TranslateMessage(&msg);
	DispatchMessage(&msg);
}
```

First, `ZeroMemory` is a Win32 function that clears out all the bits of a variable to zero, thereby "zeroing out" the object; the first parameter is a pointer to the object to zero out and the second parameter is the size, in bytes, of the object to zero out.

Moving on to the loop, for every loop cycle we call the API function `GetMessage`, which extracts the next message from the message queue and stores it in the passed-in `msg` object. The remaining three parameters of `GetMessage` are uninteresting and we can specify null (0) for them all. If the message extracted was a quit message (`WM_QUIT`) then `GetMessage` returns `false`, thereby causing the while loop to end. If the message was not a quit message then `GetMessage` returns true.

Inside the while loop, we call two more API functions: `TranslateMessage` and `DispatchMessage`. `TranslateMessage` does some key code translations into character code translations. Finally, `DispatchMessage` forwards the message off to the window procedure it is aimed for. In summary, for each cycle, the message loop gets the next message from the message queue. If the message is not a quit message then the message is sent to the appropriate window procedure to be handled.

14.4 Your Second Windows Program

The following annotated program ties in everything we have discussed in this chapter to create a basic Windows program using the steps described in the previous section:

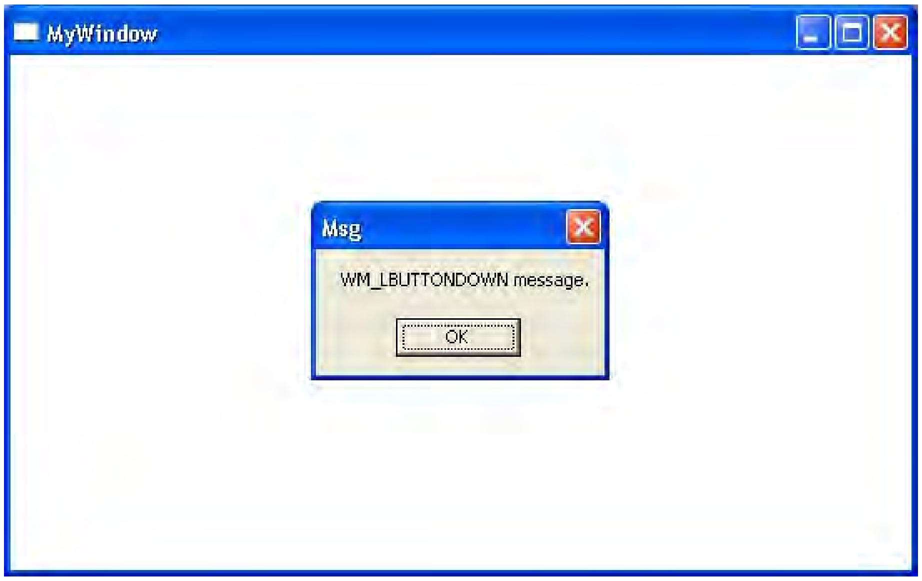
Program 14.2: Your Second Windows Program.

```
#include <windows.h>

// Store handles to the main window and application
// instance globally.
HWND      ghMainWnd = 0;
HINSTANCE ghAppInst = 0;

// Step 1: Define and implement the window procedure.
LRESULT CALLBACK
```

```
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      switch( msg )
      {
      // Handle left mouse button click message.
      case WM_LBUTTONDOWN:
            MessageBox(0, "WM_LBUTTONDOWN message.", "Msg", MB_OK);
            return 0;
      // Handle key down message.
      case WM_KEYDOWN:
            if( wParam == VK_ESCAPE )
                  DestroyWindow(ghMainWnd);
            return 0;
      // Handle destroy window message.
      case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
      // default window procedure.
      return DefWindowProc(hWnd, msg, wParam, lParam);
}

// WinMain: Entry point for a Windows application.
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Save handle to application instance.
      ghAppInst = hInstance;

      // Step 2: Fill out a WNDCLASS instance.
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = ghAppInst;
      wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
      wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
      wc.hbrBackground = (HBRUSH)::GetStockObject(WHITE_BRUSH);
      wc.lpszMenuName  = 0;
      wc.lpszClassName = "MyWndClassName";

      // Step 3: Register the WNDCLASS instance with Windows.
      RegisterClass( &wc );

      // Step 4: Create the window, and save handle in globla
      // window handle variable ghMainWnd.
      ghMainWnd = ::CreateWindow("MyWndClassName", "MyWindow",
            WS_OVERLAPPEDWINDOW, 0, 0, 500, 500, 0, 0, ghAppInst, 0);

      if(ghMainWnd == 0)
      {
            ::MessageBox(0, "CreateWindow - Failed", 0, 0);
            return false;
      }
```

```
       // Step 5: Show and update the window.
       ShowWindow(ghMainWnd, showCmd);
       UpdateWindow(ghMainWnd);

       // Step 6: Enter the message loop and don't quit until
       // a WM_QUIT message is received.
       MSG msg;
       ZeroMemory(&msg, sizeof(MSG));

       while( GetMessage(&msg, 0, 0, 0) )
       {
             TranslateMessage(&msg);
             DispatchMessage(&msg);
       }

       // Return exit code back to operating system.
       return (int)msg.wParam;
}
```

Program 14.2 outputs a window with a caption "My Window," a minimize button, a maximize button, and a close button. If you click the client area (white rectangle) a message box is displayed that indicates you generated a `WM_LBUTTONDOWN` message. Finally, you can close the window by pressing the close 'X' button or by pressing the escape key. Figure 14.14 shows the output after the left mouse button was pressed:



WM_LBUTTONDOWN message.

Figure 14.14: Program 14.2 output after the user left mouse clicked the client area.

14.5 Summary

1. The primary characteristic of a Win32 application is the GUI (Graphical User Interface). "GUI" refers to the menus, buttons, scroll bars, and other graphical objects users interact with.

2. The Win32 library is a large low-level set of functions, structures, and types that allow you to create and customize windows.

3. In order to write Win32 programs you need to include the Windows header file (#include <windows.h>); this header file gives the prototypes and definitions of various functions and types you will need to program the Win32 API.

4. Win32 programs no longer start with the `main` function. Instead, they start with the `WinMain` function. `WinMain` returns an integer, which specifies the code that is returned to the Windows operating system.

5. An `HWND` is an ID to a particular window, just like an `HINSTANCE` is an ID to a particular application. The 'H' in both of these stands for "handle": `HWND` means "windows handle" and `HINSTANCE` means "application instance handle." We need these IDs or handles, so that we can refer to individual Windows objects, such as applications and windows, which are internally maintained by the Win32 API.

6. Windows programs are event driven; that is, they sit and wait for an event to occur, and then execute some code in response to that event. An event can be a key press, a mouse click, a button press, a menu selection, scrolling the scroll bars, etc. When an event occurs, Windows adds a message to the application's message queue for which the event was aimed for. The application's message loop then retrieves and processes the messages from the message queue. When a message is processed it is dispatched to the window procedure of the window the event was targeted for. The window procedure then handles the message by executing some code in response to the message.

7. Creating and displaying a window requires several steps:

    a. You must define the window procedure of the main application window
    b. You must fill out a `WNDCLASS` instance describing some of the properties of your window
    c. You must register the `WNDCLASS` instance with Windows using the `RegisterClass` function
    d. You must create the window with the `CreateWindow` function
    e. You must show and update the window with `ShowWindow` and `UpdateWindow`, respectively
    f. You must enter the message loop so that your application can retrieve and process messages.

14.6 Exercises

These exercises should be done as modification to Program 4.2.

14.6.1 Exit Message

When the user presses the escape key, instead of exiting immediately, display a "YES/NO" style message box asking the user if he really wants to quit. If the user selects "Yes" then exit, else if the user selects "No" then do not exit.

14.6.2 Horizontal and Vertical Scroll Bars

Add horizontal and vertical scroll bars to the main window. These scroll bars only need to be displayed—they do *not* need to be functional. (Hint: See the `CreateWindow` style flags in the Win32 documentation.)

14.6.3 Multiple Windows

Instead of just displaying one window, make the application create and show two more windows. The windows' positions and dimensions should be set so that they occupy different regions of the screen, thereby making all three visible at once. Give all three windows their own window procedure. When the user left mouse clicks in one of the windows, display a message box with the message "You clicked Window #*x*," where *x* should be replaced with 1, 2, or 3 depending on which window was selected. (Hint: Create and fill out three `WNDCLASS` instances that correspond to each window and which contain the corresponding window procedure. Also create three `HWND`s (one for each window) and call `CreateWindow` three times to create each window. You will also need to show and update each window.)

14.6.4 Change the Cursor

Change the mouse cursor to something other than the arrow cursor.

14.6.5 Blue Background

Change the background of the window (or the three windows if you did Exercise 14.6.3) from white to blue (or red, green, and blue if you did Exercise 14.6.3). You will need to create a blue brush for this (or red, green, and blue if you did Exercise 14.6.3). You can create a custom brush with the following function:

```
HBRUSH CreateBrushIndirect(
  CONST LOGBRUSH *lplb   // brush information
);
```

Observe that this function returns an `HBRUSH`. This function has a parameter that specifies the details of the brush to create. Specifically, `LOGBRUSH` looks like this:

```
typedef struct tagLOGBRUSH {
  UINT     lbStyle;
  COLORREF lbColor;
  LONG     lbHatch;
} LOGBRUSH, *PLOGBRUSH;
```

The style allows you to specify a brush style. We want a solid brush, so we select `BS_SOLID`.

A `COLORREF` is a 32-bit value, where 1 byte is used to describe the shade of red, 1 byte is used to describe the shade of green, and 1 byte is used to describe the shade of blue—the so-called RGB color—and one byte is not used. We can use the following macro to create a `COLORREF`:

```
COLORREF RGB(
  BYTE byRed,    // red component of color in the range [0, 255].
  BYTE byGreen,  // green component of color in the range [0, 255].
  BYTE byBlue    // blue component of color in the range [0, 255].
);
```

`BYTE` is `typedefed` as an `unsigned char`. For the color ranges, 0 would mean zero color intensity/shade (i.e., black) and 255 would mean full intensity—bright red, green, or blue. Thus, we have 256 shades of each color component. The mixing of various shades of red, green, and blue allows us to describe millions of different colors.

The hatch member of `LOGBRUSH` is another value that allows you to describe a hatch pattern. Since we are using a solid brush, we can ignore this value—see the Win32 documentation if you are curious about why.

With all that said, we can create our blue brush like so:

```
LOGBRUSH lb;
lb.lbStyle = BS_SOLID;
lb.lbColor = RGB(0, 0, 255);
HBRUSH blueBrush = CreateBrushIndirect(&lb);
```

After you have made a window with a blue background color, try experimenting with different colors (i.e., changing the values in the RGB macro).

14.6.6 Custom Icon

Instead of using the default Windows icon, let us make our own. To do this, we need to add an icon resource to our project. To do this, go to the menu, select *View->Resource View*, as Figure 14.15 shows.

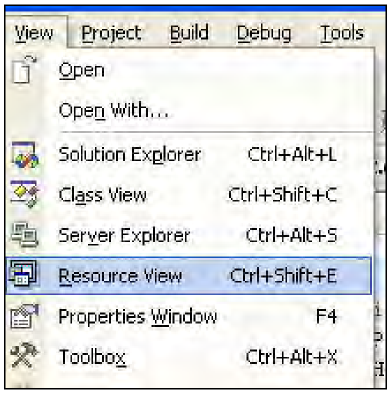


Figure 14.15: Opening the Resource View panel.

On the right hand side of Visual C++ .NET, you should see a "Resource View" panel, with the name of your project. Right click the name, select Add, then Add Resource, as Figure 14.16 shows.

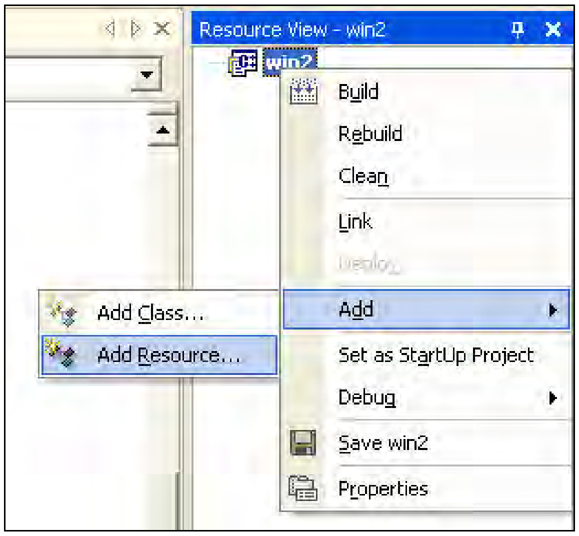


Figure 14.16: Adding a resource to the project.

A dialog box appears (Figure 14.17). Select the Icon selection and press the "New" button.

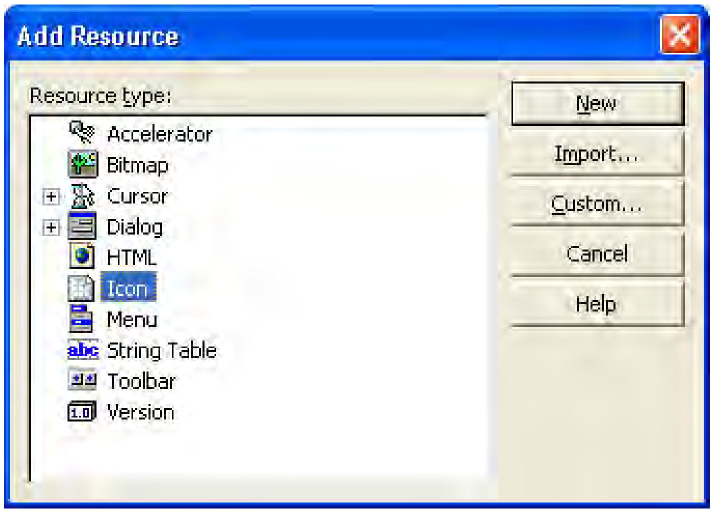


Figure 14.17: Adding an icon resource.

An icon resource will be added to your project, and the Visual C++ .NET icon editor will launch, thereby allowing you to paint your icon. Figure 14.18 shows a quick Game Institute icon that was painted in the editor.

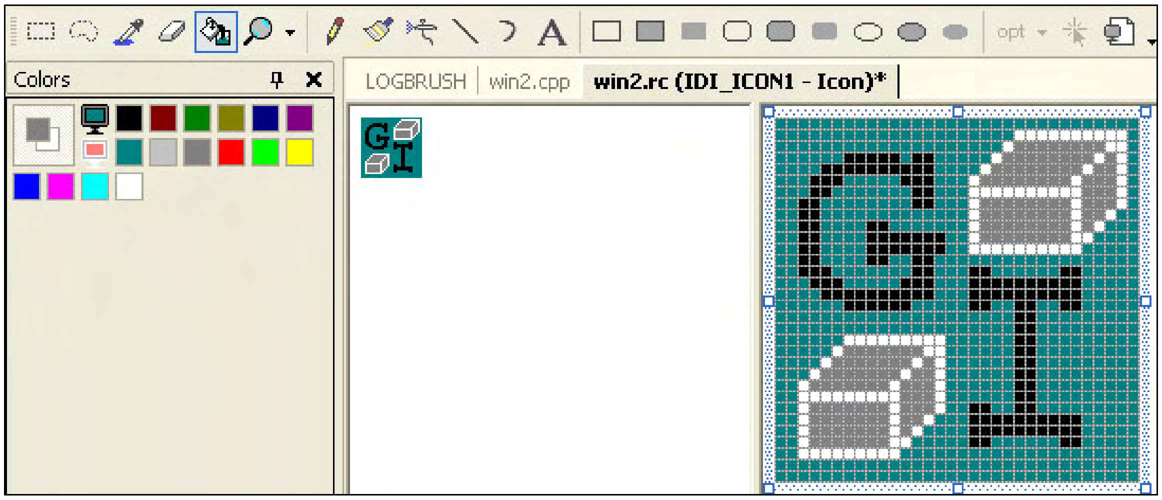


Figure 14.18: Painting an icon in the Visual C++ Resource Editor.

As you can see from the "Resource View" panel (Figure 14.19), the icon that we added was given the default name IDI_ICON1. We can change this name, but it is fine for now.

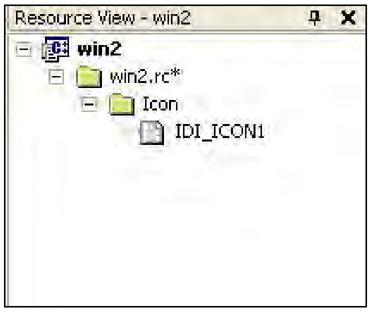


Figure 14.19: The Resource View panel shows the project resources.

To get a `HICON` handle to our icon, we use the `LoadIcon` function like so:

```
::LoadIcon(ghAppInst, MAKEINTRESOURCE(IDI_ICON1));
```

Because we are not using a system icon, we must specify the application instance for the first parameter. For the second parameter, we must pass our icon name through a macro, which will convert a numeric ID to the icon name. Also note that when we add a new resource, a new header file called "resource.h" is automatically created, which contains our resource names and symbols. In order for the application to recognize the symbol `IDI_ICON1`, you will need to #include "resource.h". Figure 14.20 shows our window, now with a custom icon:

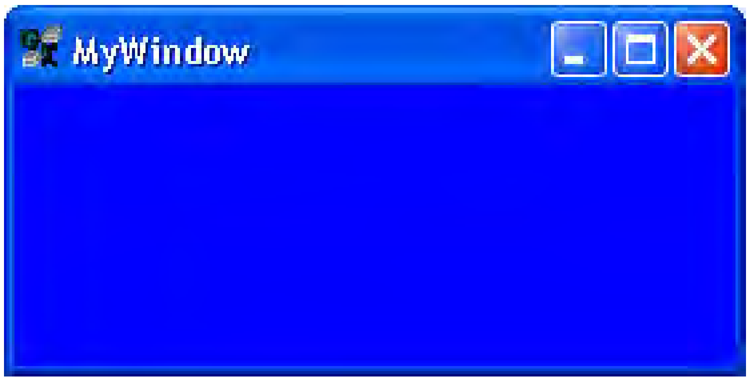


Figure 14.20: Our window with a custom icon.

Chapter 15

Introduction to GDI and Menus

Introduction

The primary theme of this chapter is **GDI (Graphics Device Interface).** GDI is the component of the Win32 API that is concerned with drawing graphics; it provides all the necessary structures and functions for drawing graphics onto your windows.

This is definitely one of the more "fun" chapters we have encountered so far this course, as we finally have enough C++ background and Windows background to make an interesting application. Moreover, the basic graphics and menu features we learn about here will provide us with a solid foundation when we begin to make some Windows games. By the end of this chapter, you will have learned how to create a basic paint program that allows you to draw various shapes with different colors and styles. Figure 15.1 shows our ultimate goal for this chapter.

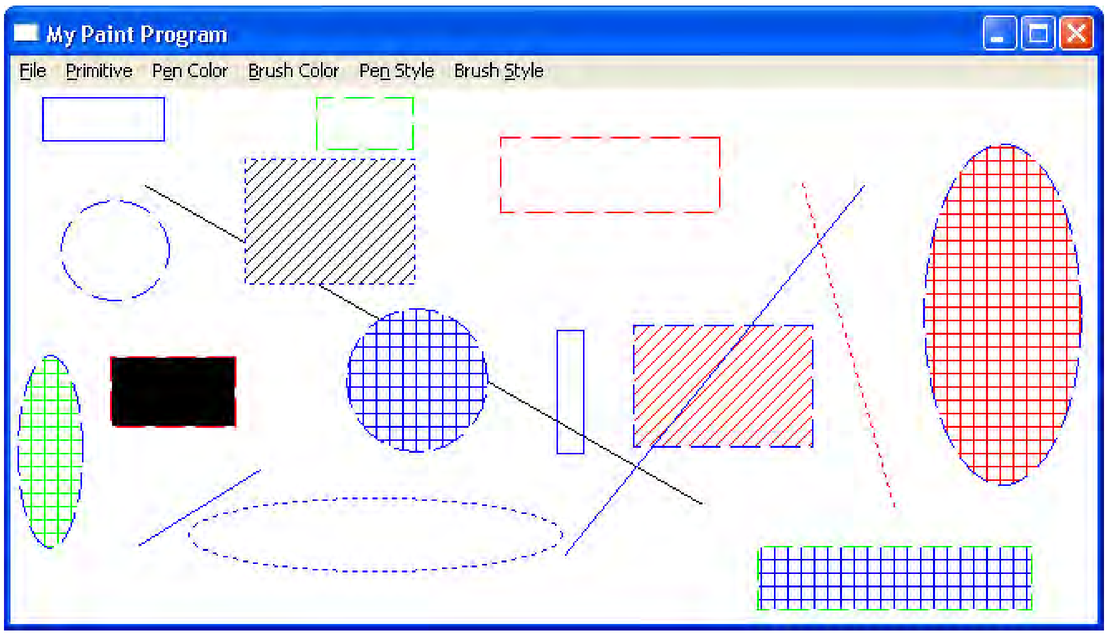


Figure 15.1: A screenshot of the paint program that we will develop in this chapter.

Chapter Objectives

- Learn how to output text onto a window, and how to draw several GDI shape primitives like lines, rectangles and ellipses.
- Understand how different pens and brushes can be used to change the way in which the GDI shapes are colored and drawn.
- Learn how to load bitmap (.bmp) images from file into our Windows programs, and how to draw them on the client area of our windows.
- Become familiar with the Visual C++ menu resource editor, and learn how to create menus with it.

15.1 Text Output

15.1.1 The `WM_PAINT` Message

Before we can begin a discussion of outputting text to a window's client area (which is a form of drawing), we need to talk about the `WM_PAINT` message. First, note that in Windows, you can have several windows open at once. For example, you may have a web browser open, a word processor open, and perhaps a couple different windows displaying the contents of various folders. You have certainly observed a time where a window *A* was partially or fully obscured by another window *B* (or even obscured by several windows, but let us keep it simple). **The key idea is that when *B* obscures a part of *A*'s client area (call the obscured region *R*), the data drawn on *R* is not saved by Windows. Consequently, when we move *B* off of *A*'s client area, so that a part of *R* becomes visible, the data previously drawn to *R* no longer exists—it was not saved.** However, you might argue that this is not what happens. That is, you have obscured windows, and then made them visible again and nothing was lost. What is actually happening is that the windows "know" when previously obscured parts become visible, and they then redraw themselves to restore the previous data. Thus, it looks like nothing was really lost. Note though, that this functionality is not automatic—the applications were programmed to repaint themselves at the appropriate time.

> **Note:** Resizing a window so that it becomes smaller can also hide a region you once painted to. When you resize back to a larger size, the data drawn to the previously hidden region will not be "remembered" by Windows—you will have to redraw it yourself. Window also sends a `WM_PAINT` message when a window is resized.

The window knows to redraw itself when part of it becomes visible because Windows will send the window a `WM_PAINT` message informing the window that it needs to repaint its client area (or a region of the client area). This means that we need to structure our programs so that the drawing code occurs in response to a `WM_PAINT` message. By placing all the drawing code in the `WM_PAINT` message handler, we can ensure that the window will repaint itself whenever necessary; that is, whenever a `WM_PAINT` occurs.

15.1.2 The Device Context

A **device context** is a software abstraction of a display device, such as the video card or a printer. We do all drawing operations through the device context (abbreviated as DC). The first question is how to obtain a DC. There are several ways. We will look at one method now, and then look at another method a few chapters down the road.

As said, we will try to structure our programs so that all the drawing will be done in the `WM_PAINT` handler. Because the `WM_PAINT` handler is a common place for drawing, the Win32 API provides a special function, which can be used there that returns a handle to a device context. More specifically, in a `WM_PAINT` handler, we can get a handle to device context (`HDC`) with the `BeginPaint` function:

```
HDC hdc = 0;
PAINTSTRUCT ps;

switch( msg )
{
case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);

      // Drawing code goes here.

      EndPaint(hWnd, &ps);
      return 0;
```

Note that `BeginPaint` takes an argument to a window handle (`hWnd`). This is necessary because a DC must be associated with some object onto which it is to render. In this context, we want to render onto a window, so we must specify the handle of the window we want to associate the DC with.

Also observe that every `BeginPaint` function call must have a corresponding `EndPaint` function call, and that the actual drawing code goes in between the two calls.

The `HDC` is a handle to a device context. The `PAINTSTRUCT` structure is defined like so:

```
typedef struct tagPAINTSTRUCT {
  HDC  hdc;
  BOOL fErase;
  RECT rcPaint;
  BOOL fRestore;
  BOOL fIncUpdate;
  BYTE rgbReserved[32];
} PAINTSTRUCT, *PPAINTSTRUCT;
```

You should only touch the first three members; the Win32 API uses the others internally. Although `BeginPaint` returns a handle to a device context, you can also get it from the `hdc` member of the `PAINTSTRUCT`. The `fErase` member is true if the background should be erased, and false otherwise. The `rcPaint` rectangle defines the smallest rectangle that needs to be updated. In this way, you do not have to repaint the entire client area, only the necessary region.

15.1.3 TextOut

In the previous section, we learned how to obtain a DC. We can now learn how to draw text. Drawing text is quite simple, and is handled with one API function. In particular, it is handled with the `TextOut` API function. Here is a sample call:

```
string s = “Hello, World!”;
TextOut(hdc, 10, 10, s.c_str(), s.size());
```

The first parameter of `TextOut` is a handle to the device context, which is associated with the window’s client area onto which we will be drawing the text. The second and third parameters are used to specify the (x, y) position, relative to the window's client rectangle in screen space (upper-left corner of client area is the origin and +y "goes down") and where the text should be displayed on the client area rectangle. The fourth parameter is the c-string to display, and the fifth parameter is the size of the c-string to display.

15.1.3 Example Program

In this sample program, we will develop an application where we output the text "Hello, World." wherever the user presses the left mouse button on the window's client area. Figure 15.2 demonstrates the product of numerous mouse clicks:

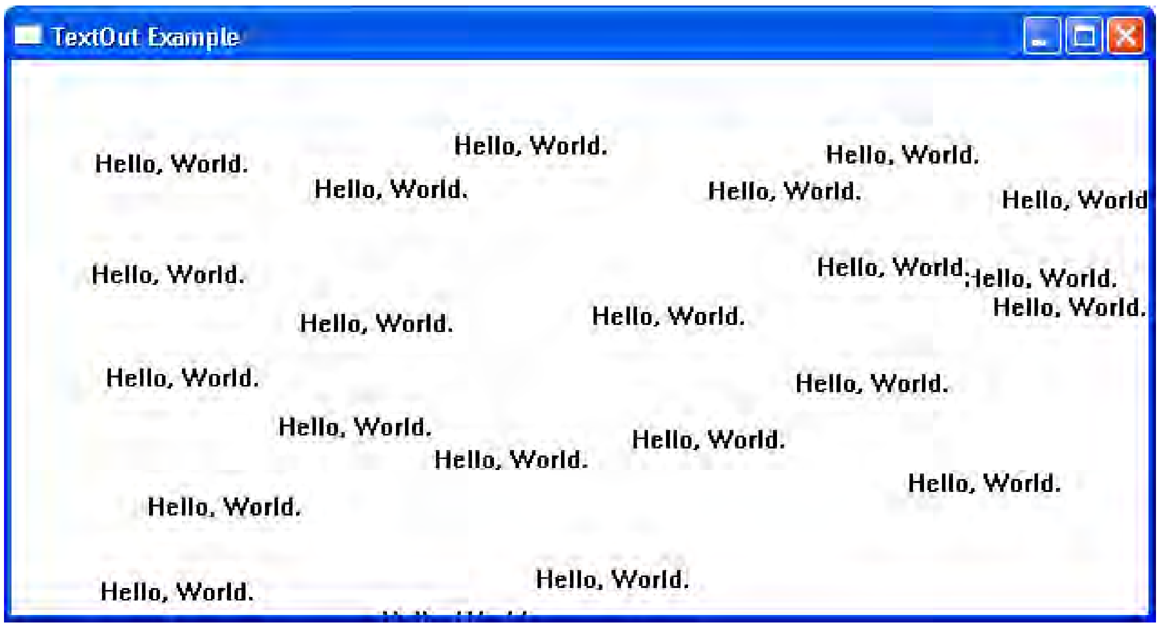


Figure 15.2: We output the text "Hello, World." wherever the user presses the left mouse button.

Recall that Windows does not save our drawn data if a part of the client area gets obscured. Therefore, we need to save all the data ourselves so that we can redraw it all when a `WM_PAINT` message occurs. To facilitate this, we maintain a global vector of `TextObjs`, which will store the position and string of every string we create. (In this program, we will create a string whenever we press the left mouse button on the window's client area.) Thus we have the following:

```
struct TextObj
{
      string s; // The string.
      POINT  p; // String position, relative to the client area rect.
};
vector<TextObj> gTextObjs;
```

Now we will create a TextObj instance and add it to gTextObjs whenever the user presses the left mouse button on the window's client area. To do this we must handle the WM_LBUTTONDOWN message, which is sent when the user presses the left mouse button:

```
TextObj to;

switch( msg )
{
// Handle left mouse button click message.
case WM_LBUTTONDOWN:
      to.s = "Hello, World.";

      // Point that was clicked is stored in the lParam.
      to.p.x = LOWORD(lParam);
      to.p.y = HIWORD(lParam);

      // Add to our global list of text objects.
      gTextObjs.push_back( to );

      InvalidateRect(hWnd, 0, false);

      return 0;
```

There are a few new ideas here. First, we note that the point the user clicked is provided in the lParam of the WM_LBUTTONDOWN message. The lParam is a 32-bit integer value, so what Windows does to save space is to store the x-coordinate in the lower 16-bits and the y-coordinate in the higher 16-bits. For convenience, the Win32 API also provides the macro LOWORD, which will extract the lower 16-bit value of a 32-bit integer and return the result, and the macro HIWORD, which will extract the higher 16-bit value of a 32-bit integer, and return the result. Thus, for the WM_LBUTTONDOWN message, we can extract the point that was clicked, relative to the client area rectangle, with the code:

```
to.p.x = LOWORD(lParam);
to.p.y = HIWORD(lParam);
```

In this program, all of our strings will be "Hello, World." so we assign a literal to our TextObj's string member:

```
to.s = "Hello, World.";
```

After creating and filling out the data members of the TextObj instance, we add it to our global container so that it is saved:

```
// Add to our global list of text objects.
gTextObjs.push_back( to );
```

Now that we have created a new TextObj, we want to display it immediately. However, all of our drawing code is in the WM_PAINT handler. Essentially, we want to force a repaint now (i.e., send a WM_PAINT message now). We can do that by calling the InvalidateRect function, which will post a WM_PAINT message.

InvalidateRect takes three parameters: the first is a handle to the window to which we want to send a WM_PAINT message; the second is a pointer to a RECT structure, which defines the region that needs to be updated—we can pass null for this value to update the entire client area; the third parameter is a Boolean value that specifies whether the region that parameter two defines should be erased with the background color prior to being drawn upon.

We create a new TextObj whenever the user presses the left mouse button. Each TextObj stores the string and its position. This provides us with all the information we need to draw the string the TextObj contains with TextOut. Thus, to draw all the TextObjs we have created (which are added to gTextObjs), all we need to do is iterate through each element in gTextObjs and call TextOut:

```
case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);

      for(int i = 0; i < gTextObjs.size(); ++i)
            TextOut(
                  hdc,
                  gTextObjs[i].p.x,
                  gTextObjs[i].p.y,
                  gTextObjs[i].s.c_str(),
                  gTextObjs[i].s.size());

      EndPaint(hWnd, &ps);
      return 0;
```

Program 15.1 shows the code in its entirety, with the concepts new to this chapter in bold font (the rest of the code is the same window creation code we discussed in the previous chapter).

Program 15.1: Text drawing program.

```
#include <windows.h>
#include <string>
#include <vector>
using namespace std;

//=========================================================
// Globals.

HWND      ghMainWnd = 0;
HINSTANCE ghAppInst = 0;

struct TextObj
{
      string s; // The string object.
      POINT  p; // The position of the string, relative to the
                // upper-left corner of the client rectangle of
                // the window.
};

vector<TextObj> gTextObjs;
```

```
// Step 1: Define and implement the window procedure.
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      // Objects for painting.
      HDC hdc = 0;
      PAINTSTRUCT ps;

      TextObj to;

      switch( msg )
      {
      // Handle left mouse button click message.
      case WM_LBUTTONDOWN:
            to.s = "Hello, World.";

            // Point that was clicked is stored in the lParam.
            to.p.x = LOWORD(lParam);
            to.p.y = HIWORD(lParam);

            // Add to our global list of text objects.
            gTextObjs.push_back( to );

            InvalidateRect(hWnd, 0, false);

            return 0;
      // Handle paint message.
      case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);

            for(int i = 0; i < gTextObjs.size(); ++i)
                  TextOut(
                        hdc,
                        gTextObjs[i].p.x,
                        gTextObjs[i].p.y,
                        gTextObjs[i].s.c_str(),
                        gTextObjs[i].s.size());

            EndPaint(hWnd, &ps);
            return 0;

      // Handle key down message.
      case WM_KEYDOWN:
            if( wParam == VK_ESCAPE )
                  DestroyWindow(ghMainWnd);

            return 0;
      // Handle destroy window message.
      case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
      // default window procedure.
      return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

```
// WinMain: Entry point for a Windows application.
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Save handle to application instance.
      ghAppInst = hInstance;

      // Step 2: Fill out a WNDCLASS instance.
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = ghAppInst;
      wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
      wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
      wc.hbrBackground = (HBRUSH)::GetStockObject(WHITE_BRUSH);
      wc.lpszMenuName  = 0;
      wc.lpszClassName = "MyWndClassName";

      // Step 3: Register the WNDCLASS instance with Windows.
      RegisterClass( &wc );

      // Step 4: Create the window, and save handle in globla
      // window handle variable ghMainWnd.
      ghMainWnd = ::CreateWindow("MyWndClassName", "TextOut Example",
            WS_OVERLAPPEDWINDOW, 200, 200, 640, 480, 0, 0,
            ghAppInst, 0);

      if(ghMainWnd == 0)
      {
            ::MessageBox(0, "CreateWindow - Failed", 0, 0);
            return false;
      }

      // Step 5: Show and update the window.
      ShowWindow(ghMainWnd, showCmd);
      UpdateWindow(ghMainWnd);

      // Step 6: Enter the message loop and don't quit until
      // a WM_QUIT message is received.
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));

      while( GetMessage(&msg, 0, 0, 0) )
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }

      // Return exit code back to operating system.
      return (int)msg.wParam;
}
```

15.2 Shape Primitives

15.2.1 Drawing Lines

Line drawing is done with two functions. The first function moves the "virtual pen" to the starting point of the line. The second function draws a line from the previously specified starting point, to a newly specified second point:

```
// Following two functions draws a line from (startX, startY)
// to (endX, endY).
MoveToEx(hdc, startX, startY, 0);
LineTo(hdc, endX, endY);
```

Both functions take a handle to the device context, as all drawing must be done through the device context. The second and third parameters for both functions are the x- and y-coordinates of a point, which is relative to the client area rectangle. For `MoveToEx`, that point is the starting point of the line. For `LineTo`, that point is the ending point of the line. The fourth parameter of `MoveToEx` returns a `POINT` object of the last "start" point that was specified, via a pointer parameter. If this value is not needed, you can specify null.

The program we will write to illustrate line drawing allows the user to define the line's "start" point by pressing the left mouse button. The user holds the left mouse button down and moves the mouse to a new point. When the user has found the point where he/she wants the line's "end" point to be, the user raises the left mouse button up. Figure 15.3 shows the program after some lines were drawn.

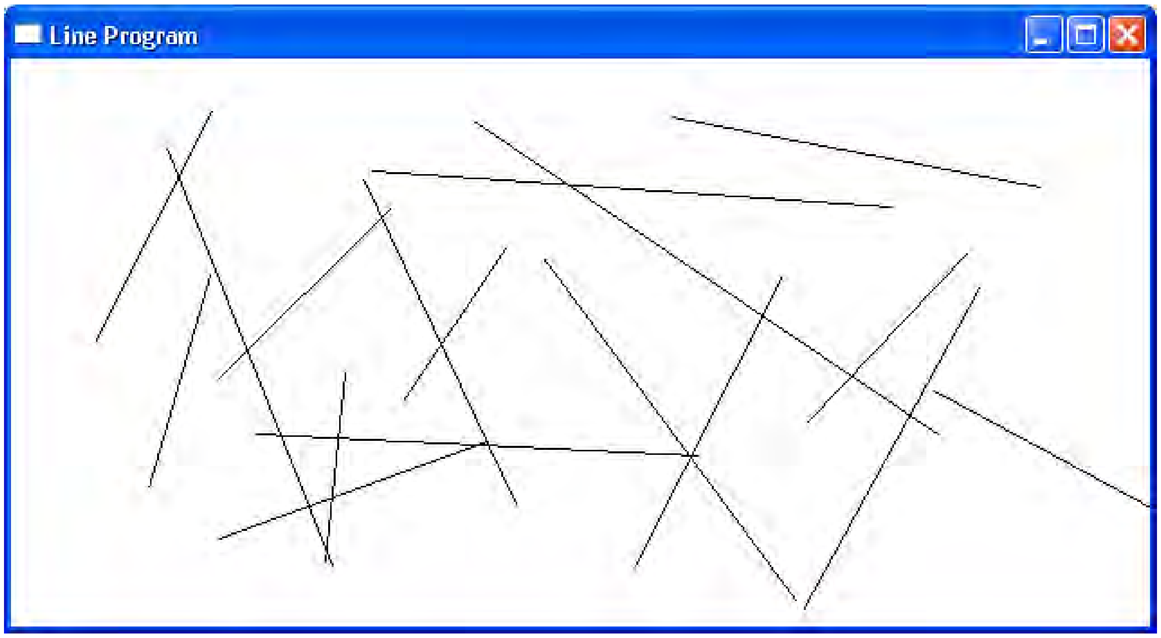


Figure 15.3: The line drawing program. Users can draw lines by holding the left mouse button down

As the user moves the mouse around looking for the "end" point, he/she will expect to see the new line being drawn in real-time. That is, a line from the "start" point to the current mouse position should constantly be drawn and updated interactively. In this way, the user can see exactly how the line will look before raising the left mouse button to make the line permanent. This functionality requires some special code. Let us get started.

First, we have the following global variables (and also a structure definition):

```
struct Line
{
	POINT p0;
	POINT p1;
};

vector<Line> gLines;
Line gLine;

bool gMouseDown = false;
```

A `Line` is simply defined by two points, `p0`, and `p1`, where `p0` is the "start" point and `p1` is the "end" point.

We recall that Windows does not save our drawn data if a part of the client area gets obscured. Therefore, we need to save all the data ourselves so that we can redraw it all when a `WM_PAINT` message occurs. To facilitate this, we maintain a global vector of `Lines`, called `gLines`, which will store the lines we create. The global variable `gLine` is our temporary line; that is, it is the line we will draw as the user moves the mouse around when deciding where the "end" point of the line should be. We do not actually add a line to `gLines` until the user has lifted the left mouse button. Finally, `gMouseDown` is a Boolean variable that denotes whether or not the left mouse button is currently down or not.

The first message we need to handle is the `WM_LBUTTONDOWN` message, which is where the line's "starting" point is defined.

```
case WM_LBUTTONDOWN:

	// Capture the mouse (we still get mouse input
	// even after the mouse cursor moves off the client area.
	SetCapture(hWnd);
	gMouseDown = true;

	// Point that was clicked is stored in the lParam.
	gLine.p0.x = LOWORD(lParam);
	gLine.p0.y = HIWORD(lParam);

	return 0;
```

Note that we set the "start" point in our temporary line. We do not actually add the line to our global line container `gLines` until the user lifts the left mouse button.

A new API function in this message handler is the SetCapture function. This function “captures” the mouse for the specified window. Capturing means that the window will continue to receive mouse messages even if the mouse moves off the window’s client area. As long as the user has the left mouse button down, we would like to have the mouse captured—we free the mouse when the user lifts the left mouse button. Finally, if a WM_LBUTTONDOWN message occurs, we know the mouse is now down, so we set our flag gMouseDown to true.

The next message we handle is the WM_MOUSEMOVE message. This message is sent whenever the mouse moves.

```
case WM_MOUSEMOVE:

      if( gMouseDown )
      {
            // Current mouse position is stored in the lParam.
            gLine.p1.x = LOWORD(lParam);
            gLine.p1.y = HIWORD(lParam);

            InvalidateRect(hWnd, 0, true);
      }

      return 0;
```

Notice that we only care about this message if the left mouse button is down (if(gMouseDown)). If it is not, we do not care about the WM_MOUSEMOVE message and do not execute any code.

So, assuming the left mouse button is down, as the mouse moves we obtain the current mouse position (given in the lParam for the WM_MOUSEMOVE message) and set it as the “end” point for the temporary line. We then invalidate the window’s client rectangle so that it is forced to repaint itself. In this way, the new temporary line will be redrawn interactively as the mouse moves. Also note that here we invalidate the rectangle with true specified for the third parameter—this will cause the background to be erased, which is necessary since we need to erase any previously drawn temporary lines. That is, every time the mouse moves we will draw a temporary line, but we do not want to accumulate these lines; we just want to draw the latest temporary line. Therefore, we must erase any old lines.

The third message we handle is the WM_LBUTTONUP message. This message is generated when the left mouse button is lifted up.

```
case WM_LBUTTONUP:

      // Release the captured mouse when the left mouse button
      // is lifted.
      ReleaseCapture();
      gMouseDown = false;

      // Current mouse position is stored in the lParam.
      gLine.p1.x = LOWORD(lParam);
      gLine.p1.y = HIWORD(lParam);

      gLines.push_back( gLine );
      InvalidateRect(hWnd, 0, true);

      return 0;
```

First, as we said before, when the left mouse button is raised, we can release our capture of the mouse; this is done with the `ReleaseCapture` function. Also, because the left mouse button has now been raised up, we set `gMouseDown` to `false`. Next, we update the "end" point of the temporary line to reflect the position of the point where the left mouse button was lifted up. Then, by raising the left mouse button up, the user has chosen where to make a permanent line. Thus, we add a copy of `gLine` to our line container:

```
gLines.push_back( gLine );
```

Finally, we invalidate the window's client rectangle so that the newly added permanent line is drawn.

Program 15.2 shows the code for the line drawing program in its entirety, with the new concepts bolded.

Program 15.2: Line drawing program.

```
#include <windows.h>
#include <string>
#include <vector>
using namespace std;

//===========================================================
// Globals.

HWND      ghMainWnd = 0;
HINSTANCE ghAppInst = 0;

struct Line
{
      POINT p0;
      POINT p1;
};

vector<Line> gLines;
Line gLine;

bool gMouseDown = false;

// Step 1: Define and implement the window procedure.
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      // Objects for painting.
      HDC hdc = 0;
      PAINTSTRUCT ps;

      switch( msg )
      {
      // Handle left mouse button click message.
      case WM_LBUTTONDOWN:
            // Capture the mouse (we still get mouse input
            // even after the mouse cursor moves off the client area.
            SetCapture(hWnd);
            gMouseDown = true;

            // Point that was clicked is stored in the lParam.
            gLine.p0.x = LOWORD(lParam);
            gLine.p0.y = HIWORD(lParam);

            return 0;
      // Message sent whenever the mouse moves.
      case WM_MOUSEMOVE:

            if( gMouseDown )
            {
                  // Current mouse position is stored in the lParam.
                  gLine.p1.x = LOWORD(lParam);
                  gLine.p1.y = HIWORD(lParam);

                  InvalidateRect(hWnd, 0, true);
            }

            return 0;
      case WM_LBUTTONUP:

            // Release the captured mouse when the left mouse
            // button is lifted.
            ReleaseCapture();
            gMouseDown = false;

            // Current mouse position is stored in the lParam.
            gLine.p1.x = LOWORD(lParam);
            gLine.p1.y = HIWORD(lParam);

            gLines.push_back( gLine );

            InvalidateRect(hWnd, 0, true);

            return 0;
      // Handle paint message.
      case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);

            if( gMouseDown )
            {
                  MoveToEx(hdc, gLine.p0.x, gLine.p0.y, 0);
                  LineTo(hdc, gLine.p1.x, gLine.p1.y);
            }

            for(int i = 0; i < gLines.size(); ++i)
            {
                  MoveToEx(hdc, gLines[i].p0.x, gLines[i].p0.y, 0);
                  LineTo(hdc, gLines[i].p1.x, gLines[i].p1.y);
            }

            EndPaint(hWnd, &ps);
            return 0;
      // Handle key down message.
      case WM_KEYDOWN:
            if( wParam == VK_ESCAPE )
                  DestroyWindow(ghMainWnd);

            return 0;
      // Handle destroy window message.
      case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
      // default window procedure.
      return DefWindowProc(hWnd, msg, wParam, lParam);
}

// WinMain: Entry point for a Windows application.
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Save handle to application instance.
      ghAppInst = hInstance;

      // Step 2: Fill out a WNDCLASS instance.
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = ghAppInst;
      wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
      wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
      wc.hbrBackground = (HBRUSH)::GetStockObject(WHITE_BRUSH);
      wc.lpszMenuName  = 0;
      wc.lpszClassName = "MyWndClassName";

      // Step 3: Register the WNDCLASS instance with Windows.
      RegisterClass( &wc );

      // Step 4: Create the window, and save handle in globla
      // window handle variable ghMainWnd.
      ghMainWnd = ::CreateWindow("MyWndClassName", "MyWindow",
            WS_OVERLAPPEDWINDOW, 200, 200, 640, 480, 0, 0,
            ghAppInst, 0);

      if(ghMainWnd == 0)
      {
            ::MessageBox(0, "CreateWindow - Failed", 0, 0);
            return false;
      }

      // Step 5: Show and update the window.
      ShowWindow(ghMainWnd, showCmd);
      UpdateWindow(ghMainWnd);

      // Step 6: Enter the message loop and don't quit until
```

```
        // a WM_QUIT message is received.
        MSG msg;
        ZeroMemory(&msg, sizeof(MSG));

        while( GetMessage(&msg, 0, 0, 0) )
        {
              TranslateMessage(&msg);
              DispatchMessage(&msg);
        }

        // Return exit code back to operating system.
        return (int)msg.wParam;
}
```

15.2.2 Drawing Rectangles

To draw a rectangle we can use the `Rectangle` API function. The `Rectangle` function takes five parameters; the first is a handle to a DC, and the last four define the dimensions of the rectangle to draw. That is, the last four parameters take the left, top, right, and bottom coordinates of the rectangle. Here is an example call.

```
Rectangle(hdc, gRect.left, gRect.top, gRect.right, gRect.bottom);
```

`gRect` is of type `RECT`, which was discussed in a Note in the previous chapter.

The program we will write in order to illustrate rectangle drawing allows the user to define the (left, top) and (right, bottom) points on the rectangle by pressing and lifting the left mouse button. Figure 15.4 shows the program after some rectangles were drawn.

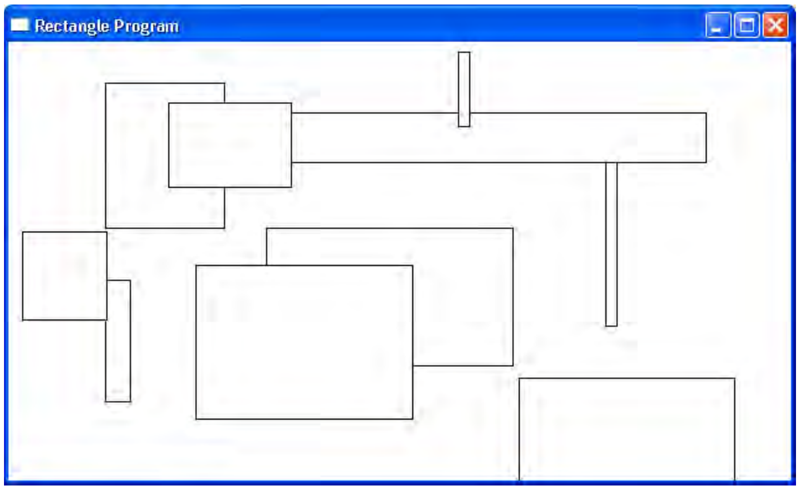


Figure 15.4: A screenshot of the rectangle sample.

In a way similar to the line drawing program, as the user moves the mouse around looking for the (right, bottom) point, he/she will expect to see the new rectangle being drawn in real-time. That is, a rectangle from the (left, top) point to the current mouse position should constantly be drawn and updated interactively. In this way, the user can see exactly how the rectangle will look before raising the left mouse button to make the rectangle permanent. The logic to do this is exactly the same as the line program. For example, we have the following global variables:

```
HWND      ghMainWnd = 0;
HINSTANCE ghAppInst = 0;

vector<RECT> gRects;
RECT gRect;

bool gMouseDown = false;
```

These are basically the same as the line program, except we have replaced the `Line` structure with `RECT`. But logically, everything is going to be the same as the line drawing program. We are just drawing rectangles instead of lines (which are described with two points, just as lines are). Therefore, rather than duplicate an analogous discussion, we will simply show the rectangle program, and bold the parts that have changed. There should be no trouble understanding the logic if the line drawing program was understood.

Program 15.3: Rectangle drawing program.

```
#include <windows.h>
#include <string>
#include <vector>
using namespace std;

//=========================================================
// Globals.
HWND      ghMainWnd = 0;
HINSTANCE ghAppInst = 0;

vector<RECT> gRects;
RECT gRect;

bool gMouseDown = false;

// Step 1: Define and implement the window procedure.
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      // Objects for painting.
      HDC hdc = 0;
      PAINTSTRUCT ps;

      switch( msg )
      {
      // Handle left mouse button click message.
      case WM_LBUTTONDOWN:

            // Capture the mouse (we still get mouse input
            // even after the mouse cursor moves off the client area.
            SetCapture(hWnd);
            gMouseDown = true;

            // Point that was clicked is stored in the lParam.
            gRect.left = LOWORD(lParam);
            gRect.top  = HIWORD(lParam);

            return 0;
      // Message sent whenever the mouse moves.
      case WM_MOUSEMOVE:

            if(gMouseDown)
            {
                  // Current mouse position is stored in the lParam.
                  gRect.right  = LOWORD(lParam);
                  gRect.bottom = HIWORD(lParam);

                  InvalidateRect(hWnd, 0, true);
            }

            return 0;
      case WM_LBUTTONUP:

            // Release the captured mouse when the left mouse button
            // is lifted.
            ReleaseCapture();
            gMouseDown = false;

            // Current mouse position is stored in the lParam.
            gRect.right  = LOWORD(lParam);
            gRect.bottom = HIWORD(lParam);

            gRects.push_back( gRect );

            InvalidateRect(hWnd, 0, true);

            return 0;
      case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);

            if( gMouseDown )
                  Rectangle(hdc, gRect.left,  gRect.top,
                                 gRect.right, gRect.bottom);

            for(int i = 0; i < gRects.size(); ++i)
                  Rectangle(hdc, gRects[i].left,  gRects[i].top,
                                 gRects[i].right, gRects[i].bottom);

            EndPaint(hWnd, &ps);
            return 0;

      // Handle key down message.
      case WM_KEYDOWN:
            if( wParam == VK_ESCAPE )
                  DestroyWindow(ghMainWnd);

            return 0;
      // Handle destroy window message.
      case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
      // default window procedure.
      return DefWindowProc(hWnd, msg, wParam, lParam);
}

// WinMain: Entry point for a Windows application.
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Save handle to application instance.
      ghAppInst = hInstance;

      // Step 2: Fill out a WNDCLASS instance.
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = ghAppInst;
      wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
      wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
      wc.hbrBackground = (HBRUSH)::GetStockObject(WHITE_BRUSH);
      wc.lpszMenuName  = 0;
      wc.lpszClassName = "MyWndClassName";

      // Step 3: Register the WNDCLASS instance with Windows.
      RegisterClass( &wc );

      // Step 4: Create the window, and save handle in global
      // window handle variable ghMainWnd.
      ghMainWnd = ::CreateWindow("MyWndClassName", "MyWindow",
            WS_OVERLAPPEDWINDOW, 200, 200, 640, 480, 0, 0,
            ghAppInst, 0);

      if(ghMainWnd == 0)
      {
            ::MessageBox(0, "CreateWindow - Failed", 0, 0);
            return false;
      }

      // Step 5: Show and update the window.
      ShowWindow(ghMainWnd, showCmd);
      UpdateWindow(ghMainWnd);

      // Step 6: Enter the message loop and don't quit until
      // a WM_QUIT message is received.
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));

      while( GetMessage(&msg, 0, 0, 0) )
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }

      // Return exit code back to operating system.
      return (int)msg.wParam;
}
```

15.2.3 Drawing Ellipses

Drawing an ellipse is done with the `Ellipse` function. What is interesting about the `Ellipse` function is that instead of specifying properties on an ellipse, we specify the dimensions of the ellipse's bounding rectangle. Figure 15.5 illustrates:

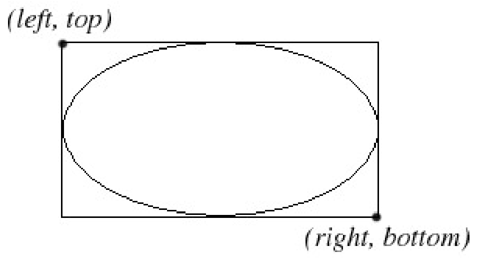


Figure 15.5: The Ellipse function draws an ellipse tightly inside the bounding rectangle specified.

Thus, it turns out that the `Ellipse` function has the exact same parameters as the `Rectangle` function. Here is an example call:

```
Ellipse(hdc, gRect.left, gRect.top, gRect.right, gRect.bottom);
```

Where `gRect` is of type `RECT`, which specifies the bounding rectangle of the ellipse.

We could write an ellipse-drawing program. However, the code would be practically the same as the rectangle sample. In fact, we can simply replace the `Rectangle` function with `Ellipse` function in Program 15.3 to draw ellipses instead of rectangles. We leave this as an exercise for you to try. Figure 15.6 shows how the output of such a program would look:

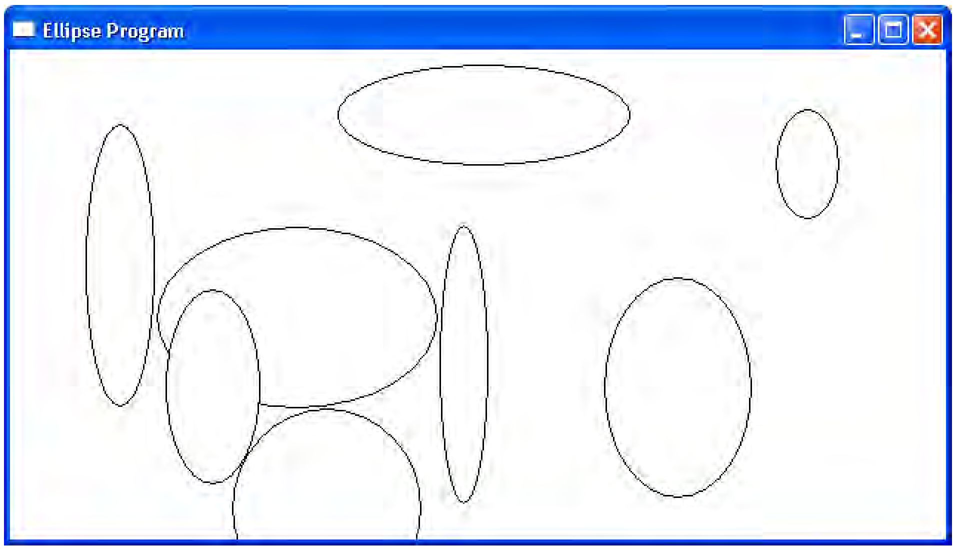


Figure 15.6: A screenshot of an ellipse drawing program.

15.3 Loading and Drawing Bitmaps

In this section, we will see how to load a .bmp image from file and display it in the client area of a window. This is not particularly difficult, but it is an important task because, later on when we begin to program some games, we will be working with bitmaps extensively. Before we begin, a brief definition of bitmaps is in order. A **bitmap** is simply a matrix of data, where each element in the matrix describes a color. We can map this color matrix to a rectangle of pixels on the monitor screen to display the bitmap image. Because the pixels are small and close together, a continuous image can be displayed on your monitor.

15.3.1 Loading

To load a bitmap, the first thing we must do is to load a bitmap as a resource. To do this, go the Visual C++ .NET menu and select *View->Resource View* as Figure 15.7 shows.

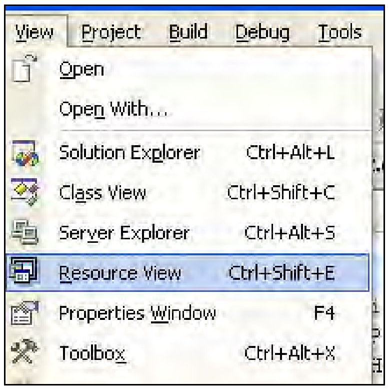


Figure 15.7: Opening the Resource View.

The "Resource View" panel should be displayed near the right side of the Visual C++ .NET interface. In the "Resource View" panel, right click your project name, and then select *Add->Add Resource* as Figure 15.8 shows.

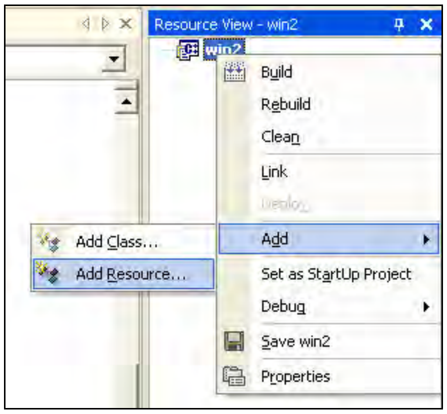


Figure 15.8: Adding a resource to the application.

A new "Add Resource" dialog will appear (Figure 15.9). Select "Bitmap" and then the "Import…" button.

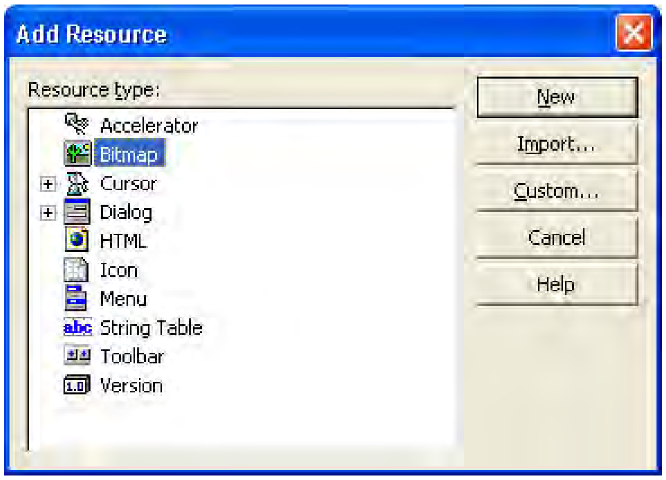


Figure 15.9: Adding a bitmap resource.

Now an "Import" dialog box appears (Figure 15.10). Use this dialog box to find the .bmp file you wish to load. Here we have placed a *gilogo.bmp* in the application project directory. Select the .bmp file you wish to load and select the "Open" button.

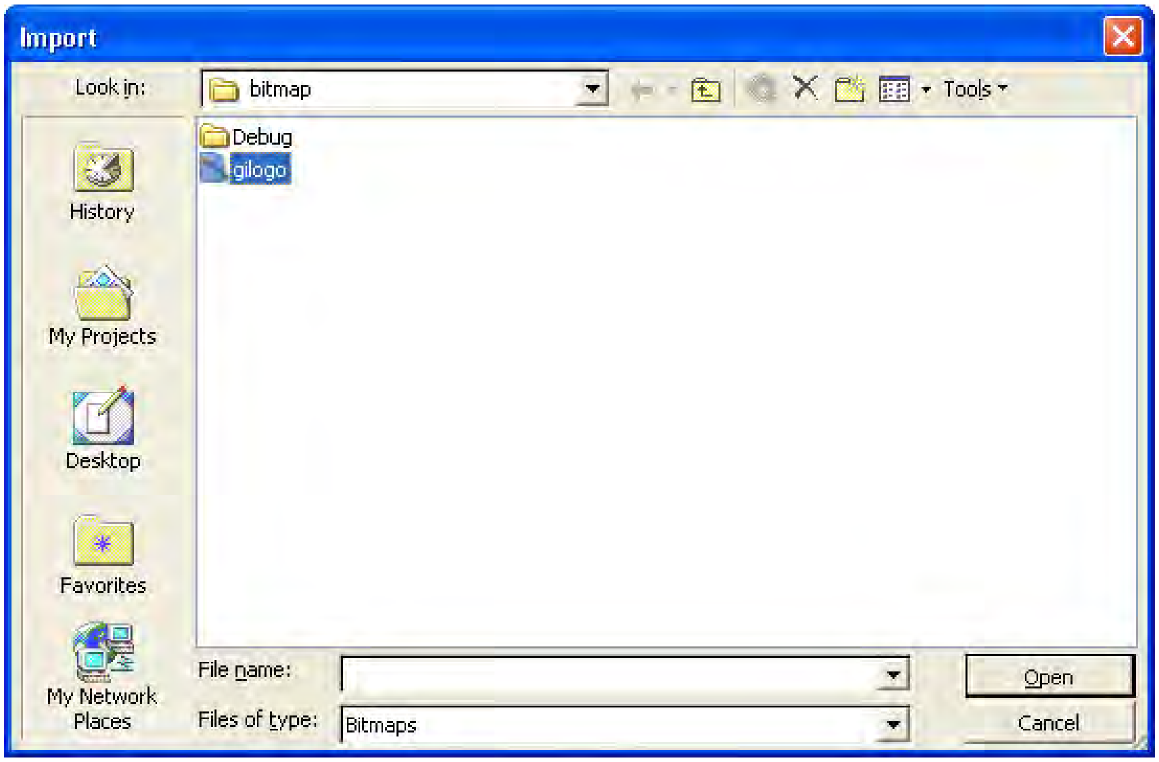


Figure 15.10: Selecting the bitmap file to import.

At this point, the .bmp file should be loaded as an application resource. Your "Resource View" panel should look like Figure 15.11.

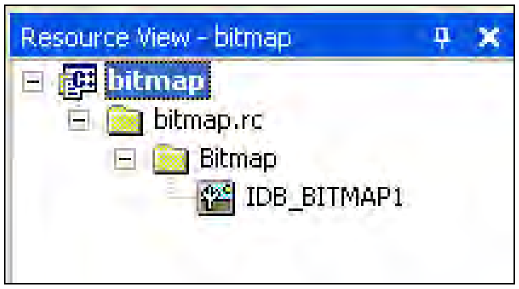


Figure 15.11: The Resource View panel.

By default, Visual C++ .NET automatically named the bitmap resource IDB_BITMAP1. You can change this if you like, but we will keep it as is for the sample program. Note that this resource name is a numeric identification symbol, which allows us to make reference to the bitmap in our program code.

Now that we have successfully loaded a bitmap resource, we can load it into our application. This is done with the `LoadBitmap` API function:

```
HBITMAP hBitMap = LoadBitmap(ghAppInst, MAKEINTRESOURCE(IDB_BITMAP1));
```

This function returns an `HBITMAP`; that is, a handle to the loaded bitmap. This function takes two arguments: the first is a handle to the application instance; the second is the numeric identification symbol name of the bitmap resource we wish to load. For the second parameter, we must pass our bitmap name through a macro (`MAKEINTRESOURCE`), which will convert a numeric ID to the bitmap name. Also note that when we add a new resource, a new header file called "resource.h" is automatically created, which contains our resource names and symbols. In order for the application to recognize the symbol `IDB_BITMAP1`, you will need to #include "resource.h".

Given a bitmap handle, we would like to get information about the bitmap, such as its width and height. To get the corresponding data structure of a bitmap we use the `GetObject` function:

```
BITMAP bitmap;
GetObject(hBitMap,        // Handle to GDI object.
          sizeof(BITMAP), // Size in bytes of GDI object.
          &bitmap);       // Instance to fill data members of.
```

This function fills out the `bitmap` instance. The `BITMAP` structure is defined like so:

```
typedef struct tagBITMAP {
  LONG   bmType;
  LONG   bmWidth;
  LONG   bmHeight;
  LONG   bmWidthBytes;
  WORD   bmPlanes;
  WORD   bmBitsPixel;
  LPVOID bmBits;
} BITMAP, *PBITMAP;
```

We only discuss the four most important data members:

`bmWidth`:	The width of the bitmap, measured in pixels.
`bmHeight`:	The height of the bitmap, measured in pixels.
`bmBitsPixel`:	The number of bits used to describe a single pixel in the bitmap (i.e., the number of bits used to describe a single color element). 24 bits per pixel is common for colored bitmaps; that is, 8-bits for red, 8-bits for green, and 8-bits for blue.
`bmBits`:	A pointer to the actual bitmap elements; that is, a pointer to the matrix array.

15.3.2 Rendering

To render a bitmap to the client area of a rectangle (i.e., copy the pixels of the bitmap over to the pixels of the client area) we must first associate the bitmap with a separate system memory device context. We do this with the `CreateCompatibleDC` function:

```
hdc = BeginPaint(hWnd, &ps);

HDC bmHDC = CreateCompatibleDC(hdc);
```

Next, we need to associate our bitmap with this new system memory device context. We can do that with the `SelectObject` function:

```
HBITMAP oldBM = (HBITMAP)SelectObject(bmHDC, hBitMap);
```

Note that the `SelectObject` function returns a handle to the previously selected object. It is considered good practice to restore the original object back to the device context when finished with the newly selected object. The first parameter of `SelectObject` is a handle to the device context into which you wish to select the GDI object; the second parameter is the handle to the GDI object you wish to select—in this case, the bitmap handle.

Now that we have our bitmap associated with a system memory device context (`bmHDC`), we can copy the pixels of the bitmap (source) over to the window's client area (destination) with the `BitBlt` function.

```
// Now copy the pixels from the bitmap bmHDC has selected
// to the pixels from the client area hdc has selected.
BitBlt(
      hdc,       // Destination DC.
      0,         // 'left' coordinate of destination rectangle.
      0,         // 'top' coordinate of destination rectangle.
      bmWidth,   // 'right' coordinate of destination rectangle.
      bmHeight,  // 'bottom' coordinate of destination rectangle.
      bmHDC,     // Bitmap source DC.
      0,         // 'left' coordinate of source rectangle.
      0,         // 'top' coordinate of source rectangle.
      SRCCOPY);  // Copy the source pixels from bitmap directly
                 // to the destination pixels (client area)
```

Finally, to clean up, we restore the original bitmap object back to the system memory device context, and then we delete the system memory device context because we are done with it:

```
SelectObject(bmHDC, oldBM);
DeleteDC(bmHDC);
EndPaint(hWnd, &ps);
```

15.3.3 Deleting

To destroy a bitmap object—that is, to free the bitmap memory—we use the `DeleteObject` function, which takes a handle to the object to delete:

```
DeleteObject(hBitMap);
```

15.3.4 Sample Program

This next sample program loads the Game Institute .bmp logo from file and displays it in the window's client area, as Figure 15.12 shows.

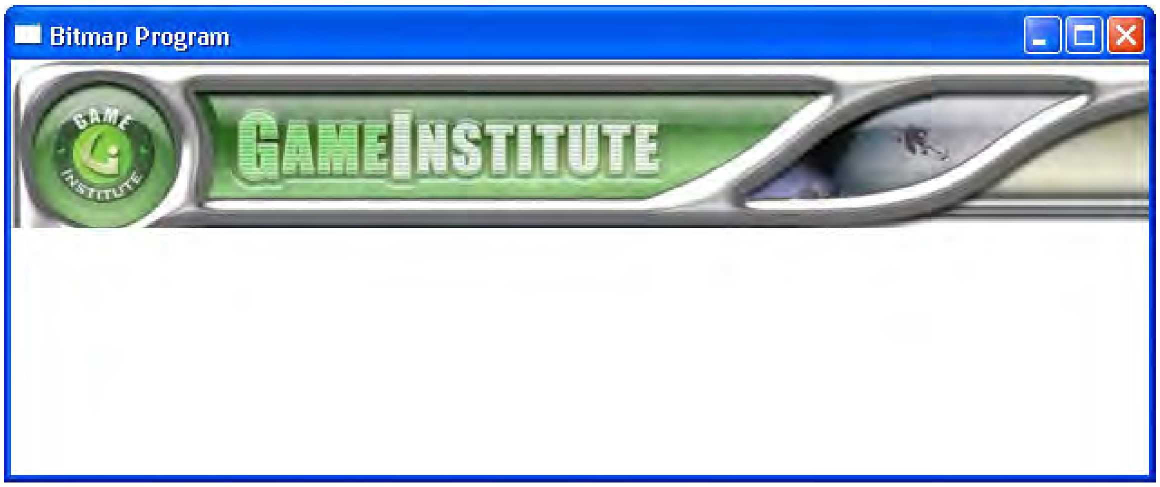


Figure 15.12: A screenshot of the bitmap sample.

> **Note:** You may notice this sample, and the other graphics samples, flicker slightly as you resize the window and under some other circumstances. We will address the flickering problem in later chapters.

There is one new key concept which this program uses which we have not discussed. It is the `WM_CREATE` message. This message is sent when the window is first created, and so, it is common to place initialization code in the `WM_CREATE` message handler. In Program 15.4, we load the bitmap in the `WM_CREATE` message handler:

```
case WM_CREATE:
      ghBitMap = LoadBitmap(ghAppInst,
                            MAKEINTRESOURCE(IDB_BITMAP1));
```

For convenience, we have bolded the new bitmap loading related segments of code.

Program 15.4: Bitmap drawing program.

```
#include <windows.h>
#include <string>
#include <vector>
#include "resource.h"
using namespace std;

//===========================================================
// Globals.

HWND      ghMainWnd = 0;
HINSTANCE ghAppInst = 0;

HBITMAP ghBitMap = 0;

// Step 1: Define and implement the window procedure.
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      // Objects for painting.
      HDC hdc = 0;
      HDC bmHDC = 0;
      PAINTSTRUCT ps;

      BITMAP bitmap = {0};

      static int bmWidth = 0;
      static int bmHeight = 0;

      HBITMAP oldBM = 0;

      switch( msg )
      {
      case WM_CREATE:
            ghBitMap = LoadBitmap(ghAppInst,
                                  MAKEINTRESOURCE(IDB_BITMAP1));

            GetObject(ghBitMap, sizeof(BITMAP), &bitmap);

            bmWidth  = bitmap.bmWidth;
            bmHeight = bitmap.bmHeight;

            return 0;
      case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);

            // Create a system memory device context.
            bmHDC = CreateCompatibleDC(hdc);
            // Hook up the bitmap to the bmHDC.
            oldBM = (HBITMAP)SelectObject(bmHDC, ghBitMap);

            // Now copy the pixels from the bitmap bmHDC has selected
            // to the pixels from the client area hdc has selected.
            BitBlt(
            hdc,      // Destination DC.
            0,        // 'left' coordinate of destination rectangle.
            0,        // 'top' coordinate of destination rectangle.
            bmWidth,  // 'right' coordinate of destination rectangle.
            bmHeight, // 'bottom' coordinate of destination rectangle.
            bmHDC,    // Bitmap source DC.
            0,        // 'left' coordinate of source rectangle.
            0,        // 'top' coordinate of source rectangle.
            SRCCOPY); // Copy the source pixels directly
                      // to the destination pixels

            // Select the originally loaded bitmap.
            SelectObject(bmHDC, oldBM);

            // Delete the system memory device context.
            DeleteDC(bmHDC);

            EndPaint(hWnd, &ps);
            return 0;

      // Handle key down message.
      case WM_KEYDOWN:
            if( wParam == VK_ESCAPE )
                  DestroyWindow(ghMainWnd);

            return 0;
      // Handle destroy window message.
      case WM_DESTROY:
            DeleteObject(ghBitMap);
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
      // default window procedure.
      return DefWindowProc(hWnd, msg, wParam, lParam);
}

// WinMain: Entry point for a Windows application.
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Save handle to application instance.
      ghAppInst = hInstance;

      // Step 2: Fill out a WNDCLASS instance.
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
```

```
	wc.hInstance     = ghAppInst;
	wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
	wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
	wc.hbrBackground = (HBRUSH)::GetStockObject(WHITE_BRUSH);
	wc.lpszMenuName  = 0;
	wc.lpszClassName = "MyWndClassName";

	// Step 3: Register the WNDCLASS instance with Windows.
	RegisterClass( &wc );

	// Step 4: Create the window, and save handle in globla
	// window handle variable ghMainWnd.
	ghMainWnd = ::CreateWindow("MyWndClassName", "Bitmap Program",
		WS_OVERLAPPEDWINDOW, 200, 200, 640, 480, 0, 0,
		ghAppInst, 0);

	if(ghMainWnd == 0)
	{
		::MessageBox(0, "CreateWindow - Failed", 0, 0);
		return false;
	}

	// Step 5: Show and update the window.
	ShowWindow(ghMainWnd, showCmd);
	UpdateWindow(ghMainWnd);

	// Step 6: Enter the message loop and don't quit until
	// a WM_QUIT message is received.
	MSG msg;
	ZeroMemory(&msg, sizeof(MSG));

	while( GetMessage(&msg, 0, 0, 0) )
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}

	// Return exit code back to operating system.
	return (int)msg.wParam;
}
```

15.4 Pens and Brushes

Just as an artist can select different pens and brushes to achieve different results, the Win32 API provides conceptual pens and brushes in code, which allows us to modify the way in which things are drawn. Via pens and brushes it was possible to draw the shapes in Figure 15.1 with different colors, hatch patterns, and styles.

15.4.1 Pens

Pens are used to draw lines, curves, and also to draw the border of solid shapes like rectangles and ellipses. The properties of a pen are described with the `LOGPEN` structure:

```
typedef struct tagLOGPEN {
  UINT     lopnStyle;
  POINT    lopnWidth;
  COLORREF lopnColor;
} LOGPEN, *PLOGPEN;
```

- `lopnStyle`: The pen style; some common styles are `PS_SOLID` (solid pen), `PS_DOT` (for dotted lines/borders), and `PS_DASH` (for dashed lines/borders).

- `lopnWidth`: The thickness of the pen, in pixels.

- `lopnColor`: The pen color.

Given a filled out `LOGPEN` instance that describes a pen, we can create a pen with those properties using the `CreatePenIndirect` function:

```
LOGPEN lp;
lp.lopnStyle   = PS_SOLID;
lp.lopnWidth.x = 1;
lp.lopnWidth.y = 1;
lp.lopnColor   = RGB(255, 0, 255);

HPEN hPen   = CreatePenIndirect(&lp);
```

When you are done with a pen, you should delete it with the `DeleteObject` function:

```
DeleteObject(hPen);
```

Once a pen is created, it must be selected by the device context before it can be used. This is done with the `SelectObject` function:

```
mOldPen   = (HPEN)SelectObject(hdc, mPen);
```

Again, `SelectObject` will return a handle to the previously selected GDI object. It is advised that you reselect the old GDI object when you are done with the new one:

```
hOldPen   = (HPEN)SelectObject(hdc, hPen);

// do work with hPen

SelectObject(hdc, hOldPen);
```

15.4.2 Brushes

Brushes are used to fill in the interiors of solid shapes like rectangles and ellipses. The properties of a brush are described with the `LOGBRUSH` structure:

```
typedef struct tagLOGBRUSH {
  UINT     lbStyle;
  COLORREF lbColor;
  LONG     lbHatch;
} LOGBRUSH, *PLOGBRUSH;
```

- `lbStyle`: The brush style; some common styles are `BS_SOLID` (solid brush), `BS_NULL` (creates a brush that does not fill in the interior regions of solid shapes), `BS_HATCHED` (creates a brush that draws hatched marks; the hatching style depends on the `lbHatch` member of the `LOGBRUSH` structure).

- `lbColor`: The brush color.

- `lbHatch`: The hatching style; some common hatching styles are `HS_BDIAGONAL` (diagonal hatch marks), and `HS_CROSS` (crossed hatch marks).

Given a filled out `LOGBRUSH` instance that describes a brush, we can create a brush with those properties using the `CreateBrushIndirect` function:

```
LOGBRUSH lb;
lb.lbStyle = BS_HATCHED;
lb.lbColor   = RGB(255, 0, 255);
lb.lbHatch   = HS_CROSS;

HBRUSH hBrush   = CreateBrushIndirect(&lp);
```

Note: If you create a brush with the style `BS_SOLID` then the value in `lbHatch` is ignored, because, by definition, a solid brush is not hatched.

When you are done with a brush, you should delete it with the `DeleteObject` function:

```
DeleteObject(hBrush);
```

Once a brush is created, it must be selected by the device context before it can be used. This is done with the `SelectObject` function:

```
hOldBrush   = (HBRUSH)SelectObject(hdc, hBrush);
```

Again, `SelectObject` will return a handle to the previously selected GDI object. It is advised that you reselect the old GDI object when you are done with the new one:

```
hOldBrush   = (HBRUSH)SelectObject(hdc, hBrush);

// do work with hBrush

SelectObject(hdc, hOldBrush);
```

15.5 Shape Classes

We are nearing the end of this chapter, where we will be able to finally implement the drawing program displayed in Figure 15.1. In that program we will need to be able to draw different kinds of shapes, so it will simplify matters if we create some shape classes to represent the three different kinds of shapes we will draw—lines, rectangles and ellipses. Moreover, we will want to use polymorphism because it will make our implementation easier if we can work directly with general shapes rather than different kinds of concrete shapes. (i.e., we can store all shapes, `LineShapes`, `RectShapes`, and `EllipseShapes` in one `Shape*` container and work with the shapes without caring about what specific kind of shape they are—review polymorphism in Module I if this is confusing.)

15.5.1 Class Definitions

First, what do lines, rectangles, and ellipses all have in common? We have seen that all shapes can be represented with two points. A line has a start and end point. A rectangle has the (left, top) and (right, bottom) points and so does an ellipse, since an ellipse is drawn based on its specified bounding rectangle. Additionally, all shapes will have a pen and brush that specifies the color and style in which it should be drawn. Thus we have the following data that is common to all shapes:

```
      POINT  mPt0;
      POINT  mPt1;

      HPEN   mhPen;
      HBRUSH mhBrush;

      HPEN mhOldPen;
      HBRUSH mhOldBrush;
```

Note that we have added `mhOldPen` and `mhOldBrush`, so that we can restore the original pen and brush after we are done drawing with `mhPen` and `mhBrush`.

Next, what can all shapes do? We will say that all shapes know how to draw themselves. However, at the general `Shape` level, we do not know what shape to draw. Therefore, we make the `draw` method an *abstract* method, which must be overridden and implemented in the derived classes. Also note that because we do all drawing through the device context, the draw method takes a parameter to the handle of a device context. In addition to drawing, we provide methods for setting the start and end points of the shape. For rectangles (and ellipses), the start point corresponds to the point (left, top) and the end point corresponds to the point (right, bottom). Thus we have the following methods that are common to all shapes:

```
void setStartPt(const POINT& p0);
void setEndPt(const POINT& p1);

virtual void draw(HDC hdc) = 0;
```

In addition to that, we will also need a constructor and destructor:

```
Shape(const POINT u, const POINT v,
      const LOGPEN& lp, const LOGBRUSH& lb);

virtual ~Shape();
```

Observe that the `Shape` class contains all the data any derived class will need. Thus, derived classes need only to override the `draw` method. The following code shows the full definition of the `Shape` class, as well as the definitions of three derived classes `LineShape`, `RectShape`, and `EllipseShape`.

```
// Shape.h
#ifndef SHAPE_H
#define SHAPE_H

#include <windows.h>

class Shape
{
public:
      Shape(const POINT u, const POINT v,
            const LOGPEN& lp, const LOGBRUSH& lb);

      virtual~Shape();

      void setStartPt(const POINT& p0);
      void setEndPt(const POINT& p1);

      virtual void draw(HDC hdc) = 0;

protected:
      POINT  mPt0;
      POINT  mPt1;
      HPEN   mhPen;
      HBRUSH mhBrush;

      HPEN mhOldPen;
      HBRUSH mhOldBrush;
};
```

```
class LineShape : public Shape
{
public:
      LineShape(const POINT u, const POINT v,
            const LOGPEN& lp, const LOGBRUSH& lb);

      void draw(HDC hdc);
};

class RectShape : public Shape
{
public:
      RectShape(const POINT u, const POINT v,
            const LOGPEN& lp, const LOGBRUSH& lb);

      void draw(HDC hdc);
};

class EllipseShape : public Shape
{
public:
      EllipseShape(const POINT u, const POINT v,
            const LOGPEN& lp, const LOGBRUSH& lb);

      void draw(HDC hdc);
};

#endif // SHAPE_H
```

15.5.2 Class Implementations

We start with the constructor of the `Shape` class. As you can see, the constructor has a `LOGPEN` and `LOGBRUSH` parameter from which we can get an `HPEN` and `HBRUSH` instance:

```
// Shape.cpp
#include "Shape.h"

Shape::Shape(const POINT u, const POINT v,
             const LOGPEN& lp, const LOGBRUSH& lb)
{
      mPt0.x = u.x;
      mPt0.y = u.y;
      mPt1.x = v.x;
      mPt1.y = v.y;
      mhPen   = CreatePenIndirect(&lp);
      mhBrush = CreateBrushIndirect(&lb);

      mhOldPen   = 0;
      mhOldBrush = 0;
}
```

A destructor should free any resources that were allocated in the constructor. The only resource we allocate in the constructor of `Shape` was a pen and brush, so we delete these GDI objects in the destructor like so:

```
Shape::~Shape()
{
      DeleteObject(mhPen);
      DeleteObject(mhBrush);
}
```

The `Shape` `set*` methods are implemented trivially:

```
void Shape::setStartPt(const POINT& p0)
{
      mPt0 = p0;
}

void Shape::setEndPt(const POINT& p1)
{
      mPt1 = p1;
}
```

Moving onto the derived classes, the `LineShape` constructor does nothing except call the parent constructor:

```
LineShape::LineShape(const POINT u, const POINT v,
                     const LOGPEN& lp, const LOGBRUSH& lb)
                    : Shape(u, v, lp, lb)
{}
```

This follows from the fact that the `LineShape` added no new data members, so there is nothing else to construct except the `Shape` part of `LineShape`.

The `LineShape` `draw` method simply selects its pen and brush before drawing (so that the line will be drawn using that pen and brush), draws the line based on the start and end point (`mPt0` and `mPt1`), and restores the original pen and brush.

```
void LineShape::draw(HDC hdc)
{
      // Select the current pen and brush.
      mhOldPen   = (HPEN)SelectObject(hdc, mhPen);
      mhOldBrush = (HBRUSH)SelectObject(hdc, mhBrush);

      // Draw the line.
      MoveToEx(hdc, mPt0.x, mPt0.y, 0);
      LineTo(hdc, mPt1.x, mPt1.y);

      // Restore the old pen and brush.
      SelectObject(hdc, mhOldPen);
      SelectObject(hdc, mhOldBrush);
}
```

The implementations for the other classes, `RectShape` and `EllipseShape`, are exactly the same as `LineShape`, except we replace the line drawing GDI functions with the `Rectangle` and `Ellipse` functions, respectively. Their implementations are as follows:

```
RectShape::RectShape(const POINT u, const POINT v,
                     const LOGPEN& lp, const LOGBRUSH& lb)
                     : Shape(u, v, lp, lb)
{
}

void RectShape::draw(HDC hdc)
{
      // Select the current pen and brush.
      mhOldPen   = (HPEN)SelectObject(hdc, mhPen);
      mhOldBrush = (HBRUSH)SelectObject(hdc, mhBrush);

      // Draw the rectangle.
      Rectangle(hdc, mPt0.x, mPt0.y, mPt1.x, mPt1.y);

      // Restore the old pen and brush.
      SelectObject(hdc, mhOldPen);
      SelectObject(hdc, mhOldBrush);
}

EllipseShape::EllipseShape(const POINT u, const POINT v,
                           const LOGPEN& lp, const LOGBRUSH& lb)
                           : Shape(u, v, lp, lb)
{
}



void EllipseShape::draw(HDC hdc)
{
      // Select the current pen and brush.
      mhOldPen   = (HPEN)SelectObject(hdc, mhPen);
      mhOldBrush = (HBRUSH)SelectObject(hdc, mhBrush);

      // Draw the ellipse.
      Ellipse(hdc, mPt0.x, mPt0.y, mPt1.x, mPt1.y);

      // Restore the old pen and brush.
      SelectObject(hdc, mhOldPen);
      SelectObject(hdc, mhOldBrush);
}
```

15.6 Menus

The last feature we need to learn about in order to implement the drawing program is how to create and use menus.

15.6.1 Creating a Menu Resource

To create a menu, you need to go to the Visual C++ .NET menu and select *View->Resource View*. When the "Resource View" panel comes up, right click the project name in it, and select *Add->Add Resource*. When the "Add Resource" dialog is displayed, select "Menu" and then press the "New" button—Figure 15.13.

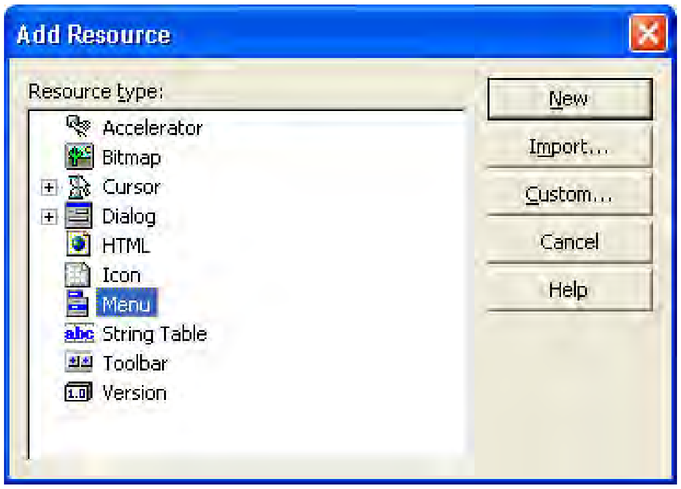


Figure 15.13: Adding a menu resource.

The Visual C++ .NET menu editor will be displayed. As with the other resources you have created, you can change the resource ID name (Figure 15.14) if you so desire, however, we will leave it set to the default, IDR_MENU1. Again, the resource ID is how we refer to a particular resource in code.

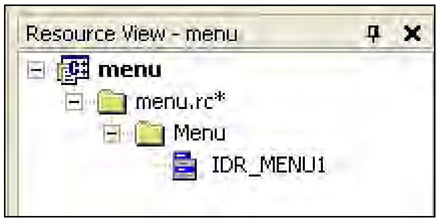


Figure 15.14: The Resource View panel.

Adding menu items to the resource is fairly intuitive with the menu editor. Just select the box and type in the menu item string name you want displayed. Figure 15.15 shows the menu items we create for the paint drawing sample program.

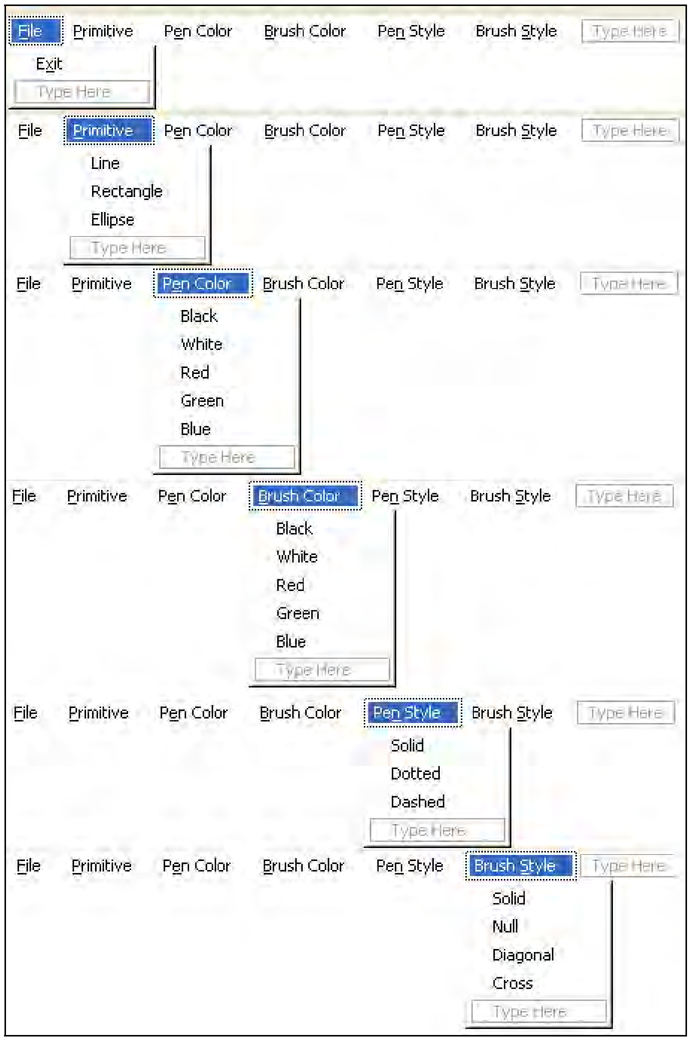
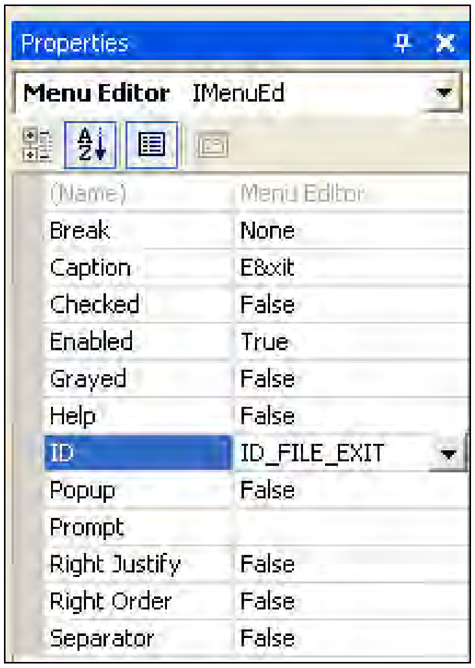


Figure 15:15: Screenshots of each menu category we made for the paint drawing program.

> **Note:** If you put an ampersand, "&", before a character in the menu item name, the character that follows the ampersand will be underlined, which means you can use the keyboard to select the menu item instead of the mouse. That is, you can "alt-key."

As you are typing the menu item name for a new menu item, notice the properties box in the lower right hand corner of Visual C++ .NET (Figure 15.16). This properties box allows you to set various properties for the menu item. The "caption" or menu item name that is displayed is only one property. To check a menu item by default, set "Checked" to true. To disable a menu item by default, set "Enabled" to false. Also observe that each menu item is given an ID. Again, this resource ID is how we will refer to a particular menu item in code. You can name the ID whatever you want, but Visual C++ .NET will typically pick a good descriptive ID name based on the menu caption.



Menu Editor IMenuEd

(Name)	Menu Editor
Break	None
Caption	E&xit
Checked	False
Enabled	True
Grayed	False
Help	False
ID	ID_FILE_EXIT
Popup	False
Prompt	
Right Justify	False
Right Order	False
Separator	False

Figure 15.16: The Menu Item properties table.

We used the following menu item IDs in our drawing program:

```
ID_FILE_EXIT
ID_PRIMITIVE_LINE
ID_PRIMITIVE_RECTANGLE
ID_PRIMITIVE_ELLIPSE
ID_PRIMITIVE_TEXT
ID_PENCOLOR_BLACK
ID_PENCOLOR_WHITE
ID_PENCOLOR_RED
ID_PENCOLOR_GREEN
ID_PENCOLOR_BLUE
ID_BRUSHCOLOR_BLACK
ID_BRUSHCOLOR_WHITE
ID_BRUSHCOLOR_RED
ID_BRUSHCOLOR_GREEN
ID_BRUSHCOLOR_BLUE
ID_PENSTYLE_SOLID
ID_PENSTYLE_NULL
ID_PENSTYLE_DOTTED
ID_PENSTYLE_DASHED
ID_BRUSHSTYLE_SOLID
ID_BRUSHSTYLE_NULL
ID_BRUSHSTYLE_DIAGONAL
ID_BRUSHSTYLE_CROSS
```

15.6.2 Loading a Menu and Attaching it to a Window

Now that we have created a menu resource, we need to get a handle to that menu in our program. We can do that with the `LoadMenu` function:

```
HMENU hMenu = LoadMenu(ghAppInst, MAKEINTRESOURCE(IDR_MENU1));
```

Recall that you will need to #include "resource.h" as that is where the menu symbols are defined, such as `IDR_MENU1`.

Now that we have a handle to a menu, we can attach it to a window when we create the window in the `CreateWindow` function:

```
ghMainWnd = ::CreateWindow("MyWndClassName", "My Paint Program",
            WS_OVERLAPPEDWINDOW, 200, 200, 640, 480, 0, hMenu,
            ghAppInst, 0);
```

Recall that a parameter of `CreateWindow` is an `HMENU`. Before, we always specified null for that parameter because we were not using a menu. But now that we have created and loaded a menu, we can pass the menu handle to `CreateWindow` to attach the menu to the window.

15.6.3 Checking Menu Items

Programmatically, we can put a check mark next to a menu item (or remove a check mark from a menu item) with the `CheckMenuItem` API function. This function is prototyped like so:

```
DWORD CheckMenuItem(
  HMENU hmenu,
  UINT uIDCheckItem,
  UINT uCheck
);
```

The first parameter, `hmenu`, is a handle to the menu we are working with. We need to specify the specific menu because a program could actually have several menus on various sub-windows. The second parameter `uIDCheckItem,` is the resource ID of the menu item to check. Finally, the third parameter `uCheck,` is either `MF_CHECKED` or `MF_UNCHECKED`. `MF_CHECKED` puts a check mark next to the menu item and `MF_UNCHECKED` removes a check mark from a menu item (if it was previously checked).
For example, to check the menu item with the ID `ID_PRIMITIVE_LINE`, we would write:

```
CheckMenuItem(hMenu, ID_PRIMITIVE_LINE, MF_CHECKED);
```

15.6.4 Selecting Menu Items

When the user selects a menu item, we will generally want to execute some code in response to that menu item selection. To facilitate this, when the user selects a menu item, a `WM_COMMAND` message is sent to the window's window procedure. The lower 16-bits of the `WM_COMMAND`'s `wParam` value stores the menu item's resource ID. Thus, in code, we can figure out which item was selected and execute the appropriate code like so:

```
// User selected a menu item.
case WM_COMMAND:
      // Determine which menu item.
      switch( LOWORD(wParam) ) // Check lower 16-bits of wParam
      {
      // User selected the "Exit" menu item.
      case ID_FILE_EXIT:
            DestroyWindow(ghMainWnd);
            return 0;

      // User selected the "Line" menu item.
      case ID_PRIMITIVE_LINE:
            // Execute code in response.
            return 0;
      // User selected the "Rectangle" menu item.
      case ID_PRIMITIVE_RECTANGLE:
            // Execute code in response.
            return 0;
      // User selected the "Ellipse" menu item.
      case ID_PRIMITIVE_ELLIPSE:
            // Execute code in response.
            return 0;
```

15.7 The Paint Sample

We now have enough background knowledge to implement the paint program shown in Figure 15.1. With the exception of the menu handling code, the program logic is very similar to the other drawing programs we have made.

The first thing we do is define some constant color values at the global scope:

```
const COLORREF BLACK = RGB(0, 0, 0);
const COLORREF WHITE = RGB(255, 255, 255);
const COLORREF RED   = RGB(255, 0, 0);
const COLORREF GREEN = RGB(0, 255, 0);
const COLORREF BLUE  = RGB(0, 0, 255);
```

This simply allows us to easily refer to some basic colors using the symbol name, instead of RGB macros.

Next, we create a global vector of `Shape` pointers, `gShapes`, which will maintain all the shapes we have created so that we can draw all the shapes whenever a `WM_PAINT` message is generated. In addition, we keep a global shape pointer `gShape`, which will be the temporary shape we will draw while the user is moving the mouse around and deciding where exactly to make the shape permanent. Notice how we can talk about shapes in general. This is because of the polymorphism. Each shape will know how to draw itself correctly based on its concrete dynamic type (that is, if it is a `LineShape` then it will know to draw a line, if it is a `RectShape` then it will know to draw a rectangle, and so on).

```
vector<Shape*> gShapes;
Shape* gShape = 0;
```

Recall that the program allows the user to draw three kinds of shapes. The primitive menu item that is selected determines the shape that will be drawn at a given time. For example, if the "Line" menu item is selected (checked) then the user can draw lines. If the "Rectangle" menu item is selected (checked) then the user can draw rectangles.

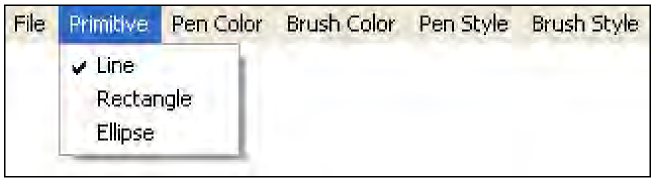


Figure 15.17: User can select which type of shape to draw via the menu.

We need to keep track of which primitive menu item is selected so that we know which kind of shape to create. To facilitate this, we keep a global variable that keeps track of the currently selected primitive:

```
int gCurrPrimSel       = ID_PRIMITIVE_LINE;
```

Later in the `WM_LBUTTONDOWN` message handler, we create the shape based on the currently selected type:

```
switch( gCurrPrimSel )
      {
      case ID_PRIMITIVE_LINE:
            gShape = new LineShape(p0, p1, gLogPen, gLogBrush);
            break;
      case ID_PRIMITIVE_RECTANGLE:
            gShape = new RectShape(p0, p1, gLogPen, gLogBrush);
            break;
      case ID_PRIMITIVE_ELLIPSE:
            gShape = new EllipseShape(p0, p1,gLogPen,gLogBrush);
            break;
      };
```

In addition to keeping track of the selected primitive type, we will also want to keep track of the selected pen and brush color, and the selected pen and brush style. For this purpose, we add the following global variables:

```
int gCurrPenColSel     = ID_PENCOLOR_BLACK;
int gCurrBrushColSel   = ID_BRUSHCOLOR_BLACK;
int gCurrPenStyleSel   = ID_PENSTYLE_SOLID;
int gCurrBrushStyleSel = ID_BRUSHSTYLE_SOLID;
```

The values to which these global variables are initialized are the program's "default" selected values. Therefore, we also need to check the corresponding menu items at the start of the program so that the default selections are checked. We implement this functionality in the `WM_CREATE` message:

```
case WM_CREATE:
      CheckMenuItem(ghMenu, ID_PRIMITIVE_LINE, MF_CHECKED);
      CheckMenuItem(ghMenu, ID_PENCOLOR_BLACK, MF_CHECKED);
      CheckMenuItem(ghMenu, ID_BRUSHCOLOR_BLACK, MF_CHECKED);
      CheckMenuItem(ghMenu, ID_PENSTYLE_SOLID, MF_CHECKED);
      CheckMenuItem(ghMenu, ID_BRUSHSTYLE_SOLID, MF_CHECKED);
      return 0;
```

This way, the default-selected menu items will be checked when the window is first created.

We also keep a global instance of a `LOGPEN` and `LOGBRUSH`:

```
LOGPEN gLogPen;
LOGBRUSH gLogBrush;
```

We update the data members of these structures interactively as the user makes new menu selections. For instance, if the user selects the menu item with the ID `ID_PENSTYLE_DOTTED` then we update `gLogPen` like so:

```
case ID_PENSTYLE_DOTTED:
      CheckMenuItem(ghMenu, ID_PENSTYLE_DOTTED, MF_CHECKED);
      CheckMenuItem(ghMenu, gCurrPenStyleSel, MF_UNCHECKED);
      gCurrPenStyleSel = ID_PENSTYLE_DOTTED;
      gLogPen.lopnStyle = PS_DOT; // update pen style
      return 0;
```

Also notice how we check the newly selected item, uncheck the previously selected item, then update `gCurrPenStyleSel`.

When a shape is created, we pass `gLogPen` and `gLogBrush` into the constructor. For example,

```
gShape = new LineShape(p0, p1, gLogPen, gLogBrush);
```

Because we update `gLogPen` and `gLogBrush` immediately as the user selects new colors and styles from the menu, these objects always reflect the current user menu selections. Therefore, any object created will always be created with the currently selected pen and brush colors, and pen and brush styles, which is the exact functionality which should happen.

The majority of code in the paint-drawing program has to do with the menu and updating the primitive type that is selected, the colors of the pen and brush, and the styles of the pen and brush. The code is long, but simple. For example, the part of the code that checks for which primitive type the user selected is as follows:

```
case WM_COMMAND:
      switch( LOWORD(wParam) )
      {

      //=======================================
      // Primitive Types (Shape Types)
      //=======================================
      case ID_PRIMITIVE_LINE:
            CheckMenuItem(ghMenu, ID_PRIMITIVE_LINE, MF_CHECKED);
            CheckMenuItem(ghMenu, gCurrPrimSel, MF_UNCHECKED);
            gCurrPrimSel = ID_PRIMITIVE_LINE;
            return 0;
      case ID_PRIMITIVE_RECTANGLE:
            CheckMenuItem(ghMenu, ID_PRIMITIVE_RECTANGLE, MF_CHECKED);
            CheckMenuItem(ghMenu, gCurrPrimSel, MF_UNCHECKED);
            gCurrPrimSel = ID_PRIMITIVE_RECTANGLE;
            return 0;
      case ID_PRIMITIVE_ELLIPSE:
            CheckMenuItem(ghMenu, ID_PRIMITIVE_ELLIPSE, MF_CHECKED);
            CheckMenuItem(ghMenu, gCurrPrimSel, MF_UNCHECKED);
            gCurrPrimSel = ID_PRIMITIVE_ELLIPSE;
            return 0;
```

It is all quite redundant. We simply update which menu items should be checked, and update the global `gCurrPrimSel` variable so that it reflects the currently selected item. We execute similar code for the pen color menu items, the brush color menu items, the pen style menu items, and the brush style menu items. As mentioned, the code is all largely the same, so we will not discuss it further.

You should now be able to understand the drawing program. The code is listed in Program 15.5.

Program 15.5: Paint drawing program.

```
#include <windows.h>
#include <string>
#include <vector>
#include "Shape.h"
#include "resource.h"
using namespace std;

//=========================================================
// Globals.

const COLORREF BLACK = RGB(0, 0, 0);
const COLORREF WHITE = RGB(255, 255, 255);
const COLORREF RED   = RGB(255, 0, 0);
const COLORREF GREEN = RGB(0, 255, 0);
const COLORREF BLUE  = RGB(0, 0, 255);

HWND      ghMainWnd = 0;
HINSTANCE ghAppInst = 0;
HMENU     ghMenu    = 0;
```

```
vector<Shape*> gShapes;
Shape* gShape = 0;

bool gMouseDown = false;

int gCurrPrimSel       = ID_PRIMITIVE_LINE;
int gCurrPenColSel     = ID_PENCOLOR_BLACK;
int gCurrBrushColSel   = ID_BRUSHCOLOR_BLACK;
int gCurrPenStyleSel   = ID_PENSTYLE_SOLID;
int gCurrBrushStyleSel = ID_BRUSHSTYLE_SOLID;

LOGPEN gLogPen;
LOGBRUSH gLogBrush;

//=========================================================

// Step 1: Define and implement the window procedure.
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      // Objects for painting.
      HDC hdc = 0;
      PAINTSTRUCT ps;

      // Local POINT variables we will use in some of the case
      // statements.
      POINT p0;
      POINT p1;

      switch( msg )
      {
      case WM_CREATE:
            CheckMenuItem(ghMenu, ID_PRIMITIVE_LINE, MF_CHECKED);
            CheckMenuItem(ghMenu, ID_PENCOLOR_BLACK, MF_CHECKED);
            CheckMenuItem(ghMenu, ID_BRUSHCOLOR_BLACK, MF_CHECKED);
            CheckMenuItem(ghMenu, ID_PENSTYLE_SOLID, MF_CHECKED);
            CheckMenuItem(ghMenu, ID_BRUSHSTYLE_SOLID, MF_CHECKED);
            return 0;

      case WM_COMMAND:
            switch( LOWORD(wParam) )
            {
            //=========================================
            // File Menu
            //=========================================
            case ID_FILE_EXIT:
                  DestroyWindow(ghMainWnd);
                  return 0;
            //=========================================
            // Primitive Types (Shape Types)
            //=========================================
            case ID_PRIMITIVE_LINE:
                  CheckMenuItem(ghMenu, ID_PRIMITIVE_LINE, MF_CHECKED);
                  CheckMenuItem(ghMenu, gCurrPrimSel, MF_UNCHECKED);
                  gCurrPrimSel = ID_PRIMITIVE_LINE;
                  return 0;
            case ID_PRIMITIVE_RECTANGLE:
             CheckMenuItem(ghMenu, ID_PRIMITIVE_RECTANGLE, MF_CHECKED);
                  CheckMenuItem(ghMenu, gCurrPrimSel, MF_UNCHECKED);
                  gCurrPrimSel = ID_PRIMITIVE_RECTANGLE;
                  return 0;
            case ID_PRIMITIVE_ELLIPSE:
               CheckMenuItem(ghMenu, ID_PRIMITIVE_ELLIPSE, MF_CHECKED);
                  CheckMenuItem(ghMenu, gCurrPrimSel, MF_UNCHECKED);
                  gCurrPrimSel = ID_PRIMITIVE_ELLIPSE;
                  return 0;
            //========================================
            // Pen Colors
            //========================================
            case ID_PENCOLOR_BLACK:
                  CheckMenuItem(ghMenu, ID_PENCOLOR_BLACK, MF_CHECKED);
                  CheckMenuItem(ghMenu, gCurrPenColSel, MF_UNCHECKED);
                  gCurrPenColSel = ID_PENCOLOR_BLACK;
                  gLogPen.lopnColor = BLACK;
                  return 0;
            case ID_PENCOLOR_WHITE:
                  CheckMenuItem(ghMenu, ID_PENCOLOR_WHITE, MF_CHECKED);
                  CheckMenuItem(ghMenu, gCurrPenColSel, MF_UNCHECKED);
                  gCurrPenColSel = ID_PENCOLOR_WHITE;
                  gLogPen.lopnColor = WHITE;
                  return 0;
            case ID_PENCOLOR_RED:
                  CheckMenuItem(ghMenu, ID_PENCOLOR_RED, MF_CHECKED);
                  CheckMenuItem(ghMenu, gCurrPenColSel, MF_UNCHECKED);
                  gCurrPenColSel = ID_PENCOLOR_RED;
                  gLogPen.lopnColor = RED;
                  return 0;
            case ID_PENCOLOR_GREEN:
                  CheckMenuItem(ghMenu, ID_PENCOLOR_GREEN, MF_CHECKED);
                  CheckMenuItem(ghMenu, gCurrPenColSel, MF_UNCHECKED);
                  gCurrPenColSel = ID_PENCOLOR_GREEN;
                  gLogPen.lopnColor = GREEN;
                  return 0;
            case ID_PENCOLOR_BLUE:
                  CheckMenuItem(ghMenu, ID_PENCOLOR_BLUE, MF_CHECKED);
                  CheckMenuItem(ghMenu, gCurrPenColSel, MF_UNCHECKED);
                  gCurrPenColSel = ID_PENCOLOR_BLUE;
                  gLogPen.lopnColor = BLUE;
                  return 0;
            //========================================
            // Brush Colors
            //========================================
            case ID_BRUSHCOLOR_BLACK:
                CheckMenuItem(ghMenu, ID_BRUSHCOLOR_BLACK, MF_CHECKED);
                 CheckMenuItem(ghMenu, gCurrBrushColSel, MF_UNCHECKED);
                  gCurrBrushColSel = ID_BRUSHCOLOR_BLACK;
                  gLogBrush.lbColor = BLACK;
                  return 0;
            case ID_BRUSHCOLOR_WHITE:
                CheckMenuItem(ghMenu, ID_BRUSHCOLOR_WHITE, MF_CHECKED);
                 CheckMenuItem(ghMenu, gCurrBrushColSel, MF_UNCHECKED);
                  gCurrBrushColSel = ID_BRUSHCOLOR_WHITE;
                  gLogBrush.lbColor = WHITE;
                  return 0;
```

```
            case ID_BRUSHCOLOR_RED:
                  CheckMenuItem(ghMenu, ID_BRUSHCOLOR_RED, MF_CHECKED);
                 CheckMenuItem(ghMenu, gCurrBrushColSel, MF_UNCHECKED);
                  gCurrBrushColSel = ID_BRUSHCOLOR_RED;
                  gLogBrush.lbColor = RED;
                  return 0;
            case ID_BRUSHCOLOR_GREEN:
                CheckMenuItem(ghMenu, ID_BRUSHCOLOR_GREEN, MF_CHECKED);
                 CheckMenuItem(ghMenu, gCurrBrushColSel, MF_UNCHECKED);
                  gCurrBrushColSel = ID_BRUSHCOLOR_GREEN;
                  gLogBrush.lbColor = GREEN;
                  return 0;
            case ID_BRUSHCOLOR_BLUE:
                 CheckMenuItem(ghMenu, ID_BRUSHCOLOR_BLUE, MF_CHECKED);
                 CheckMenuItem(ghMenu, gCurrBrushColSel, MF_UNCHECKED);
                  gCurrBrushColSel = ID_BRUSHCOLOR_BLUE;
                  gLogBrush.lbColor = BLUE;
                  return 0;
            //========================================
            // Pen Styles
            //========================================
            case ID_PENSTYLE_SOLID:
                  CheckMenuItem(ghMenu, ID_PENSTYLE_SOLID, MF_CHECKED);
                 CheckMenuItem(ghMenu, gCurrPenStyleSel, MF_UNCHECKED);
                  gCurrPenStyleSel = ID_PENSTYLE_SOLID;
                  gLogPen.lopnStyle = PS_SOLID;
                  return 0;
            case ID_PENSTYLE_DOTTED:
                 CheckMenuItem(ghMenu, ID_PENSTYLE_DOTTED, MF_CHECKED);
                 CheckMenuItem(ghMenu, gCurrPenStyleSel, MF_UNCHECKED);
                  gCurrPenStyleSel = ID_PENSTYLE_DOTTED;
                  gLogPen.lopnStyle = PS_DOT;
                  return 0;
            case ID_PENSTYLE_DASHED:
                 CheckMenuItem(ghMenu, ID_PENSTYLE_DASHED, MF_CHECKED);
                 CheckMenuItem(ghMenu, gCurrPenStyleSel, MF_UNCHECKED);
                  gCurrPenStyleSel = ID_PENSTYLE_DASHED;
                  gLogPen.lopnStyle = PS_DASH;
                  return 0;
            //========================================
            // Brush Styles
            //========================================
            case ID_BRUSHSTYLE_SOLID:
                CheckMenuItem(ghMenu, ID_BRUSHSTYLE_SOLID, MF_CHECKED);
               CheckMenuItem(ghMenu, gCurrBrushStyleSel, MF_UNCHECKED);
                  gCurrBrushStyleSel = ID_BRUSHSTYLE_SOLID;
                  gLogBrush.lbStyle = BS_SOLID;
                  return 0;
            case ID_BRUSHSTYLE_NULL:
                 CheckMenuItem(ghMenu, ID_BRUSHSTYLE_NULL, MF_CHECKED);
               CheckMenuItem(ghMenu, gCurrBrushStyleSel, MF_UNCHECKED);
                  gCurrBrushStyleSel = ID_BRUSHSTYLE_NULL;
                  gLogBrush.lbStyle = BS_NULL;
                  return 0;
            case ID_BRUSHSTYLE_DIAGONAL:
             CheckMenuItem(ghMenu, ID_BRUSHSTYLE_DIAGONAL, MF_CHECKED);
               CheckMenuItem(ghMenu, gCurrBrushStyleSel, MF_UNCHECKED);
```

```
                  gCurrBrushStyleSel = ID_BRUSHSTYLE_DIAGONAL;
                  gLogBrush.lbStyle = BS_HATCHED;
                  gLogBrush.lbHatch = HS_BDIAGONAL;
                  return 0;
            case ID_BRUSHSTYLE_CROSS:
                CheckMenuItem(ghMenu, ID_BRUSHSTYLE_CROSS, MF_CHECKED);
               CheckMenuItem(ghMenu, gCurrBrushStyleSel, MF_UNCHECKED);
                  gCurrBrushStyleSel = ID_BRUSHSTYLE_CROSS;
                  gLogBrush.lbStyle = BS_HATCHED;
                  gLogBrush.lbHatch = HS_CROSS;
                  return 0;
            }
      // Handle left mouse button click message.
      case WM_LBUTTONDOWN:

            // Capture the mouse (we still get mouse input
            // even after the mouse cursor moves off the client area.
            SetCapture(hWnd);
            gMouseDown = true;

            // Point that was clicked is stored in the lParam.
            p0.x = LOWORD(lParam);
            p0.y = HIWORD(lParam);

            // We don't know the end point yet, so set to zero.
            p1.x = 0;
            p1.y = 0;

            // Create the shape based on what shape the user has
            // selected in the menu.
            switch( gCurrPrimSel )
            {
            case ID_PRIMITIVE_LINE:
                  gShape = new LineShape(p0, p1, gLogPen, gLogBrush);
                  break;
            case ID_PRIMITIVE_RECTANGLE:
                  gShape = new RectShape(p0, p1, gLogPen, gLogBrush);
                  break;
            case ID_PRIMITIVE_ELLIPSE:
                  gShape = new EllipseShape(p0, p1,gLogPen,gLogBrush);
                  break;
            };

            return 0;
      // Message sent whenever the mouse moves.
      case WM_MOUSEMOVE:
            if(gMouseDown)
            {
                  // Current mouse position is stored in the lParam.
                  p1.x = LOWORD(lParam);
                  p1.y = HIWORD(lParam);

                  // Update the end point of the current temporary
                  // shape based on the mouse position.
                  gShape->setEndPt(p1);

                  // Repaint the window so the temporary shape
```

```
                    // is redrawn interactively as the mouse moves.
                    InvalidateRect(hWnd, 0, true);
              }

              return 0;
       case WM_LBUTTONUP:

              // Release the captured mouse when the left mouse button
              // is lifted.
              ReleaseCapture();
              gMouseDown = false;

              // Current mouse position is stored in the lParam.
              p1.x = LOWORD(lParam);
              p1.y = HIWORD(lParam);

              // Update the end point of the current temporary shape
              // based on the mouse position.
              gShape->setEndPt(p1);

              // The user lifted the left mouse button, so the shape
              // becomes permanent, so add it to the shape container.
              gShapes.push_back( gShape );

              // Repaint the window so the new permanent shape will
              // be displayed.
              InvalidateRect(hWnd, 0, true);

              return 0;
       case WM_PAINT:
              hdc = BeginPaint(hWnd, &ps);

              // Only draw temporary shape if the mouse is down.
              if( gMouseDown )
                    gShape->draw(hdc);

              // Draw all the permenent shapes.
              for(int i = 0; i < gShapes.size(); ++i)
                    gShapes[i]->draw(hdc);

              EndPaint(hWnd, &ps);

       // Handle key down message.
       case WM_KEYDOWN:
              if( wParam == VK_ESCAPE )
                    DestroyWindow(ghMainWnd);

              return 0;
       // Handle destroy window message.
       case WM_DESTROY:
              PostQuitMessage(0);
              return 0;
       }
       // Forward any other messages we didn't handle to the
       // default window procedure.
       return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

```
// WinMain: Entry point for a Windows application.
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Save handle to application instance.
      ghAppInst = hInstance;

      // Step 2: Fill out a WNDCLASS instance.
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = ghAppInst;
      wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
      wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
      wc.hbrBackground = (HBRUSH)::GetStockObject(WHITE_BRUSH);
      wc.lpszMenuName  = 0;
      wc.lpszClassName = "MyWndClassName";

      // Step 3: Register the WNDCLASS instance with Windows.
      RegisterClass( &wc );

      // Step 4: Create the window, and save handle in globla
      // window handle variable ghMainWnd.
      ghMenu = LoadMenu(ghAppInst, MAKEINTRESOURCE(IDR_MENU1));
      ghMainWnd = ::CreateWindow("MyWndClassName", "My Paint Program",
            WS_OVERLAPPEDWINDOW, 200, 200, 640, 480, 0, ghMenu,
            ghAppInst, 0);

      if(ghMainWnd == 0)
      {
            ::MessageBox(0, "CreateWindow - Failed", 0, 0);
            return false;
      }

      // Step 5: Show and update the window.
      ShowWindow(ghMainWnd, showCmd);
      UpdateWindow(ghMainWnd);

      // Step 6: Enter the message loop and don't quit until
      // a WM_QUIT message is received.
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));

      while( GetMessage(&msg, 0, 0, 0) )
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }

      for(int i = 0; i < gShapes.size(); ++i)
            delete gShapes[i];

      // Return exit code back to operating system.
      return (int)msg.wParam;
}
```

15.8 Summary

1. Windows sends a window a `WM_PAINT` message whenever a window needs to repaint its client area (or a region of the client area). A window may need to repaint its client area when it is resized, or when a previously obscured part of it becomes visible. The program can explicitly send a `WM_PAINT` message with the `InvalidateRect` function. A Windows program should know how to draw all of its data, so that if all or part of its client area drawing data is erased (by resizing the window or obscuring the window, for example) it can all be restored at the appropriate time when the next `WM_PAINT` is sent to the window.

2. A device context is a software abstraction of a display device, such as the video card or a printer. We do all drawing operations through the device context (abbreviated as DC). As such, we must pass a handle to a device context (HDC) to all GDI related functions. One way to obtain an HDC is with the `BeginPaint` function.

3. To draw text onto the client area of a window, we use the `TextOut` function. To draw a line onto the client area of a window, we use the `MoveToEx` function to specify the line's start point, and the `LineTo` function to specify the line's end point. To draw a rectangle onto the client area of a window, we use the `Rectangle` function. Finally, to draw an ellipse onto the client area of a window, we use the `Ellipse` function. Recall that when specifying an ellipse we specify its bounding rectangle; that is, an ellipse that fits tightly into that bounding rectangle will be drawn.

4. A bitmap is a matrix of color elements, or pixels. To load a bitmap we must first load the .bmp file as a resource. After that, we can load the bitmap resource into our program and obtain a handle to it with the `LoadBitmap` function. To copy the pixels in the bitmap to the window's client area we use the `BitBlt` function. When we are finished with a bitmap, we should delete it with the `DeleteObject` function.

5. Pens and brushes allow us to alter the way in which shapes are drawn with GDI. In particular, we can draw shapes with different colors, styles, and patterns. Pens are used to draw lines, curves, and also to draw the border of solid shapes like rectangles and ellipses. The properties of a pen are described with the `LOGPEN` structure. Brushes are used to fill in the interiors of solid shapes like rectangles and ellipses. The properties of a brush are described with the `LOGBRUSH` structure.

6. Menus allow the user to make selections. To load a menu, we must first create a menu resource in the Visual C++ resource editor. After we have finished editing the menu resource, we can load it with the `LoadMenu` function. To attach the menu to a window, we can pass in a handle to the menu (`HMENU` returned from `LoadMenu`) to the `hMenu` parameter of `CreateWindow`. We can check/uncheck a menu item with the `CheckMenuItem` function. When the user selects a menu item, Windows sends the window a `WM_COMMAND` message. The lower 16-bits of the `wParam` parameter of this message specify the numeric resource ID of the menu item that was selected. In this way, we can test which menu item was selected and execute the appropriate consequential code.

15.9 Exercises

15.9.1 Colors

Modify Program 15.5 by adding additional pen and brush colors (you pick the colors), which the user can select from the menu and draw with.

15.9.2 Styles

Look up the `LOGPEN` and `LOGBRUSH` structures in the Win32 documentation to see some of the additional styles and hatch patterns that are supported. Modify Program 15.5 by adding additional pen and brush styles/hatch patterns (you pick the styles/patterns), which the user can select from the menu and draw with.

15.9.3 Cube

Figure 15.18 shows a cube. As you can see, a cube is made up of twelve lines. Write a function that draws a cube to the client area of a window. Then add a new "cube" primitive to Program 15.5 that the user can select from the menu and draw.

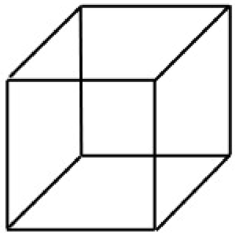


Figure 15.18: A cube outline drawn with twelve lines.

15.9.4 Undo Feature

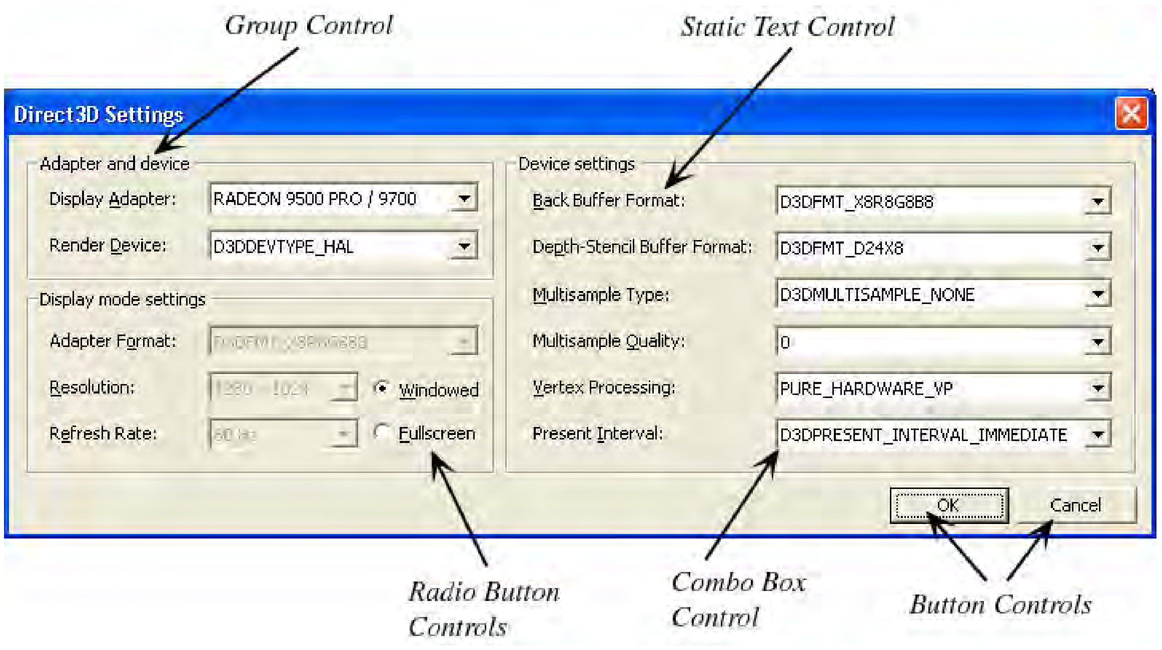
Modify Program 15.5 by implementing an undo feature that erases the last shape that was drawn (add an undo menu item). Be sure to avoid memory leaks.

Chapter 16

Introduction to Dialogs and Controls

Introduction

The final Win32 API-related topics we need to discuss are dialog boxes and controls. A dialog box is a type of window, which is commonly used to display a variety of controls the user can set and configure. Figure 16.1 shows a dialog box that is used in the demos programs for the DirectX Graphics Programming course here at Game Institute.



Direct3D Settings

Adapter and device
Display Adapter: RADEON 9500 PRO / 9700
Render Device: D3DDEVTYPE_HAL

Display mode settings
Adapter Format: D3DFMT_X8R8G8B8
Resolution: 1280 x 1024
Refresh Rate: 60 Hz
[✓ Windowed] Fullscreen

Device settings
Back Buffer Format: D3DFMT_X8R8G8B8
Depth-Stencil Buffer Format: D3DFMT_D24X8
Multisample Type: D3DMULTISAMPLE_NONE
Multisample Quality: 0
Vertex Processing: PURE_HARDWARE_VP
Present Interval: D3DPRESENT_INTERVAL_IMMEDIATE

Figure 16.1: An example of a dialog box with controls.

As you can see from Figure 16.1, controls and dialog boxes go hand in hand.

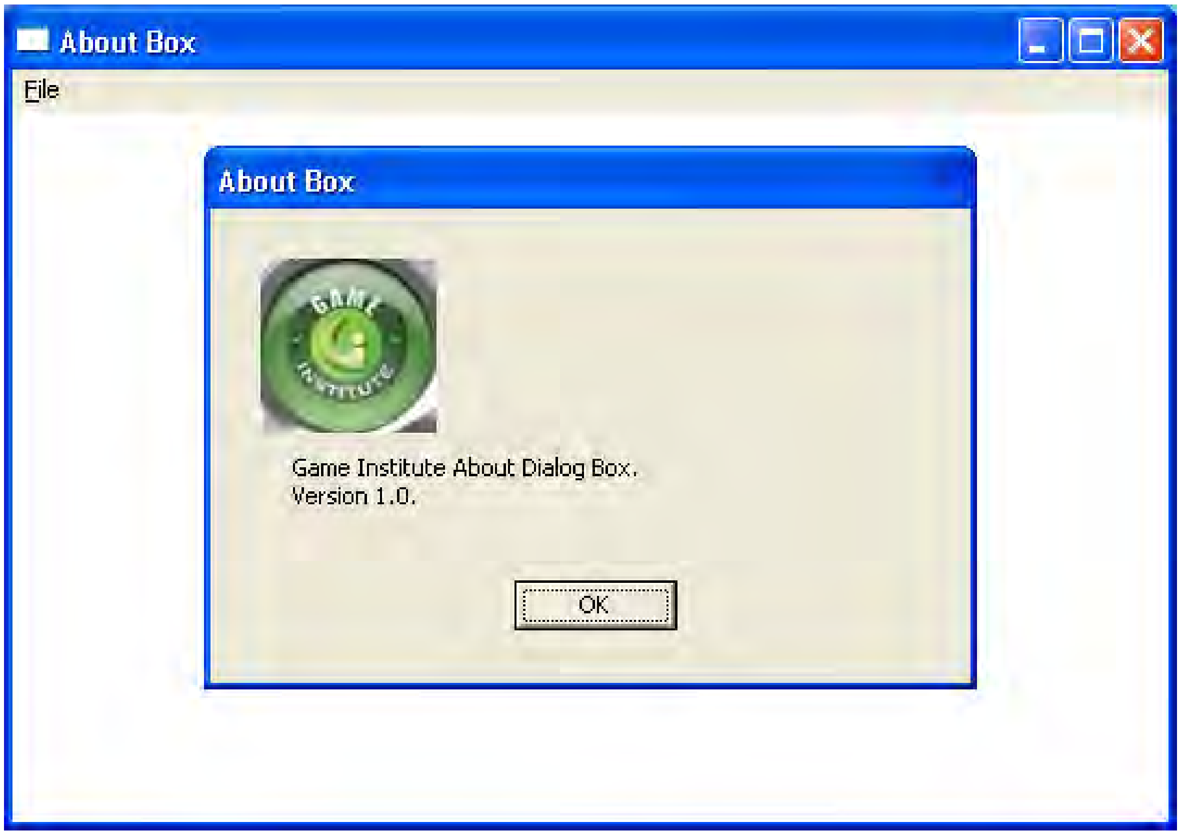
Chapter Objectives

- Learn how to create modal and modeless dialog boxes, and how to distinguish between the two.
- Discover how to create and design dialog boxes with the Visual C++ resource editor.
- Become familiar with several Win32 controls such as static text controls, picture box controls, edit box controls, radio button controls, button controls, and combo box controls.

Note: As with the previous two Win32 API chapters in this book, we cannot possibly cover everything. Therefore, if you are interested in learning how to create the various other Win32 controls you may have seen, but which we do not cover here (e.g., slider controls, color controls, spin controls, tree controls), then you will need to consult MSDN or a book devoted to the Win32 API. The standard "bible" text on the Win32 API is *Programming Windows 5th Edition* By Charles Petzold.

16.1 Modal Dialog Boxes; The Static Text Control; The Button Control

A **modal dialog box** is a dialog box that does not close until the user has made some sort of selection (i.e., the user cannot switch to another window until the modal dialog box has ended). Typically, an application will use a modal dialog box when it needs immediate input from the user, and cannot continue until it receives that input. For our first program, we will illustrate a modal dialog box by creating a simple window that can display an "About" dialog box. Figure 16.2 shows our goal:



Game Institute About Dialog Box.
Version 1.0.

Figure 16.2: The dialog box is launched by going to File and selecting the "About" menu item.

16.1.1 Designing the Dialog Box

Dialogs are created and edited in the Visual C++ Resource editor, just like menus and icons. Select the *Add Resource* button and select a "Dialog" from the "Add Resource" dialog and then select the "New" button—Figure 16.3.

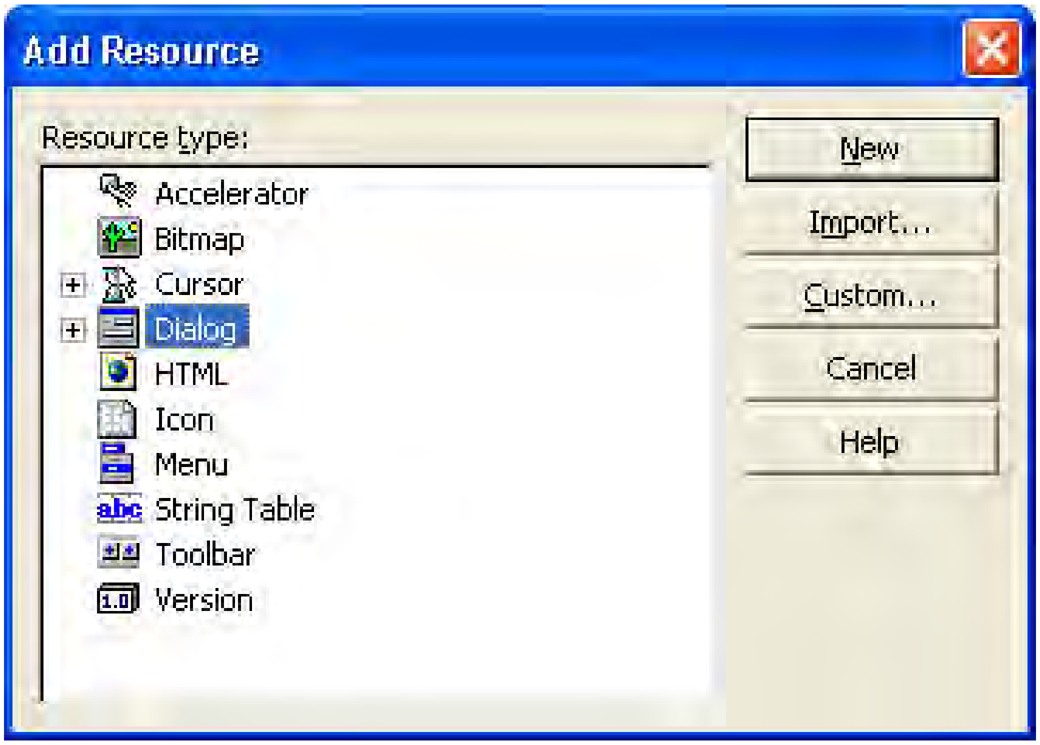


Figure 16.3: Select "Dialog" and then press the "New" button.

At this point, the Dialog Editor will open up. Before we begin designing the dialog box, let us rename the dialog resource ID to IDD_ABOUTBOX—Figure 16.4.

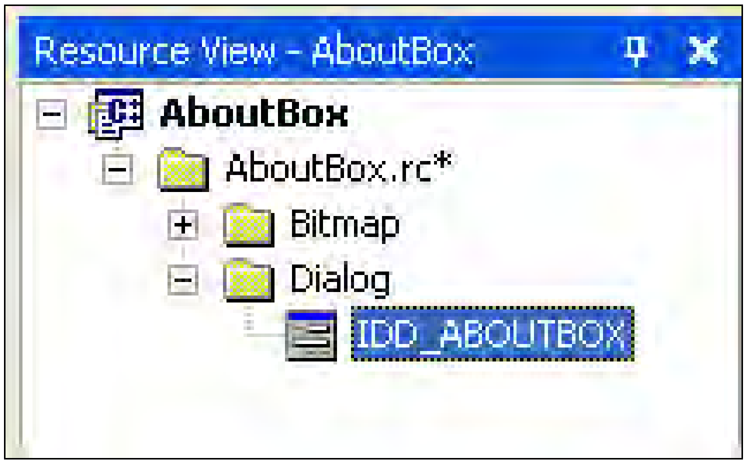


Figure 16.4: Renaming the dialog resource ID to IDD_ABOUTBOX.

Now, design your dialog box as Figure 16.5 shows.

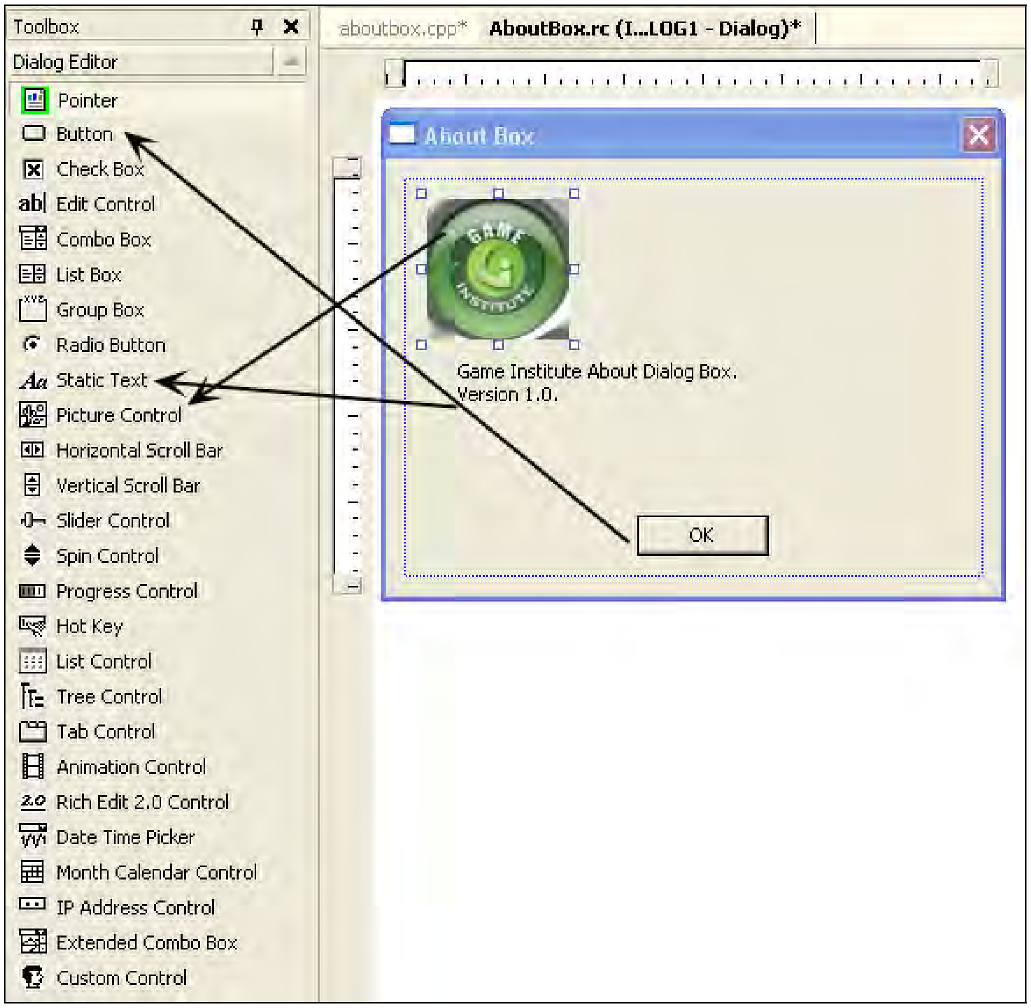


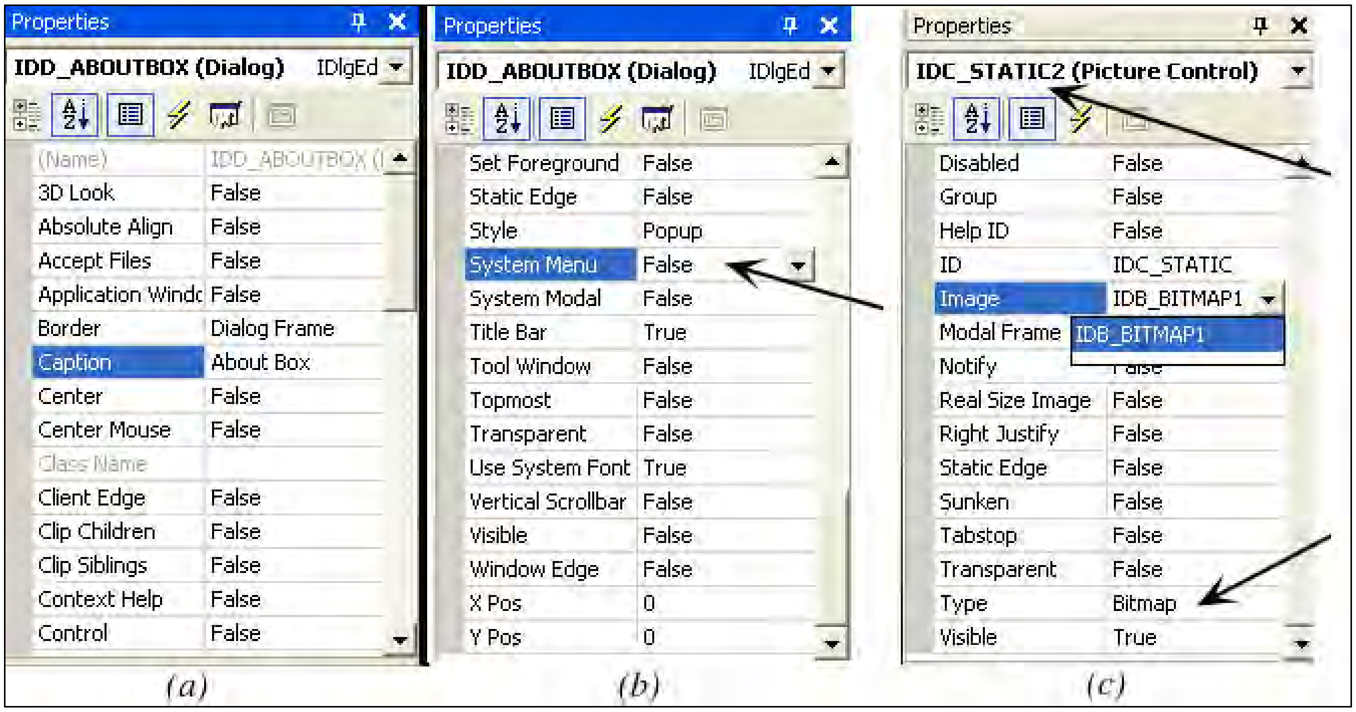
Figure 16.5: Designing the dialog. To add a control to the dialog, select the control from the left hand side control listing and drag it onto the dialog.

To delete a control, select the control from the dialog and press the delete key on your keyboard. To add a control to the dialog, select the control from the left hand side control listing and drag it onto the dialog. You can move controls around the dialog by selecting them and dragging them about; controls can also be resized by dragging the borders of the control windows.

First, change the dialog box caption from "Dialog" to something else. We chose to rename it as "About Box." You can rename the dialog box caption by selecting the dialog box and editing the "Caption" property—Figure 16.6*a*. The dialog properties should be displayed in the lower right corner of Visual C++ .NET when you select the dialog. While we are editing the dialog box properties, also set the "System Menu" property to false—Figure 16.6*b*. This removes the 'X' close button from the dialog box—typically, modal dialog boxes do not have system menus.

Second, change the static text from the default text to "Game Institute About Dialog Box. Version 1.0." Just like menu items, each control has a list of properties, which are displayed in the lower right corner of Visual C++ .NET when you select the control. You can change the caption of the static text control by editing the "Caption" property.

Next, load the Game Institute logo into the picture box (you can use another image if you like, of course). First, you need to load the bitmap image as a resource. We showed how to load a bitmap resource in the previous chapter. In our program, we keep the default bitmap resource ID, namely, IDB_BITMAP1. Now, select the picture control so that its properties are displayed. Change the "Type" property to "Bitmap" and then change the "Image" property to the bitmap resource ID which you want displayed in the picture box. In our example, this is IDB_BITMAP1—Figure 16.6*c*.


Properties

IDD_ABOUTBOX (Dialog) IDlgEd

(Name)	IDD_ABOUTBOX (
3D Look	False
Absolute Align	False
Accept Files	False
Application Windc	False
Border	Dialog Frame
Caption	About Box
Center	False
Center Mouse	False
Class Name	
Client Edge	False
Clip Children	False
Clip Siblings	False
Context Help	False
Control	False

(a)

Properties

IDD_ABOUTBOX (Dialog) IDlgEd

Set Foreground	False
Static Edge	False
Style	Popup
System Menu	False
System Modal	False
Title Bar	True
Tool Window	False
Topmost	False
Transparent	False
Use System Font	True
Vertical Scrollbar	False
Visible	False
Window Edge	False
X Pos	0
Y Pos	0

(b)

Properties

IDC_STATIC2 (Picture Control)

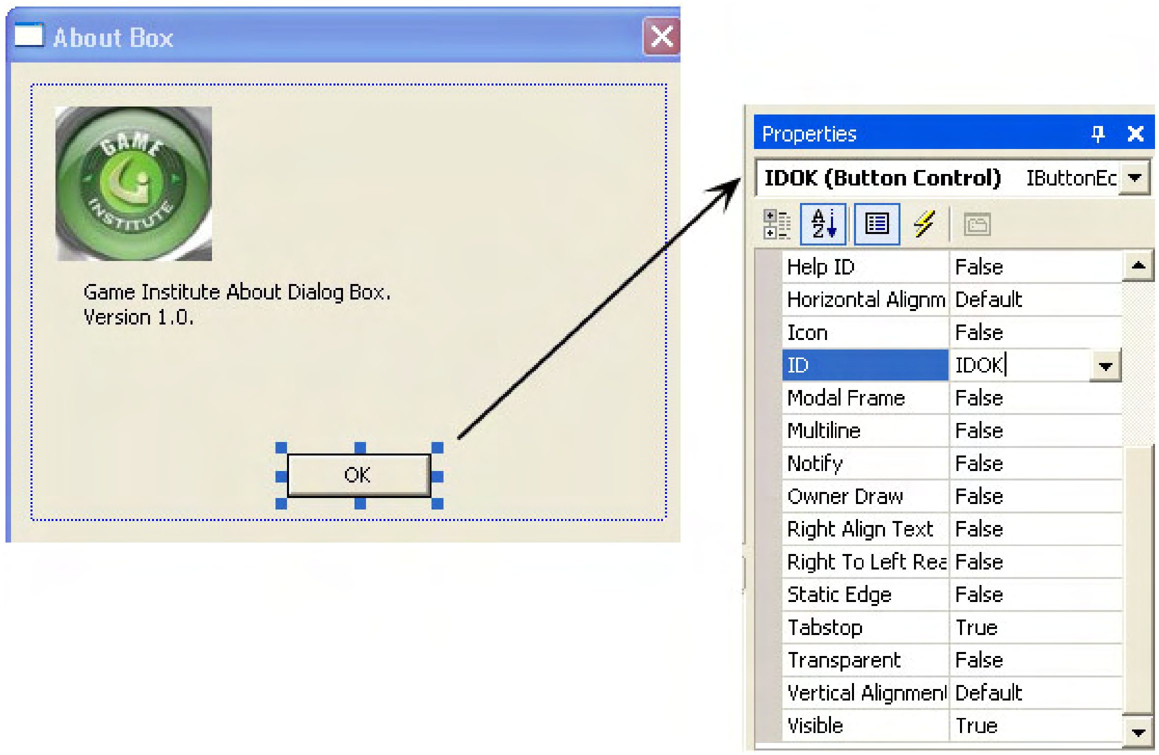
Disabled	False
Group	False
Help ID	False
ID	IDC_STATIC
Image	IDB_BITMAP1
Modal Frame	IDB_BITMAP1
Notify	False
Real Size Image	False
Right Justify	False
Static Edge	False
Sunken	False
Tabstop	False
Transparent	False
Type	Bitmap
Visible	True

(c)

Figure 16.6: (a) Changing the dialog box's caption. (b) Removing the dialog box's system menu. (c) Setting the picture control type to bitmap, and loading the bitmap IDB_BITMAP1 as the picture box's image.

Your loaded image logo should now be displayed in the picture box, as it is in Figure 16.5.

Note that we do not care about the resource ID of the picture box control or the static text control. This is because these controls are static and do not change, in general. Thus we will not need to access them from our program. However, we should take a brief look at the properties of the OK button control. Recall that the button was placed on the dialog box by default. If we select the OK button control, the properties window looks like Figure 16.7. The key observation for now is that the resource ID that was given to it: IDOK. That name sounds reasonable, so we will keep it, but we must remember it because we will need to know the ID so that we can execute the appropriate code when the user presses the button in the program. In particular, we will kill the modal dialog when the user presses the OK button.



Game Institute About Dialog Box.
Version 1.0.

Help ID	False
Horizontal Alignm	Default
Icon	False
ID	IDOK
Modal Frame	False
Multiline	False
Notify	False
Owner Draw	False
Right Align Text	False
Right To Left Rea	False
Static Edge	False
Tabstop	True
Transparent	False
Vertical Alignment	Default
Visible	True

Figure 16.7: The OK Button's control properties.

16.1.2 Modal Dialog Box Theory

A dialog box is a kind of window, and as such, has its own window procedure function. A dialog box window procedure looks similar to a regular window procedure except that it returns `BOOL`, which is `typedef`ed as `int` by the Win32 API. Here is a typical dialog procedure declaration:

```
BOOL CALLBACK
AboutDlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam);
```

A special message that is specific to dialog boxes is the `WM_INITDIALOG` message. This message is an analog to the `WM_CREATE` message we handle for non-dialog windows. Typically, we handle the `WM_INITDIALOG` message so that we can initialize the controls of the dialog to default values. In addition, the `WM_INITDIALOG` is a good place to obtain window handles to each control on the dialog (remember, controls are child windows and the dialog is the parent window).

After we have implemented a dialog window procedure, we can display a modal dialog with the `DialogBox` function:

```
DialogBox(
      ghAppInst,                      // Application instance.
      MAKEINTRESOURCE(IDD_ABOUTBOX),  // Dialog resource ID.
      hWnd,                           // Parent window of dialog box.
      AboutDlgProc);                  // Ptr to dialog box window procedure.
```

Note that `DialogBox` is used only for modal dialog boxes. We will illustrate how to create and show modeless dialog boxes in the next section.

In the sample program we developed, we display the "About" dialog box when the user selects the "About" menu item—see Figure 16.8.

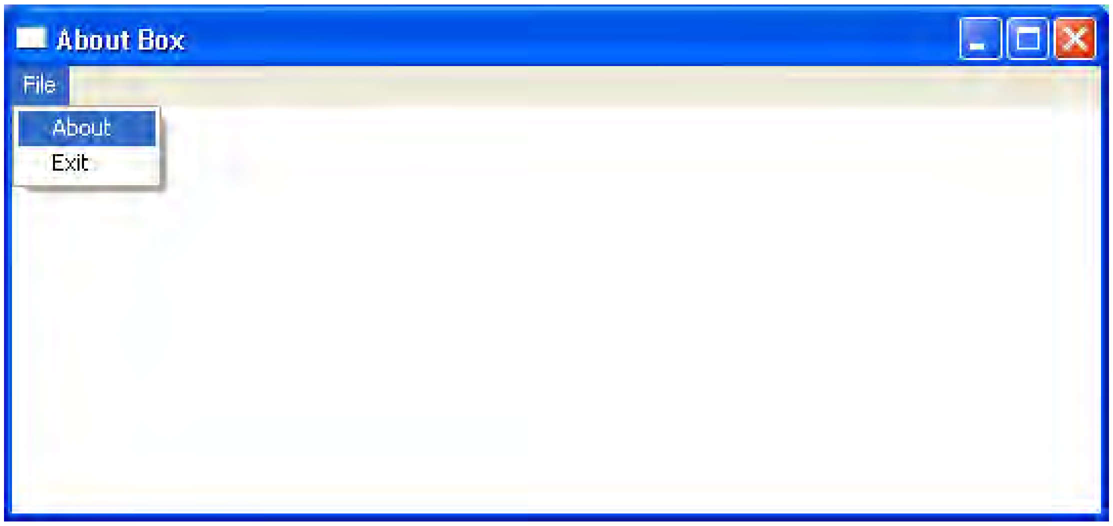


Figure 16.8: The user can launch the "About" dialog box by selecting the "About" menu item.

In the main window procedure, we execute the `DialogBox` function in response to the user selecting the "About" menu item like so:

```
// User selected About menu item, so show the modal dialog box.
case ID_FILE_ABOUT: // About menu item resource ID.
      DialogBox(
            ghAppInst, // Application instance.
            MAKEINTRESOURCE(IDD_ABOUTBOX), // Dialog resource ID.
            hWnd, // Parent window of dialog box.
            AboutDlgProc); // Ptr to dialog box window procedure.
      return 0;
```

To close a modal dialog box, we use the `EndDialog` function. This function is used only for modal dialog boxes—modeless dialog boxes are destroyed via another approach. By convention, a modal dialog box is typically ended when the user presses the OK or CANCEL buttons of the dialog box. In our "About" dialog box we only have an OK button. Now we need to address how button clicks are handled with dialog boxes. When a button is clicked, a `WM_COMMAND` message is sent to the parent window, which would be the dialog window. This is similar to what happens when a menu item is clicked. The lower 16-bits of the `wParam` parameter specify the resource ID of the button that was clicked. Consequently, in the dialog procedure, we can check if a particular button was clicked by handling the `WM_COMMAND` message and examining the lower 16-bits of the `wParam`:

```
case WM_COMMAND:
      // The low 16-bits of the wParam stores the resource
      // ID of the button control the user pressed.  So from
      // that information, we can determine which button was pressed
```

```
        switch(LOWORD(wParam))
        {
        // Did the user press the OK button (resource ID = IDOK)?
        // If so, close the dialog box.
        case IDOK:
              EndDialog(
                    hDlg, // Handle to dialog to end.
                    0);   // Return code--generally always zero.
              return true;
        }
        break;
```

In summary:

- Create and design the graphical layout of a dialog box with the Visual C++ Resource Editor.
- A dialog box is a kind of window and, as such, needs its own dialog procedure.
- A modal dialog can be displayed with the `DialogBox` function.
- A modal dialog can be ended with the `EndDialog` function.
- When a button control is pushed, it sends a `WM_COMMAND` message to the parent dialog's window procedure.

16.1.3 The About Box Sample

Now that we have created and designed a dialog resource, and have discussed the necessary background for showing and ending modal dialog boxes, we present the code for the "About" dialog box sample, as displayed in Figure 16.2. The concepts related to dialog boxes have been bolded.

Program 16.1: This program displays a window with a menu. The menu has an "About" menu item, which, when clicked, displays an about dialog box. See Figure 16.2 for an illustration.

```
#include <windows.h>
#include "resource.h" // For dialog and menu resource.

// Store handles to the main window and application
// instance globally.
HWND      ghMainWnd = 0;
HINSTANCE ghAppInst = 0;
HMENU     ghMenu    = 0;

// Define a window procedure for the dialog box:
// A dialog box is a kind of window, and as such we need to
// define a window procedure for it.
BOOL CALLBACK
AboutDlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
      switch( msg )
      {
      // A WM_INITDIALOG message is sent right before the
      // dialog is displayed.  This gives you a chance to
      // initialize any dialog related variables.  For an
      // About Box, we do not need to do any initialization
      // so we just return true.
      case WM_INITDIALOG:
            return true;

      // We learned in the last chapter that a WM_COMMAND
      // message is sent when the user selects a menu item
      // from the menu.  In addition to that, a WM_COMMAND
      // message is also sent when the user presses a
      // button control.
      case WM_COMMAND:
            // The low 16-bits of the wParam stores the resource
            // ID of the button control the user pressed.  So from
            // that information, we can determine which button was
            // pressed.
            switch(LOWORD(wParam))
            {
            // Did the user press the OK button (resource ID = IDOK)?
            // If so, close the dialog box.
            case IDOK:
                  EndDialog(
                        hDlg, // Handle to dialog to end.
                        0);   // Return code--generally always zero.
                  return true;
            }
            break;
      }
      return false;
}

// Step 1: Define and implement the window procedure.
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      switch( msg )
      {
      case WM_COMMAND:
            switch( LOWORD(wParam) )
            {
            case ID_FILE_EXIT:
                  DestroyWindow(ghMainWnd);
                  return 0;
            // User click the “About” menu item?
            case ID_FILE_ABOUT:
                  DialogBox(
                  ghAppInst, // Application instance.
                  MAKEINTRESOURCE(IDD_ABOUTBOX), // Dialog resource ID.
                  hWnd, // Parent window of dialog box.
                  AboutDlgProc); // Ptr to dialog box window procedure.
                  return 0;
            }
            return 0;
      // Handle destroy window message.
      case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
```

```
	// default window procedure.
	return DefWindowProc(hWnd, msg, wParam, lParam);
}

// WinMain: Entry point for a Windows application.
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
	// Save handle to application instance.
	ghAppInst = hInstance;

	// Step 2: Fill out a WNDCLASS instance.
	WNDCLASS wc;
	wc.style         = CS_HREDRAW | CS_VREDRAW;
	wc.lpfnWndProc   = WndProc;
	wc.cbClsExtra    = 0;
	wc.cbWndExtra    = 0;
	wc.hInstance     = ghAppInst;
	wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
	wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
	wc.hbrBackground = (HBRUSH)::GetStockObject(WHITE_BRUSH);
	wc.lpszMenuName  = 0;
	wc.lpszClassName = "MyWndClassName";

	// Step 3: Register the WNDCLASS instance with Windows.
	RegisterClass( &wc );

	// Step 4: Create the window, and save handle in globla
	// window handle variable ghMainWnd.
	ghMenu = LoadMenu(ghAppInst, MAKEINTRESOURCE(IDR_MENU1));
	ghMainWnd = ::CreateWindow("MyWndClassName", "About Box",
		WS_OVERLAPPEDWINDOW, 0, 0, 540, 380, 0, ghMenu,
		ghAppInst, 0);

	if(ghMainWnd == 0){
		::MessageBox(0, "CreateWindow - Failed", 0, 0);
		return false;
	}

	// Step 5: Show and update the window.
	ShowWindow(ghMainWnd, showCmd);
	UpdateWindow(ghMainWnd);

	// Step 6: Enter the message loop and don't quit until
	// a WM_QUIT message is received.
	MSG msg;
	ZeroMemory(&msg, sizeof(MSG));
	while( GetMessage(&msg, 0, 0, 0) )
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}

	// Return exit code back to operating system.
	return (int)msg.wParam;
}
```

16.2 Modeless Dialog Boxes; The Edit Control

A **modeless dialog box** is a dialog box that behaves like a normal window, but is still a dialog box. Unlike modal dialog boxes, the user can switch to other windows as desired. Therefore, the modeless dialog box acts like an independent freestanding window. In fact, you can make a modeless dialog box your primary application window. Take a look at the Calculator program—Figure 16.9—that ships with Windows and you will observe that it is a pure dialog application.

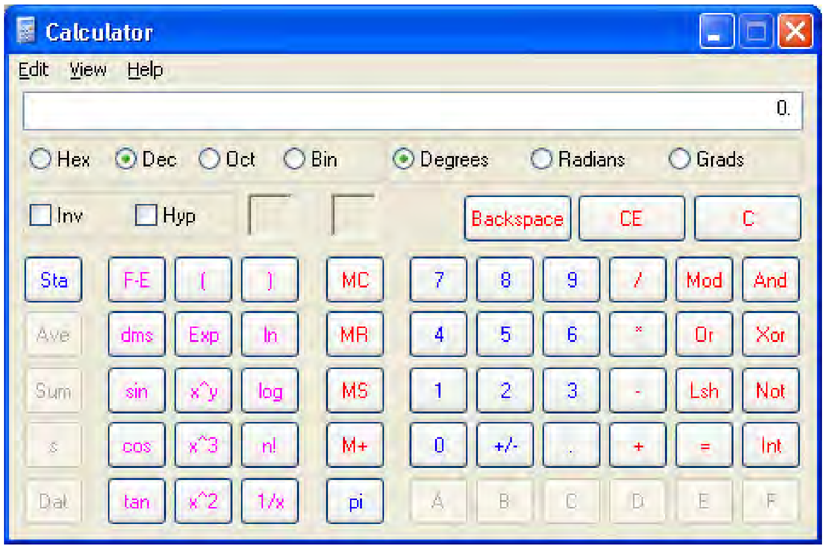

Calculator
Edit View Help
0.
Hex Dec Oct Bin Degrees Radians Grads
Inv Hyp
Backspace CE C
Sta F-E ( ) MC 7 8 9 / Mod And
Ave dms Exp ln MR 4 5 6 * Or Xor
Sum sin x^y log MS 1 2 3 - Lsh Not
s cos x^3 n! M+ 0 +/- . + = Int
Dat tan x^2 1/x pi A B C D E F

Figure 16.9: The Windows Calculator program is a pure dialog based application.

Typically, an application uses a modeless dialog box as a tool dialog. That is, the user wants the dialog to be floating around so that it can be accessed quickly and efficiently. Adobe Photoshop is a good example of a program that uses a floating tool dialog.

The rest of the programs we create in this chapter will be purely dialog based and use a modeless dialog box. The next sample program illustrates how to use an edit box control. But before we get to that, we must first learn how modeless dialog boxes are created and destroyed (recall it is different from modal dialog boxes).

16.2.1 Modeless Dialog Box Theory

First, as with modal dialog boxes, modeless dialog boxes are also a kind of window and as such, they too have a window procedure. The window procedure is implemented in the same way a modal dialog box procedure is, so we will not discuss it further.

Let us now examine how modeless dialog boxes are created. To create a modeless dialog box, we use the `CreateDialog` function instead of the `DialogBox` function:

```
HWND ghDlg = CreateDialog(
      hInstance,                   // Application instance.
      MAKEINTRESOURCE(IDD_MSGDLG), // Dialog resource ID.
      0,                           // Parent window--null for no parent.
      EditDlgProc);                // Dialog window procedure.
```

The parameters are similar to that of `DialogBox`. We note, however, that `CreateDialog` returns a window handle to the created dialog box window.

After a dialog is created, we need to show it with `ShowWindow`:

```
// Show the dialog.
ShowWindow(ghDlg, showCmd);
```

> **Note:** There is a dialog property you can set in the dialog resource editor called "Visible." If the property "Visible" is set to true then the dialog will be shown by default and a call to `ShowWindow` is unnecessary. Try it.

To destroy a modeless dialog, we use the `DestroyWindow` function instead of the `EndDialog` function:

```
// If the dialog was closed (user pressed 'X' button)
// then terminate the dialog.
case WM_CLOSE:
      DestroyWindow(hDlg);
      return true;
```

The final difference between a modeless dialog box and a modal dialog box has to do with the way messages are processed; we must do a little more work with a modeless dialog. In particular, we need to intercept messages aimed for the dialog box from the message loop and forward them to the dialog box procedure. This is done for us with modal dialog boxes. To do this interception we use the `IsDialogMessage` function. This function returns true if the message is indeed a dialog message, and if it is a dialog message it will also dispatch the message to the dialog's window procedure. Here is the rewritten message loop:

```
while( GetMessage( &msg, 0, 0, 0 ) )
{
      // Is the message a dialog message?  If so the function
      // IsDialogMessage will return true and then dispatch
      // the message to the dialog window procedure.
      // Otherwise, we process as the message as normal.
      if( ghDlg == 0 || !IsDialogMessage(ghDlg, &msg ) )
      {
            // Process message as normal.
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
}
```

Essentially, the line

```
IsDialogMessage(ghDlg, &msg )
```

can be read as "is the message stored in `msg` aimed for the dialog box referred to by `ghDlg`?"

Also note that if the dialog handle is null (`ghDlg == 0`, which means the dialog was destroyed or has not been created yet) then the 'if' statement will be true and the message will be processed normally.

16.2.2 The Edit Box Sample: Designing the Dialog Resource

We now need to design the dialog for our second sample program. This sample program will have an edit box, which allows the user to input some text. The program will also have a button. When the user presses the button a message box will be displayed. The message box will contain whatever text the user entered into the edit box. The goal of this program is to illustrate how text is extracted from an edit box control. Figure 16.10 shows a screenshot.

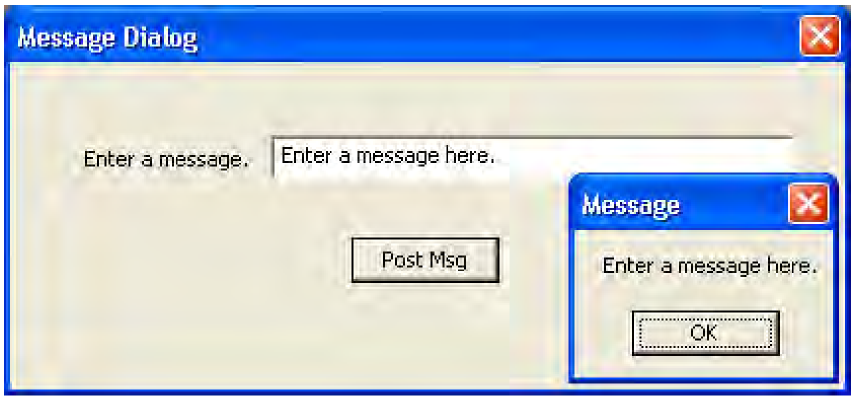


Figure 16.10: This program extracts the text entered in the edit box and displays it in a message box whenever the user presses the "Post Msg" button.

The first task is to design a dialog box in the resource editor as Figure 16.11 shows:

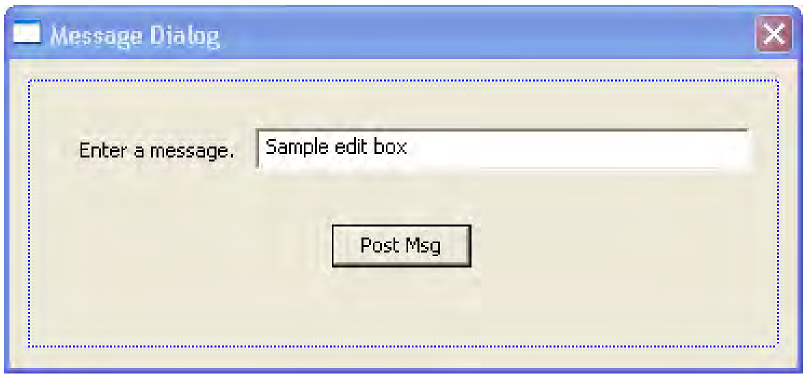


Figure 16.11: Designing the message dialog.

In our program, we have used the following resource IDs for the various controls.

- Dialog box -- IDD_MSGDLG
- Edit box -- IDC_MSGTEXT
- Static text -- IDC_STATIC
- Button -- IDB_MSG

Note that we keep the "System Menu" set to true in this example, so that the dialog has the 'X' close button in the upper right hand corner.

16.2.3 The Edit Box Sample

Now that we have created the dialog resource, let us examine some of the program logic we will use. One of the key ideas of this sample is extracting the text from the edit box control. In order to do this, we need a handle to the control so we can access it. To obtain a handle to a child window on a dialog, we can use the `GetDlgItem` function:

```
HWND hEditBox = GetDlgItem(hDlg, IDC_MSGTEXT);
```

The first parameter is a `HWND` to the dialog on which the control "lives", and the second parameter is the resource ID of the child control. This function then returns an `HWND` to that child control.

Given an `HWND` to the edit box control, the program can manually set the text in the edit box with the `SetWindowText` function:

```
// Set some default text in the edit box.
SetWindowText(hEditBox, "Enter a message here.");
```

The first parameter is a handle to the window in which we want to set some text, and the second parameter is the string we wish to set. Why would the program manually set text in an edit box when the user is supposed to input text there? There are a variety of reasons why a program might need to do this. For example, we might want to fill the edit box with some default text, as we do in Program 16.2. The reverse operation is to extract text from the message box. To do that we use the `GetWindowText` function:

```
// Text buffer to be filled with string user entered into edit control.
char msgText[256];

// Extract the text from the edit box.
GetWindowText(hEditBox, msgText, 256);
```

The first parameter is a handle to the window from which we want to extract the text (in this case, the edit box), the second parameter is a pointer to a buffer which will receive the text, and the third parameter is the maximum number of characters to read—we do not want to read more characters than can fit in our buffer. If the text stored in the edit box is longer than the maximum number of characters you specify to extract then the string is truncated.

Let us look at some of the specifics of Program 16.2. By default, we want the edit box to contain the string "Enter a message here." So, in the dialog procedure, in the `WM_INITDIALOG` message handler, we obtain a handle to that edit box, and also set the default text:

```
case WM_INITDIALOG:
      hEditBox = GetDlgItem(hDlg, IDC_MSGTEXT);

      // Set some default text in the edit box.
      SetWindowText(hEditBox, "Enter a message here.");
      return true;
```

`hEditBox` is declared at a higher scope and saved statically in the dialog procedure.

When the button "Post Msg" is pressed, we extract the text from the edit box (be it the default text or text which the user entered), and display it in a message box:

```
case WM_COMMAND:
      switch(LOWORD(wParam))
      {
      case IDB_MSG:
            // Extract the text from the edit box.
            GetWindowText(hEditBox, msgText, 256);

            // Now display the text in the edit box in
            // a message box.
            MessageBox(0, msgText, "Message", MB_OK);
            return true;
      }
      return true;
```

Finally, when the user presses the close button (the 'X' in the upper right hand corner), we destroy the dialog and exit the application:

```
case WM_CLOSE:
      DestroyWindow(hDlg);
      return true;
case WM_DESTROY:
      PostQuitMessage(0);
      return true;
```

We now present the edit box sample in its entirety, with key lines of code bolded:

Program 16.2: The Edit Box Control Sample. This program shows how to set/get text to and from an edit box control. It also illustrates how to create a modeless dialog box.

```
#include <windows.h>
#include <string>
#include "resource.h"
using namespace std;

// Dialog handle.
HWND ghDlg = 0;

// Dialog window procedure.
BOOL CALLBACK
EditDlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
      // Text buffer to be filled with string user entered
      // into edit control.
      char msgText[256];

      // Handle to the edit box control (ID = IDC_MSGTEXT).
      static HWND hEditBox = 0;

      switch( msg )
      {
      case WM_INITDIALOG:
            // Controls are child windows to the dialog they lie on.
            // In order to get and send information to and from a
            // control we will need a handle to it.  So save a handle
            // to the edit box control as the dialog is being
            // initialized.  Recall that we get a handle to a child
            // control on a dialog box with the GetDlgItem.
            hEditBox = GetDlgItem(hDlg, IDC_MSGTEXT);

            // Set some default text in the edit box.
            SetWindowText(hEditBox, "Enter a message here.");
            return true;

      case WM_COMMAND:
            switch(LOWORD(wParam))
            {
            case IDB_MSG:
                  // Extract the text from the edit box.
                  GetWindowText(hEditBox, msgText, 256);

                  // Now display the text in the edit box in
                  // a message box.
                  MessageBox(0, msgText, "Message", MB_OK);
                  return true;
            }
            return true;

      // If the dialog was closed (user pressed 'X' button)
      // then terminate the dialog.
      case WM_CLOSE:
            DestroyWindow(hDlg);
            return true;
      case WM_DESTROY:
            PostQuitMessage(0);
            return true;
      }

      return false;
}

int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Create the modeless dialog window.  This program is a pure
      // dialog window application, and the dialog window is the
      // "main" window.
      ghDlg = CreateDialog(
            hInstance, // Application instance.
            MAKEINTRESOURCE(IDD_MSGDLG), // Dialog resource ID.
            0,  // Parent window--null for no parent.
            EditDlgProc); // Dialog window procedure.

      // Show the dialog.
      ShowWindow(ghDlg, showCmd);

      // Enter the message loop.
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));

      while( GetMessage( &msg, 0, 0, 0 ) )
      {
            // Is the message a dialog message?  If so the function
            // IsDialogMessage will return true and then dispatch
            // the message to the dialog window procedure.
            // Otherwise, we process as the message as normal.
            if( ghDlg == 0 || !IsDialogMessage(ghDlg, &msg ) )
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
      }

      return (int)msg.wParam;
}
```

16.3 Radio Buttons

We now turn our attention to radio button controls. The program we will develop to illustrate radio buttons will allow the user to make a selection. After the user makes a selection and presses the OK button, the program will output a message box based on the user's selection. Consider Figure 6.12.

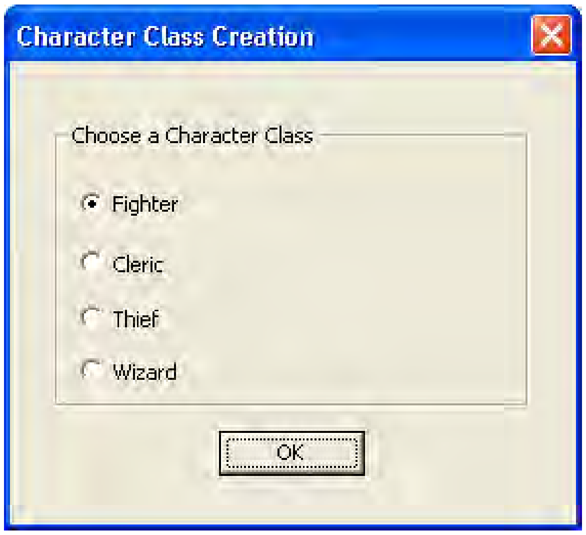


Figure 6.12: A screen shot of the radio button sample we develop in this section.

Radio buttons are used to provide users with a set of options, where one, and only one, option in the group must be selected at any time. Figure 6.12 shows how we might use radio buttons to implement a character class selection for a role playing game. Observe how we surround the radio buttons in a group control; a group control functions merely as an aesthetic tool to provide a visual grouping of related controls. Figure 16.1 also gives an example of radio buttons which allow the user to switch between windowed mode and fullscreen mode.

16.3.1 Designing the Radio Dialog Resource

Our first task in making the radio button sample program is to create a dialog resource and design it so that it looks like Figure 16.12. We use the following control resource names:

- Dialog box -- IDD_RADIODLG
- Group box -- IDC_STATIC
- Fighter radio button -- IDC_RADIO_FIGHTER
- Cleric radio button -- IDC_RADIO_CLERIC
- Thief radio button -- IDC_RADIO_THIEF
- Wizard radio button -- IDC_RADIO_WIZARD
- OK button -- IDOK

16.3.2 Implementing the Radio Button Sample

The key API function needed when implementing radio buttons is the `CheckRadioButton` function. This function updates the GUI of a group of radio buttons by selecting a new radio button and deselecting the previously selected radio button. Here is an example of how it would be called:

```
CheckRadioButton(hDlg,
      IDC_RADIO_FIGHTER, // First radio button in group
      IDC_RADIO_WIZARD,  // Last radio button in group
      IDC_RADIO_FIGHTER);// Button to select.
```

This function call would check the "Fighter" radio button and deselect any previously selected radio button. The key idea is that a group of radio button IDs must be consecutive for this function to work (the function assumes they are consecutive). If you create the radio buttons in the resource dialog one after another, they will be made consecutive. To understand why this is necessary, first observe how we specify the first radio button and last radio button ID to the `CheckRadioButton` function. If the radio buttons IDs are consecutive, the first and last ID essentially marks the range of the radio button IDs. Thus the function can iterate one-by-one over each ID and update its graphical appearance (selected/deselected).

Let us now examine the program implementation details. The first important task we do is to select a default radio button. We do this when the dialog box is being initialized:

```
case WM_INITDIALOG:
      // Select the default radio button.
      // Note: Assumes the radio buttons were created sequentially.
      CheckRadioButton(hDlg,
            IDC_RADIO_FIGHTER, // First radio button in group
            IDC_RADIO_WIZARD,  // Last radio button in group
            IDC_RADIO_FIGHTER);// Button to select.
      return true;
```

We will also want to maintain the currently selected radio button, so we include a static variable that maintains that value:

```
static int classSelection = IDC_RADIO_FIGHTER;
```

We initialize it to the default value. When the user selects a new radio button, we update this value. When a radio button is selected, Windows sends the dialog procedure a `WM_COMMAND` message, just as it does for a regular button control. The lower 16-bits of the `wParam` stores the resource ID of the button selected:

```
// Was *any* radio button selected?
case IDC_RADIO_FIGHTER:
case IDC_RADIO_CLERIC:
case IDC_RADIO_THIEF:
case IDC_RADIO_WIZARD:
      // Yes, one of the radio buttons in the group was selected,
      // so select the new one (stored in LOWORD(wParam)) and
      // deselect the other to update the radio button GUI.

      // Note: Assumes the radio buttons were created sequentially.
      CheckRadioButton(hDlg,
            IDC_RADIO_FIGHTER, // First radio button in group
            IDC_RADIO_WIZARD,  // Last radio button in group
            LOWORD(wParam));   // Button to select.

      // Save currently selected radio button.
      classSelection = LOWORD(wParam);
      return true;
```

Observe that we do not test for each individual radio button selection; rather we test for *any* selection. `LOWORD(wParam)` will give us the actual button pressed, which we pass on to the `CheckRadioButton` function, which will handle selecting the correct button and deselecting the others.

Finally, the last key idea of this program is the message box. More specifically, when the user presses the OK button, we want the program to display a message specific to the character (radio button) selected. This is easy enough since we have saved the current radio button selected in the `classSelection` variable. Thus we could do a simple 'if' statement to output an appropriate message based on the selection. However, instead we add the following variable to the dialog procedure that contains our messages:

```
string classNames[4] =
{
      "You selected the Fighter.",
      "You selected the Cleric.",
      "You selected the Thief.",
      "You selected the Wizard."
};
```

We then output the message like so:

```
case IDOK:

      // Now display the class the user selected in a message box.
      MessageBox(
           0,
           classNames[classSelection-IDC_RADIO_FIGHTER].c_str(),
           "Message",
           MB_OK);
      return true;
```

The interesting line is:

```
classNames[classSelection-IDC_RADIO_FIGHTER].c_str()
```

The best way to explain this is to look at the actual numeric values of the resource IDs. If you open resource.h, you may see something similar to the following:

```
#define IDC_RADIO_FIGHTER               1001
#define IDC_RADIO_CLERIC                1002
#define IDC_RADIO_THIEF                 1003
#define IDC_RADIO_WIZARD                1004
```

Again, the resource identifiers are just unique ways of identifying the resource items.

Our array `classNames` is a four-element array, so we cannot index into it with values such as 1001, 1002, 1003, or 1004. What we do is subtract the first ID in the group. The first resource in the group is `IDC_RADIO_FIGHTER`, which is actually the number 1001. So this would give:

```
IDC_RADIO_FIGHTER - IDC_RADIO_FIGHTER = 1001 – 1001 = 0
IDC_RADIO_CLERIC  - IDC_RADIO_FIGHTER = 1002 – 1001 = 1
IDC_RADIO_THIEF   - IDC_RADIO_FIGHTER = 1003 – 1001 = 2
IDC_RADIO_WIZARD  - IDC_RADIO_FIGHTER = 1004 – 1001 = 3
```

The result of these subtractions gives us the corresponding text index for `classNames`! So for example, if `classSelection == IDC_RADIO_THIEF`, we would have:

```
IDC_RADIO_THIEF - IDC_RADIO_FIGHTER = 1003 – 1001 = 2
```

and

```
classNames[2] == "You selected the Thief."
```

This is the exact string we want to use in this example.

Program 16.3 shows the complete code listing for the radio button program.

Program 16.3:

```
#include <windows.h>
#include <string>
#include "resource.h"
using namespace std;

// Dialog handle.
HWND ghDlg = 0;

// Dialog window procedure.
BOOL CALLBACK
MsgDlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
      // Default to fighter.
      static int classSelection = IDC_RADIO_FIGHTER;
      string classNames[4] =
      {
            "You selected the Fighter.",
            "You selected the Cleric.",
            "You selected the Thief.",
            "You selected the Wizard."
      };
      switch( msg )
      {
      case WM_INITDIALOG:
            // Select the default radio button.
            // Note: Assumes the radio buttons were created
            // sequentially.
            CheckRadioButton(hDlg,
                  IDC_RADIO_FIGHTER, // First radio button in group
                  IDC_RADIO_WIZARD,  // Last radio button in group
                  IDC_RADIO_FIGHTER);// Button to select.

            return true;
      case WM_COMMAND:
            switch(LOWORD(wParam))
            {
            // Was *any* radio button selected?
            case IDC_RADIO_FIGHTER:
            case IDC_RADIO_CLERIC:
            case IDC_RADIO_THIEF:
            case IDC_RADIO_WIZARD:

            // Yes, one of the radio buttons in the group was selected,
            // so select the new one (stored in LOWORD(wParam)) and
            // deselect the other to update the radio button GUI.

            // Note: Assumes the radio buttons were created
            // sequentially.
            CheckRadioButton(hDlg,
                  IDC_RADIO_FIGHTER,// First radio button in group
                  IDC_RADIO_WIZARD, // Last radio button in group
                  LOWORD(wParam));  // Button to select.

             // Save currently selected radio button.
            classSelection = LOWORD(wParam);
            return true;
            case IDOK:

                  // Now display the class the user selected in
                  //  a message box.
                  MessageBox(
                  0,
                  classNames[classSelection-IDC_RADIO_FIGHTER].c_str(),
                  "Message",
                   MB_OK);
                  return true;
            }
            return true;
      case WM_CLOSE:
            DestroyWindow(hDlg);
            return true;
      case WM_DESTROY:
            PostQuitMessage(0);
            return true;
      }
      return false;
}
```

```
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Create the modeless dialog window.
      ghDlg = CreateDialog(
            hInstance, // Application instance.
            MAKEINTRESOURCE(IDD_RADIODLG), // Dialog resource ID.
            0,  // Parent window--null for no parent.
            MsgDlgProc); // Dialog window procedure.

      // Show the dialog.
      ShowWindow(ghDlg, showCmd);

      // Enter the message loop.
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));
      while( GetMessage( &msg, 0, 0, 0 ) )
      {
            // Is the message a dialog message?  If so the function
            // IsDialogMessage will return true and then dispatch
            // the message to the dialog window procedure.
            // Otherwise, we process as the message as normal.
            if( ghDlg == 0 || !IsDialogMessage(ghDlg, &msg ) )
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
      }
      return (int)msg.wParam;
}
```

16.4 Combo Boxes

The last program we will write in this chapter illustrates the combo box control. A combo box can be described as a drop down list box. The program we will implement has a combo box, edit box, and button control. The user can enter text in the edit box control. When the user presses the button, the text in the edit box will be added to the combo box. In addition, when the user selects an item from the combo box, a message box of that item's text is shown.

Figure 16.13 shows a screenshot of the program we will develop:

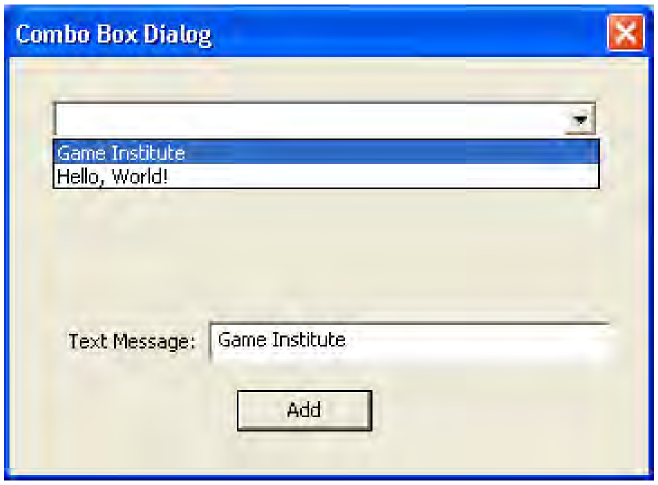


Figure 16.13: A screenshot of the combo box application. The user can add text to the combo box by writing some text in the edit control and pressing the "Add" button.

16.4.1 Designing the Combo Box Dialog Resource

Our first task in making the combo box sample program is to create a dialog resource and design it such that it looks like Figure 16.13. We use the following control resource names:

- Dialog box -- IDD_COMBODLG
- Static text -- IDC_STATIC
- Add button -- IDC_ADDBUTTON
- Edit box -- IDC_EDIT_MSG
- Combo box -- IDC_COMBOBOX

16.4.2 Implementing the Combo Box Sample

The first key implementation detail to note is that we need to get handles to the window controls on the dialog box. We obtain these handles as the dialog box is being initialized:

```
hComboBox  = GetDlgItem(hDlg, IDC_COMBOBOX);
hEditBox   = GetDlgItem(hDlg, IDC_EDIT_MSG);
hAddButton = GetDlgItem(hDlg, IDC_ADDBUTTON);
```

These variables are declared statically in the dialog window procedure like so:

```
// Handles to the combo box controls.
static HWND hComboBox  = 0;
static HWND hEditBox   = 0;
static HWND hAddButton = 0;
```

The second key implementation concept is how to add the text from the edit box to the combo box. We know that we can extract the edit box text using the `GetWindowText` function, but how do we add an item to the combo box? To add an item to the combo box we need to send it a message. (What is interesting about the combo box is that Windows creates a window procedure for it internally.) In particular, we need to send a `CB_ADDSTRING` message to the combo box, where the CB stands for combo box. We can do this with the `SendMessage` function:

```
SendMessage(
    hComboBox,        // Handle to window to send message to.
    CB_ADDSTRING,     // Add string message.
    0,                // No WPARAM for this message.
    (LPARAM)msgText);// The LPARAM is a pointer to the string to add.
```

`msgText` is some array of `char`s representing the string to add.

Recall that we stated that when the user selects an item from the combo box we will display a message box showing the string item that was selected from the combo box. We can detect when the user interacts with a combo box item because a `WM_COMMAND` is sent to the dialog window procedure. The `HIWORD` of the `wParam` parameter gives the notification code, which explains specifically how the user interacted with the combo box. If the notification code is `CBN_SELENDOK` then the user selected an item from the combo box:

```
case WM_COMMAND:
      switch(HIWORD(wParam))
      {
      // User selected a combo box item.
      case CBN_SELENDOK:
```

> **Note:** Some other possible notification codes for a combo box are:
>
> CBN_CLOSEUP: Sent when the combo box drop down list is closed.
> CBN_DBLCLK: Sent when an item in the combo box is double-clicked.
> CBN_DROPDOWN: Sent when the combo box drop down list is opened.
> CBN_KILLFOCUS: Sent when the combo box loses focus.
> CBN_SETFOCUS: Sent when the combo box gains focus.
>
> All these special messages allow you to execute code when certain actions happen to the combo box. There are more—see the Win32 documentation for full details.

To display a message box showing the string item that was selected from the combo box, we need to send the combo box two messages. In the first message, we need to ask for the index (the strings added to a combo box are stored in an array-like structure) of the string that was selected. We can do that like so:

```
index = SendMessage(hComboBox, CB_GETCURSEL, 0, 0);
```

Where `CB_GETCURSEL` may be read as: “get the index of the currently selected item.” Given the index of the currently selected item in the combo box, we can get the actual string at that index by sending another message `CB_GETLBTEXT`:

```
char msgText[256];
SendMessage(hComboBox, CB_GETLBTEXT, (WPARAM)index, (LPARAM)msgText);
```

The `WPARAM` is the index of the string to get, and the `LPARAM` returns a pointer to the string.

Program 16.4 shows the complete code listing for the combo box program.

Program 16.4: The combo box program.

```
#include <windows.h>
#include <string>
#include "resource.h"
using namespace std;

// Dialog handle.
HWND ghDlg = 0;

// Dialog window procedure.
BOOL CALLBACK
MsgDlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
      // Text buffer to be filled with string user entered
      // into edit control.
      char msgText[256];

      // Handles to the combo box controls.
      static HWND hComboBox  = 0;
      static HWND hEditBox   = 0;
      static HWND hAddButton = 0;

      int index = 0;

      switch( msg )
      {
      case WM_INITDIALOG:
            // Controls are child windows to the dialog they lie on.
            // In order to get and send information to and from a
            // control we will need a handle to it.  So save a handle
            // to the controls as the dialog is being initialized.
            // Recall that we get a handle to a child control on a
            // dialog box with the GetDlgItem.
            hComboBox  = GetDlgItem(hDlg, IDC_COMBOBOX);
            hEditBox   = GetDlgItem(hDlg, IDC_EDIT_MSG);
            hAddButton = GetDlgItem(hDlg, IDC_ADDBUTTON);

            return true;
      case WM_COMMAND:
            switch(HIWORD(wParam))
            {
            // User selected a combo box item.
            case CBN_SELENDOK:
                  index = SendMessage(hComboBox, CB_GETCURSEL, 0, 0);
                  SendMessage(hComboBox, CB_GETLBTEXT, (WPARAM)index,
                             (LPARAM)msgText);
                  MessageBox(0, msgText, "Combo Message", MB_OK);
                  return true;
            }
            switch(LOWORD(wParam))
            {

            // User pressed the "Add" button.
            case IDC_ADDBUTTON:
                  // Get the text from the edit box.
                  GetWindowText(hEditBox, msgText, 256);

                  // Add the text to the combo box only if the
                  // user entered a string that is greater than zero.
                  if( strlen(msgText) > 0 )
                        SendMessage(
                            hComboBox,
                            CB_ADDSTRING,
                            0,
                            (LPARAM)msgText);

                  return true;
            }

            return true;

      case WM_CLOSE:
            DestroyWindow(hDlg);
            return true;

      case WM_DESTROY:
            PostQuitMessage(0);
            return true;
      }

      return false;
}

int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      // Create the modeless dialog window.
      ghDlg = CreateDialog(
            hInstance, // Application instance.
            MAKEINTRESOURCE(IDD_COMBODLG), // Dialog resource ID.
            0,  // Parent window--null for no parent.
            MsgDlgProc); // Dialog window procedure.

      // Show the dialog.
      ShowWindow(ghDlg, showCmd);

      // Enter the message loop.
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));

      while( GetMessage( &msg, 0, 0, 0 ) )
      {
            // Is the message a dialog message?  If so the function
            // IsDialogMessage will return true and then dispatch
            // the message to the dialog window procedure.
            // Otherwise, we process as the message as normal.
            if( ghDlg == 0 || !IsDialogMessage(ghDlg, &msg ) )
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
      }
      return (int)msg.wParam;
}
```

16.5 Summary

1. A dialog box is a type of window which is commonly used to display a variety of controls the user can set and configure.

2. A modal dialog box is a dialog box that does not close until the user has made some sort of selection (i.e., the user cannot switch to another window until the modal dialog box is ended). Typically, an application will use a modal dialog box when it needs immediate input from the user, and cannot continue until it receives that input.

3. A modeless dialog box is a dialog box that behaves like a normal window, but is still a dialog box. Unlike modal dialog boxes, the user can switch to other windows as desired. Therefore, the modeless dialog box acts like an independent window. In fact, you can make a modeless dialog box your primary application window. Typically, an application uses a modeless dialog box as a tool dialog. That is, the user wants the dialog to be floating around so that it can be accessed quickly and efficiently. Adobe Photoshop is a good example of a program that uses a floating tool dialog.

4. We create and design dialog boxes in the Visual C++ Resource Editor. Each control, which is not static, should be given a resource ID so that the application can refer to that control in code. Remember to include "resource.h" in your source code. Also note that you can modify various properties of the dialog and its controls through the properties window.

5. We can create and show a modal dialog box with the `DialogBox` function and close a modal dialog box with the `EndDialog` function. We can create a modeless dialog box with the `CreateDialog` function and show it with `ShowWindow`; we can close a modeless dialog box with the `DestroyWindow` function.

6. We can set the text in an edit box control with the `SetWindowText` function, and we can extract the text from an edit box control with the `GetWindowText` function.

7. Radio buttons are used to provide users with a set of options, where one, and only one, option in the group must be selected at any time. A radio button sends a `WM_COMMAND` message to its parent dialog window procedure when it is clicked. The lower 16-bits of the `wParam` contains the resource ID of the button that was selected. We can update the graphics of a group of radio buttons with the `CheckRadioButton` function; this function selects a specified radio button and deselects the previously selected button.

8. A combo box can be described as a drop down list box. What is interesting about the combo box is that Windows creates a window procedure for it internally. Consequently, we communicate with a combo box by sending it messages. To add an item to the combo box we need to send it a `CB_ADDSTRING` message. To get the index of an item in the combo box the user selects we need to send the combo box a `CB_GETCURSEL` message. Given the index of the currently selected item in the combo box, we can get the actual string at that index by sending a `GETLBTEXT` message.

16.6 Exercises

The following exercises are research oriented exercises where you will need to spend some time looking up the solution in the Win32 documentation or on the Internet. An online resource of the Win32 documentation is available at www.msdn.microsoft.com.

16.6.1 List Box

Rewrite Program 16.4, but instead of using a combo box, use a list box, which is essentially the same thing, but a list box is not "drop down;" rather, it is always "expanded." (Hint: List box messages begin with LB; for example, `LB_ADDSTRING`.)

16.6.2 Checkbox Controls

Implement a program that displays the modeless dialog box in Figure 16.14:

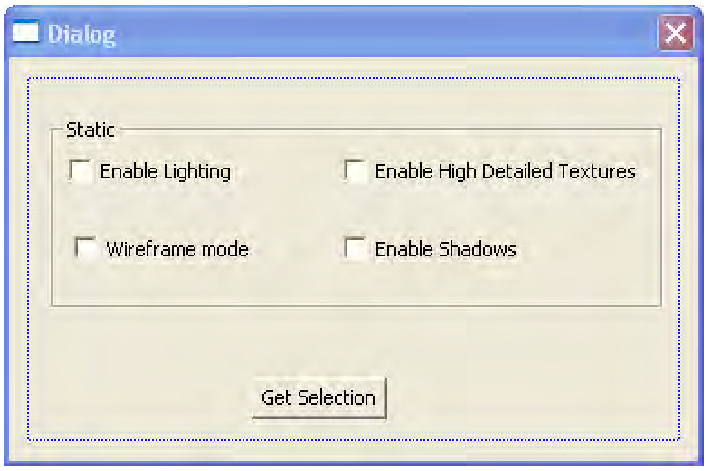
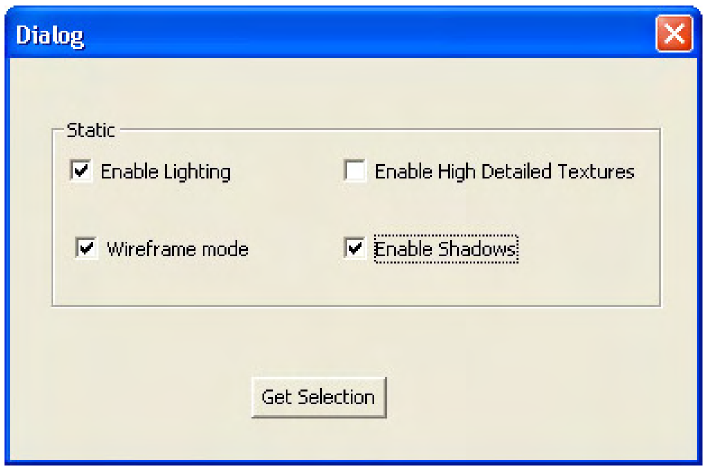


Figure 16.14: The dialog layout for the checkbox exercise.

This might be how an options screen looks for a game where you can customize the graphics by enabling and disabling features with checkbox controls. When the user presses the "Get Selection" button, a message box should be displayed that corresponds to the user selections. For example, if the boxes are checked like so:



[✓] Enable Lighting
[ ] Enable High Detailed Textures
[✓] Wireframe mode
[✓] Enable Shadows

Figure 16.15: The checkbox dialog after the user checked some items.

Then the message box should output the message:

"Lighting on, wireframe mode on, detailed textures off, shadows on."

<u>Hints</u>:

- Investigate the `BM_SETCHECK` message and the `BM_GETCHECK` messages.

16.6.3 File Save and Open Dialogs

No doubt you are familiar with the common file save and open dialogs found in Windows. Figure 16.16 shows an example of the common save dialog from Microsoft Word.

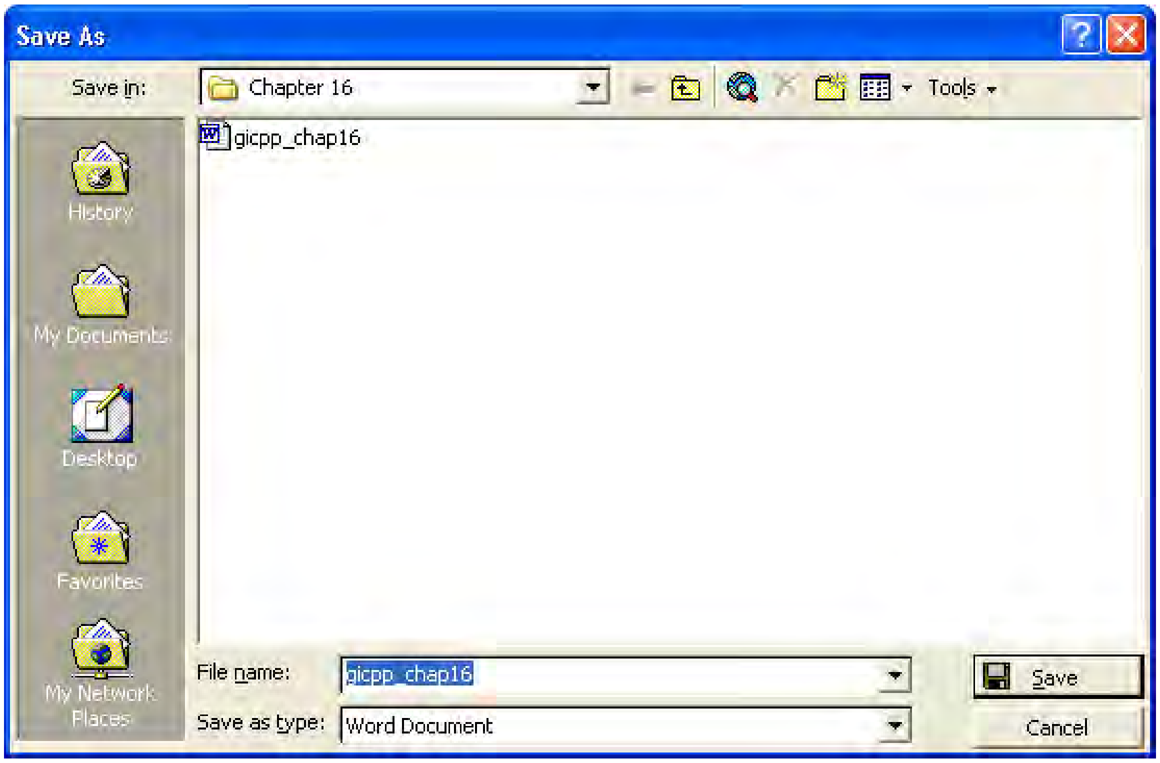


Figure 16.16: The standard Win32 Save Dialog.

You will notice that Visual C++ .NET uses the same common save and open dialogs as well. Most likely, other Windows programs you use also employ these same common dialogs.

The reason why many programs use these common dialogs is because they are prewritten by the Win32 API, and therefore, are straightforward to include, and also because they provide a standard user interface among Windows applications. A standard user interface is a good thing because users experienced with other Windows programs can quickly become experienced with your program if you provide a standard interface.

Your assignment for this exercise is to create a dialog application like that in Figure 16.17.

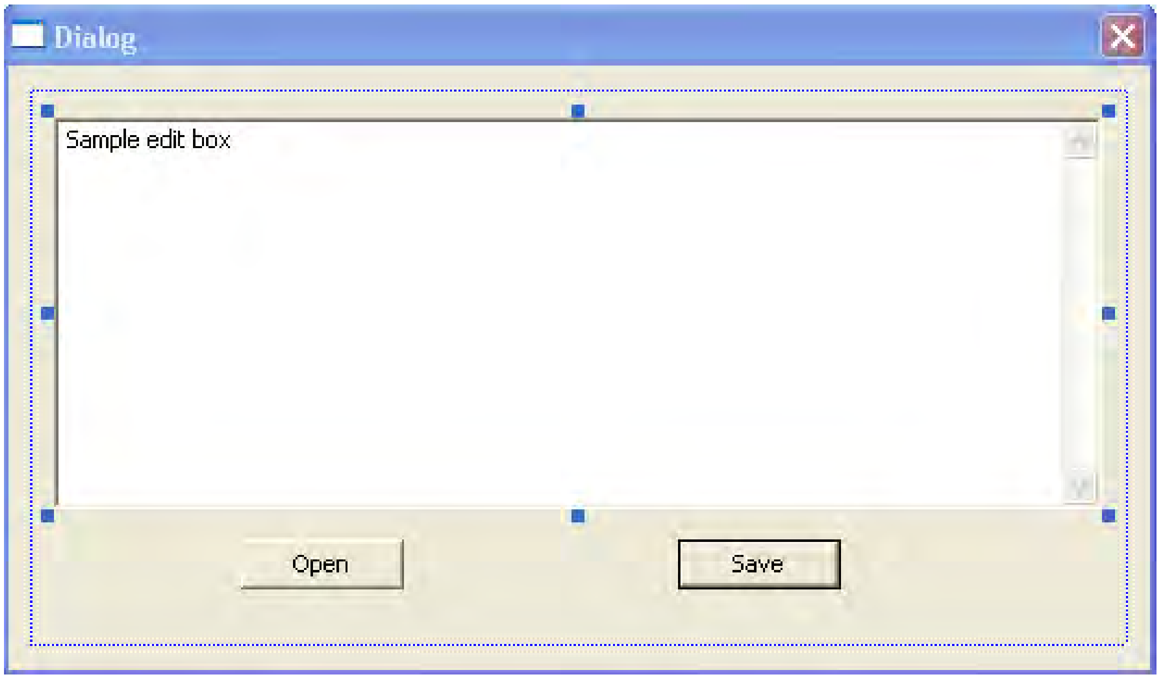


Figure 16.17: The dialog box you are to design for the File Save and File Open exercise.

Note that the “white rectangle” is an edit box—you can adjust the size of the edit box. In this program the user can enter text into the edit box. Then when the user presses the “Save” button, the common save dialog box should open. After the user has specified a filename in which to save the work, the program should proceed to save the text entered in the edit box to the file (using file output mechanisms: see Chapter 8 in Module I).

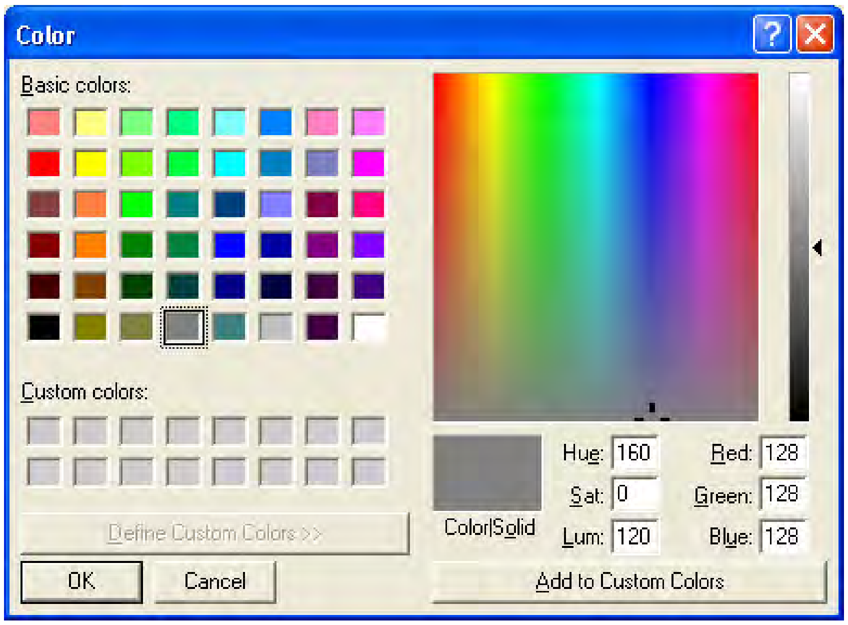
In addition to the save feature, your program should also be able to load text files. When the user presses the “Open” button, the common open dialog box should appear. After the user has specified the text filename in which to load, the program should proceed to load the text file (using file input mechanisms: see Chapter 8 in Module I), and copy the loaded text into the edit box. In this way, you will have a simple word processor that can save and load text files to and from the edit box.

<u>Hints</u>:

- You will want to modify some of the edit box’s properties. In particular, you will want to set “Multiline” to true, so that the text will wrap to the next line. You will also want to add a vertical scroll bar in case the text exceeds the visible viewing rectangle; you can do this by setting “Vertical Scroll” to true. Finally, you will also want to set “Auto HScroll” to false so that the text will not scroll horizontally—we want it to drop to the next line.
- You will need to include the common dialog header file (commdlg.h).
- Look up the `OPENFILENAME` structure.
- Look up the `GetOpenFileName` and `GetSaveFileName` functions.

16.6.4 Color Dialog

In addition to the standard file save and open dialogs, Windows also provides a standard color dialog (Figure 16.18).



Hue: 160
Sat: 0
Lum: 120
Red: 128
Green: 128
Blue: 128

Figure 16.18: The standard color dialog.

Write a program that displays the standard color dialog. Your program does not need to do anything with the dialog—it only needs to display the dialog.

Hints:

- You will need to include the common dialog header file (commdlg.h).
- Look up the `CHOOSECOLOR` structure and the `ChooseColor` function.

Chapter 17

Timing, Animation and Sprites

Introduction

We now have enough C++ and Win32 programming background to begin work on a 2D game. However, in order to accomplish such a task, we have one more stepping stone to cross; namely, we need to discuss some techniques related to animation and 2D graphics. Those issues are timing, double buffering, and sprites.

Chapter Objectives

- Learn how to use the Windows multimedia timer functions, so that we can keep track of how much time has elapsed per frame, for smooth and consistent animation.

- Discover how to do basic 2D computer animation by moving graphical objects over small increments as time passes.

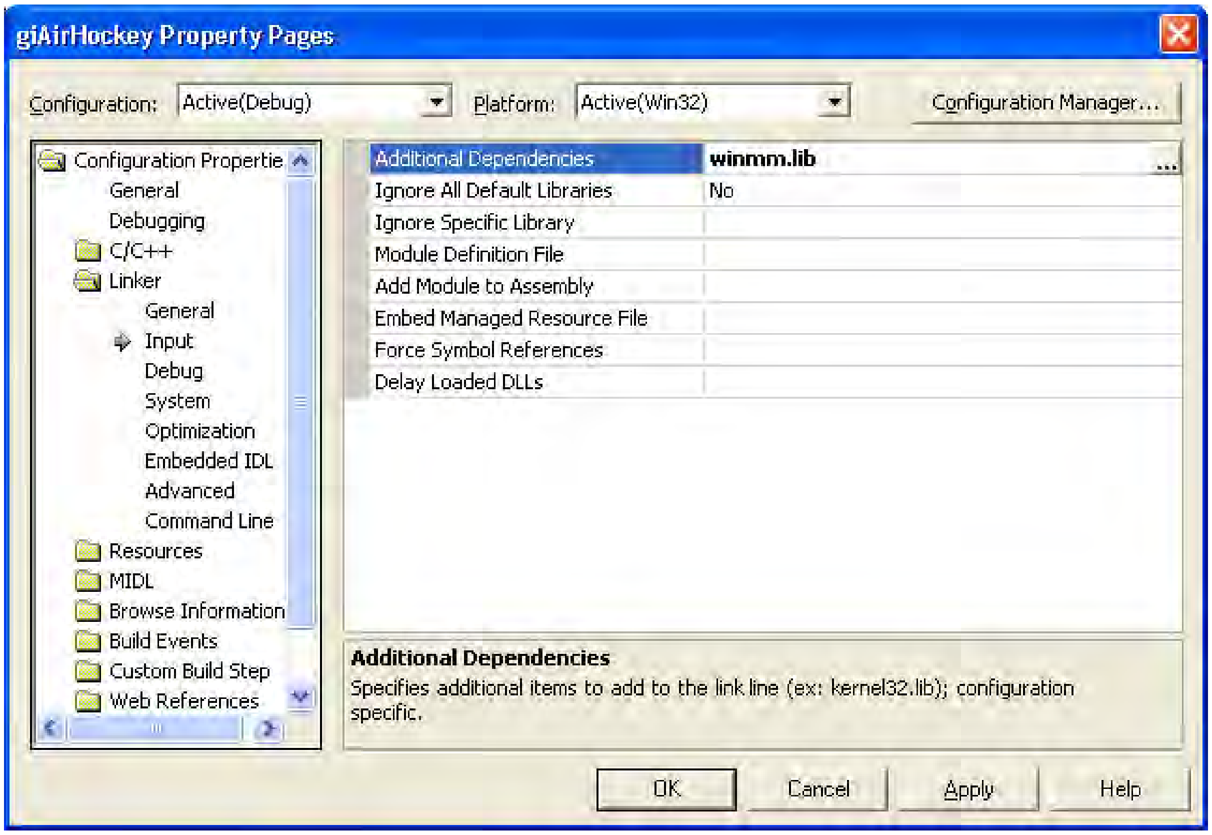
- Understand the technique of double buffering and how it works to avoid flicker.

- Learn how to draw complex 2D image bitmaps, which are not rectangular, via sprites using the GDI raster operations.

17.1 Timing and Frames Per Second

17.1.1 The Windows Multimedia Timer Functions

The Windows media library (#include <mmsystem.h> and link winmm.lib) provides a timer function called `timeGetTime`. This function returns the number of *milliseconds* that has elapsed since Windows started. In order to use this function, you will need to link winmm.lib, which is not linked by default. To do this in VC++ .NET, go to the "Solution Explorer," right click on your project name and select "Properties" from the popup menu. On the "Property Pages" dialog, select *Linker->Input*. Next, enter in "winmm.lib" in the "Additional Dependencies" field, as Figure 17.1 shows. Finally, press the "Apply" button and then the "OK" button.


giAirHockey Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

Additional Dependencies	winmm.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	

Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

Figure 17.1: Linking the winmm.lib file.

Often we will want the time in seconds, and not milliseconds. We can convert a time t_{ms} ms in milliseconds to a time t_s s in seconds by multiplying t_{ms} ms by the ratio 1 s / 1000 ms (there is one second per one-thousand milliseconds), as follows:

$$t_{ms}\text{ ms}\cdot\frac{1\text{ s}}{1000\text{ ms}}=\frac{t_{ms}\cdot 1\text{ s}}{1000}=t_s\text{ s}$$

Observe how the millisecond units (ms) cancel and we are left with only units of seconds. Also note that 1 s / 1000 ms = 1, and therefore, we can multiply any number by it (that is, 1) without actually changing it—we are changing units, not the physical meaning.

Recall that `timeGetTime` returns the number of *milliseconds* that has elapsed since Windows started. This number in and of itself is not particularly interesting. What is interesting is that if we call this function at time t_0, and then again at a later time t_1, we can compute the elapsed time Δt (change in time) as follows:

$$\Delta t = t_1 - t_0$$

We can use the result to determine how much time has passed between two successive calls to `timeGetTime`.

For measuring very short time intervals (such as the time elapsed between frames) at a high resolution, the `timeGetTime` function may not be accurate enough. Instead, you should use the more precise **performance timer.** Using the performance timer requires a small additional step; namely, we must first obtain its frequency (number of counts per second) so that we can convert the time units (i.e., "counts") measured by the performance counter to seconds. The frequency is obtained with the function `QueryPerformanceFrequency`:

```
// Get the performance timer frequency.
__int64 cntsPerSec = 0;
bool perfExists = QueryPerformanceFrequency((LARGE_INTEGER*)&cntsPerSec)!=0;
if( !perfExists )
{
      MessageBox(0, "Performance timer does not exist!", 0, 0);
      return 0;
}
```

Notice that this function returns the result into a 64-bit integer value. It also returns true if the performance timer is available (it is on all modern CPUs), and false if not.

To obtain a scaling factor that will convert performance counter units to seconds, we just divide 1 by the frequency. This gives us seconds per count.

```
// Get a scaling factor which will convert the timer units to seconds.
double timeScale = 1.0 / (double)cntsPerSec;
```

Then to get the current time, we use the `QueryPerformanceCounter` function:

```
__int64 time = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&time);
```

To convert the value to seconds, we multiply by the scaling factor which is in units of seconds per count. This has the effect of canceling out the performance counter units, leaving us with units in seconds. The idea is the same as when we cancelled out the milliseconds unit with the `timeGetTime` function:

```
double timeInSeconds = (double)(time) * timeScale;
```

Note that you still need to include <mmsystem.h> and link with winmm.lib to use the performance timer.

17.1.2 Computing the Time Elapsed Per Frame

One of the applications for which we will use timers is to compute the elapsed time between **frames.** A frame is one complete image in an animation sequence. For example, a game running at sixty frames per second is displaying sixty slightly different images per second. The rapid successive display of slightly different images over time gives the illusion of a smooth and continuous animation. As an aside, in practice, an animation running at approximately thirty frames per second is enough to fool the eye into seeing a smooth and continuous animation.

How do we compute the elapsed time between frames? If we saved the time of the last frame t_{i-1} using `timeGetTime`, and we get the time of the current frame t_i using `timeGetTime`, it follows that the time difference between the current frame i and the last frame $i-1$ is given by:

$$\Delta t = t_i - t_{i-1}$$

The real issue is how to do this in code. First of all, we are going to modify our message loop into a more "game friendly" version. We do this for two reasons: 1) games do not typically interact with Windows as much as other applications and therefore are not as dependent upon the Win32 messaging system and 2) this will prepare you for how the message loop will be written in other Game Institute courses, and other game programming related literature.

Essentially, our new message loop can be summarized in the following words: If there is a Windows message that needs to be processed, then process it; otherwise, execute our game code, which will typically be updating and drawing a frame. In order to implement this behavior, we need a function to determine if there is a message that needs to be processed. The Win32 API provides such a function called `PeekMessage`, which returns true if there is a message that needs to be processed; otherwise, it returns false. Thus our new message loop looks like this:

```
while(msg.message != WM_QUIT)
{
      // IF there is a Windows message then process it.
      if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
      // ELSE, do game stuff.
      else
      {
           ...
      }
}
```

Note that if there is a message, then `PeekMessage` also extracts the message into the first argument `msg`, just as `GetMessage` does. Also, the last parameter `PM_REMOVE` simply indicates that the message should be removed from the application message queue once the message is processed.

Our revised message loop is often termed the **game loop**, since it is the loop where we process most of our game related happenings. Note that in this course we will still use some Windows messages, but as you move on to study DirectX or OpenGL, you will begin to phase out more and more Windows messages. Also, we will commonly call each game loop cycle a **frame** (not to be confused with an image frame), because it is common to draw a frame (image) for each game loop cycle and drawing is generally the task that takes the most time to complete. Thus there is a correspondence between a game loop cycle and the image frame being drawn during that cycle.

We return to the question about implementing the time elapsed per frame $\Delta t = t_i - t_{i-1}$. Essentially, for each frame we get the time t_i, and this time will then become the last time for the next frame. Here is the game loop with the Δt calculations added. Note that we represent Δt with the variable `deltaTime`. In addition, the variable `currTime` corresponds to t_i, and `lastTime` corresponds to t_{i-1}.

```
// Get the current time.
float lastTime = (float)timeGetTime();

while(msg.message != WM_QUIT)
{
      // IF there is a Windows message then process it.
      if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
      // ELSE, do game stuff.
      else
      {
            // Get the time now.
            float currTime  = (float)timeGetTime();

            // Compute the differences in time from the last
            // time we checked.  Since the last time we checked
            // was the previous loop iteration, this difference
            // gives us the time between loop iterations...
            // or, I.e., the time between frames.
            float deltaTime = (currTime - lastTime)*0.001f;

            //
            // [...] Do Work this Frame (I.e., update and draw frame)
            //

            // We are at the end of the loop iteration, so
            // prepare for the next loop iteration by making
            // the "current time" the "last time."
            lastTime = currTime;
      }
}
```

Observe that for the very first frame, there is no last frame, and so we initialize the last time variable outside the message loop, before we get the time of the first frame:

```
float lastTime = (float)timeGetTime();
```

17.1.3 Computing the Frames Per Second

We now present a function that computes the frames per second:

```
//=========================================================
// Name: DrawFramesPerSecond
// Desc: This function is called every frame and updates
//       the frame per second display in the main window
//       caption.
//=========================================================
void DrawFramesPerSecond(float deltaTime)
{
      // Make static so the variables persist even after
      // the function returns.
      static int   frameCnt    = 0;
      static float timeElapsed = 0.0f;
      static char  buffer[256];

      // Function called implies a new frame, so increment
      // the frame count.
      ++frameCnt;

      // Also accumulate how much time has passed since the
      // last frame.
      timeElapsed += deltaTime;

      // Has one second passed?
      if( timeElapsed >= 1.0f )
      {
            // Yes, so compute the frames per second.
            // FPS = frameCnt / timeElapsed, but since we
            // compute only when timeElapsed = 1.0, we can
            // reduce to:
            // FPS = frameCnt / 1.0 = frameCnt.

            sprintf(buffer, "--Frames Per Second = %d", frameCnt);

            // Add the frames per second string to the main
            // window caption--that is, we'll display the frames
            // per second in the window's caption bar.
            string newCaption = gWndCaption + buffer;

            // Now set the new caption to the main window.
            SetWindowText(ghMainWnd, newCaption.c_str());

            // Reset the counters to prepare for the next time
            // we compute the frames per second.
            frameCnt    = 0;
            timeElapsed = 0.0f;
      }
}
```

Items to note:

1. The function `SetWindowText`, which we previously used to set the text of an edit box control, is actually generic to all windows, and can be used to set the text of any window. In main windows, `SetWindowText` sets the text of the window's caption bar.

2. We wait until a second of time has elapsed before we make the frames per second computation. This is done for a few reasons. First, it is inefficient to make the calculation too frequently. Second, the frames per second are not constant, and so, by waiting for a second, we get a good average, which is more meaningful. And third, if we output the updated frames per second count every frame, it would appear too quickly to see!

17.2 Double Buffering

17.2.1 Motivation

In the drawing programs in Chapter 15, you probably noticed some flickering. Flickering occurs when you draw one thing and then draw something over it quickly afterwards. The process of drawing over a previously drawn pixel is called **overdraw.** The eye, which is quite sensitive to change and movement, picks up on overdraw and a flickering phenomenon is seen.

Although flickering may be acceptable for some applications that are not graphics intensive, it is not acceptable for games. Therefore, we are motivated to look for a solution, which double buffering provides. Note that double buffering is an important concept, which is also used in Direct3D programming (the graphics aspect of DirectX) and in OpenGL.

17.2.2 Theory

The problem with overdraw is not so much overdraw in and of itself, but rather that we can see the overdraw. More specifically, because we are drawing to the client rectangle, which we can always see, we are literally watching the image get drawn. Thus the user can notice the brief flicker in color when an overdraw occurs—we see the color of the first graphic object that was drawn, and then a fraction of a second later, a new color is drawn as we draw an object on top of it. Our solution to the flickering problem does not necessarily need to solve the overdraw problem. It only needs to keep the viewer from seeing the overdraw. This is where double buffering comes in.

The idea of **double buffering** is quite simple: instead of drawing directly to the client area of the window, we instead draw to an **off screen surface** called the **backbuffer,** which will be a blank bitmap that will represent our virtual "canvas." Since the backbuffer is not directly visible, overdraw is not a problem. That is, we will not see the overdraw as it happens because we cannot see the backbuffer—we are no longer watching the image get "painted." Once we have drawn all the graphics to the backbuffer, we will have a completed 2D image on the backbuffer. We can then copy the backbuffer image contents to the window client area all at once with the `BitBlt` function. Since the backbuffer contains the completed frame, the only thing we will need to draw to the window client area is the contents of the backbuffer. Therefore, we will have zero overdraw on the client area window, and thus avoid flicker.

The process of copying the backbuffer image to the window client area is known as **presenting.** For example, saying, "we *present* the backbuffer," means we copy the backbuffer to the window client area.

> **Note:** Drawing graphical objects are not the only things that are drawn. Windows usually clears the window client area background by default. This means that even if we use double buffering, we will still get overdraw: Windows clears the client area (1st draw) and then we present the backbuffer (2nd draw). To fix this, we simply instruct Windows not to clear the window client area. To do this, we specify a "null brush" for the window when we fill out the `hbrBackground` member of the `WNDCLASS`:
>
> ```
> wc.hbrBackground = (HBRUSH)::GetStockObject(NULL_BRUSH);
> ```
>
> This way, we will only draw to the window client area once—when we present the backbuffer.

17.2.3 Implementation

To implement double buffering, we will create a class to represent the backbuffer. Here is the class definition:

```
// BackBuffer.h
// By Frank Luna
// August 24, 2004.

#ifndef BACKBUFFER_H
#define BACKBUFFER_H

#include <windows.h>

class BackBuffer
{
public:
      BackBuffer(HWND hWnd, int width, int height);
      ~BackBuffer();

      HDC getDC();

      int width();
      int height();

      void present();

private:
      // Make copy constructor and assignment operator private
      // so client cannot copy BackBuffers.  We do this because
      // this class is not designed to be copied because it
      // is not efficient--copying bitmaps is slow (lots of memory).
```

```
      // In addition, most applications will probably only need one
      // BackBuffer anyway.
      BackBuffer(const BackBuffer& rhs);
      BackBuffer& operator=(const BackBuffer& rhs);
private:
      HWND    mhWnd;
      HDC     mhDC;
      HBITMAP mhSurface;
      HBITMAP mhOldObject;
      int     mWidth;
      int     mHeight;
};

#endif // BACKBUFFER_H
```

Let us begin analysis of this class by first looking at the data members.

1. `mhWnd`: A handle to the main window. We need this handle to obtain a device context associated with the main window.

2. `mhDC`: A handle to a *system memory* device context, upon which we will associate the backbuffer bitmap. We need this because all GDI functions are done through a device context. So in order to draw to the backbuffer bitmap, we need a device context associated with it. Note that this device context is not the window device context, which is associated with the main window's client area.

3. `mhSurface`: A handle to the bitmap that serves as our backbuffer. That is, our render target—the bitmap we will draw onto.

4. `mhOldObject`: A handle to the previous bitmap that was loaded by the device context. Recall that it is "good form" to restore the original GDI object when done with the new one.

5. `mWidth`: The width of the bitmap matrix in pixels; in other words, how many horizontal pixels across the bitmap surface.

6. `mHeight`: The height of the bitmap matrix in pixels; in other words, how many vertical pixels across the bitmap surface.

A backbuffer is represented with a bitmap and an associated device context, which is needed to render onto that bitmap.

Let us now examine the methods, one-by-one.

1. `BackBuffer(HWND hWnd, int width, int height);`

   The constructor is responsible for creating a backbuffer. It takes three parameters; the first is a handle to the main window from which we will get the associated window device context. This is needed because we need to create the backbuffer bitmap in a way that is compatible with the window's client area, in order that we can copy the backbuffer contents to the window's client area. The width and height parameters are simply used to specify the backbuffer dimensions and these dimensions should match the window's client area dimensions so that there is a one-to-one pixel correspondence between the backbuffer and window's client area. The constructor is implemented like so:

```
// BackBuffer.cpp
// By Frank Luna
// August 24, 2004.

#include "BackBuffer.h"

BackBuffer::BackBuffer(HWND hWnd, int width, int height)
{
      // Save a copy of the main window handle.
      mhWnd = hWnd;

      // Get a handle to the device context associated with
      // the window.
      HDC hWndDC = GetDC(hWnd);

      // Save the backbuffer dimensions.
      mWidth  = width;
      mHeight = height;

      // Create system memory device context that is compatible
      // with the window one.
      mhDC = CreateCompatibleDC(hWndDC);

      // Create the backbuffer surface bitmap that is compatible
      // with the window device context bitmap format.  That is
      // the surface we will render onto.
      mhSurface = CreateCompatibleBitmap(
            hWndDC, width, height);

      // Done with window DC.
      ReleaseDC(hWnd, hWndDC);

      // At this point, the back buffer surface is uninitialized,
      // so lets clear it to some non-zero value.  Note that it
      // needs to be non-zero.  If it is zero then it will mess
      // up our sprite blending logic.

      // Select the backbuffer bitmap into the DC.
      mhOldObject = (HBITMAP)SelectObject(mhDC, mhSurface);

      // Select a white brush.
      HBRUSH white    = (HBRUSH)GetStockObject(WHITE_BRUSH);
      HBRUSH oldBrush = (HBRUSH)SelectObject(mhDC, white);

      // Clear the backbuffer rectangle.
      Rectangle(mhDC, 0, 0, mWidth, mHeight);

      // Restore the original brush.
      SelectObject(mhDC, oldBrush);
}
```

2. `~BackBuffer();`

   The destructor is responsible for cleaning up any resources the constructor has allocated. We observe that the constructor allocates two resources: 1) a system memory device context, and 2) a bitmap. Thus we have the following destructor implementation:

```
BackBuffer::~BackBuffer()
{
	SelectObject(mhDC, mhOldObject);
	DeleteObject(mhSurface);
	DeleteDC(mhDC);
}
```

Note that we restore the original selected bitmap before deleting anything. Again, it is just “good form” to restore the original GDI object when done with the new one.

3. `HDC getDC();`

   This function is trivial and simply returns a copy of the backbuffer’s associated device context:

```
HDC BackBuffer::getDC()
{
	return mhDC;
}
```

4. `int width();`

   Another trivial method, this one returns the width of the backbuffer bitmap in pixels.

```
int BackBuffer::width()
{
	return mWidth;
}
```

5. `int height();`

   This method returns the height of the backbuffer bitmap in pixels.

```
int BackBuffer::height()
{
	return mHeight;
}
```

6. `void present();`

   This method presents the contents of the backbuffer to the main window’s client area. In order to do this, the method needs a handle to the device context associated with the main window’s client area; this can be obtained from the internal `mhWnd` data member. The implementation of this method is fairly simple; it copies the pixels from the backbuffer to the window’s client area using the `BitBlt` function. We copied bitmaps to the client area in Chapter 15. At that time we loaded our bitmap data from .bmp files. Here, we are actually drawing to the bitmap and generating it ourselves, and then we copy it to the client area.

```
void BackBuffer::present()
{
      // Get a handle to the device context associated with
      // the window.
      HDC hWndDC = GetDC(mhWnd);

      // Copy the backbuffer contents over to the
      // window client area.
      BitBlt(hWndDC, 0, 0, mWidth, mHeight,
                  mhDC, 0, 0, SRCCOPY);

      // Always free window DC when done.
      ReleaseDC(mhWnd, hWndDC);
}
```

One thing worth emphasizing is that we should release the window DC when we are finished with it. The window DC is a valuable GDI resource and we should only “hold on” to it when we need to use it to draw.

Now that we have our backbuffer set up, let us review the basic process of how we will use it in the game loop. First, we only need one backbuffer per application, and so we will declare it as a global variable:

```
BackBuffer* gBackBuffer = 0;
[...]
// Create the backbuffer.
gBackBuffer = new BackBuffer(
      ghWindowDC,
      gClientWidth,
      gClientHeight);
```

Observe how the dimensions of the backbuffer match the client area dimensions.

Remember that for each game loop cycle we draw a frame of animation. Thus, for every animated application we will have the following pattern executed per loop cycle:

1. Compute the time difference from the previous frame Δt.
2. Update the positions of the graphical objects slightly.
3. Draw the graphical objects to the backbuffer.
4. Present the backbuffer contents to the main window’s client area.

The astute reader may object, and point out that we do all of our drawing in a `WM_PAINT` message handler. This is no longer the case with an animation intensive application. Here we want to update and draw the graphics continuously. However, this brings up something else to think about. Namely, if we are no longer drawing in a `WM_PAINT` message handler, how do we obtain a device context to the window’s client area? Before, we always used `BeginPaint`, which may only be called in a `WM_PAINT` message handler. As you already saw from the `BackBuffer` class implementation, the Win32 API provides another function, which can give us a handle to a device context associated with a window's client area; the function is called `GetDC`:

```
// Get a DC associated with the window's client area.
HDC hWndDC = GetDC(mhWnd);
```

The `GetDC` function takes a parameter to a window handle (`HWND`), which specifies the window with which we want to associate the device context. The `GetDC` function then returns a handle to such a device context.

17.3 Tank Animation Sample

Figure 17.2 shows a screenshot of the Tank animation sample we will write in this section.

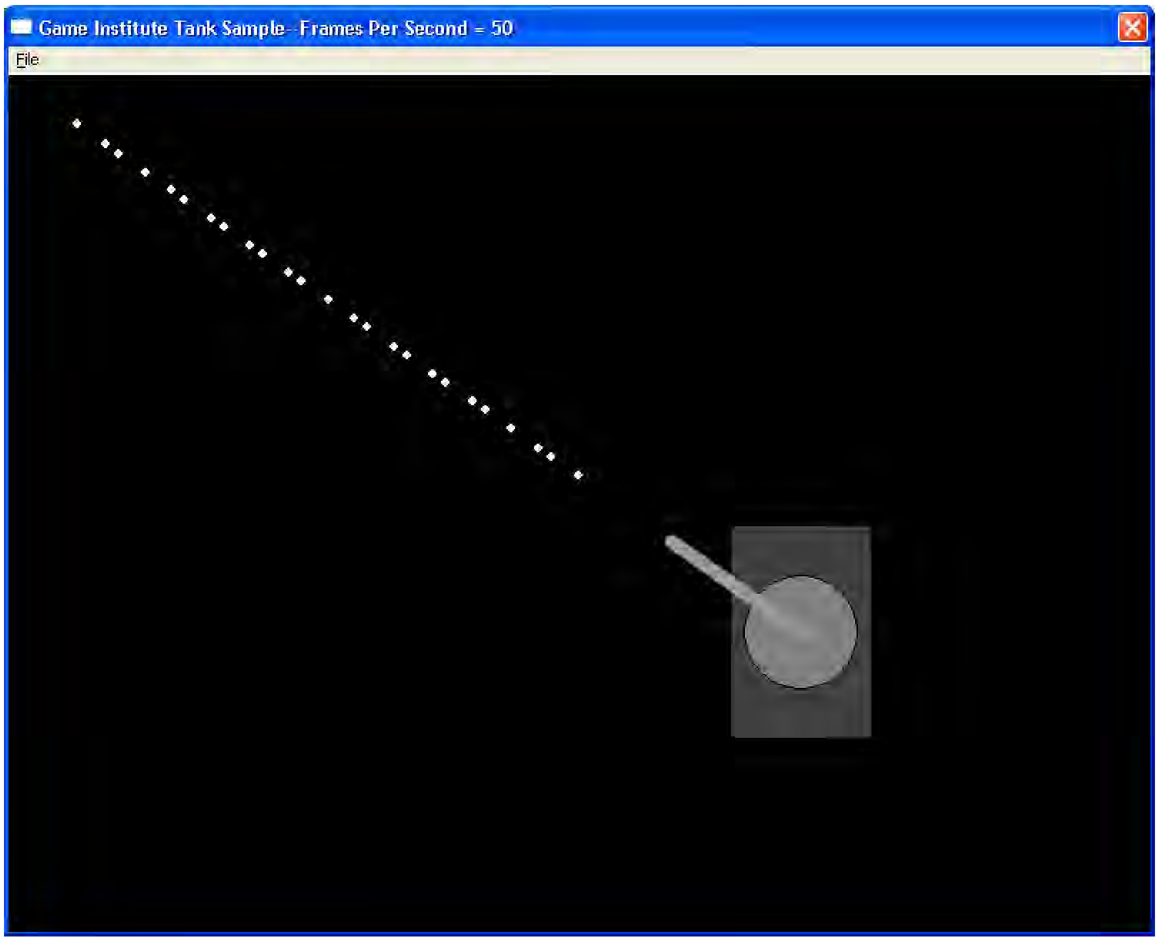


Figure 17.2: A screenshot of the Tank sample.

The tank is drawn using a rectangle for the tank base, an ellipse for the gun base, and a thick line (i.e., pen width > 1) for the gun. You can move the tank up and down and from side to side with the 'W', 'S', 'A' and 'D' keys. You can rotate the gun with the 'Q' and 'E' keys. Finally, you can fire bullets with the spacebar key. The bullets are modeled using ellipses.

Be aware that this program uses a 2D vector class called `Vec2`. This class is remarkably similar to the `Vector3` class we developed in Chapter 7 so please take a moment to review the vector mathematics discussed in that chapter if you do not recall the concepts. We will be using vectors to determine directions. For example, we will need to determine the direction a bullet should travel. In addition, we will sometimes interpret the components of vectors as points.

Before we begin an analysis of the tank program, let us first look at the global variables the program uses; the comments explain their purpose:

```
HWND        ghMainWnd  = 0; // Main window handle.
HINSTANCE   ghAppInst  = 0; // Application instance handle.
HMENU       ghMainMenu = 0; // Menu handle.

// The backbuffer we will render onto.
BackBuffer* gBackBuffer = 0;

// The text that will appear in the main window's caption bar.
string gWndCaption = "Game Institute Tank Sample";

// Client rectangle dimensions we will use.
const int gClientWidth  = 800;
const int gClientHeight = 600;

// Center point of client rectangle.
const POINT gClientCenter =
{
      gClientWidth  / 2,
      gClientHeight / 2
};

// Pad window dimensions so that there is room for window
// borders, caption bar, and menu.
const int gWindowWidth  = gClientWidth  + 6;
const int gWindowHeight = gClientHeight + 52;

// Client area rectangle, which we will use to detect
// if a bullet travels "out-of-bounds."
RECT gMapRect = {0, 0, 800, 600};

// Vector to store the center position of the tank,
// relative to the client area rectangle.
Vec2 gTankPos(400.0f, 300.0f);

// Handle to a pen we will use to draw the tank's gun.
HPEN gGunPen;

// A vector describing the direction the tank's gun
// is aimed in.  The vector’s magnitude denotes the
// length of the gun.
Vec2 gGunDir(0.0f, -120.0f);

// A list, where we will add bullets to as they are fired.
// The list stores the bullet positions, so that we can
// draw an ellipse at the position of each bullet.
list<Vec2> gBulletList;
```

17.3.1 Creation

The very first thing we need to do is initialize some of our resources. To do this, we need a valid handle to the main window, and therefore, the `WM_CREATE` message is a good place to do resource acquisition. We have two resources we need to create. First, we need to create the pen, which we will use to draw the tank gun. This pen needs to be somewhat thick, so we specify 10 units for its width. Finally, we create the backbuffer. Here is the implementation for the `WM_CREATE` message handler:

```
case WM_CREATE:

      // Create the tank's gun pen.
      lp.lopnColor   = RGB(150, 150, 150);
      lp.lopnStyle   = PS_SOLID;
      lp.lopnWidth.x = 10;
      lp.lopnWidth.y = 10;
      gGunPen = CreatePenIndirect(&lp);

      // Create the backbuffer.
      gBackBuffer = new BackBuffer(
            hWnd,
            gClientWidth,
            gClientHeight);

      return 0;
```

Where `lp` is a `LOGPEN`.

17.3.2 Destruction

The application destruction process should free any resource allocated in the application creation process. Thus we need to delete the pen we created and the backbuffer as well. The natural place to do such resource deletion is in the `WM_DESTROY` message handler:

```
case WM_DESTROY:
      DeleteObject(gGunPen);
      delete gBackBuffer;
      PostQuitMessage(0);
      return 0;
```

17.3.3 Input

We said that you can move the tank up and down and from side to side with the 'W', 'S', 'A' and 'D' keys, that you can rotate the gun with the 'Q' and 'E' keys, and that you can fire bullets with the spacebar key. Implementing such functionality is simply a matter of handling the `WM_KEYDOWN` message:

```
case WM_KEYDOWN:
      switch(wParam)
      {
      // Move left.
      case 'A':
            gTankPos.x -= 5.0f;
            break;
      // Move right.
      case 'D':
            gTankPos.x += 5.0f;
            break;
      // Move up--remember in Windows coords, -y = up.
      case 'W':
            gTankPos.y -= 5.0f;
            break;
      // Move down.
      case 'S':
            gTankPos.y += 5.0f;
            break;
      // Rotate tank gun to the left.
      case 'Q':
            gGunDir.rotate(-0.1f);
            break;
      // Rotate tank gun to the right.
      case 'E':
            gGunDir.rotate(0.1f);
            break;
      // Fire a bullet.
      case VK_SPACE:
            gBulletList.push_back(gTankPos + gGunDir);
            break;
      }
      return 0;
```

As you can see, pressing either the 'A', 'W', 'S', or 'D' key simply updates the tank's position slightly along the appropriate axis. The 'Q' and 'E' keys rotate the tank's gun. We will discuss how `Vec2::rotate` is implemented in Section 17.3.6. For now, just realize that this rotates the gun's direction vector by some angle in a circular fashion.

Finally, pressing the spacebar button (symbolized with `VK_SPACE`), adds a bullet to our global list of bullets. Recall that the bullet list stores the positions of the bullets. We will update the bullets in another function, but when we first create the bullet (add it to the list) we want the bullet to be created at the tip of the gun, not the center point of the tank. Thus we have to do some vector addition to get that gun tip point. That is, `gTankPos + gGunDir`. Figure 17.3 shows what this means geometrically.

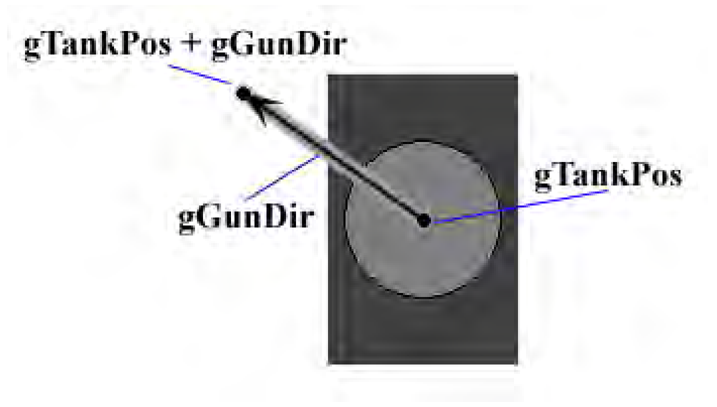


Figure 17.3: The position of the gun's tip point is given by gTankPos + gGunDir.

17.3.4 Updating and Drawing

We are now ready to examine the game loop for the tank program. However, the implementation is a bit lengthy, so let us first look at a general roadmap of the function:

1. Compute the time elapsed between frames (Δt).
2. Draw a black rectangle spanning the entire backbuffer to clear the backbuffer to black. This provides our background.
3. Draw the tank to the backbuffer, which includes the base rectangle, the circular gun base, and the gun itself.
4. Iterate over the entire bullet list, and for each bullet, update the bullet position and draw the bullet to the backbuffer.
5. Draw the frames per second into the Window Caption bar.
6. Present the backbuffer contents to the main window's client area.

The implementation is as follows:

```
while(msg.message != WM_QUIT)
{
      // IF there is a Windows message then process it.
      if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
      // ELSE, do game stuff.
      else
      {
            // Get the time now.
            float currTime  = (float)timeGetTime();

            // Compute the differences in time from the last
            // time we checked.  Since the last time we checked
```

```
        // was the previous loop iteration, this difference
        // gives us the time between loop iterations...
        // or, I.e., the time between frames.
        float deltaTime = (currTime - lastTime)*0.001f;

        // Get the backbuffer DC.
        HDC bbDC = gBackBuffer->getDC();

        // Clear the entire backbuffer black.  This gives
        // up a black background.
        HBRUSH oldBrush = (HBRUSH)SelectObject(bbDC,
                                      GetStockObject(BLACK_BRUSH));
        Rectangle(bbDC, 0, 0, 800, 600);

        // Draw the base of the tank--a rectangle surrounding
        // the tank's center position point.
        SelectObject(bbDC, GetStockObject(DKGRAY_BRUSH));
        Rectangle(bbDC,
              (int)gTankPos.x - 50,
              (int)gTankPos.y - 75,
              (int)gTankPos.x + 50,
              (int)gTankPos.y + 75);

        // Draw the gun base--an ellipse surrounding
        // the tank's center position point.
        SelectObject(bbDC, GetStockObject(GRAY_BRUSH));
        Ellipse(bbDC,
              (int)gTankPos.x - 40,
              (int)gTankPos.y - 40,
              (int)gTankPos.x + 40,
              (int)gTankPos.y + 40);

        // Draw the gun itself--a line from the tank's
        // center position point to the tip of the gun.
        HPEN oldPen = (HPEN)SelectObject(bbDC, gGunPen);
        MoveToEx(bbDC, (int)gTankPos.x, (int)gTankPos.y, 0);
        LineTo(bbDC,
              (int)(gTankPos.x + gGunDir.x),
              (int)(gTankPos.y + gGunDir.y));

        // Draw any bullets that where fired.
        SelectObject(bbDC, GetStockObject(WHITE_BRUSH));
        SelectObject(bbDC, oldPen);

        // Bullet velocity is 5X the gun's direction's
        // magnitude.
        Vec2 bulletVel = gGunDir * 5.0f;
        list<Vec2>::iterator i = gBulletList.begin();
        while( i != gBulletList.end() )
        {
              // Update the bullet position.
              *i += bulletVel * deltaTime;

              // Get POINT form.
              POINT p = *i;

              // Only draw bullet if it is still inside the
```

```
                    // map boundaries, otherwise, delete it.
                    if( !PtInRect(&gMapRect, p) )
                          i = gBulletList.erase(i);
                    else
                    {
                          // Draw bullet as a circle.
                          Ellipse(bbDC,
                                p.x - 4,
                                p.y - 4,
                                p.x + 4,
                                p.y + 4);

                          ++i; // Next in list.
                    }
              }

              SelectObject(bbDC, oldBrush);

              DrawFramesPerSecond(deltaTime);

              // Now present the backbuffer contents to the main
              // window client area.
              gBackBuffer->present(ghWindowDC);

              // We are at the end of the loop iteration, so
              // prepare for the next loop iteration by making
              // the "current time" the "last time."
              lastTime = currTime;

              // Free 20 miliseconds to Windows so we don't hog
              // the system resources.
              Sleep(20);
        }
}
```

A new function that we have not discussed is the `Sleep` function. This Win32 function takes a single parameter, which specifies the number of milliseconds to sleep. Sleeping is defined as suspending execution of the current application so that Windows is free to perform other processes.

Despite being long, the game loop implementation is fairly straightforward. The only tricky part might be updating the bullets, so let us examine that section more closely. First, we define a bullet's velocity to be in the direction the gun is aimed, but five times the magnitude. Recall that velocity describes a speed (magnitude) and the direction of travel.

```
Vec2 bulletVel = gGunDir * 5.0f;
```

Given the velocity, we update the bullet's position like so:

```
*i += bulletVel * deltaTime;
```

But what exactly is `bulletVel * deltaTime`? To see this, we must go to the definition of velocity, which is the change in position over time:

$$\vec{v} = \frac{\Delta\vec{p}}{\Delta t} \Rightarrow \Delta\vec{p} = \vec{v} \cdot \Delta t$$

That is, the change in position of the bullet $\Delta\vec{p}$ (displacement) over Δt seconds is $\Delta\vec{p} = \vec{v} \cdot \Delta t$. So the formula $\Delta\vec{p} = \vec{v} \cdot \Delta t$ tells us how much the position $\vec{p}$ needs to be displaced given the velocity $\vec{v}$, over a time of Δt seconds. Recall that Δt is the time elapsed between frames, thus this formula tells us how much to displace a point $\vec{p}$ per frame given the velocity $\vec{v}$; that is, $\vec{p}' = \vec{p} + \Delta\vec{p} = \vec{p} + \vec{v} \cdot \Delta t$ —see Figure 17.4.

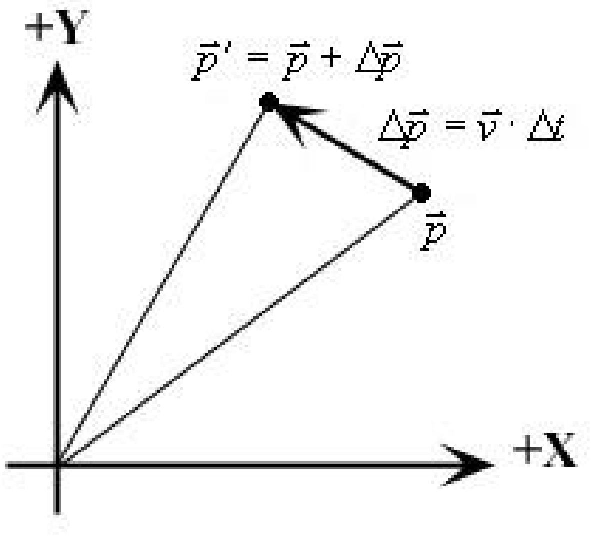

+Y
$\vec{p}' = \vec{p} + \Delta\vec{p}$
$\Delta\vec{p} = \vec{v} \cdot \Delta t$
$\vec{p}$
+X

Figure 17.4: Displacement. The displaced point $\vec{p}'$ equals $\vec{p} + \Delta\vec{p}$, where $\Delta\vec{p} = \vec{v} \cdot \Delta t$. Note that this figure shows a "typical" coordinate system. Recall that in Windows coordinates, +Y goes "down." However, the idea of displacement is the same, nonetheless.

Note that the value Δt will typically be very small: if we are running at 30 frames per second, then Δt will approximately being 1/30th of a second. Thus, the displacement vector $\Delta\vec{p}$ will also be small. These small displacements over time give a smooth continuous animation.

Finally, the `std::list::erase` method is a method that allows us to delete an element in the list given an iterator to it:

```
i = gBulletList.erase(i);
```

This function deletes the iterator `i` and returns an iterator to the next element in the list.

17.3.5 Point Rotation

We stated in Section 17.3.3 that we are able to rotate the gun's directional vector with the code:

```
// Rotate tank gun to the left.
case 'Q':
      gGunDir.rotate(-0.1f);
      break;
// Rotate tank gun to the right.
case 'E':
      gGunDir.rotate(0.1f);
      break;
```

However, we did not elaborate on how the `Vec2::rotate` function worked. Let us examine that now.

The implementation to `Vec2::rotate` looks like so:

```
Vec2& Vec2::rotate(float t)
{
      x = x * cosf(t) - y * sinf(t);
      y = y * cosf(t) + x * sinf(t);

      return *this;
}
```

The mathematical operations taking place in the implementation do not make any sense until we derive the rotation equations, which we will do now.

Consider Figure 17.5, where we have a given point $(x, \quad y)$, which makes an angle α with the x-axis, and we want to know the coordinates of that point if we rotate it by an angle θ in a counterclockwise direction. That is, we want to know $(x', \quad y')$.

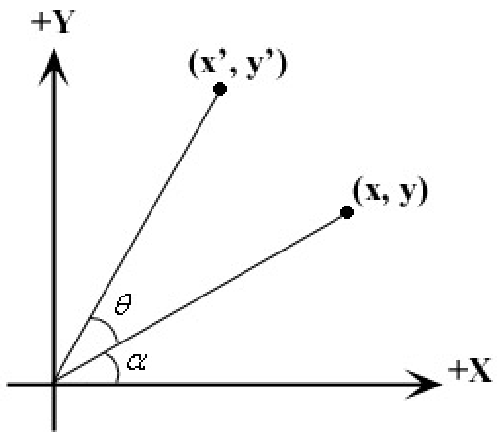


Figure 17.5: Rotating a point (x, y) by and angle θ to a new point (x', y').

Trigonometry dictates that:

(1) $$x = R\cos(\alpha) \\ y = R\sin(\alpha)$$

and similarly that:

(2) $$x' = R\cos(\alpha + \theta) \\ y' = R\sin(\alpha + \theta)$$

Moreover, there is a trigonometric identity for angle sum relations:

(3) $$\cos(\alpha + \theta) = \cos(\alpha)\cos(\theta) - \sin(\alpha)\sin(\theta) \\ \sin(\alpha + \theta) = \sin(\alpha)\cos(\theta) + \cos(\alpha)\sin(\theta)$$

Thus, (2) can be rewritten as:

(4) $$x' = R\cos(\alpha)\cos(\theta) - R\sin(\alpha)\sin(\theta) \\ y' = R\sin(\alpha)\cos(\theta) + R\cos(\alpha)\sin(\theta)$$

However, we note that the $R\cos(\alpha)$ and $R\sin(\alpha)$ factors in equations (4) can be substituted with *x* and *y*, respectively, due to the relationships specified in (1). Thus, the rotated point in terms of the original point and the angle of rotation θ is:

The 2D Rotation Counterclockwise Rotation Formula.

(5) $$x' = x\cos(\theta) - y\sin(\theta) \\ y' = y\cos(\theta) + x\sin(\theta)$$

And we can now see that the implementation of `Vec2::rotate` is a direct application of equations (5).

17.3.6 Tank Application Code

To conclude the Tank sample discussion, we now present the main application code in its entirety so that you can see everything together at once, instead of in separate parts. However, be sure to download the complete project from the Game Institute C++ Course Website so that you see the entire project as a whole with the other .h/.cpp files (BackBuffer.h/.cpp, and Vec2.h/.cpp).

Program 17.1: The Tank Sample Main Application Code. You still need the other files like Sprite.h/.cpp, BackBuffer.h/.cpp, and Vec2.h/.cpp to compile. To obtain these files download the entire project off of the Game Institute C++ Course Website.

```
// tank.cpp
// By Frank Luna
// August 24, 2004.

//=========================================================
// Includes
//=========================================================
#include <string>
#include "resource.h"
#include "BackBuffer.h"
#include "Vec2.h"
#include <list>
using namespace std;

//=========================================================
// Globals
//=========================================================
HWND        ghMainWnd  = 0; // Main window handle.
HINSTANCE   ghAppInst  = 0; // Application instance handle.
HMENU       ghMainMenu = 0; // Menu handle.

// The backbuffer we will render onto.
BackBuffer* gBackBuffer = 0;

// The text that will appear in the main window's caption bar.
string gWndCaption = "Game Institute Tank Sample";

// Client rectangle dimensions we will use.
const int gClientWidth  = 800;
const int gClientHeight = 600;

// Center point of client rectangle.
const POINT gClientCenter =
{
      gClientWidth  / 2,
      gClientHeight / 2
};

// Pad window dimensions so that there is room for window
// borders, caption bar, and menu.
const int gWindowWidth  = gClientWidth  + 6;
const int gWindowHeight = gClientHeight + 52;

// Client area rectangle, which we will use to detect
// if a bullet travels "out-of-bounds."
RECT gMapRect = {0, 0, 800, 600};

// Vector to store the center position of the tank,
// relative to the client area rectangle.
Vec2 gTankPos(400.0f, 300.0f);

// Handle to a pen we will use to draw the tank's gun.
HPEN gGunPen;

// A vector describing the direction the tank's gun
// is aimed in.  The vector's magnitude denotes the
// length of the gun.
```

```
Vec2 gGunDir(0.0f, -120.0f);

// A list, where we will add bullets to as they are fired.
// The list stores the bullet positions, so that we can
// draw an ellipse at the position of each bullet.
list<Vec2> gBulletList;

//=========================================================
// Function Prototypes
//=========================================================

bool InitMainWindow();
int  Run();
void DrawFramesPerSecond(float deltaTime);

LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

//=========================================================
// Name: WinMain
// Desc: Program execution starts here.
//=========================================================

int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      ghAppInst = hInstance;

      // Create the main window.
      if( !InitMainWindow() )
      {
            MessageBox(0, "Window Creation Failed.", "Error", MB_OK);
            return 0;
      }

      // Enter the message loop.
      return Run();
}

//=========================================================
// Name: InitMainWindow
// Desc: Creates the main window upon which we will
//       draw the game graphics onto.
//=========================================================
bool InitMainWindow()
{
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = ghAppInst;
      wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
      wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
      wc.hbrBackground = (HBRUSH)::GetStockObject(NULL_BRUSH);
      wc.lpszMenuName  = 0;
      wc.lpszClassName = "MyWndClassName";

      RegisterClass( &wc );

      // WS_OVERLAPPED | WS_SYSMENU: Window cannot be resized
      // and does not have a min/max button.
      ghMainMenu = LoadMenu(ghAppInst, MAKEINTRESOURCE(IDR_MENU));
      ghMainWnd = ::CreateWindow("MyWndClassName",
            gWndCaption.c_str(), WS_OVERLAPPED | WS_SYSMENU,
            200, 200, gWindowWidth, gWindowHeight, 0,
            ghMainMenu, ghAppInst, 0);

      if(ghMainWnd == 0)
      {
            ::MessageBox(0, "CreateWindow - Failed", 0, 0);
            return 0;
      }

      ShowWindow(ghMainWnd, SW_NORMAL);
      UpdateWindow(ghMainWnd);

      return true;
}

//=========================================================
// Name: Run
// Desc: Encapsulates the message loop.
//=========================================================
int Run()
{
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));

      // Get the current time.
      float lastTime = (float)timeGetTime();

      while(msg.message != WM_QUIT)
      {
            // IF there is a Windows message then process it.
            if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
            // ELSE, do game stuff.
            else
            {
                  // Get the time now.
                  float currTime  = (float)timeGetTime();

                  // Compute the differences in time from the last
                  // time we checked.  Since the last time we checked
                  // was the previous loop iteration, this difference
                  // gives us the time between loop iterations...
                  // or, I.e., the time between frames.
                  float deltaTime = (currTime - lastTime)*0.001f;
```

```
// Get the backbuffer DC.
HDC bbDC = gBackBuffer->getDC();

// Clear the entire backbuffer black.  This gives
// up a black background.
HBRUSH oldBrush = (HBRUSH)SelectObject(bbDC,
                    GetStockObject(BLACK_BRUSH));
Rectangle(bbDC, 0, 0, 800, 600);

// Draw the base of the tank--a rectangle surrounding
// the tank's center position point.
SelectObject(bbDC, GetStockObject(DKGRAY_BRUSH));
Rectangle(bbDC,
      (int)gTankPos.x - 50,
      (int)gTankPos.y - 75,
      (int)gTankPos.x + 50,
      (int)gTankPos.y + 75);

// Draw the gun base--an ellipse surrounding
// the tank's center position point.
SelectObject(bbDC, GetStockObject(GRAY_BRUSH));
Ellipse(bbDC,
      (int)gTankPos.x - 40,
      (int)gTankPos.y - 40,
      (int)gTankPos.x + 40,
      (int)gTankPos.y + 40);

// Draw the gun itself--a line from the tank's
// center position point to the tip of the gun.
HPEN oldPen = (HPEN)SelectObject(bbDC, gGunPen);
MoveToEx(bbDC, (int)gTankPos.x, (int)gTankPos.y, 0);
LineTo(bbDC,
      (int)(gTankPos.x + gGunDir.x),
      (int)(gTankPos.y + gGunDir.y));

// Draw any bullets that where fired.
SelectObject(bbDC, GetStockObject(WHITE_BRUSH));
SelectObject(bbDC, oldPen);

// Bullet velocity is 5X the gun's direction's
// magnitude.
Vec2 bulletVel = gGunDir * 5.0f;
list<Vec2>::iterator i = gBulletList.begin();
while( i != gBulletList.end() )
{
      // Update the bullet position.
      *i += bulletVel * deltaTime;

      // Get POINT form.
      POINT p = *i;

      // Only draw bullet if it is still inside the
      // map boundaries, otherwise, delete it.
      if( !PtInRect(&gMapRect, p) )
            i = gBulletList.erase(i);
      else
      {
```

```
                              // Draw bullet as a circle.
                              Ellipse(bbDC,
                                    p.x - 4,
                                    p.y - 4,
                                    p.x + 4,
                                    p.y + 4);

                              ++i; // Next in list.
                        }
                  }

                  SelectObject(bbDC, oldBrush);

                  DrawFramesPerSecond(deltaTime);

                  // Now present the backbuffer contents to the main
                  // window client area.
                  gBackBuffer->present();

                  // We are at the end of the loop iteration, so
                  // prepare for the next loop iteration by making
                  // the "current time" the "last time."
                  lastTime = currTime;

                  // Free 20 miliseconds to Windows so we don't hog
                  // the system resources.
                  Sleep(20);
             }
       }
      // Return exit code back to operating system.
      return (int)msg.wParam;
}

//===========================================================
// Name: DrawFramesPerSecond
// Desc: This function is called every frame and updates
//       the frame per second display in the main window
//       caption.
//===========================================================
void DrawFramesPerSecond(float deltaTime)
{
      // Make static so the variables persist even after
      // the function returns.
      static int   frameCnt    = 0;
      static float timeElapsed = 0.0f;
      static char  buffer[256];

      // Function called implies a new frame, so increment
      // the frame count.
      ++frameCnt;

      // Also increment how much time has passed since the
      // last frame.
      timeElapsed += deltaTime;

      // Has one second passed?
      if( timeElapsed >= 1.0f )
      {
            // Yes, so compute the frames per second.
            // FPS = frameCnt / timeElapsed, but since we
            // compute only when timeElapsed = 1.0, we can
            // reduce to:
            // FPS = frameCnt / 1.0 = frameCnt.

            sprintf(buffer, "--Frames Per Second = %d", frameCnt);

            // Add the frames per second string to the main
            // window caption--that is, we'll display the frames
            // per second in the window's caption bar.
            string newCaption = gWndCaption + buffer;

            // Now set the new caption to the main window.
            SetWindowText(ghMainWnd, newCaption.c_str());

            // Reset the counters to prepare for the next time
            // we compute the frames per second.
            frameCnt    = 0;
            timeElapsed = 0.0f;
      }
}

//=========================================================
// Name: WndProc
// Desc: The main window procedure.
//=========================================================

LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      LOGPEN lp;

      switch( msg )
      {
      // Create application resources.
      case WM_CREATE:

            // Create the tank's gun pen.
            lp.lopnColor   = RGB(150, 150, 150);
            lp.lopnStyle   = PS_SOLID;
            lp.lopnWidth.x = 10;
            lp.lopnWidth.y = 10;
            gGunPen = CreatePenIndirect(&lp);

            // Create the backbuffer.
            gBackBuffer = new BackBuffer(
                  hWnd,
                  gClientWidth,
                  gClientHeight);

            return 0;

      case WM_COMMAND:
            switch(LOWORD(wParam))
            {
```

```
            // Destroy the window when the user selects the 'exit'
            // menu item.
            case ID_FILE_EXIT:
                  DestroyWindow(ghMainWnd);
                  break;
            }
            return 0;

      case WM_KEYDOWN:
            switch(wParam)
            {
            // Move left.
            case 'A':
                  gTankPos.x -= 5.0f;
                  break;
            // Move right.
            case 'D':
                  gTankPos.x += 5.0f;
                  break;
            // Move up--remember in Windows coords, -y = up.
            case 'W':
                  gTankPos.y -= 5.0f;
                  break;
            // Move down.
            case 'S':
                  gTankPos.y += 5.0f;
                  break;
            // Rotate tank gun to the left.
            case 'Q':
                  gGunDir.rotate(-0.1f);
                  break;
            // Rotate tank gun to the right.
            case 'E':
                  gGunDir.rotate(0.1f);
                  break;
            // Fire a bullet.
            case VK_SPACE:
                  gBulletList.push_back(gTankPos + gGunDir);
                  break;
            }
            return 0;

      // Destroy application resources.
      case WM_DESTROY:
            DeleteObject(gGunPen);
            delete gBackBuffer;
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
      // default window procedure.
      return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

17.4 Sprites

17.4.1 Theory

Although the tank from the tank sample is reminiscent of early classic computer gaming, its graphics are obviously extremely primitive. A more contemporary solution is for an artist to paint a detailed tank image using some image editing software, such as Adobe Photoshop™, and perhaps base it on a photograph of a real tank. The artist can then save the image, say as a .bmp file, which we can load and use in our programs. This can lead us to some better-looking graphics. For example, Figure 17.6 shows a bitmap of a military jet we could use in a 2D jet fighter style game.

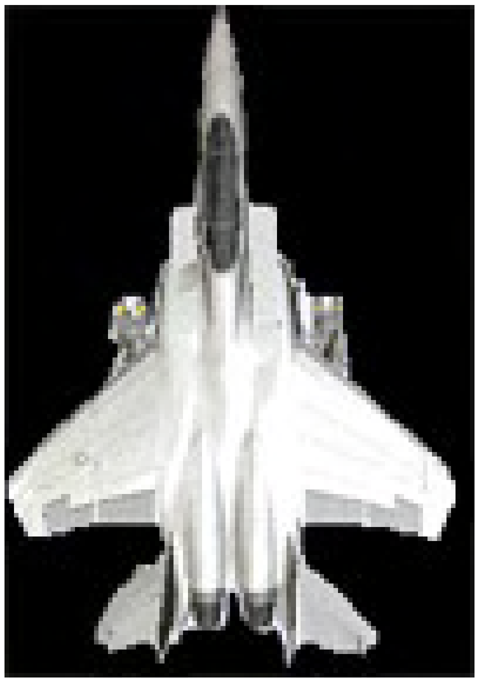


Figure 17.6: A bitmap of a fighter jet.

However, there is a problem. Bitmaps are rectangular, by definition. The actual jet part of Figure 17.6 is not rectangular, but as you can see, it lies on a rectangular background. However, when we draw these bitmaps we do not want this black background to be drawn—Figure 17.7 illustrates the problem and Figure 17.8 shows the desired "correct" output.

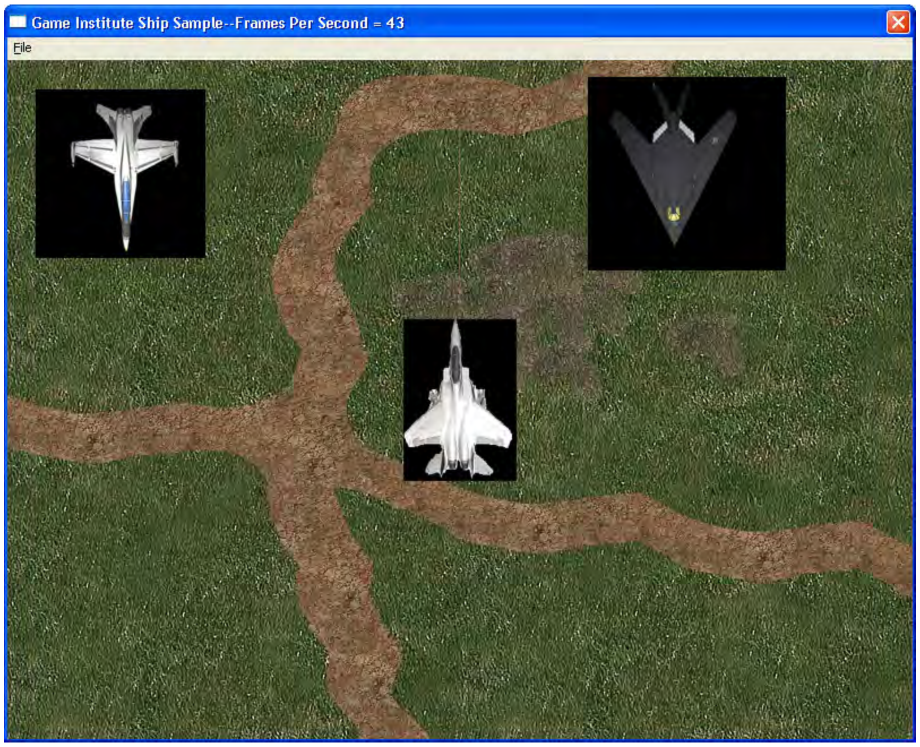


Figure 17.7: Drawing bitmaps incorrectly. The problem is that the bitmap pixels that are not part of the actual image show through; that is, the black background pixels are visible.

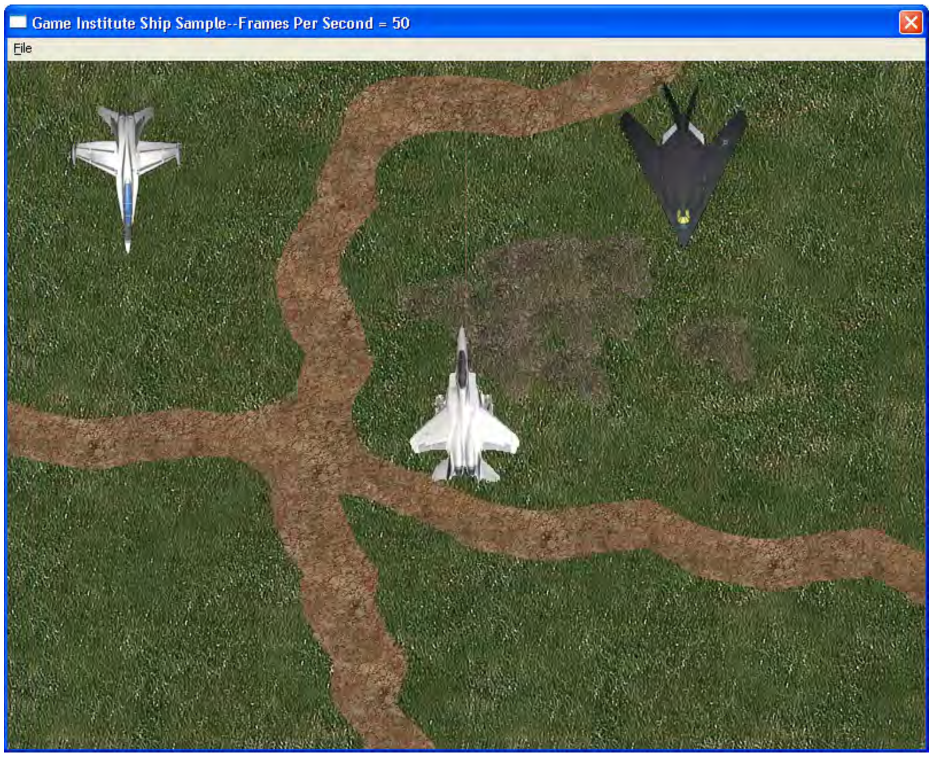


Figure 17.8: Drawing bitmaps correctly—only the desired image pixels are drawn.

The task at hand is to figure out a way to not draw the black background part of the bitmap. These images we use to represent game objects, such as jets, missiles and such, are commonly referred to as **sprites.**

The solution to this problem lies in the way we combine the pixels of the source bitmap (sprite) and the pixels of the destination bitmap (backbuffer). First, for each sprite we will create a corresponding **mask bitmap.** This bitmap will mark the pixels of the sprite which should be drawn to the backbuffer and mark the pixels of the sprite which should not be drawn to the backbuffer. It is important to realize that the mask bitmap must be of the same dimensions as the image bitmap so that the *ij*th pixel in the image corresponds with the *ij*th pixel in the mask. Figure 17.9*a* shows a jetfighter bitmap image, Figure 17.9*b* shows its mask, and Figure 17.9*c* shows the result when the image is combined with the mask. (We show how this combination is done in the next paragraph; our goal here is to intuitively show what is happening.)

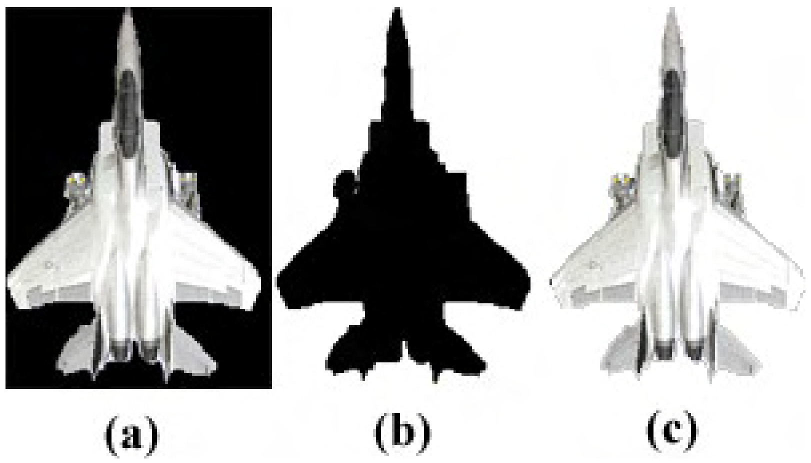


Figure 17.9: (a) The image bitmap. (b) The mask, marking the pixels that should be drawn. (c) The result after combining the image and the mask.

In the mask, the black pixels mark the pixels of the sprite that should be drawn to the backbuffer and the white pixels mark the pixels of the sprite that should not be drawn to the backbuffer.

To draw a sprite, we first draw the mask to the backbuffer using the raster operation `SRCAND`, instead of `SRCCOPY`. Recall that `SRCCOPY` simply copies the source pixels directly over the destination pixels—it overwrites whatever is there. On the other hand, `SRCAND` combines each source pixel with its corresponding destination pixel to produce the new final destination color, like so: F = D & S, where F is the final color eventually written to the destination pixel, D was the previous destination pixel color, and S is the source pixel color. This combination is a bitwise AND. (Hopefully your bitwise operations are well understood. If they are not, you may wish to review Chapter 12**.)** **The AND operation is a key point because when we AND a pixel color with a black pixel we get black (zero) in all circumstances, and when we AND a pixel color with a white pixel we do not modify the original pixel (it is like multiplying by one):**

1. D & S = 0x00?????? & 0x00000000 = 0x00000000 (S = Black)
2. D & S = 0x00?????? & 0x00FFFFFF = 0x00?????? (S = White)

The question marks simply mean the left-hand-side can be any value.

What does this amount to? It means that when we draw the mask to the backbuffer, we draw black to the backbuffer where the mask is black (D & Black = Black) and we leave the backbuffer pixels unchanged where the mask is white (D & White = D).

> **Note:** In case you are wondering how we can AND colors, recall that colors are typically represented as 32-bit integers, where 8-bits are used for the red component, 8-bits are used for the green component, and 8-bits are used for the blue component. (8-bits are not used.) That is what a `COLORREF` is: it is `typedef`ed as a 32-bit integer. In hexadecimal, the format of the `COLORREF` type looks like so:
>
> 0x00bbggrr
>
> Where red takes the rightmost 8-bits, green takes the next 8-bits, blue takes the next 8-bits, and the top 8-bits is not used and just set to zero.
>
> We can do bitwise operations on integers and thus `COLORREF`s. Incidentally, the color black would be described as 0x00000000 (each 8-bit color component is 0), and white is represented as 0x00FFFFFF (each 8-bit color component is 255).

At this point, we have drawn black to the backbuffer pixels that correspond to the pixels in the sprite image we wish to draw—Figure 17.10. The next step is to draw the actual sprite image onto the backbuffer.

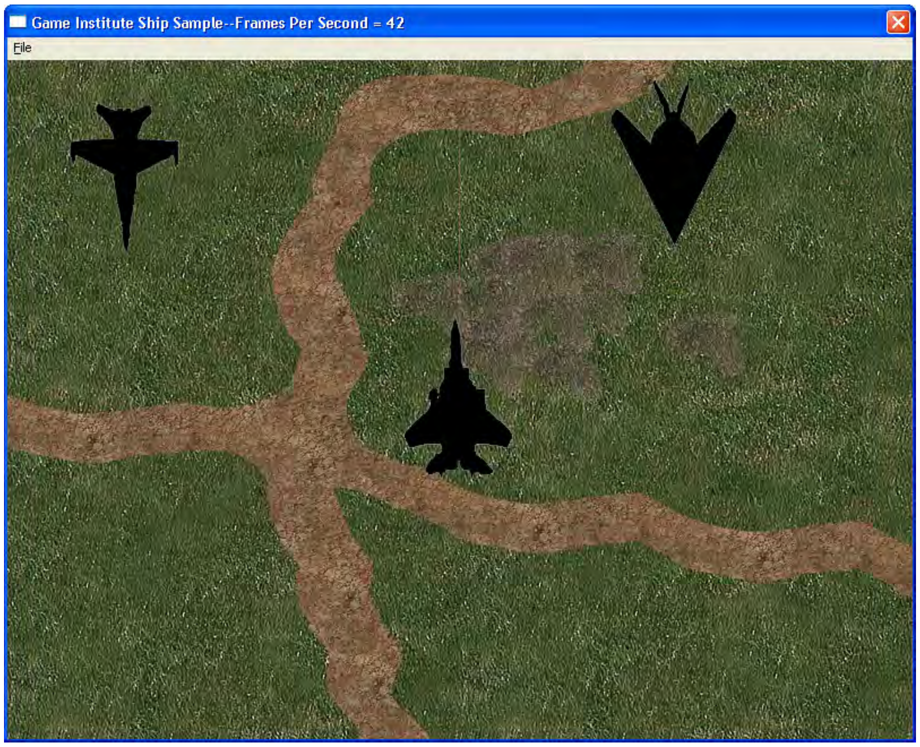


Figure 17.10: The backbuffer after the mask bitmap is drawn to it. It marks the pixels black where we will copy the image bitmap to.

However, when we draw the sprite image, we need to use the raster operation `SRCPAINT`, which specifies that the destination and source pixels be combined like so: F = D | S. The OR operation is a key point because, where the destination (backbuffer) is black, it copies all the source pixels: F = Black | S = S. This is exactly what we want, because we marked the pixels black (when we drew the mask) which correspond to the spite pixels we want to draw. Moreover, where the source (sprite) is black, it leaves the destination backbuffer color unchanged because, F = D | Black = D. Again, this is exactly what we want. We do not want to draw the black background part of the sprite bitmap. The end result is that only the sprite pixels we want to draw are drawn to the backbuffer—Figure 17.8.

17.4.2 Implementation

To facilitate the drawing of sprites, we will create a `Sprite` class, which will handle all the drawing details discussed in the previous section. In addition, it will handle the allocation and deallocation of the resources associated with sprites. Here is the class definition:

```
// Sprite.h
// By Frank Luna
// August 24, 2004.

#ifndef SPRITE_H
#define SPRITE_H

#include <windows.h>
#include "Circle.h"
#include "Vec2.h"

class Sprite
{
public:
	Sprite(HINSTANCE hAppInst, int imageID, int maskID,
		const Circle& bc, const Vec2& p0, const Vec2& v0);

	~Sprite();

	int width();
	int height();

	void update(float dt);
	void draw(HDC hBackBufferDC, HDC hSpriteDC);

public:
	// Keep these public because they need to be
	// modified externally frequently.
	Circle    mBoundingCircle;
	Vec2      mPosition;
	Vec2      mVelocity;

private:
	// Make copy constructor and assignment operator private
	// so client cannot copy Sprites.  We do this because
	// this class is not designed to be copied because it
```

```
      // is not efficient--copying bitmaps is slow (lots of memory).
      Sprite(const Sprite& rhs);
      Sprite& operator=(const Sprite& rhs);

protected:
      HINSTANCE mhAppInst;
      HBITMAP   mhImage;
      HBITMAP   mhMask;
      BITMAP    mImageBM;
      BITMAP    mMaskBM;
};

#endif // SPRITE_H
```

We will begin by analyzing the data members first.

1. `mBoundingCircle`: A circle that approximately describes the area of the sprite. We will discuss and implement the `Circle` class in the next chapter. It is not required in this chapter yet.

2. `mPosition`: The center position of the sprite rectangle.

3. `mVelocity`: The velocity of the sprite—the direction and speed the sprite is moving in.

4. `mhAppInst`: A handle to the application instance.

5. `mhImage`: A handle to the sprite image bitmap.

6. `mhMask`: A handle to the sprite mask bitmap.

7. `mImageBM`: A structure containing the sprite image bitmap info.

8. `mMaskBM`: A structure containing the sprite mask bitmap info.

Now we describe the methods.

1.
   ```
   Sprite(HINSTANCE hAppInst, int imageID, int maskID,
          const Circle& bc, const Vec2& p0, const Vec2& v0);
   ```

   The constructor takes several parameters. The first is a handle to the application instance, which is needed for the `LoadBitmap` function. The second and third parameters are the resource IDs of the image bitmap and mask bitmap, respectively. The fourth parameter specifies the sprite's bounding circle; the fifth parameter specifies the sprite's initial position, and the sixth parameter specifies the sprite's initial velocity. This constructor does four things. First, it initializes some of the sprite's data members. It also loads the image and mask bitmaps given the resource IDs. Additionally, it obtains the corresponding `BITMAP` structures for both the image and mask bitmaps. Finally, it verifies that the image bitmap dimensions equal the mask bitmap dimensions. Here is the implementation:

```
Sprite::Sprite(HINSTANCE hAppInst, int imageID, int maskID,
                   const Circle& bc, const Vec2& p0, const Vec2& v0)
{
      mhAppInst = hAppInst;

      // Load the bitmap resources.
      mhImage = LoadBitmap(hAppInst, MAKEINTRESOURCE(imageID));
      mhMask  = LoadBitmap(hAppInst, MAKEINTRESOURCE(maskID));

      // Get the BITMAP structure for each of the bitmaps.
      GetObject(mhImage, sizeof(BITMAP), &mImageBM);
      GetObject(mhMask,  sizeof(BITMAP), &mMaskBM);

      // Image and Mask should be the same dimensions.
      assert(mImageBM.bmWidth  == mMaskBM.bmWidth);
      assert(mImageBM.bmHeight == mMaskBM.bmHeight);

      mBoundingCircle = bc;
      mPosition       = p0;
      mVelocity       = v0;
}
```

2. `~Sprite();`

   The destructor is responsible for deleting any resources we allocated in the constructor. The only resources we allocated were the bitmaps, and so we delete those in the destructor:

```
Sprite::~Sprite()
{
      // Free the resources we created in the constructor.
      DeleteObject(mhImage);
      DeleteObject(mhMask);
}
```

3. `int width();`

   This method returns the width, in pixels, of the sprite.

```
int Sprite::width()
{
      return mImageBM.bmWidth;
}
```

4. `int height();`

   This method returns the height, in pixels, of the sprite.

```
int Sprite::height()
{
      return mImageBM.bmHeight;
}
```

5. `void update(float dt);`

The `update` function essentially does what we did in the tank sample for the bullet. That is, it displaces the sprite's position by some small displacement vector, given the sprite velocity (data member) and a small change in time (`dt`).

```
void Sprite::update(float dt)
{
      // Update the sprites position.
      mPosition += mVelocity * dt;

      // Update bounding circle, too.  That is, the bounding
      // circle moves with the sprite.
      mBoundingCircle.c = mPosition;
}
```

6. `void draw(HDC hBackBufferDC, HDC hSpriteDC);`

This function draws the sprite as discussed in Section 17.4.1. This method takes two parameters. The first is a handle to the backbuffer device context, which we will need to render onto the backbuffer. The second is a handle to a second system memory device context with which we will associate our sprites. Remember, everything in GDI must be done through a device context. In order to draw a sprite bitmap onto the backbuffer, we need a device context associated with the sprite, as well as the backbuffer.

```
void Sprite::draw(HDC hBackBufferDC, HDC hSpriteDC)
{
      // The position BitBlt wants is not the sprite's center
      // position; rather, it wants the upper-left position,
      // so compute that.

      int w = width();
      int h = height();

      // Upper-left corner.
      int x = (int)mPosition.x - (w / 2);
      int y = (int)mPosition.y - (h / 2);

      // Note: For this masking technique to work, it is assumed
      // the backbuffer bitmap has been cleared to some
      // non-zero value.

      // Select the mask bitmap.
      HGDIOBJ oldObj = SelectObject(hSpriteDC, mhMask);

      // Draw the mask to the backbuffer with SRCAND.  This
      // only draws the black pixels in the mask to the backbuffer,
      // thereby marking the pixels we want to draw the sprite
      // image onto.
      BitBlt(hBackBufferDC, x, y, w, h, hSpriteDC, 0, 0, SRCAND);

      // Now select the image bitmap.
      SelectObject(hSpriteDC, mhImage);
```

```
        // Draw the image to the backbuffer with SRCPAINT.  This
        // will only draw the image onto the pixels that where previously
        // marked black by the mask.
        BitBlt(hBackBufferDC, x, y, w, h, hSpriteDC, 0, 0, SRCPAINT);

        // Restore the original bitmap object.
        SelectObject(hSpriteDC, oldObj);
}
```

17.5 Ship Animation Sample

We will now describe how to make the Ship sample, which is illustrated by the screenshot shown in Figure 17.8. In this program, the user can control only the F-15 jet. The other jets remain motionless (animating the other jets will be left as an exercise for you to complete). The user uses the ‘A’ and ‘D’ keys to move horizontally, and the ‘W’ and ‘S’ keys to move vertically. The spacebar key fires a missile. In essence, this program is much like the tank program, but with better graphics.

17.5.1 Art Resources

We require the following art assets:

- A background image with mask—Figure 17.11*a*.
- An F-15 Jet image with mask—Figure 17.11*b*.
- An F-18 Jet image with mask—Figure 17.11*c*.
- An F-117 Jet image with mask—Figure 17.11*d*.
- A missile image with mask—Figure 17.11*e*.

Observe that for the background image, the mask is all black, indicating that we want to draw the entire background image. Although this is somewhat wasteful since we do not need to mask the background, it allows us to work generally with the `Sprite` class. We will use the `Sprite` class to draw the background as well.

We use the following resource IDs for these art assets:

Background image bitmap: `IDB_BACKGROUND`
Background mask bitmap: `IDB_BACKGROUNDMASK`
F-15 image bitmap: `IDB_F15`
F-15 mask bitmap: `IDB_F15MASK`
F-18 image bitmap: `IDB_F18`
F-18 mask bitmap: `IDB_F18MASK`
F-117 image bitmap: `IDB_F117`
F-117 mask bitmap: `IDB_F117MASK`
Missile image bitmap: `IDB_BULLET`
Missile mask bitmap: `IDB_BULLETMASK`

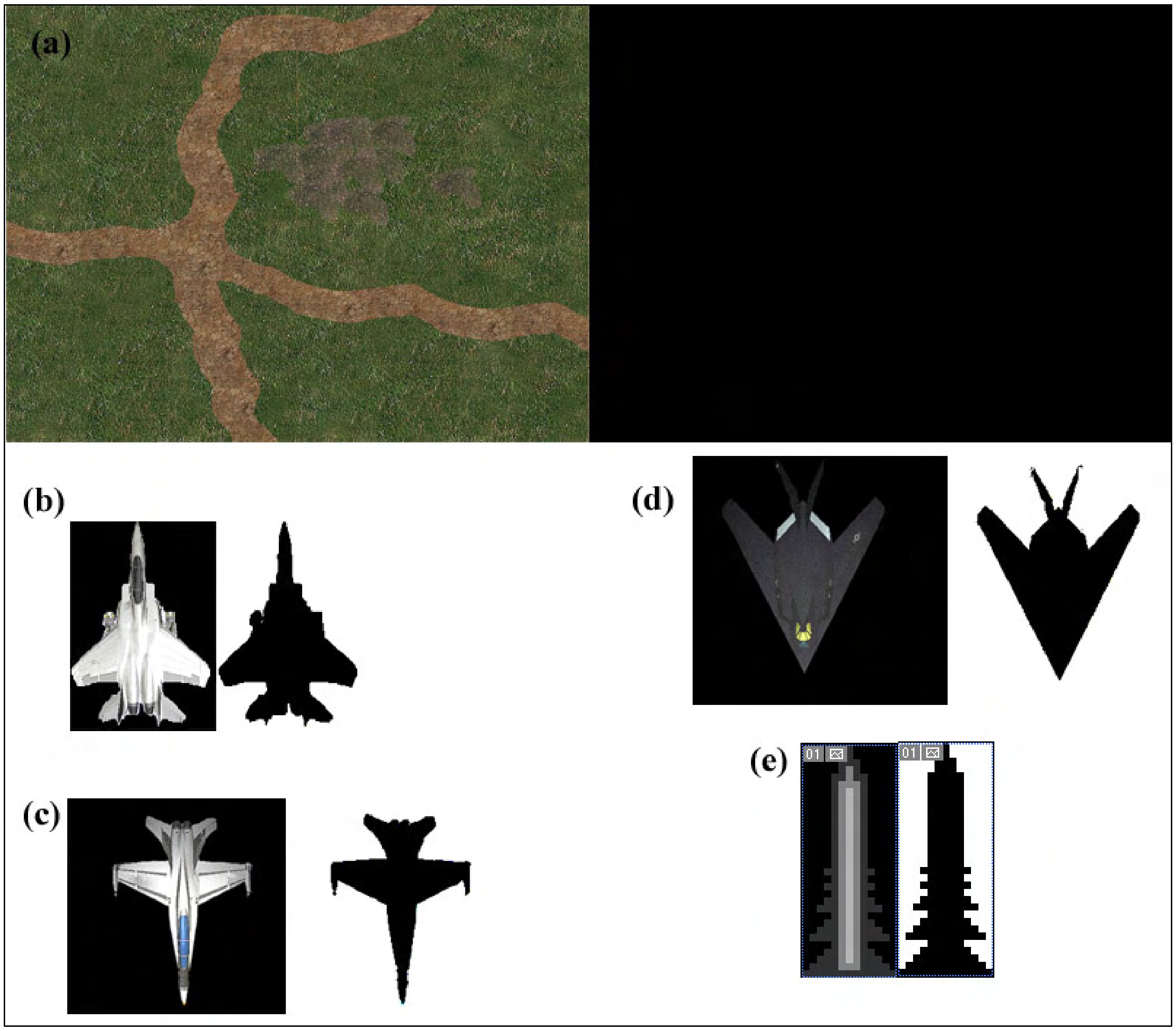


Figure 17.11: Art assets used in the Ship sample.

17.5.2 Program Code

There is not much to explain for the Ship sample. The only real difference between this sample and the Tank sample is that we are using sprites now instead of GDI shape functions. Other than that, it follows the same format. Moreover, the program is heavily commented. Therefore, we will simply present the main application code.

Program 17.2: The Ship Sample Main Application Code. You still need the other files like Sprite.h/.cpp, BackBuffer.h/.cpp, Circle.h/.cpp and Vec2.h/.cpp to compile. To obtain these files download the entire project off of the Game Institute C++ Course Website.

```
// ship.cpp
// By Frank Luna
// August 24, 2004.
```

```
//===========================================================
// Includes
//===========================================================
#include <string>
#include "resource.h"
#include "Sprite.h"
#include "BackBuffer.h"
#include <list>
using namespace std;

//===========================================================
// Globals
//===========================================================
HWND        ghMainWnd  = 0;
HINSTANCE   ghAppInst  = 0;
HMENU       ghMainMenu = 0;
HDC         ghSpriteDC = 0;

BackBuffer* gBackBuffer = 0;
Sprite*     gBackground = 0;
Sprite*     gF15        = 0;
Sprite*     gF18        = 0;
Sprite*     gF117       = 0;
Sprite*     gBullet     = 0;

list<Vec2> gBulletPos;
RECT gMapRect = {0, 0, 800, 600};

string gWndCaption = "Game Institute Ship Sample";

// Client dimensions exactly equal dimensions of
// background bitmap.  This is found by inspecting
// the bitmap in an image editor, for example.
const int gClientWidth  = 800;
const int gClientHeight = 600;

// Center point of client rectangle.
const POINT gClientCenter =
{
      gClientWidth  / 2,
      gClientHeight / 2
};

// Pad window dimensions so that there is room for window
// borders, caption bar, and menu.
const int gWindowWidth  = gClientWidth  + 6;
const int gWindowHeight = gClientHeight + 52;

//===========================================================
// Function Prototypes
//===========================================================
bool InitMainWindow();
int  Run();
void DrawFramesPerSecond(float deltaTime);
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

```
//=========================================================
// Name: WinMain
// Desc: Program execution starts here.
//=========================================================
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      ghAppInst = hInstance;

      // Create the main window.
      if( !InitMainWindow() )
      {
            MessageBox(0, "Window Creation Failed.", "Error", MB_OK);
            return 0;
      }

      // Enter the message loop.
      return Run();
}

//=========================================================
// Name: InitMainWindow
// Desc: Creates the main window upon which we will
//       draw the game graphics onto.
//=========================================================
bool InitMainWindow()
{
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = ghAppInst;
      wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
      wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
      wc.hbrBackground = (HBRUSH)::GetStockObject(NULL_BRUSH);
      wc.lpszMenuName  = 0;
      wc.lpszClassName = "MyWndClassName";

      RegisterClass( &wc );

      // WS_OVERLAPPED | WS_SYSMENU: Window cannot be resized
      // and does not have a min/max button.
      ghMainMenu = LoadMenu(ghAppInst, MAKEINTRESOURCE(IDR_MENU));
      ghMainWnd = ::CreateWindow("MyWndClassName",
            gWndCaption.c_str(), WS_OVERLAPPED | WS_SYSMENU,
            200, 200, gWindowWidth, gWindowHeight, 0,
            ghMainMenu, ghAppInst, 0);

      if(ghMainWnd == 0)
      {
            ::MessageBox(0, "CreateWindow - Failed", 0, 0);
            return 0;
      }

      ShowWindow(ghMainWnd, SW_NORMAL);
      UpdateWindow(ghMainWnd);

      return true;
}

//=========================================================
// Name: Run
// Desc: Encapsulates the message loop.
//=========================================================
int Run()
{
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));

      // Get the current time.
      float lastTime = (float)timeGetTime();

      while(msg.message != WM_QUIT)
      {
            // IF there is a Windows message then process it.
            if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
            // ELSE, do game stuff.
            else
            {
                  // Get the time now.
                  float currTime  = (float)timeGetTime();

                  // Compute the differences in time from the last
                  // time we checked.  Since the last time we checked
                  // was the previous loop iteration, this difference
                  // gives us the time between loop iterations...
                  // or, I.e., the time between frames.
                  float deltaTime = (currTime - lastTime)*0.001f;

                  // Clamp speed to 100 units per second.
                  if(gF15->mVelocity.length() > 100.0f)
                        gF15->mVelocity.normalize() *= 100.0f;

                  // Update ship.
                  gF15->update(deltaTime);

                  // Make sure F15 stays in the map boundary.
                  if( gF15->mPosition.x < gMapRect.left )
                  {
                        gF15->mPosition.x = (float)gMapRect.left;
                        gF15->mVelocity.x = 0.0f;
                        gF15->mVelocity.y = 0.0f;
                  }
                  if( gF15->mPosition.x > gMapRect.right )
                  {
                        gF15->mPosition.x = (float)gMapRect.right;
                        gF15->mVelocity.x = 0.0f;
                        gF15->mVelocity.y = 0.0f;
                  }
                  if( gF15->mPosition.y < gMapRect.top )
                  {
                        gF15->mPosition.y = (float)gMapRect.top;
                        gF15->mVelocity.x = 0.0f;
                        gF15->mVelocity.y = 0.0f;
                  }
                  if( gF15->mPosition.y > gMapRect.bottom )
                  {
                        gF15->mPosition.y = (float)gMapRect.bottom;
                        gF15->mVelocity.x = 0.0f;
                        gF15->mVelocity.y = 0.0f;
                  }

                  // Draw objects.
                  gBackground->draw(gBackBuffer->getDC(), ghSpriteDC);
                  gF15->draw(gBackBuffer->getDC(), ghSpriteDC);
                  gF18->draw(gBackBuffer->getDC(), ghSpriteDC);
                  gF117->draw(gBackBuffer->getDC(), ghSpriteDC);

                  list<Vec2>::iterator i = gBulletPos.begin();
                  while( i != gBulletPos.end() )
                  {
                        Vec2 bulletVelocity(0.0f, -300.0f);

                        // Update the position.
                        *i += bulletVelocity * deltaTime;

                        POINT p = *i;

                        // Only draw bullet if it is still inside
                        // the map boundaries.
                        // Otherwise, delete it.
                        if( !PtInRect(&gMapRect, p) )
                              i = gBulletPos.erase(i);
                        else
                        {
                              gBullet->mPosition = *i;
                              gBullet->draw(
                                    gBackBuffer->getDC(),
                                    ghSpriteDC);
                              ++i; // Next in list.
                        }
                  }

                  DrawFramesPerSecond(deltaTime);

                  // Now present the backbuffer contents to the main
                  // window client area.
                  gBackBuffer->present();

                  // We are at the end of the loop iteration, so
                  // prepare for the next loop iteration by making
                  // the "current time" the "last time."
                  lastTime = currTime;

                  // Free 20 miliseconds to Windows so we don't hog
                  // the system resources.
                  Sleep(20);
             }
      }
      // Return exit code back to operating system.
      return (int)msg.wParam;
}

//=========================================================
// Name: DrawFramesPerSecond
// Desc: This function is called every frame and updates
//       the frame per second display in the main window
//       caption.
//=========================================================
void DrawFramesPerSecond(float deltaTime)
{
      // Make static so the variables persist even after
      // the function returns.
      static int   frameCnt    = 0;
      static float timeElapsed = 0.0f;
      static char  buffer[256];

      // Function called implies a new frame, so increment
      // the frame count.
      ++frameCnt;

      // Also increment how much time has passed since the
      // last frame.
      timeElapsed += deltaTime;

      // Has one second passed?
      if( timeElapsed >= 1.0f )
      {
            // Yes, so compute the frames per second.
            // FPS = frameCnt / timeElapsed, but since we
            // compute only when timeElapsed = 1.0, we can
            // reduce to:
            // FPS = frameCnt / 1.0 = frameCnt.

            sprintf(buffer, "--Frames Per Second = %d", frameCnt);

            // Add the frames per second string to the main
            // window caption--that is, we'll display the frames
            // per second in the window's caption bar.
            string newCaption = gWndCaption + buffer;

            // Now set the new caption to the main window.
            SetWindowText(ghMainWnd, newCaption.c_str());

            // Reset the counters to prepare for the next time
            // we compute the frames per second.
            frameCnt    = 0;
            timeElapsed = 0.0f;
      }
}
```

```
//=========================================================
// Name: WndProc
// Desc: The main window procedure.
//=========================================================

LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      Circle bc; // Not needed in this demo, leave default.
      Vec2   p0(gClientCenter);
      Vec2   v0(0.0f, 0.0f);

      switch( msg )
      {
      // Create application resources.
      case WM_CREATE:

            // Create the sprites
            gBackground = new Sprite(ghAppInst,
                  IDB_BACKGROUND, IDB_BACKGROUNDMASK, bc, p0, v0);

            gF15 = new Sprite(ghAppInst, IDB_F15, IDB_F15MASK,
                  bc, p0, v0);

            p0.x = 100;
            p0.y = 100;
            gF18 = new Sprite(ghAppInst, IDB_F18, IDB_F18MASK,
                  bc, p0, v0);

            p0.x = 600;
            p0.y = 100;
            gF117 = new Sprite(ghAppInst, IDB_F117, IDB_F117MASK,
                  bc, p0, v0);

            p0.x = 0.0f;
            p0.y = 0.0f;
            gBullet = new Sprite(ghAppInst, IDB_BULLET, IDB_BULLETMASK,
                  bc, p0, v0);


            // Create system memory DCs
            ghSpriteDC = CreateCompatibleDC(0);

            // Create the backbuffer.
            gBackBuffer = new BackBuffer(
                  hWnd,
                  gClientWidth,
                  gClientHeight);

            return 0;

      case WM_COMMAND:
            switch(LOWORD(wParam))
            {
            // Destroy the window when the user selects the 'exit'
            // menu item.
            case ID_FILE_EXIT:
                  DestroyWindow(ghMainWnd);
                  break;
            }
            return 0;

      case WM_KEYDOWN:
            switch(wParam)
            {
            // Accelerate left.
            case 'A':
                  gF15->mVelocity.x -= 5.0f;
                  break;
            // Accelerate right.
            case 'D':
                  gF15->mVelocity.x += 5.0f;
                  break;
            // Accelerate up (remember +y goes down and -y goes up)
            case 'W':
                  gF15->mVelocity.y -= 5.0f;
                  break;
            // Accelerate down.
            case 'S':
                  gF15->mVelocity.y += 5.0f;
                  break;
            case VK_SPACE:
                  // Add a bullet to the bullet list.
                  gBulletPos.push_back(gF15->mPosition);
                  break;
            }
            return 0;

      // Destroy application resources.
      case WM_DESTROY:
            delete gBackground;
            delete gF15;
            delete gF18;
            delete gF117;
            delete gBullet;
            delete gBackBuffer;
            DeleteDC(ghSpriteDC);
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
      // default window procedure.
      return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

17.6 Summary

1. The Windows media library (#include <mmsystem.h> and link winmm.lib) provides a timer function called `timeGetTime`. This function returns the number of *milliseconds* that has elapsed since Windows started. In order to use this function, you will need to link winmm.lib, which is not linked by default. We can convert a time t_{ms} ms in milliseconds to a time t_s s in seconds by multiplying t_{ms} ms by the ratio 1 s / 1000 ms. The number of *milliseconds* that has elapsed since Windows started is not particularly interesting, but what is interesting is that if we call this function at time t_0, and then again at a later time t_1, we can compute the elapsed time Δt (change in time) as follows: $\Delta t = t_1 - t_0$. This is used to determine how much time has elapsed per frame.

2. A frame is one complete image in an animation sequence. For example, a game running at sixty frames per second is displaying sixty slightly different images per second. The rapid successive display of slightly different images over time gives the illusion of a smooth and continuous animation.

3. We create a new message loop, which is more game-friendly, called the game loop. This loop focuses less on Windows messages and more on processing game tasks. It can be summarized like so: If there is a Windows message that needs to be processed then process it, otherwise, execute our game code, which will typically be updating and drawing a frame. In order to implement this behavior we need a function to determine if there is a message that needs to be processed. The Win32 API provides such a function called `PeekMessage`, which returns true if there is a message that needs to be processed. Otherwise, it returns false.

4. Flickering occurs when you draw one thing and then draw something over it quickly afterwards. The process of drawing over a previously drawn pixel is called overdraw**.** The eye, which is quite sensitive to change and movement, picks up on overdraw and a flickering phenomenon is seen. To solve the flickering problem we use a double buffering technique. The idea of double buffering is quite simple: instead of drawing directly to the client area of the window, we instead draw to an off screen surface called the backbuffer, which will be a blank bitmap that will represent our virtual "canvas." Since the backbuffer is not directly visible, overdraw is not a problem. That is, we will not see the overdraw as it happens because we cannot see the backbuffer—we are no longer watching the image get "painted." Once we have drawn all the graphics to the backbuffer, we will have a completed 2D image on the backbuffer. We can then copy the backbuffer image contents to the window client area all at once with the `BitBlt` function. Since the backbuffer contains the completed frame, the only thing we will need to draw to the window client area is the contents of the backbuffer. Therefore, we will have zero overdraw on the client area window, and thus avoid flicker. To keep consistent with Direct3D vocabulary, we will refer to the process of copying the backbuffer image to the window client area as presenting. For example, saying, "we *present* the backbuffer," means we copy the backbuffer to the window client area.

5. Given a velocity $\vec{v}$, the change in position of a point $\Delta\vec{p}$ (displacement) over Δt seconds is $\Delta\vec{p} = \vec{v} \cdot \Delta t$. Recall that Δt is the time elapsed between frames, thus this formula tells us how much to displace a point per frame given the velocity $\vec{v}$. To displace a point $\vec{p}$, we simply add the displacement vector to it: $\vec{p}' = \vec{p} + \Delta\vec{p}$.

6. We can rotate a point (or vector) $(x, \quad y)$, about the origin of a coordinate system by an angle θ, to a new point $(x', \quad y')$, using the following equations:

$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = y\cos(\theta) + x\sin(\theta)$$

7. We call the images we use to represent game objects "sprites" (e.g., jets, missiles, explosions, trees, characters). Sprites are not typically rectangular, but bitmaps always are. In order to not draw the part of the bitmap that does not make up the sprite, we use a mask bitmap, which essentially marks the pixels on the sprite image bitmap that should be drawn.

17.7 Exercises

17.7.1 Colors

What is the hex value for the colors: red, green, blue, yellow, cyan, magenta, black, and white? (Hint: Cyan is a mix of green and blue. Yellow is a mix of green and red. Magenta is a mix of red and blue.)

17.7.2 Draw Order

Is the order in which you draw the graphics to the backbuffer important? Explain.

17.7.3 Masking

Explain, in your own words, how a bitmap masking technique prevents the part of the bitmap that does not make up the sprite from being drawn. Also, why do you think they call the mask bitmap a "mask?"

17.7.4 Make Your Own Sprite

Draw your own 2D sprite image and its corresponding mask bitmap in an image editor. Then using the `Sprite` class, load and draw your sprite to the window.

17.7.5 Bouncing Ball

Create a ball sprite, or use an ellipse, and write a program where the ball bounces around the client area rectangle. When the ball hits the edge "wall," it should reflect off the wall in the correct direction.

17.7.6 Modify the Ship Program

Modify the Ship sample program and animate the other jets. How you move them is up to you. Perhaps oscillate their movement from side-to-side, or have them follow a circular trajectory—it is open ended.

17.7.7 Pong

Write a pong game as Figure 17.12 shows. If you need to see how this game plays, you can find an online pong game at http://www.80smusiclyrics.com/games.shtml.

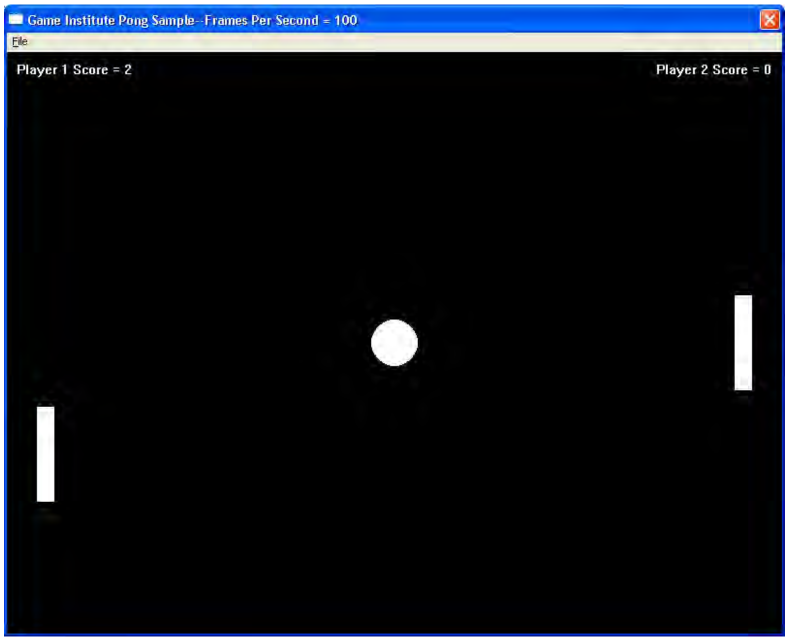


Figure 17.12: A Screenshot of a typical pong game.

17.7.8 More on Animation

In this chapter, the animation we did consisted of updating the positions of our objects, such as the tank/ship position, and bullet/missile positions, over time. However, let us recall the traditional animator's technique, where the artist actually draws each frame of an animation on a different page of paper, and where each frame is slightly different then the previous. By doing this, the artist can create an animation by rapidly flipping the pages (frames) of animation. There is no reason why we cannot do this in code as well. That is, we can load in an array of sprite images that represent the different sprite animation frames, and then for each game loop cycle, we cycle to the next sprite animation frame. For example, Figure 17.13 shows a list of explosion animation frames. When an explosion occurs, we draw the first explosion frame. On the subsequent game loop cycle, we draw the second explosion frame, and on the following game loop cycle, we draw the third explosion frame, and so on, until we have drawn all the explosion frames.

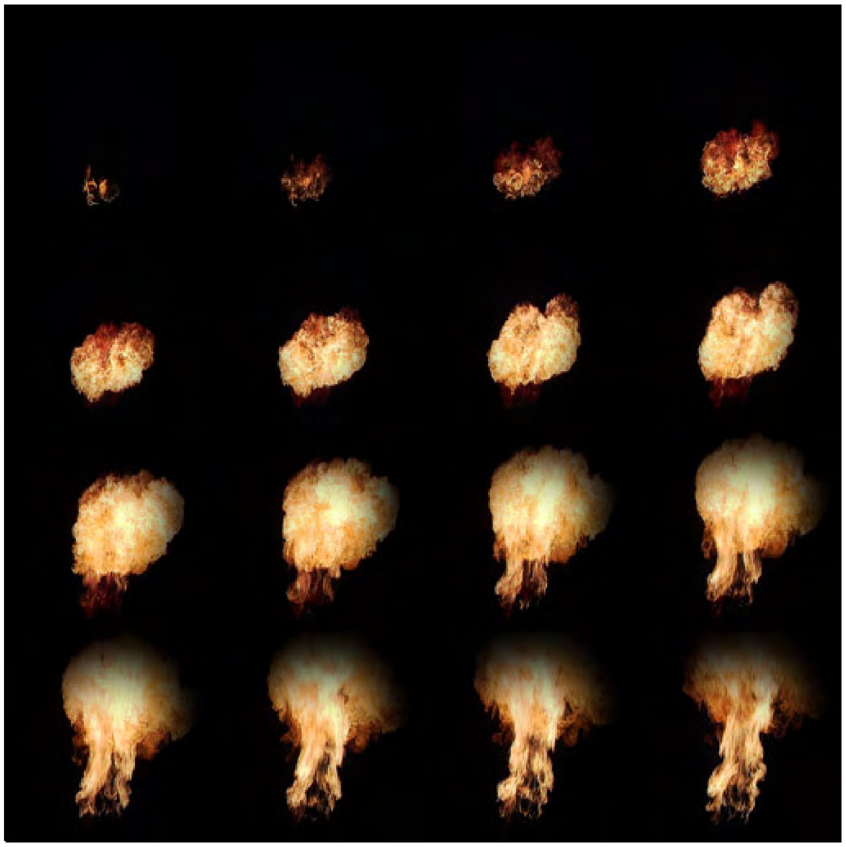


Figure 17.13: Frames of an explosion sprite animation.

Your assignment for this exercise is to find a sprite animation sequence on the Internet (or make your own if you are so inclined), and to animate it on the client area of a window. That is, to cycle through frames of animation every cycle of the game loop until the animation is over.

Implementation Hints. For each sprite animation frame, you will need a sprite. So if your entire sprite animation has 16 frames, you will represent it as:

```
Sprite* explosionAnim[16];

// Load all 16 animation frames into the elements
// of explosionAnim

// […]

int currAnim = 0; // Start at first animation frame.

// In the game loop:

// Draw the current sprite animation frame.
explosionAnim[currAnim]->draw(…);

// Draw the next animation frame next cycle
currAnim++;
```

Chapter 18

The Air Hockey Game

Introduction

After much hard work, we have finally reached the culmination of this course—the development of a simple but functional game. Figure 18.1 on the next page shows our goal: a 2D Air Hockey game. In addition to solidifying many of the course topics with this program, we also take this opportunity to introduce a three-stage software development guide:

1. Analysis: In the analysis stage, we spend some time studying the software requirements in detail. In particular, we identify which problems will need to be solved in order to implement the software with said requirements; in other words, a feature list. To aid in the identification of all problems that must be solved, a clear picture of how the program ought to run should be realized. For games, concept art and storyboards work well.

2. Design: In the design stage, we do two things. First, we devise algorithmic solutions to the problems identified in the Analysis stage. Second, we make decisions on how the actual program should be constructed; that is, what *classes* of objects do we need to model the simulation, how are the objects related, how will they interact with each other, and what kind of data structures do we need? Thus, we say that the design stage consists of an algorithmic part—that is, devising algorithmic solutions to specific problems—and a software engineering part—that is, determining how the actual program should be organized.

3. Implementation: The final stage is implementing our design in a programming language in order to actually create the software we have designed. If the design step is done correctly, the transition from design to implementation is rather smooth. In fact, the design stage is where the real work is done; that is where all the problems are solved. In theory, any competent programmer should be able to take a design specification and implement the program. Of course, reality tells a different story: unanticipated problems surface, features need to be extended and/or added, and miscalculations of software design decisions become detrimental to the implementation process. On the bright side, at least you know that you are in good company, as this happens even to the best and the brightest.

Chapter Objectives

- Become familiar with the three-stage software development process of analysis, design, and implementation.

- Practice the three-stage software development process by analyzing an Air Hockey game, designing an Air Hockey game, and implementing an Air Hockey game.

18.1 Analysis

As briefly noted in the introduction, in the Analysis stage we identify the problems that will need to be solved in order to implement the software. A clear picture of what we want the software to look like helps the Analysis stage of software development. Let us look at a picture of the Air Hockey game we want to develop.

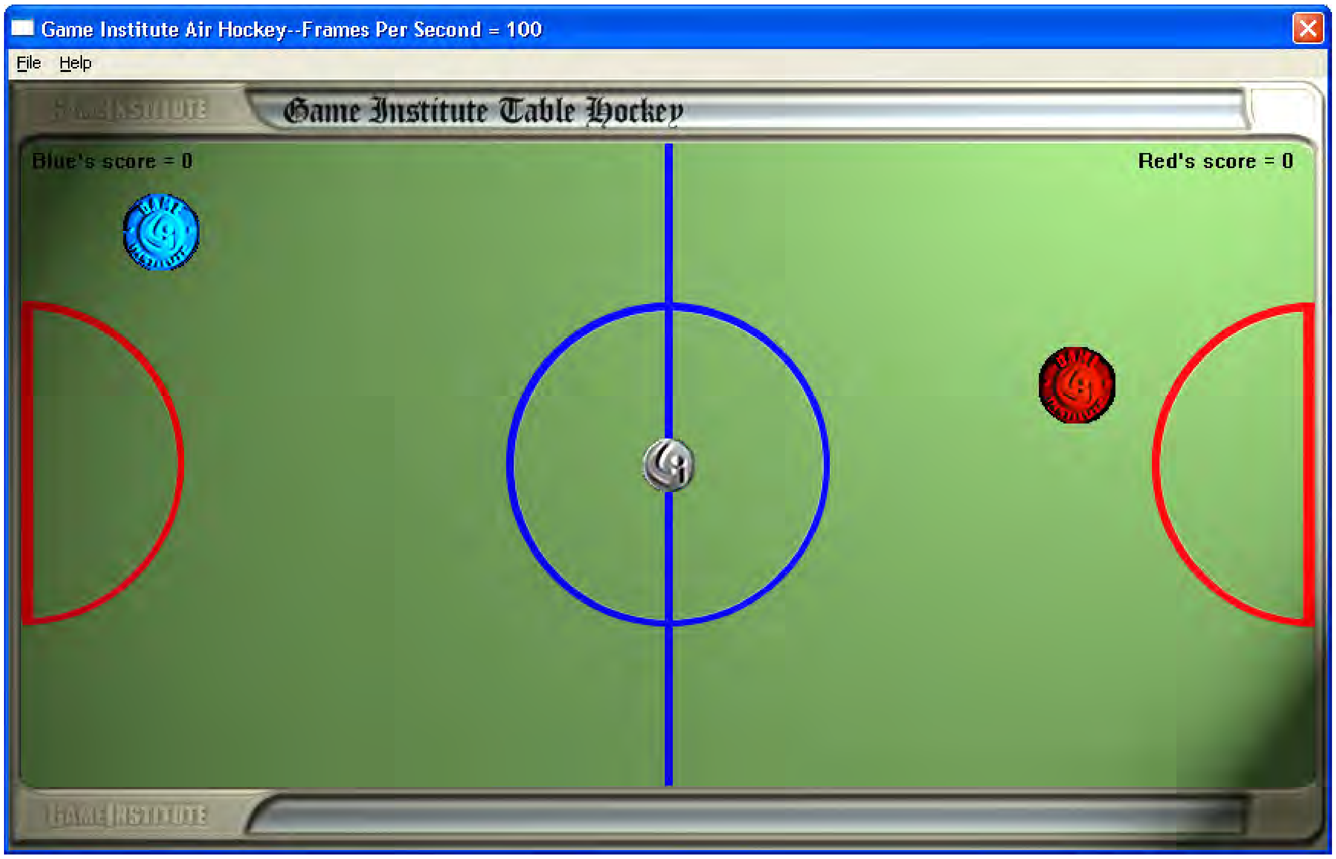

Game Institute Air Hockey--Frames Per Second = 100
File
Help
Game Institute Table Hockey
Blue's score = 0
Red's score = 0

Figure18.1: A Screenshot of the Air Hockey Game.

We proceed as follows:

1. We first identify the objects in our game.
2. We then identify how the game requires these objects to behave.
3. Finally, we identify the specific problems that will need to be solved in order to implement the stated game requirements.

18.1.1 Object Identification

We identify eight game objects in our Air Hockey simulation.

- The game board, which defines the boundaries for the game action. All game play must occur inside the game board boundaries.

- The blue paddle, which is the instrument the player controls in order to hit the puck.

- The red paddle, which is the instrument the computer controls in order to hit the puck.

- The puck, which is the object each player tries to hit into the other player's goal box.

- The blue paddle's "side". That is, the rectangular boundaries that the blue paddle must stay in. In particular, we stipulate that the blue paddle must stay in the game board boundaries and, additionally, cannot cross the center blue line.

- The red paddle's "side". That is, the rectangular boundaries that the red paddle must stay in. In particular, we stipulate that the red paddle must stay in the game board boundaries and, additionally, cannot cross the center blue line.

- The blue player's goal box.

- The red player's goal box.

18.1.2 Game Behavior and Corresponding Problems to Solve

We now identify seven game features and the corresponding problem for each that will need to be solved in order to implement the feature. Note that we are using the word "feature" in the broadest sense of the word; that is, we are not referring to special features, but rather general game behavioral features; in other words, how the game ought to behave.

- **Blue Paddle Motion.** The player (human) controls the blue paddle and we specify that it is to be moved with the mouse. Consequently, it should move in the direction and speed that the mouse cursor position moves.

    Problem: We need to be able to determine the velocity of the mouse during any given frame of animation.

- **Red Paddle Motion.** The computer (artificial intelligence) controls the red paddle.

    Problem: We need to be able to create fairly convincing artificial intelligence for the red paddle so that it behaves in a manner that roughly simulates a human player.

- **Puck / Paddle Collision.** When a paddle, be it blue or red, hits the puck, a physically realistic response should follow.

  Problem: We need to determine how two circular masses (paddle and puck) physically respond when they collide.

- **Puck / Wall Collision.** When the puck collides with a wall on the game board then it should respond in a physically realistic way. We separate this collision problem from the puck paddle collision because the objects colliding are different, and hence respond differently.

  Problem: We need to determine how an immovable line (2D wall) and a circle (the puck) physically respond when they collide.

- **Paddle / Wall Collision.** A paddle is not allowed to move outside of its "side." We defined what is meant by a player "side" in Section 18.1.1. Collision between a paddle and a wall is different than collision between a puck and wall because a player (human or computer) holds the paddle down. This implies that the player can apply external forces to the paddle to directly influence its position. Thus, when a paddle collides with any game board wall, we do not do a physical response. Instead, we simply say that the paddle can move anywhere on its "side" that the player specifies as long as it remains on its side. So if the player tries to move out-of-bounds then we only need to force the paddle inbounds.

  Problem: If a player (human or computer) attempts to move the paddle out-of-bounds we must override that action and force the paddle to stay inbounds.

- **Pause / Unpause.** The game should be able to be paused and unpaused, which stops and resumes play, respectively.

  Problem: When the player pauses the game, we must be able to stop all game activity. When the player unpauses the game, we must be able to resume all game activity from the point preceding the pause.

- **Detecting a Score.** A player scores a point when the puck enters the opponent's goal.

  Problem: We must be able to detect when a point (center of the puck) intersects a goal box (modeled as a rectangle).

18.2 Design

In the design stage, we do two things. First, we devise algorithmic solutions to the problems identified in the Analysis stage. And second, we make decisions on how the actual program should be constructed. Initially we will discuss how we will solve the problems specified in Section 18.1.2, and afterwards we will discuss how we might design our Air Hockey game from a software engineering perspective.

18.2.1 Algorithms

18.2.1.1 Mouse Velocity

Recall that velocity is a vector quantity representing speed and direction, and speed is distance per unit of time. The player controls the blue paddle with the mouse, so the velocity of the mouse directly determines the velocity of the paddle at any instant. (For us, an instant means a frame since a frame is the smallest increment of time we have.) To find the instantaneous velocity, we need to find the direction and distance the mouse has moved over the course of Δt seconds (time between the previous frame and current frame). To do this, we save the mouse position of the previous frame $\vec{r}_{i-1}$. We then obtain the mouse position for the current frame $\vec{r}_i$. Thus the mouse displaced $\vec{r}_i - \vec{r}_{i-1} = \Delta\vec{r}$ over the time period of one frame Δt (Figure 18.2). The mouse velocity for a given frame is:

$$\vec{v} = \frac{\vec{r}_i - \vec{r}_{i-1}}{\Delta t} = \frac{\Delta\vec{r}}{\Delta t}$$

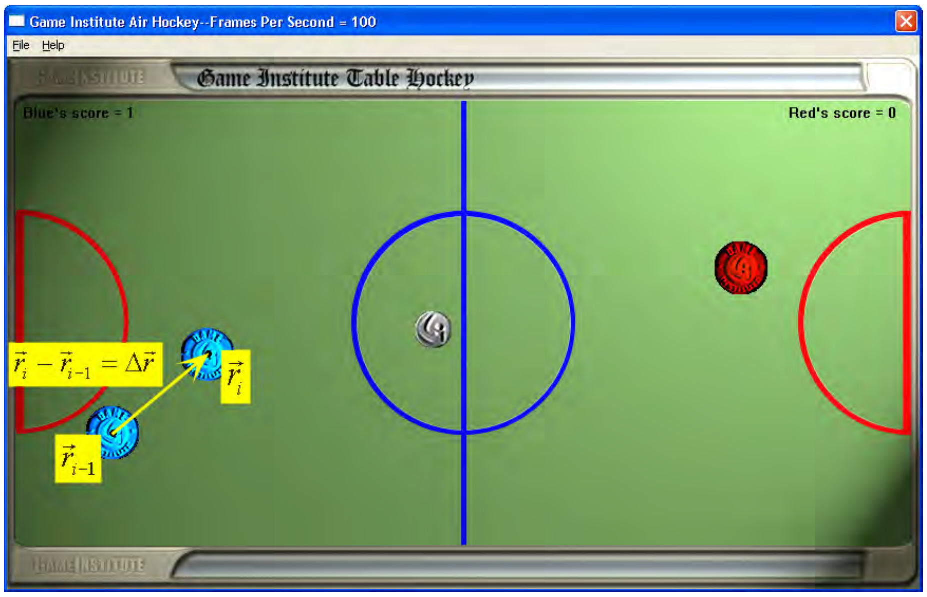

Game Institute Air Hockey--Frames Per Second = 100
File Help
Game Institute Table Hockey
Blue's score = 1
Red's score = 0
$\vec{r}_i - \vec{r}_{i-1} = \Delta\vec{r}$
$\vec{r}_i$
$\vec{r}_{i-1}$

Figure18.2: Computing the mouse displacement between frames.

18.2.1.2 Red Paddle Artificial Intelligence

We need to be able to create fairly convincing AI (artificial intelligence) for the red paddle so that it behaves in a manner that roughly simulates a human player. What we suggest is that when the puck enters the red player's side, the red paddle (computer controlled) will simply move directly towards the puck to hit it. After the red paddle hits the puck we will halt the red paddle for a tenth of a second for some "recovery time." This "recovery time" is used to model the fact that when a human air hockey player hits the puck, a human experiences a slight shock delay after impact. When the puck leaves the red player's side then the AI will move the red paddle to a "defense" position near its goal so that it is in a "ready" position. We note that the AI we create here is not very "good" AI because the "move directly toward the puck" is not the smartest strategy and often causes the red paddle to hit the puck into its own goal. However, it is satisfactory for our purposes.

Assuming the puck lies on the red player's side, in order to move the red paddle in the direction of the puck we need to compute the direction from the red paddle to the puck. This is done with a simple point subtraction followed by a vector normalization, as shown in Figure 18.3.

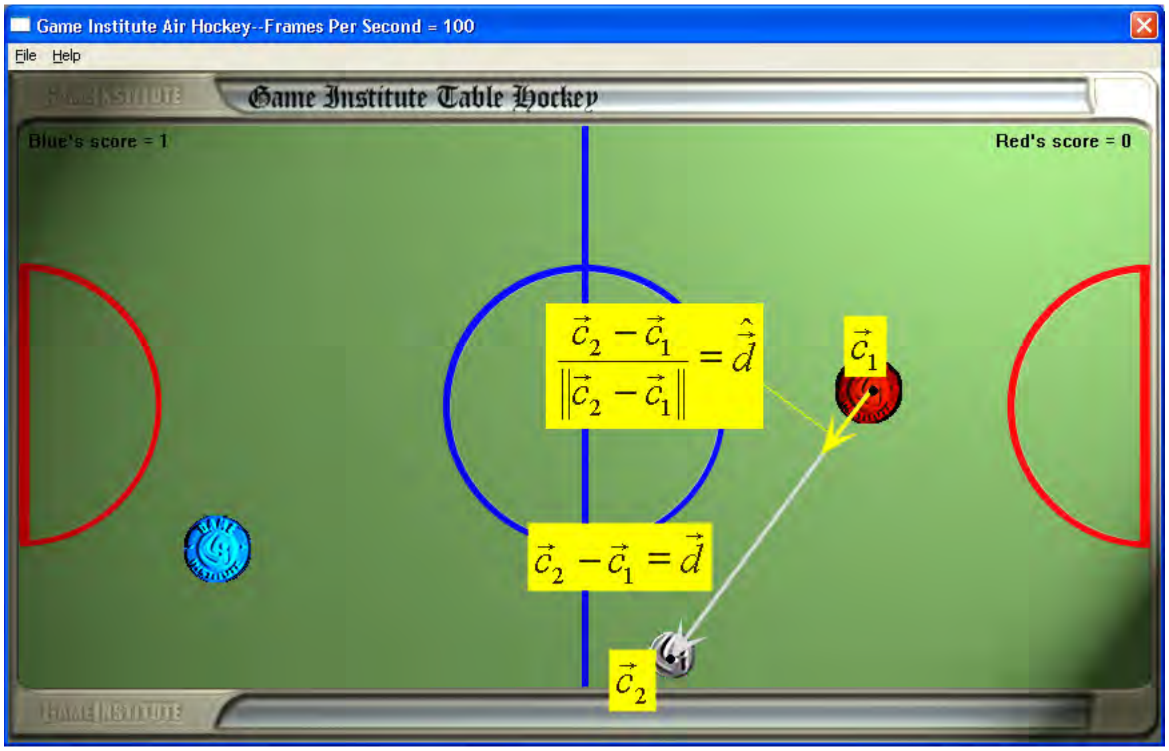

Game Institute Air Hockey--Frames Per Second = 100
File
Help
Game Institute Table Hockey
Blue's score = 1
Red's score = 0
$\frac{\vec{c}_2-\vec{c}_1}{\|\vec{c}_2-\vec{c}_1\|}=\hat{\vec{d}}$
$\vec{c}_1$
$\vec{c}_2-\vec{c}_1=\vec{d}$
$\vec{c}_2$

Figure 18.3: Here we compute the normal vector that is aimed from the red paddle to the puck.

Here, $\hat{\vec{d}}$ gives the direction, which is one part of velocity, but it does not give the magnitude (meaning the speed) of the red paddle since it is a unit vector. What we will do is define a speed constant for the red paddle. That is, the red paddle will always move at this defined speed (`RED_SPEED`). In order to give $\hat{\vec{d}}$ some speed so that it transforms from a purely directional vector to a velocity vector $\vec{v}$ (specifically the red paddle velocity) we simply scale it by `RED_SPEED`:

$\vec{v} =$ `RED_SPEED *` $\hat{\vec{d}}$

This velocity $\vec{v}$ aims in the direction of the puck and therefore will move the red paddle towards the puck at the speed `RED_SPEED`. Observe that because the puck is moving each frame, we must recalculate $\vec{v}$ every frame to get the new direction.

If the puck lies in the blue player's side, we want to move the red paddle to a "defense" position in front of its goal. The logic is the same as moving the red paddle towards the puck, but instead we now want to move the red paddle towards the defense position. Once the red paddle reaches the defense position it stops. So again we need to find the velocity that aims the red paddle towards the defense position, which incidentally we define to be (700, 200) in Windows coordinates and relative to the upper-left corner of the client area rectangle. Let $\vec{x} = \langle 700, \quad 200 \rangle$ and let $\vec{c}_1$ be the red paddle position then the direction from the red paddle to the defense position is:

$$\hat{\vec{d}} = \frac{\vec{x} - \vec{c}_1}{\left\| \vec{x} - \vec{c}_1 \right\|}$$

To transform the direction to a velocity vector, we multiply by the speed:

$\vec{v} =$ `RED_SPEED *` $\hat{\vec{d}}$

The last problem regarding the AI is to work out is how to handle the "recovery time." The solution is quite simple: when the red paddle hits the puck, we will set a variable like so:

```
recoveryTime = 0.1
```

We will then put a conditional statement in the program:

```
If( recoveryTime <= 0.0 )
    UpdateRedPaddle
Else
    RedPaddleCannotMove
```

In this way, the red paddle cannot move unless there is zero or less recovery time.

In addition, for each frame we will decrement `recoveryTime` by the time elapsed Δt. Thus, after 0.1 seconds has elapsed the recovery time will again be 0.0, which means that we can update the red paddle again. This is the exact functionality we seek. We essentially disable the red paddle for 0.1 seconds after it hits the puck in order for the paddle to "recover."

18.2.1.3 Puck Paddle Collision

When a paddle, be it blue or red, hits the puck, a physically realistic response should follow. Thus, we must be able to determine how two circular masses (paddle and puck) physically respond when they collide. The physics of this collision requires a lengthy discussion, so we have given it its own separate section in Section 18.4. What follows now is an informal conceptual explanation. A more rigorous explanation is given in Section 18.4.

Before we can even discuss anything about collisions, we need some criterion to determine if the paddle and puck collided at all. From Figure 18.4 it follows that if the length from $\vec{c}_1$ to $\vec{c}_2$ is less than or equal to the sum of the radii then the circles must intersect or touch, which implies a collision. This will be our criterion for testing whether a collision took place. In pseudocode that is,

Collision Test

```
if $\|\vec{c}_2 - \vec{c}_1\| \le R_1 + R_2$
      Intersect = true;
Else
      Intersect = false;
```

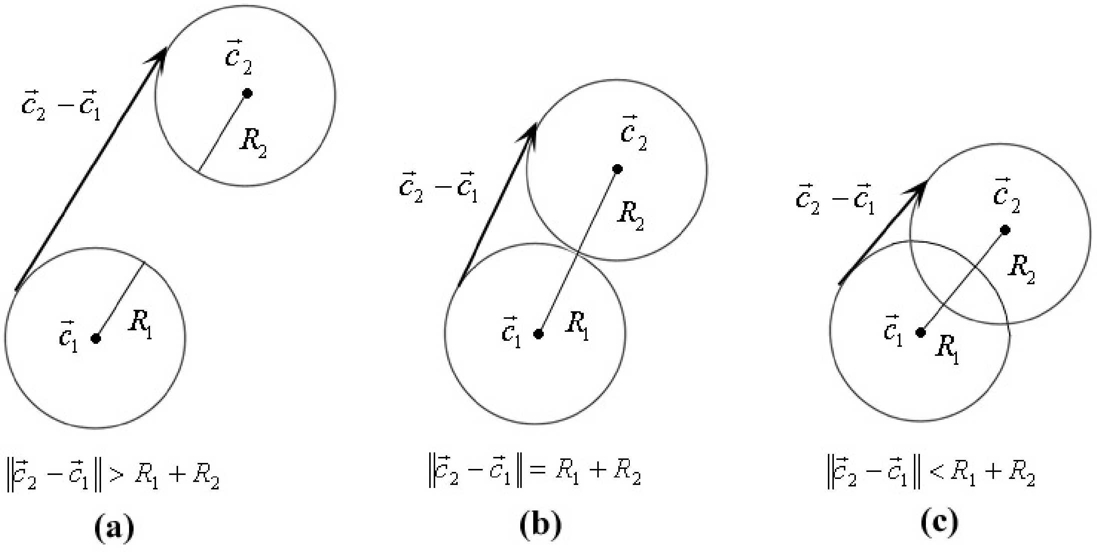
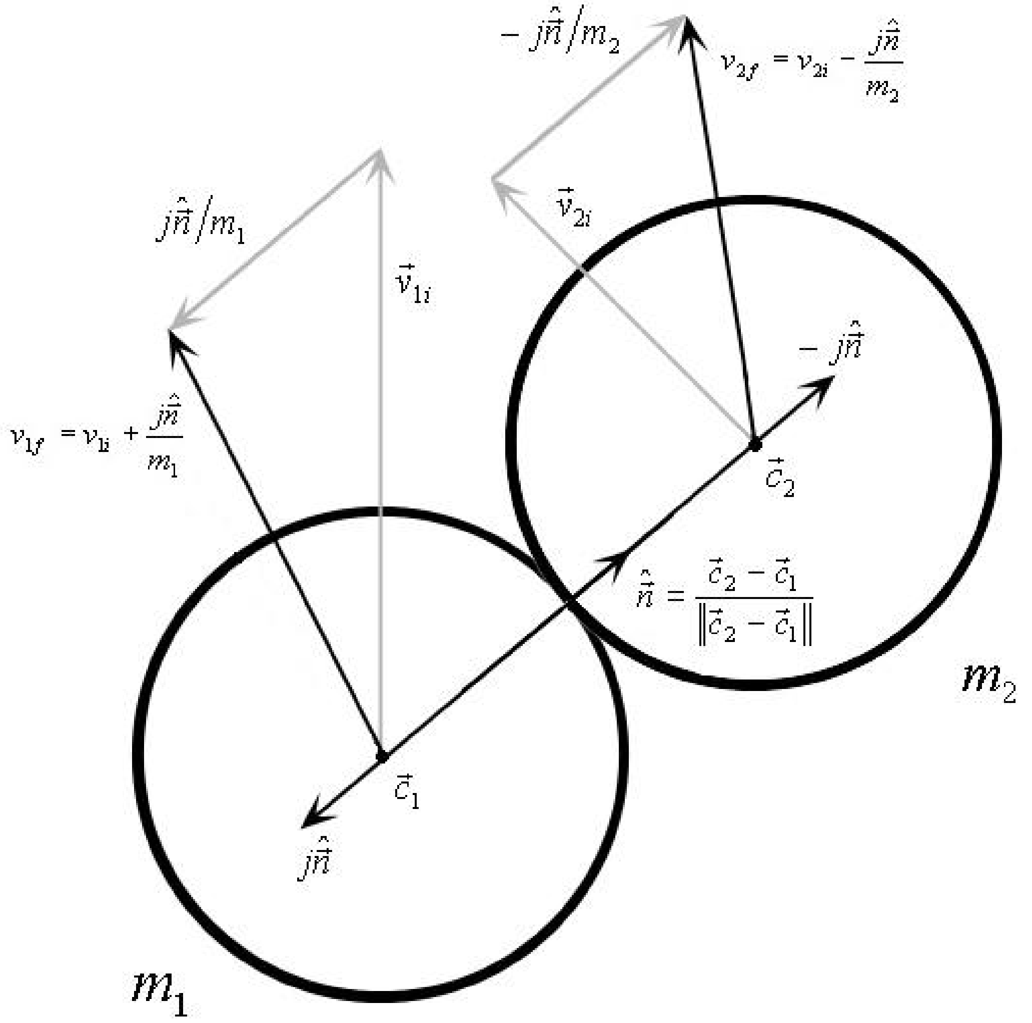


Figure 18.4: (a) The circles are not touching or interpenetrating. (b) The circles are tangents (i.e., touching). (c) The circles are interpenetrating.

Consider Figure 18.5, where two circular masses m_1 and m_2 collide. Intuitively, each object ought to "feel" an equal and opposite impulse in the direction $\hat{n}$, which is perpendicular to the collision point.

This vector $\hat{\vec{n}}$ is called the **collision normal.** Incidentally, $\hat{\vec{n}}$ can be computed by normalizing the vector that goes from $\vec{c}_1$ to $\vec{c}_2$; that is,

$$\hat{\vec{n}} = \frac{\vec{c}_2 - \vec{c}_1}{\left\|\vec{c}_2 - \vec{c}_1\right\|}$$



$-j\hat{\vec{n}}/m_2$
$v_{2f} = v_{2i} - \frac{j\hat{\vec{n}}}{m_2}$
$\vec{v}_{2i}$
$j\hat{\vec{n}}/m_1$
$\vec{v}_{1i}$
$-j\hat{\vec{n}}$
$v_{1f} = v_{1i} + \frac{j\hat{\vec{n}}}{m_1}$
$\vec{c}_2$
$\hat{\vec{n}} = \frac{\vec{c}_2 - \vec{c}_1}{\left\|\vec{c}_2 - \vec{c}_1\right\|}$
m_2
$\vec{c}_1$
$j\hat{\vec{n}}$
m_1

Figure 18.5: A collision. The impulse vectors lie on the line of the collision normal.

Here we use the word “impulse” very loosely. We give its formal physical definition in Section 18.4. For now, think of impulse as a vector quantity that changes the velocity (direction or magnitude or both) of an object in the direction of the impulse. In addition to velocities, the mass of the objects plays a role in the response. For example, if m_2 is more massive than m_1 we would expect the impulse of the collision to not effect m_2 as drastically as m_1. To compensate for the mass, we can divide the impulse vector by the mass of the object. That way, if the object is massive then the impulse vector will be less influential, and if the object is not so massive then the impulse vector will be more influential. Based on this discussion, if $j\hat{\vec{n}}$ and $-j\hat{\vec{n}}$ are the impulses (remember the impulse vectors are in the direction $\hat{\vec{n}}$ or opposite of $\hat{\vec{n}}$) and if $\vec{v}_{1i}$ and $\vec{v}_{2i}$ are the initial velocities of m_1 and m_2, respectively, before the collision, then the velocities after the collision should be computed as follows:

(1) $$v_{1f} = v_{1i} + \frac{j\hat{\vec{n}}}{m_1}$$

(2) $$v_{2f} = v_{2i} - \frac{j\hat{\vec{n}}}{m_2}$$

The reason equation (2) has a negative impulse is because we reasoned each object ought to "feel" an equal and *opposite* impulse in the direction $\hat{\vec{n}}$, the key word being "opposite". Also note that *j* is actually negative, and we compensate for that by negating the signs; that is why the impulse vectors in Figure 18.5 may seem backward. But just understand that the negative sign in the *j* will reverse them.

Let us go over equations (1) and (2) briefly. We said that the impulse would change the direction of the velocity of an object in the direction of the impulse, which summing the initial velocity with the impulse vector does (e.g., $v_{2i} + j\hat{\vec{n}}$). We also said that the mass of the object should determine how much influence the impulse vector has to change the initial velocity. We include the mass factor by dividing the impulse vector by the mass. Thus, if the mass is large then the effect of the impulse vector diminishes and if the mass is small then the effect of the impulse vector increases. Dividing the impulse vector by the mass gives us equations (1) and (2).

We still do not know what *j* is, and therefore we do not know the impulse vector (we just know its direction $\hat{\vec{n}}$ and that it has some magnitude *j*). It turns out that the magnitude of the impulse *j* is determined by the relative velocity of m_1 and m_2, $\hat{\vec{n}}$, and the masses, which is not surprising.

By **relative velocity** we mean the velocity of mass m_1 relative to mass m_2; in other words, how are the velocities moving with respect to each other? We can compute the velocity of m_1 relative to m_2 like so: $\vec{v}_{12} = \vec{v}_1 - \vec{v}_2$, where $\vec{v}_{12}$ means the velocity of mass m_1 relative to the velocity of mass m_2. (Note then that $\vec{v}_1 = \vec{v}_2 + \vec{v}_{12}$.) For example, consider the two cars in Figure (18.6*a*).

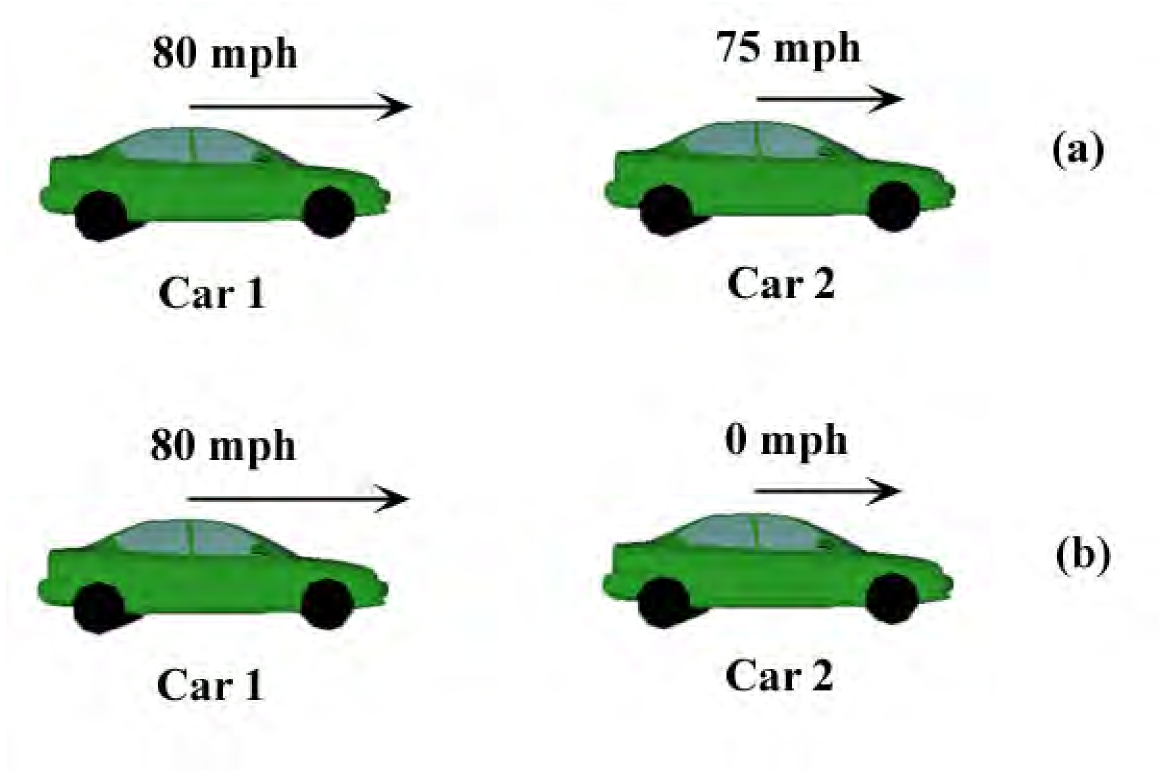


Figure 18.6: Relative velocities.

Car *1* is moving faster than Car *2* and as such, Car *1* will eventually collide with Car *2*. However, Car *1* is only going 5 miles per hour faster than Car *2*. When Car *1* hits Car *2*, will Car *2* feel an 80 miles per hour "impact?" Common sense tells us this would not be the case. Relative to Car *2*, Car *1* is only going 5 miles per hour and so Car *2* will only feel a 5 miles per hour "impact" ($\vec{v}_{12} = \vec{v}_1 - \vec{v}_2 = 80 - 75 = 5$ miles per hour).

Consider Figure (18.6*b*). Here, Car *2* is stopped, so Car *1* is going 80 miles per hour relative to Car *2*. Consequently, Car *2* will this time feel an 80 miles per hour "impact" ($\vec{v}_{12} = \vec{v}_1 - \vec{v}_2 = 80 - 0 = 80$ miles per hour). The point of this discussion is that when talking about two objects colliding, it only makes sense to talk in relative terms.

We will derive *j* in Section 18.4, but for now we will accept without proof that,

$$(3) \qquad j = \frac{-2\left(\vec{v}_{12,i} \cdot \hat{\vec{n}}\right)}{\left(\frac{1}{m_1} + \frac{1}{m_2}\right)}$$

With (3) we can rewrite (1) and (2) as,

(4) $$v_{1f} = v_{1i} + \frac{-2\left(\vec{v}_{12,i} \cdot \hat{\vec{n}}\right)\hat{\vec{n}}}{\left(\frac{m_1}{m_1} + \frac{m_1}{m_2}\right)}$$

(5) $$v_{2f} = v_{2i} - \frac{-2\left(\vec{v}_{12,i} \cdot \hat{\vec{n}}\right)\hat{\vec{n}}}{\left(\frac{m_2}{m_1} + \frac{m_2}{m_2}\right)}$$

Observe the negative sign that comes out of *j*, which was mentioned before.

We now know everything we need to solve for v_{1f} and v_{2f}; the initial velocities are given, the masses are given, the relative initial velocity is known, and $\hat{\vec{n}}$ can be found via geometric means.

Equations (4) and (5) are pretty general, but we can simplify things by looking at our specific paddle-puck collision case. Suppose m_1 is the paddle and m_2 is the puck. Because the player holds down the paddle (either human or AI), we say that the paddle is not influenced by the collision at all (a player can easily stand his/her ground against a small air hockey puck). Physically, we can view this as the paddle m_1 having a mass of infinity, which makes equation (4) reduce to:

(6) $$v_{1f} = v_{1i} + 0 = v_{1i},$$

This says that the collision does not change the velocity of the paddle.

Similarly, an infinitely massive paddle causes equation (5) to reduce to:

Collision Response of the Puck

(7) $$\boxed{v_{2f} = v_{2i} + 2\left(\vec{v}_{12,i} \cdot \hat{\vec{n}}\right)\hat{\vec{n}}}$$

Equation (7) is really the only equation we will need in our implementation. This equation will be used to compute the new velocity of the puck after it collides with a paddle.

Incidentally, if $\vec{v}_{12,i} \cdot \hat{\vec{n}} < 0$, it means that the objects m_1 and m_2 are actually already moving away from each other, which means no collision response is necessary.

18.2.1.4 Puck Wall Collision

We need to determine how an immovable line (2D wall) and a circle (the puck) physically respond when they collide. Simple real world observation tells us that if an air hockey puck is traveling with a velocity $\vec{I}$ (we call $\vec{I}$ the **incident** vector) and hits a wall, then it will *reflect* and have a new velocity $\vec{R}$—see Figure 18.7 (we approximate a zero speed loss from the bounce which is only approximately true). Observe that the reflection vector $\vec{R}$ has the same magnitude as the incident vector $\vec{I}$; the only difference is the direction is reflected. So our task at hand is, given the incoming velocity $\vec{I}$, to compute the reflected velocity $\vec{R}$. To do this, let us again examine Figure 18.7.

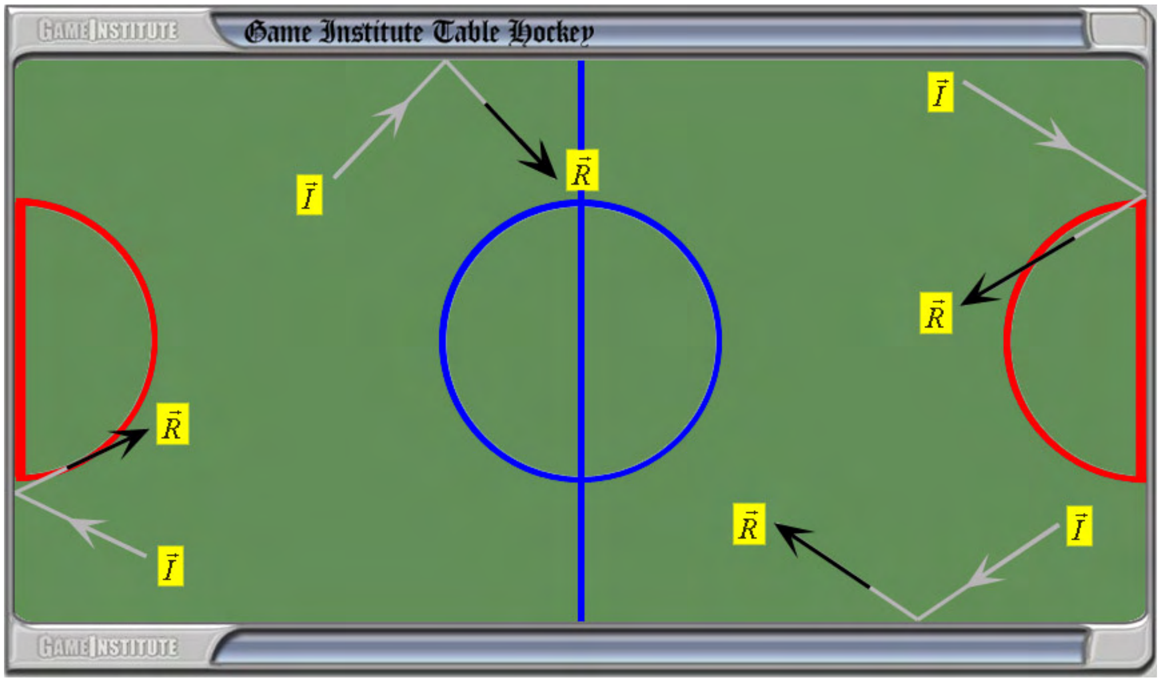


Figure 18.7: When a puck hits an edge it reflects. We assume the puck does not lose speed during the collision.

Figure 18.7 shows four cases that correspond to the four edges off of which the puck can bounce. When the puck bounces off the left or right edge, we notice that the difference between $\vec{I}$ and $\vec{R}$ is simply that that the vector x-components are opposite. Therefore, to reflect off the left or right edge we just negate the x-component of $\vec{I}$ to get $\vec{R}$. Similarly, when the puck bounces off the top or bottom edge, we notice that the difference between $\vec{I}$ and $\vec{R}$ is simply that the vector y-components are opposite. Therefore, to reflect off the top or bottom edge we just negate the y-component of $\vec{I}$ to get $\vec{R}$.

If (x, y) is the puck center point and r its radius, then we have the following puck/wall collision algorithm:

```
// Reflect velocity off left edge (if we hit the left edge).
if( x - r < left )
      mPuck->mVelocity.x *= -1.0f;
```

```
// Reflect velocity off right edge (if we hit the right edge).
if(x + r > right )
      mPuck->mVelocity.x *= -1.0f;

// Reflect velocity off top edge (if we hit the top edge).
if(y - r < top )
      mPuck->mVelocity.y *= -1.0f;

// Reflect velocity off bottom edge (if we hit the bottom edge).
if(y + r > bottom )
      mPuck->mVelocity.y *= -1.0f;
```

18.2.1.5 Paddle Wall Collision

If a player (human or computer) attempts to move the paddle out-of-bounds, we must override that action and force the paddle to stay inbounds. To solve this problem we will define a rectangle that surrounds each player's side—see Figure 18.8.

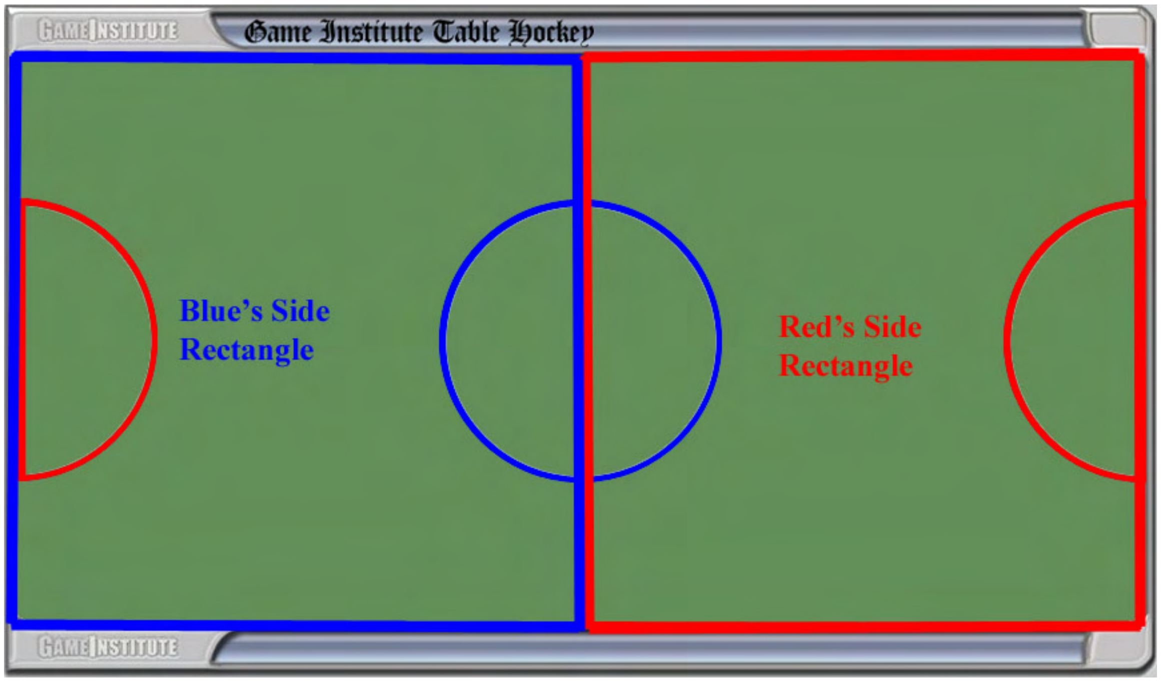


Figure 18.8: The rectangles marking the blue "side" and red "side."

If the paddle goes outside the rectangle, we will simply force it back in. A circle with radius *r* and center point (*x*, *y*) (modeling a paddle) is outside a rectangle R = {left, top, right, bottom} if the following compound logical statement is true:

$$x - r < left \;\; || \;\; x + r > right \;\; || \;\; y - r < top \;\; || \;\; y + r > bottom$$

Again, we are using Windows coordinates where +y goes "down." Essentially, the above condition asks if the circle crosses any of the rectangle edges. For example, if $x - r < left$ is true then it means part (or all) of the circle is outside the left edge—see Figure 18.9.

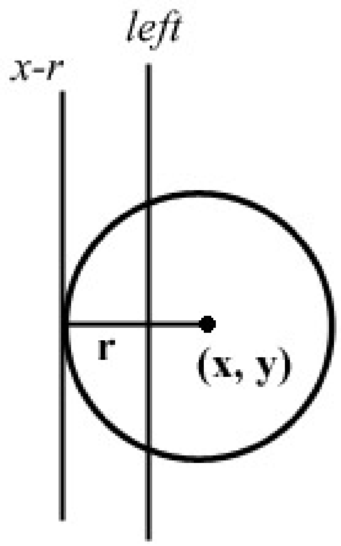


Figure 18.9: Criteria for determining if a circle crosses an edge boundary.

We can force a circle back inside the rectangle by adjusting the circle's center point based on the rectangle edge that was crossed. For instance, from Figure 18.9 it follows that the smallest x-coordinate a circle can have while still being inside the rectangle is `R.left + r`. So if the circle crosses the left edge of the rectangle we modify the circle's center x-coordinate to be `R.left + r`:

```
// Did the circle cross the rectangle's left edge?
if(p.x - r < R.left)
      p.x = R.left + r; // Yes, so modify center
```

The test and modification for the other edges is analogous:

```
// Did the circle cross the rectangle's right edge?
if(p.x + r > R.right)
      p.x = R.right - r; // Yes, so modify center

// Did the circle cross the rectangle's top edge?
if(p.y - r < R.top)
      p.y = R.top + r; // Yes, so modify center

// Did the circle cross the rectangle's bottom edge?
if(p.y + r > R.bottom)
      p.y = R.bottom - r; // Yes, so modify center
```

18.2.1.6 Pausing/Unpausing

When the player pauses the game, we must be able to stop all game activity, and when the player unpauses the game we must be able to resume all game activity from the point preceding the pause. This might seem like a difficult problem at first glance, but it turns out to be extremely simple to handle. We keep a Boolean variable `paused`, which is true if the game is paused and false otherwise. Then, when we go to update the game we can do a quick check:

```
If( not paused )
    UpdateGame
Else
    DoNotUpdateGame
```

In this way, if the game is paused, then we do not update the game. If the game is not paused, then we do update the game.

18.2.1.7 Detecting a Score

We must be able to detect when a point (center of the puck) intersects a goal box (modeled as a rectangle). This problem is similar to Section 18.2.1.5, but instead of wanting to detect when a circle goes outside a rectangle, we want to detect when a *point* goes *inside* a rectangle. (We could detect when a circle—the puck—goes inside a rectangle, but just using the point gives a good approximation for detecting a score.) A point (*x, y*) (the center of the puck) is inside a rectangle (goal box) R = {left, top, right, bottom} if and only if the following logical expression is true:

$$x >= left \ \&\& \ x <= right \ \&\& \ y >= top \ \&\& \ y <= bottom$$

This implies:

- $x >= left$: *x* is to the right of the left edge or lies on the left edge.
- $x <= right$: *x* is to the left of the right edge or lies on the right edge.
- $y >= top$: *y* is below the top edge or lies on the top edge.
- $y <= bottom$: *y* is above the bottom edge or lies on the bottom edge.

If all four of these conditions are true then we can conclude that the point is inside the rectangle or lies on the edge of a rectangle, which for our purposes we consider to be inside. Again, everything is in Windows coordinates.

18.2.2 Software Design

From the previous chapter, we already know how we are going to handle the graphics of the game. We will use double buffering and timers to provide smooth animation, and we will use sprites to draw the background game board, the paddles, and the puck.

For the paddle-puck collision detection, we will need to determine if a paddle hits the puck. Since both the paddle and puck are circular, it is natural to model these objects' areas with a circle. Thus, we shall create a `Circle` class as follows:

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include "Vec2.h"

class Circle
{
public:
      Circle();
      Circle(float R, const Vec2& center);

      bool hits(Circle& A, Vec2& normal);

      float r; // radius
      Vec2  c; // center point
};
#endif // CIRCLE_H
```

The only method this class contains other than the two constructors is the `hits` method. This method returns `true` if the circle object invoking the method hits (or intersects) the circle passed in as a parameter `A`, otherwise it returns `false`. If the two circles do hit each other then the method also returns the collision normal via the reference parameter `normal`.

In addition to circles, many of the Air Hockey game elements are rectangular; specifically, the game board, the player side regions, and the goal box. To facilitate representation of these elements we define a `Rect` class:

```
#ifndef RECT_H
#define RECT_H

#include "Circle.h"

class Rect
{
public:
      Rect();
      Rect(const Vec2& a, const Vec2& b);
      Rect(float x0, float y0, float x1, float y1);

      void forceInside(Circle& A);
```

```
	bool isPtInside(const Vec2& pt);

	Vec2 minPt;
	Vec2 maxPt;
};

#endif // RECT_H
```

You might suggest that we just use the Win32 `RECT` structure. This would work. However, we want to add some new methods, so we might as well implement our own. Moreover, we want to use `Vec2` objects for points, as their components are floating-point, whereas the coordinates of `RECT` are integers.

With `RECT` we used coordinates left, top, right, bottom to define a rectangle. But if we think about it, the point (left, top) defines the minimum point of the rectangle and the point (right, bottom) defines the maximum point of the rectangle (note we are using Windows coordinates where +y goes "down"). So instead of using left, top, right, bottom data members, we just use a `minPt` (for the minimum point) and `maxPt` (for the maximum point).

1. `void forceInside(Circle& A);`

   This method forces the `Circle A` to be inside the `Rect` object that invokes this method. This method implements the algorithm discussed in Section 18.2.1.5 for keeping the paddle inside its boundary rectangle.

2. `bool isPtInside(const Vec2& pt);`

   This method returns `true` if the point (represented as a vector) `pt` is inside the `Rect` object that invokes this method, otherwise the method returns `false`. This method implements the algorithm discussed in Section 18.2.1.7 for detecting when the puck's center point is inside the goal box rectangle.

In an effort to organize the Air Hockey program code, we have decided to encapsulate the code specific to the Air Hockey game in a separate class called `AirHockeyGame`. In this way, the Windows specific code such as window creation, the window procedure, and the message loop, will be separated from the game code (i.e., in separate files). This makes the implementation cleaner and more structured. In large projects, organization is significant, if you want to avoid getting lost in complexity. The `AirHockeyGame` class is defined like so:

```
#ifndef AIR_HOCKEY_GAME_H
#define AIR_HOCKEY_GAME_H

#include <windows.h>
#include "Sprite.h"
#include "Rect.h"
#include "Circle.h"

class AirHockeyGame
{
public:
      AirHockeyGame(HINSTANCE hAppInst, HWND hMainWnd,
            Vec2 wndCenterPt);
      ~AirHockeyGame();

      void pause();
      void unpause();

      void update(float dt);
      void draw(HDC hBackBufferDC, HDC hSpriteDC);

private:
      void updateBluePaddle(float dt);
      void updateRedPaddle(float dt);
      void updatePuck(float dt);
      bool paddlePuckCollision(Sprite* paddle);
      void increaseScore(bool blue);

private:
      HINSTANCE mhAppInst;
      HWND      mhMainWnd;
      Vec2      mWndCenterPt;

      int mBlueScore;
      int mRedScore;

      bool mPaused;

      const float MAX_PUCK_SPEED;
      const float RED_SPEED;

      float mRedRecoverTime;

      Sprite* mGameBoard;
      Sprite* mBluePaddle;
      Sprite* mRedPaddle;
      Sprite* mPuck;

      POINT mLastMousePos;
      POINT mCurrMousePos;

      Rect mBlueBounds;
      Rect mRedBounds;
      Rect mBoardBounds;
      Rect mBlueGoal;
      Rect mRedGoal;
};

#endif // AIR_HOCKEY_GAME_H
```

We begin by examining the data members.

1. `mhAppInst`: A handle to the application instance.
2. `mhMainWnd`: A handle to the main window.
3. `mWndCenterPt`: Specifies the center point of the window's client area.
4. `mBlueScore`: Integer to keep track of how many points blue has scored.

5. `mRedScore`: Integer to keep track of how many points red has scored.
6. `mPaused`: A Boolean value that is true if the game is currently paused, and false otherwise.
7. `MAX_PUCK_SPEED`: A constant that specifies the maximum speed the puck can move.
8. `RED_SPEED`: A constant that specifies the speed at which the red (AI) player moves the paddle.
9. `mRedRecoverTime`: Used to implement the recovery time idea as discussed in Section 18.2.1.2. That is, this value stores the remaining time before the red paddle can move again.
10. `mGameBoard`: The sprite that represents the game board graphic.
11. `mBluePaddle`: The sprite that represents the blue paddle graphic.
12. `mRedPaddle`: The sprite that represents the red paddle graphic.
13. `mPuck`: The sprite that represents the puck graphic.
14. `mLastMousePos`: Used to store the mouse cursor position from the previous frame.
15. `mCurrMousePos`: Used to store the mouse cursor of the current frame.
16. `mBlueBounds`: A rectangle describing the blue side.
17. `mRedBounds`: a rectangle describing the red side.
18. `mBoardBounds`: A rectangle describing the entire game board; that is, blue's side and red's side.
19. `mBlueGoal`: A rectangle describing the blue side goal box.
20. `mRedGoal`: A rectangle describing the red side goal box.

We now turn attention to the methods:

1. `void pause();`

   This method is used to pause the game.

2. `void unpause();`

   This method is used to unpause the game.

3. `void update(float dt);`

   This method updates all of the game objects; essentially it calls all of the private "helper" update methods.

4. `void draw(HDC hBackBufferDC, HDC hSpriteDC);`

   This method draws all the sprites to the backbuffer, and it also draws the current red and blue scores to the backbuffer.

5. `void updateBluePaddle(float dt);`

   This method updates the blue paddle's position after `dt` seconds have passed. It computes the current mouse velocity as Section 18.2.1.1 describes, and updates the position accordingly. In addition, it also ensures that the blue paddle stays inbounds (18.2.1.5).

6. `void updateRedPaddle(float dt);`

   This method updates the red paddle's position after `dt` seconds have passed using the AI algorithm as described in Section 18.2.1.1. In addition, it also ensures that the red paddle stays inbounds (18.2.1.5).

7. `void updatePuck(float dt);`

   This method updates the puck's position after `dt` seconds have passed. It computes the reflected velocity if the puck hits a wall (18.2.1.4). It also detects paddle-puck collisions, and if there is a collision then it computes the new velocity of the puck after the collision (18.2.1.3). In addition to making sure the puck stays within the game board boundaries, it also checks to see if the puck's center made it into one of the goal boxes (18.2.1.7); i.e., did a player score a goal?

8. `bool paddlePuckCollision(Sprite* paddle);`

   This method is called internally by `updatePuck`. This method actually performs the mathematical calculations of the paddle-puck collision (18.2.1.3). Note that this function takes a pointer to the sprite representing the paddle as a parameter. In this way, the function can be called for either the blue or red paddle.

9. `void increaseScore(bool blue);`

   This method simply increases the score of the blue player if blue is true, otherwise it increases the score of the red player. Furthermore, after increasing the score this method resets the puck to the middle of the game board.

18.3 Implementation

With our design work accomplished in the preceding section, the implementation stage is relatively straightforward.

18.3.1 Circle

The implementation to the circle class is as follows:

```
#include "Circle.h"

Circle::Circle()
: r(0.0f), c(0.0f, 0.0f)
{

}
```

```
Circle::Circle(float R, const Vec2& center)
: r(R), c(center)
{
}

bool Circle::hits(Circle& A, Vec2& normal)
{
      Vec2 u = A.c - c;

      if( u.length() <= r + A.r )
      {
            normal = u.normalize();

            // Make sure circles never overlap--at most
            // they can be tangent.
            A.c = c + (normal * (r + A.r));

            return true;
      }
      return false;
}
```

Most of this is clear; the "hit" criterion comes straight from Section 18.2.1.3 from the "Collision Test" box. One interesting thing we do, however, is if the circles are interpenetrating then we adjust the center of `A` so that they are no longer interpenetrating. The circles can become interpenetrating in code since our position updates occur in discrete steps; however, this should never happen physically. Therefore, as soon as we encounter interpenetrating circles, we modify the center of `A` to make them tangent (i.e., touching).

18.3.2 Rect

Likewise, the `Rect` class implementation is straightforward as well:

```
#include "Rect.h"

Rect::Rect()
{
}

Rect::Rect(const Vec2& a, const Vec2& b)
: minPt(a), maxPt(b)
{
}

Rect::Rect(float x0, float y0, float x1, float y1)
: minPt(x0, y0), maxPt(x1, y1)
{
}

void Rect::forceInside(Circle& A)
{
      Vec2 p  = A.c;
      float r = A.r;

      // Modify coordinates to force inside.
      if(p.x - r < minPt.x)
            p.x = minPt.x + r;
      if(p.x + r > maxPt.x)
            p.x = maxPt.x - r;

      if(p.y - r < minPt.y)
            p.y = minPt.y + r;
      if(p.y + r > maxPt.y)
            p.y = maxPt.y - r;

      // Save forced position.
      A.c = p;
}

bool Rect::isPtInside(const Vec2& pt)
{
      return pt.x >= minPt.x && pt.y >= minPt.y &&
            pt.x <= maxPt.x && pt.y <= maxPt.y;
}
```

The implementation of `forceInside` comes straight from the work done in Section 18.2.1.5, and the implementation of `isPtInside` comes straight from the work done in Section 18.2.1.7.

18.3.3 AirHockeyGame

We now present the implementation to the `AirHockeyGame` class. The implementation comes directly from our design discussion in Section 18.2. For the most part, there are no new concepts here and therefore we will just present the code. However, Figure 18.10 may help in understanding where the game rectangle coordinates came from.

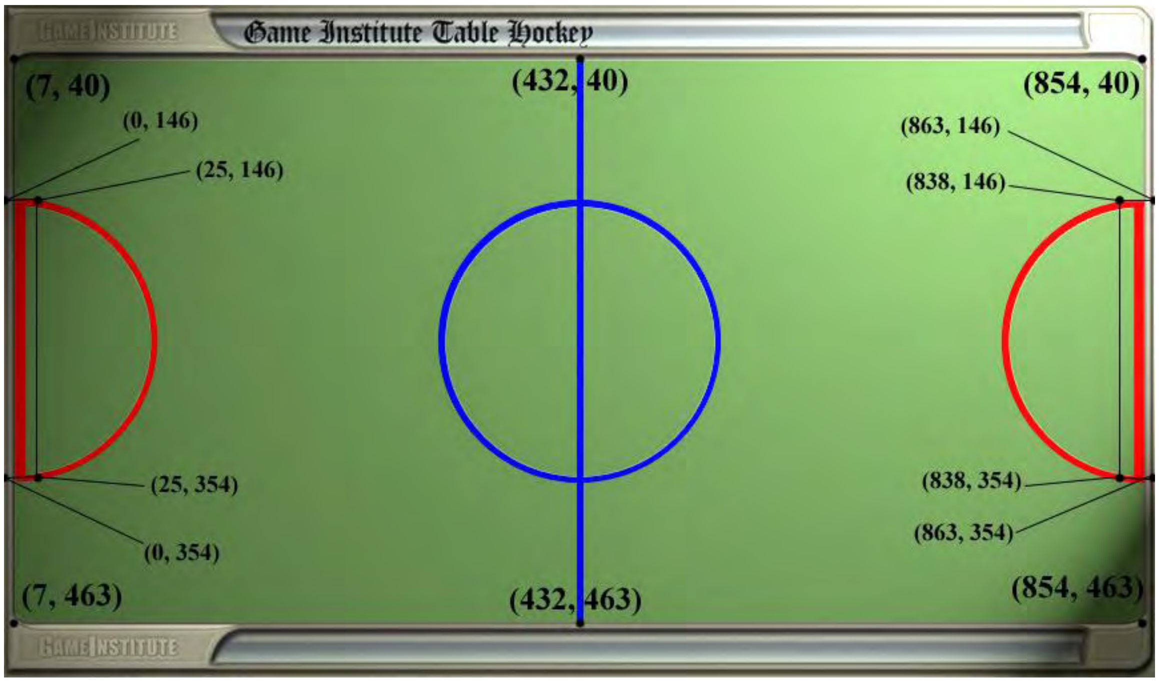


Figure 18.10: The coordinates of the various game rectangles defined.

```
#include "AirHockeyGame.h"
#include <cstdio>
#include "resource.h" // For Bitmap resource IDs

AirHockeyGame::AirHockeyGame(HINSTANCE hAppInst,
                             HWND hMainWnd,
                             Vec2 wndCenterPt)
 : MAX_PUCK_SPEED(1000.0f), RED_SPEED(300.0f)
{
      // Save input parameters.
      mhAppInst    = hAppInst;
      mhMainWnd    = hMainWnd;
      mWndCenterPt = wndCenterPt;

      // Players start game with score of zero.
      mBlueScore = 0;
      mRedScore  = 0;

      // The game is initially paused.
      mPaused    = true;

      // No recovery time for red to start.
      mRedRecoverTime = 0.0f;

      // Create the sprites:

      Circle bc;
      Vec2   p0 = wndCenterPt;
      Vec2   v0(0.0f, 0.0f);
      mGameBoard = new Sprite(mhAppInst, IDB_GAMEBOARD,
            IDB_GAMEBOARD_MASK,     bc, p0, v0);

      bc.c = p0;
      bc.r = 18.0f; // Puck radius = 18
      mPuck = new Sprite(mhAppInst, IDB_PUCK,
            IDB_PUCK_MASK, bc, p0, v0);

      p0.x = 700;
      p0.y = 200;

      bc.c = p0;
      bc.r = 25.0f; // Paddle radius = 25
      mRedPaddle = new Sprite(mhAppInst, IDB_REDPADDLE,
            IDB_PADDLE_MASK, bc, p0, v0);

      p0.x = 100;
      p0.y = 100;

      bc.c = p0;
      bc.r = 25.0f; // Paddle radius = 25
      mBluePaddle = new Sprite(mhAppInst, IDB_BLUEPADDLE,
            IDB_PADDLE_MASK, bc, p0, v0);

      // Initialize the rectangles.
      mBlueBounds  = Rect(7, 40, 432, 463);
      mRedBounds   = Rect(432, 40, 854, 463);
      mBoardBounds = Rect(7, 40, 854, 463);
      mBlueGoal    = Rect(0, 146, 25, 354);
      mRedGoal     = Rect(838, 146, 863, 354);
}

AirHockeyGame::~AirHockeyGame()
{
      delete mGameBoard;
      delete mBluePaddle;
      delete mRedPaddle;
      delete mPuck;
}

void AirHockeyGame::pause()
{
      mPaused = true;

      // Game is unpaused--release capture on mouse.
      ReleaseCapture();

      // Show the mouse cursor when paused.
      ShowCursor(true);
}

void AirHockeyGame::unpause()
{
      // Fix cursor to paddle position.
      POINT p = mBluePaddle->mPosition;
      ClientToScreen(mhMainWnd, &p);
      SetCursorPos(p.x, p.y);
      GetCursorPos(&mLastMousePos);

      mPaused = false;

      // Capture the mouse when not paused.
      SetCapture(mhMainWnd);

      // Hide the mouse cursor when not paused.
      ShowCursor(false);
}

void AirHockeyGame::update(float dt)
{
      // Only update the game if the game is not paused.
      if( !mPaused )
      {
            updateBluePaddle(dt);
            updateRedPaddle(dt);
            updatePuck(dt);

            // Decrease recovery time as time passes.
            if( mRedRecoverTime > 0.0f )
                  mRedRecoverTime -= dt;
      }
}

void AirHockeyGame::draw(HDC hBackBufferDC, HDC hSpriteDC)
{
      // Draw the sprites.
      mGameBoard->draw(hBackBufferDC, hSpriteDC);
      mBluePaddle->draw(hBackBufferDC, hSpriteDC);
      mRedPaddle->draw(hBackBufferDC, hSpriteDC);
      mPuck->draw(hBackBufferDC, hSpriteDC);


      // Draw the player scores.
      char score[32];
      sprintf(score, "Blue's score = %d", mBlueScore);

      SetBkMode(hBackBufferDC, TRANSPARENT);
      TextOut(hBackBufferDC, 15, 45, score, (int)strlen(score));

      sprintf(score, "Red's score = %d", mRedScore);
      TextOut(hBackBufferDC, 740, 45, score, (int)strlen(score));
}

void AirHockeyGame::updateBluePaddle(float dt)
{
      GetCursorPos(&mCurrMousePos);

      // Change in mouse position.
      int dx = mCurrMousePos.x - mLastMousePos.x;
      int dy = mCurrMousePos.y - mLastMousePos.y;

      Vec2 dp((float)dx, (float)dy);
      // Velocity is change in position with respect to time.
      mBluePaddle->mVelocity = dp / dt;

      // Update the blue paddle's position.
      mBluePaddle->update(dt);

      // Make sure the blue paddle stays inbounds.
      mBlueBounds.forceInside(mBluePaddle->mBoundingCircle);
      mBluePaddle->mPosition = mBluePaddle->mBoundingCircle.c;

      // The current position is now the last mouse position.
      mLastMousePos = mBluePaddle->mPosition;

      // Keep mouse cursor fixed to paddle.
      ClientToScreen(mhMainWnd, &mLastMousePos);
      SetCursorPos(mLastMousePos.x, mLastMousePos.y);
}

void AirHockeyGame::updateRedPaddle(float dt)
{
      // The red paddle's AI is overly simplistic: When the
      // puck moves into red's boundary, the red paddle
      // simply moves directly towards the puck to hit it.
      // When the puck leaves red's boundaries, the red
      // paddle returns to the center of its boundary.

      if( mRedRecoverTime <= 0.0f )
      {
            // Is the puck in red's boundary?  If yes, then
            // move the red paddle directly toward the puck.
            if( mRedBounds.isPtInside(mPuck->mPosition) )
            {
                  // Vector directed from paddle to puck.
                Vec2 redVel = mPuck->mPosition - mRedPaddle->mPosition;

                  redVel.normalize();
                  redVel *= RED_SPEED;
                  mRedPaddle->mVelocity = redVel;
            }
            // If no, then move the red paddle to the point (700, 200).
            else
            {
                  Vec2 redVel = Vec2(700, 200) - mRedPaddle->mPosition;
                  if(redVel.length() > 5.0f)
                  {
                        redVel.normalize();
                        redVel *= RED_SPEED;
                        mRedPaddle->mVelocity = redVel;
                  }
                  // Within 5 units--close enough.
                  else
                        mRedPaddle->mVelocity = Vec2(0.0f, 0.0f);
            }

            // Update the red paddle's position.
            mRedPaddle->update(dt);
            // Make sure the red paddle stays inbounds.
            mRedBounds.forceInside(mRedPaddle->mBoundingCircle);
            mRedPaddle->mPosition = mRedPaddle->mBoundingCircle.c;
      }
}

void AirHockeyGame::updatePuck(float dt)
{
      paddlePuckCollision(mBluePaddle);

      // If red hits the puck then make a small 1/10th of a second
      // delay before the red paddle can move away as sort of a
      // "recovery period" after the hit.  This is to model the
      // fact that when a human player hits something, it takes
      // a short period of time to recover from the collision.
      if(paddlePuckCollision(mRedPaddle))
            mRedRecoverTime = 0.1f;

      // Clamp puck speed to some maximum velocity; this provides
      // good stability.
      if( mPuck->mVelocity.length() >= MAX_PUCK_SPEED )
            mPuck->mVelocity.normalize() *= MAX_PUCK_SPEED;

      // Did the puck hit a wall?  If so, reflect about edge.
      Circle puckCircle = mPuck->mBoundingCircle;
      if( puckCircle.c.x - puckCircle.r < mBoardBounds.minPt.x )
            mPuck->mVelocity.x *= -1.0f;
      if( puckCircle.c.x + puckCircle.r > mBoardBounds.maxPt.x )
            mPuck->mVelocity.x *= -1.0f;
      if( puckCircle.c.y - puckCircle.r < mBoardBounds.minPt.y )
            mPuck->mVelocity.y *= -1.0f;
      if( puckCircle.c.y + puckCircle.r > mBoardBounds.maxPt.y )
            mPuck->mVelocity.y *= -1.0f;

      // Make sure puck stays inbounds of the gameboard.
      mBoardBounds.forceInside(mPuck->mBoundingCircle);
      mPuck->mPosition = mPuck->mBoundingCircle.c;

      mPuck->update(dt);

      if( mBlueGoal.isPtInside(mPuck->mPosition) )
            increaseScore(false);

      if( mRedGoal.isPtInside(mPuck->mPosition) )
            increaseScore(true);
}

bool AirHockeyGame::paddlePuckCollision(Sprite* paddle)
{
      Vec2 normal;
      if(paddle->mBoundingCircle.hits(mPuck->mBoundingCircle, normal))
      {
            // Hit updates cirle's position.  So update pucks
            // position as well since the two correspond.
            mPuck->mPosition = mPuck->mBoundingCircle.c;

            //*******************
            // Apply equation (7)
            //*******************

            // Compute the paddle's velocity relative to the puck's
            // velocity.
            Vec2 relVel = paddle->mVelocity - mPuck->mVelocity;

            // Get the component of the relative velocity along
            // the normal.
            float impulseMag = relVel.dot(normal);

            // Are the objects getting closer together?
            if( impulseMag >= 0.0f )
            {
                  // Project the relative velocity onto the normal.
                  Vec2 impulse = impulseMag * normal;

                  // Add the impulse to the puck.
                  mPuck->mVelocity += 2.0f * impulse;

                  return true;
            }
      }
      return false;
}

void AirHockeyGame::increaseScore(bool blue)
{
      if( blue )
            ++mBlueScore;
      else
            ++mRedScore;

      // A point was just scored, so reset puck to center and
      // pause game.
      mPuck->mPosition = Vec2(mWndCenterPt.x, mWndCenterPt.y);
      mPuck->mVelocity = Vec2(0.0f, 0.0f);
      mPuck->mBoundingCircle.c = Vec2(mWndCenterPt.x, mWndCenterPt.y);

      // After score, pause the game so player can prepare for
      // next round.
      pause();
}
```

18.3.4 Main Application Code

Finally, we show the main application code, which creates the window, implements the window procedure, and enters the message loop. We have seen most of this code before.

```
// giAirHockey.cpp
// By Frank Luna
// August 24, 2004.
```

```
//===========================================================
// Includes
//===========================================================

#include <string>
#include "resource.h"
#include "AirHockeyGame.h"
#include "BackBuffer.h"
using namespace std;

//===========================================================
// Globals
//===========================================================

HWND        ghMainWnd  = 0;
HINSTANCE   ghAppInst  = 0;
HMENU       ghMainMenu = 0;
HDC         ghSpriteDC = 0;

BackBuffer*    gBackBuffer = 0;
AirHockeyGame* gAirHockey  = 0;

string gWndCaption = "Game Institute Air Hockey";

// Client dimensions exactly equal dimensions of
// background bitmap.  This is found by inspecting
// the bitmap in an image editor, for example.
const int gClientWidth  = 864;
const int gClientHeight = 504;

// Center point of client rectangle.
const POINT gClientCenter =
{
      gClientWidth  / 2,
      gClientHeight / 2
};

// Pad window dimensions so that there is room for window
// borders, caption bar, and menu.
const int gWindowWidth  = gClientWidth  + 6;
const int gWindowHeight = gClientHeight + 52;

//===========================================================
// Function Prototypes
//===========================================================

bool InitMainWindow();
int  Run();
void DrawFramesPerSecond(float deltaTime);

LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

BOOL CALLBACK
AboutBoxProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam);
```

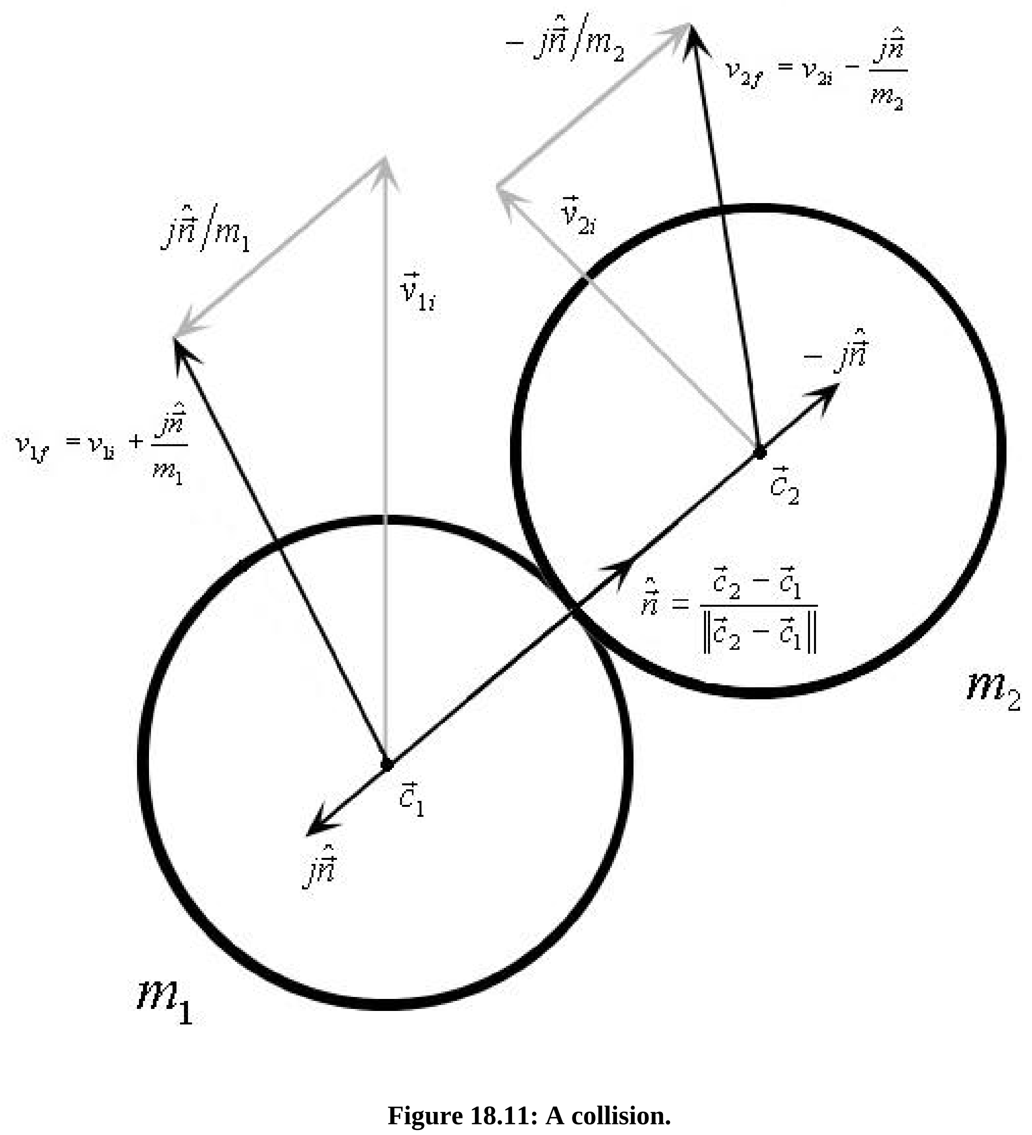
```
//=========================================================
// Name: WinMain
// Desc: Program execution starts here.
//=========================================================
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            PSTR cmdLine, int showCmd)
{
      ghAppInst = hInstance;

      // Create the main window.
      if( !InitMainWindow() )
      {
            MessageBox(0, "Window Creation Failed.", "Error", MB_OK);
            return 0;
      }

      // Enter the message loop.
      return Run();
}

//=========================================================
// Name: InitMainWindow
// Desc: Creates main window for drawing game graphics
//=========================================================
bool InitMainWindow()
{
      WNDCLASS wc;
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = ghAppInst;
      wc.hIcon         = ::LoadIcon(0, IDI_APPLICATION);
      wc.hCursor       = ::LoadCursor(0, IDC_ARROW);
      wc.hbrBackground = (HBRUSH)::GetStockObject(NULL_BRUSH);
      wc.lpszMenuName  = 0;
      wc.lpszClassName = "MyWndClassName";

      RegisterClass( &wc );

      // WS_OVERLAPPED | WS_SYSMENU: Window cannot be resized
      // and does not have a min/max button.
      ghMainMenu = LoadMenu(ghAppInst, MAKEINTRESOURCE(IDR_MENU));
      ghMainWnd = ::CreateWindow("MyWndClassName",
            gWndCaption.c_str(), WS_OVERLAPPED | WS_SYSMENU,
            200, 200, gWindowWidth, gWindowHeight, 0,
            ghMainMenu, ghAppInst, 0);

      if(ghMainWnd == 0){
            ::MessageBox(0, "CreateWindow - Failed", 0, 0);
            return 0;
      }

      ShowWindow(ghMainWnd, SW_NORMAL);
      UpdateWindow(ghMainWnd);
      return true;
}
```

```
//=========================================================
// Name: Run
// Desc: Encapsulates the message loop.
//=========================================================
int Run()
{
      MSG msg;
      ZeroMemory(&msg, sizeof(MSG));

      // Get the current time.
      float lastTime = (float)timeGetTime();

      float timeElapsed = 0.0f;

      while(msg.message != WM_QUIT)
      {
            // IF there is a Windows message then process it.
            if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
            // ELSE, do game stuff.
            else
            {
                  // Get the time now.
                  float currTime  = (float)timeGetTime();

                  // Compute the differences in time from the last
                  // time we checked.  Since the last time we checked
                  // was the previous loop iteration, this difference
                  // gives us the time between loop iterations...
                  // or, I.e., the time between frames.
             float deltaTime = max((currTime - lastTime)*0.001f, 0.0f);

                  timeElapsed += deltaTime;

                  // Only update once every 1/100 seconds.
                  if( timeElapsed >= 0.01 )
                  {
                        // Update the game and draw everything.
                        gAirHockey->update((float)timeElapsed);

                        // Draw every frame.
                        gAirHockey->draw(gBackBuffer->getDC(),
                                         ghSpriteDC);

                        DrawFramesPerSecond(timeElapsed);

                        // Now present the backbuffer contents to
                        // the main window client area.
                        gBackBuffer->present();
                        timeElapsed = 0.0;
                  }

                  // We are at the end of the loop iteration, so
                  // prepare for the next loop iteration by making
                  // the "current time" the "last time."
                  lastTime = currTime;
            }
      }
      // Return exit code back to operating system.
      return (int)msg.wParam;
}

//=========================================================
// Name: DrawFramesPerSecond
// Desc: This function is called every frame and updates
//       the frame per second display in the main window
//       caption.
//=========================================================
void DrawFramesPerSecond(float deltaTime)
{
      // Make static so the variables persist even after
      // the function returns.
      static int   frameCnt    = 0;
      static float timeElapsed = 0.0f;
      static char  buffer[256];

      // Function called implies a new frame, so increment
      // the frame count.
      ++frameCnt;

      // Also increment how much time has passed since the
      // last frame.
      timeElapsed += deltaTime;

      // Has one second passed?
      if( timeElapsed >= 1.0f )
      {
            // Yes, so compute the frames per second.
            // FPS = frameCnt / timeElapsed, but since we
            // compute only when timeElapsed = 1.0, we can
            // reduce to:
            // FPS = frameCnt / 1.0 = frameCnt.

            sprintf(buffer, "--Frames Per Second = %d", frameCnt);

            // Add the frames per second string to the main
            // window caption--that is, we'll display the frames
            // per second in the window's caption bar.
            string newCaption = gWndCaption + buffer;

            // Now set the new caption to the main window.
            SetWindowText(ghMainWnd, newCaption.c_str());

            // Reset the counters to prepare for the next time
            // we compute the frames per second.
            frameCnt    = 0;
            timeElapsed = 0.0f;
      }
}
```

```
//=========================================================
// Name: WndProc
// Desc: The main window procedure.
//=========================================================
LRESULT CALLBACK
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
      switch( msg )
      {
      // Create application resources.
      case WM_CREATE:

            // Create the hockey game.
            gAirHockey = new AirHockeyGame(
                  ghAppInst, hWnd, gClientCenter);

            // Create system memory DCs
            ghSpriteDC = CreateCompatibleDC(0);

            // Create the backbuffer.
            gBackBuffer = new BackBuffer(
                  hWnd,
                  gClientWidth,
                  gClientHeight);

            return 0;

      case WM_COMMAND:
            switch(LOWORD(wParam))
            {
            // Destroy the window when the user selects the 'exit'
            // menu item.
            case ID_FILE_EXIT:
                  DestroyWindow(ghMainWnd);
                  break;
            // Display the about dialog box when the user selects
            // the 'about' menu item.
            case ID_HELP_ABOUT:
                  DialogBox(ghAppInst, MAKEINTRESOURCE(IDD_ABOUTBOX),
                        ghMainWnd, AboutBoxProc);
                  break;
            }
            return 0;

      // Left mouse button to unpause the game.
      case WM_LBUTTONDOWN:
            gAirHockey->unpause();
            return 0;

      // Right mouse button to pause the game.
      case WM_RBUTTONDOWN:
            gAirHockey->pause();
            return 0;

      // Destroy application resources.
      case WM_DESTROY:
            delete gAirHockey;
            delete gBackBuffer;
            DeleteDC(ghSpriteDC);
            PostQuitMessage(0);
            return 0;
      }
      // Forward any other messages we didn't handle to the
      // default window procedure.
      return DefWindowProc(hWnd, msg, wParam, lParam);
}

//===========================================================
// Name: AboutBoxProc
// Desc: The window procedure of the about dialog box.
//===========================================================
BOOL CALLBACK
AboutBoxProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
      switch( msg )
      {
      case WM_INITDIALOG:
            return true;

      case WM_COMMAND:
            switch(LOWORD(wParam))
            {
            // Terminate the dialog when the user presses the
            // OK button.
            case IDOK:
                  EndDialog(hDlg, 0);
                  break;
            }
            return true;
      }
      return false;
}
```

18.4 Collision Physics Explanation (Optional)

In Section 18.2.1.3, "Puck Paddle Collision," we said we would discuss the physics of the puck-paddle collision more rigorously. Recall Figure 18.11, where two circular masses m_1 and m_2 collide. Intuitively, each object ought to "feel" an equal and opposite impulse in the direction $\hat{\vec{n}}$, which is perpendicular to the collision point. Our first task at hand is to discuss some fundamental physics principles so that we can give a precise definition to "impulse."



$-j\hat{\vec{n}}/m_2$
$v_{2f} = v_{2i} - \frac{j\hat{\vec{n}}}{m_2}$
$\vec{v}_{2i}$
$j\hat{\vec{n}}/m_1$
$\vec{v}_{1i}$
$-j\hat{\vec{n}}$
$v_{1f} = v_{1i} + \frac{j\hat{\vec{n}}}{m_1}$
$\vec{c}_2$
$\hat{\vec{n}} = \frac{\vec{c}_2 - \vec{c}_1}{\|\vec{c}_2 - \vec{c}_1\|}$
m_2
$\vec{c}_1$
$j\hat{\vec{n}}$
m_1

Figure 18.11: A collision.

Given our definition of impulse and also of linear momentum, we can derive equations (1) and (2), which we restate here:

(1) $$v_{1f} = v_{1i} + \frac{j\hat{\vec{n}}}{m_1}$$

(2) $$v_{2f} = v_{2i} - \frac{j\hat{\vec{n}}}{m_2}$$

> **Note:** Throughout this discussion we use the subscript notation i to indicate an initial value and the subscript f to indicate a final value. In the context of collisions, the initial value will mean "before the collision" and the final value will mean "after the collision."

These equations allow us to compute the final velocity of the masses after the collision, given the initial velocities before the collision and the magnitude of the impulse; that is, j.

Once we have obtained (1) and (2) we must still solve for *j*. However, in order to do so we need some more physics background. We proceed to state the law of conservation of momentum and the conservation of kinetic energy for elastic collisions. Given these two laws, we have enough relationships to solve for *j*. And once we know *j*, we will know everything we need to solve for v_{1f} and v_{2f}; the initial velocities are given, the masses are given, the initial velocities are known, and $\hat{\vec{n}}$ can be found via geometric means.

Finally, before we begin, note that the mathematics and physics may be overwhelming for students who have never taken a course in mechanics. For this reason, we have made this section optional. You should make an effort to understand this material, but if you find yourself completely lost then you can fall back on Section 18.2.1.3, which gives a more intuitive explanation. You will not be tested on any of the material in Section 18.4.

18.4.1 Linear Momentum

Definition: *A physical object has a property called* ***linear momentum*** $\vec{p}$*, which is defined to be the product of its mass m and velocity* $\vec{v}$*. That is,* $\vec{p} = m\vec{v}$*. A system of objects has a* ***total linear momentum*** $\vec{P}$*, which is the vector sum of the linear momentum of all the objects in the system. That is,* $\vec{P} = \vec{p}_1 + \vec{p}_2 + \ldots + \vec{p}_n = m_1\vec{v}_1 + m_2\vec{v}_2 + \ldots + m_n\vec{v}_n$ *for n objects in the system. (By* ***system*** *we mean the objects we are considering; for example, in our puck-paddle collision the puck and paddle are the two objects we are considering and thus form our system.)*

Conceptually, the magnitude of an object's linear momentum determines how hard it is to change the object's path of motion. For example, consider a truck driving down a steep hill. It is both massive and moving at a fairly quick velocity (at least relative to a bystander). The product of a massive object and a high speed means a large linear momentum magnitude, which means the truck's motion is not easily altered. And this makes sense intuitively; it will take a *lot* of force to change the motion of the truck.

Conversely, consider a soccer ball rolling down a slight incline. The soccer ball has a small mass and is not moving very fast, which means it has a small linear momentum magnitude, which means the soccer ball's motion is easily altered. And this also makes sense intuitively. A person can easily alter the motion of the soccer ball by giving it a kick. It is important to remember that the linear momentum is a product of both the mass and the velocity. For example, although a bullet is not very massive, when fired it is difficult to change its path because of its high velocity.

18.4.2 Newton's Second Law of Motion

Definition: *Newton's second law of motion defines the net force, at some particular time, acting on an object to be equal to its instantaneous change in linear momentum. In order to avoid calculus notation we will approximate "instantaneous" as a very short time interval* Δt*. Then,* $\vec{F} = \Delta\vec{p}/\Delta t = \left(\vec{p}_f - \vec{p}_i\right)/\Delta t$*, which reads the net force equals the change in linear momentum over a short time period* Δt*. Note that the change linear momentum* $\Delta\vec{p}$ *is just the difference in momentum from some initial momentum* $\vec{p}_i$ *to some final momentum* $\vec{p}_f$*; that is,* $\Delta\vec{p} = \vec{p}_f - \vec{p}_i$*.*

We can rewrite Newton's second law as follows:

(8) $$\vec{F} = \Delta\vec{p}/\Delta t \Rightarrow \Delta\vec{p} = \vec{F}\cdot\Delta t$$

In words, this says that the change in linear momentum is equal to the applied force times the length of time the force is applied. Intuitively this makes sense. If we apply a large (relatively speaking) force to an object for one second, then we would expect to see a large change in its linear momentum.

Conversely, if we apply a weak force for one second, then we would expect to see a small change in its linear momentum.

On the other hand, we might also apply a weak (relatively speaking) force to an object but for a long period of time—say sixty seconds. This could make the change in linear momentum large assuming that the force is not extremely weak. So we see from the relationship $\Delta\vec{p} = \vec{F}\cdot\Delta t$ that both the force strength and the duration for which it is applied determines the change in linear momentum.

18.4.3 Impulse Defined

Definition: *When a constant force* $\vec{F}$ *(may actually be a varying force but we do not need to handle such complexities here so we simplify by making the force constant) is applied to an object, over some time* Δt*, it changes the linear momentum of the object (*$\vec{F}\cdot\Delta t = \Delta\vec{p}$*). In the context of collisions, we call this change in momentum the* ***impulse*** *and denote it with the symbol* $\vec{J}$*. That is,* $\vec{J} = \vec{F}\cdot\Delta t = \Delta\vec{p}$*.*

Essentially, impulse is just a special name given to the change in linear momentum due to a collision, which implies a strong force for a short amount of time.

Given the initial linear momentum $\vec{p}_i = m\vec{v}_i$ of an object before a collision and the impulse $\vec{J}$ it undergoes during the collision, we can find the final linear momentum $\vec{p}_f = m\vec{v}_f$ of the object after the collision using basic algebra:

$$\Delta\vec{p} = \vec{p}_f - \vec{p}_i \Rightarrow \vec{p}_f = \vec{p}_i + \Delta\vec{p}$$

But $\vec{J} = \vec{F} \cdot \Delta t = \Delta\vec{p}$, so:

(9) $$\vec{p}_f = \vec{p}_i + \vec{J}$$

Typically, we are not so concerned with the final linear momentum after the collision, but rather we are concerned with the final velocity after the collision. Easily enough, the final velocity follows directly from (9):

$$\vec{p}_f = \vec{p}_i + \vec{J}$$
$$m\vec{v}_f = m\vec{v}_i + \vec{J}$$

(10) $$v_f = \vec{v}_i + \frac{\vec{J}}{m}$$

This result is important because it will allow us to solve for the final velocity for an object after a collision in terms of the initial velocity and impulse. Observe that formula (10) gives us the form from which formulas (1) and (2) come. Although we used $\vec{J} = j\hat{\vec{n}}$, the form is the same.

18.4.4 Newton's Third Law of Motion

Definition: *Newton's third law of motion states that if an object* m_1 *exerts a force* $\vec{F}_{12}$ *on object* m_2 *then object* m_2 *exerts an equal and opposite force* $\vec{F}_{21} = -\vec{F}_{12}$ *on object* m_1.

A direct consequence of Newton's third law of motion is the **law of conservation of linear momentum,** which states that in a closed and isolated system, the total (sum) linear momentum remains constant. By *closed* we mean no objects enter or leave the system. By *isolated* we mean that no net external forces from outside the system can act on the system. So if our system is two spheres and neither of the two objects leaves the system and if no new objects enter the system, and if no net external force acts on any object in the system then the *total* linear momentum of that two-sphere system will always stay the same. Observe that we emphasize the word "total" here; the linear momentum of each object may change but the total linear momentum of the system will not change, according to the law of conservation of linear momentum. That is,

(11) $$m_1\vec{v}_{1i} + m_2\vec{v}_{2i} = m_1\vec{v}_{1f} + m_2\vec{v}_{2f}$$

This reads that the total momentum at some initial time is equal to the total momentum at some final time. Equation (11) is specific to a system consisting of two objects only.

How does Newton's third law imply the conservation of linear momentum? For simplicity, let us explain with our preceding two-sphere system. There are two cases:

1. The spheres in the system do not collide. In this situation the total momentum of the system obviously stays the same. That is, no external net forces are acting on the system and therefore the objects will not change velocity. And the mass of the objects are not changing, thus the total linear momentum of the two spheres $m_1\vec{v}_{1i} + m_2\vec{v}_{2i} = m_1\vec{v}_{1f} + m_2\vec{v}_{2f}$ remains the same (i.e., constant).

2. The objects collide. A collision will change the objects' velocities. However, just because the velocity of the individual objects in the system changes, it does *not* mean the total linear momentum of the system changes. On the contrary, it still remains constant, but we must prove this.

Consider Figure 18.12.

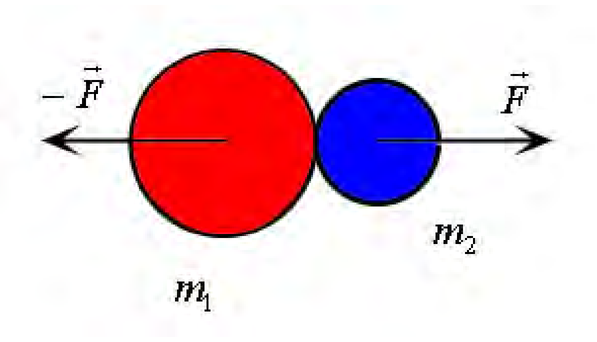


Figure 18.12: When the two masses collide they exert equal and opposite forces on each other for the duration of the collision.

During the collision, which lasts Δt seconds, object m_1 exerts an average force $\vec{F}$ during that short time interval on object m_2. Thus object m_2 experiences a change in linear momentum of $\Delta p_2 = \vec{F} \cdot \Delta t = \vec{J}$. However, by Newton's third law then, object m_2 exerts an equal and opposite force on object m_1. Thus, object m_1 experiences a change in linear momentum $\Delta p_1 = -\vec{F} \cdot \Delta t = -\vec{J}$. If we sum the total change in linear momentum that occurs in the system we have:

$$\Delta p_1 + \Delta p_2 = -\vec{F} \cdot \Delta t + \vec{F} \cdot \Delta t = -\vec{J} + \vec{J} = \vec{0}$$

This shows that even though the linear momentum of each object in the system changed, the total linear momentum of the system did not change. Thus total linear momentum was conserved, which we wanted to prove.

18.4.5 Kinetic Energy and Elastic Collisions

Definition: *It is said that an object in motion possesses* ***kinetic energy.*** *The kinetic energy of an object is defined as:* $E_K = 1/2 \cdot m \cdot v^2$*, where m is the mass of the object, and* $v = \|v\|$ *is the magnitude of the velocity vector.*

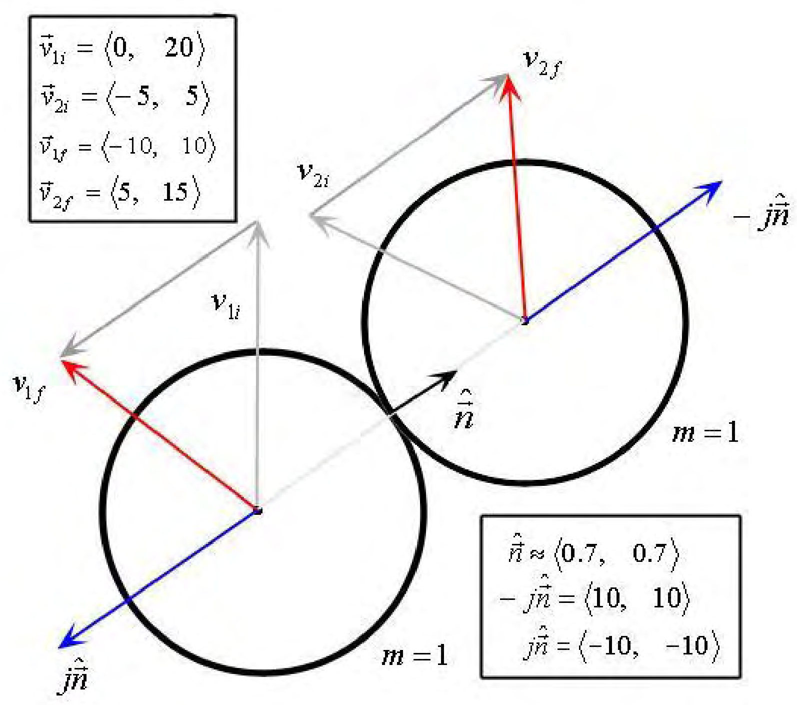
Definition: *If there is no loss of kinetic energy in a collision then we say that the collision is an* ***elastic collision.*** *Although in reality no collision is elastic (some energy is always lost), some events are almost elastic, such as two billiard balls striking each other. Although, intuitively we know that two billiard balls cannot truly be elastic because we hear a sound when they strike, thus we know some energy was transferred to sound energy.*

Elastic collisions are important for approximating *almost* elastic collisions. More specifically, if we assume the collision is elastic then we know the *total* level of kinetic energy before the collision is equal to the *total* level of kinetic energy after the collision. In other words, we say that kinetic energy is conserved. For a two-object system of m_1 and m_2, that is,

$$(20) \qquad \frac{1}{2}m_1{v_{1i}}^2 + \frac{1}{2}m_2{v_{2i}}^2 = \frac{1}{2}m_1{v_{1f}}^2 + \frac{1}{2}m_2{v_{2f}}^2$$

This reads that the total initial kinetic energy of the system is equal to the total final kinetic energy of the system after the collision.

Consider the elastic collision shown in Figure 18.13. It is elastic because kinetic energy is conserved. In addition, observe that we have made the objects to have the same mass, that is $m = 1$; this makes the calculations simpler.



$\vec{v}_{1i} = \langle 0, 20 \rangle$
$\vec{v}_{2i} = \langle -5, 5 \rangle$
$\vec{v}_{1f} = \langle -10, 10 \rangle$
$\vec{v}_{2f} = \langle 5, 15 \rangle$
v_{2f}
v_{2i}
$-j\hat{\vec{n}}$
v_{1i}
v_{1f}
$\hat{\vec{n}}$
$m = 1$
$\hat{\vec{n}} \approx \langle 0.7, 0.7 \rangle$
$-j\hat{\vec{n}} = \langle 10, 10 \rangle$
$j\hat{\vec{n}} = \langle -10, -10 \rangle$
$m = 1$
$j\hat{\vec{n}}$

Figure 18.13: An elastic collision.

Let us show that linear momentum is conserved; that is, the total initial linear momentum before the collision is equal to the total final linear momentum after the collision.

$$m_1\vec{v}_{1i} + m_2\vec{v}_{2i} = m_1\vec{v}_{1f} + m_2\vec{v}_{2f}$$

$$m\langle 0, \quad 20\rangle + m\langle -5, \quad 5\rangle = m\langle -10, \quad 10\rangle + m\langle 5, \quad 15\rangle$$

$$\langle -5, \quad 25\rangle = \langle -5, \quad 25\rangle$$

Now let us show that this collision is really elastic by showing that kinetic energy is conserved.

$$\frac{1}{2}m_1 v_{1i}{}^2 + \frac{1}{2}m_2 v_{2i}{}^2 = \frac{1}{2}m_1 v_{1f}{}^2 + \frac{1}{2}m_2 v_{2f}{}^2$$

$$v_{1i}{}^2 + v_{2i}{}^2 = v_{1f}{}^2 + v_{2f}{}^2$$

$$\left[0^2 + 20^2\right] + \left[(-5)^2 + 5^2\right] = \left[(-10)^2 + 10^2\right] + \left[5^2 + 15^2\right]$$

$$450 = 450$$

This shows that the kinetic energy remained the same before and after the collision (no energy loss).

Let us take a moment to make an observation about Figure 18.13. We note that the relative velocity projected onto $\hat{\vec{n}}$ is equal and opposite before and after the collision:

(21) $$(\vec{v}_{1i} - \vec{v}_{2i}) \cdot \hat{\vec{n}} = -(\vec{v}_{1f} - \vec{v}_{2f}) \cdot \hat{\vec{n}}$$

$$(\langle 0, \quad 20\rangle - \langle -5, \quad 5\rangle) \cdot \langle 0.7, \quad 0.7\rangle = -(\langle -10, \quad 10\rangle - \langle 5, \quad 15\rangle) \cdot \langle 0.7, \quad 0.7\rangle$$

$$\langle 5, \quad 15\rangle \cdot \langle 0.7, \quad 0.7\rangle = -\langle -15, \quad -5\rangle \cdot \langle 0.7, \quad 0.7\rangle$$

$$5(0.7) + 15(0.7) = 15(0.7) + 5(0.7)$$

$$14 = 14$$

This makes sense given the fact that kinetic energy is conserved; we would suspect that the total relative motion of the system along the line of impulse remains unchanged. But we wonder whether the equation for relative motion before and after the collision is true in general for elastic collisions. That is, is equation (21) always true for elastic collisions?

To begin deriving (21), in general from fundamental physical principles, let us select a convenient coordinate system. Figure 18.14 shows that we choose a coordinate system where the *x*-axis is parallel and coincident with $\hat{\vec{n}}$.

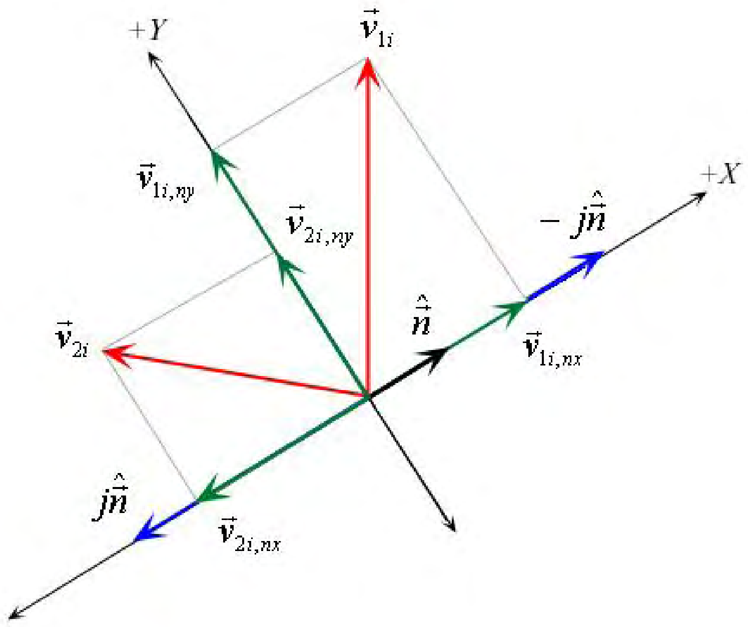


Figure 18.14: The coordinate system with the x-axis coincident to the collision normal. For clarity we have left out drawing the object bodies and have just kept the vectors. Moreover, we translate the vectors parallel to themselves so that the vectors originate with the origin; note that this translation is perfectly legal—a parallel transport does not alter the magnitude or direction of the vector.

Note that Figure 18.14 is a general figure; that is, we are not looking at a particular collision case, but rather we are looking at *any* collision case.

Let us now find the components of our velocities $\vec{v}_{1i}$ and $\vec{v}_{2i}$ relative to this coordinate system. The x-component of the velocities are found by projecting the vectors onto the normal vector $\hat{\vec{n}}$:

$$\vec{v}_{1i,nx} = (\vec{v}_{1i} \cdot \hat{\vec{n}})\hat{\vec{n}}$$
$$\vec{v}_{2i,nx} = (\vec{v}_{2i} \cdot \hat{\vec{n}})\hat{\vec{n}}$$

The y-components of the velocities are found by taking the difference between the velocity vector and the projected x-component:

$$\vec{v}_{1i,ny} = \vec{v}_{1i} - (\vec{v}_{1i} \cdot \hat{\vec{n}})\hat{\vec{n}}$$
$$\vec{v}_{2i,ny} = \vec{v}_{2i} - (\vec{v}_{2i} \cdot \hat{\vec{n}})\hat{\vec{n}}$$

Thus, our velocities relative to this new coordinate system can be expressed in terms of their components along the x- and y-axis as:

$$\vec{v}_{1i,n} = (\vec{v}_{1i} \cdot \hat{\vec{n}})\hat{\vec{n}} + \vec{v}_{1i} - (\vec{v}_{1i} \cdot \hat{\vec{n}})\hat{\vec{n}}$$
$$\vec{v}_{2i,n} = (\vec{v}_{2i} \cdot \hat{\vec{n}})\hat{\vec{n}} + \vec{v}_{2i} - (\vec{v}_{2i} \cdot \hat{\vec{n}})\hat{\vec{n}}$$

From the conservation of kinetic energy for elastic collisions and from the conservation of linear momentum, we can relate our total initial velocities

$$\frac{1}{2}m_1 v_{1i,n}^2 + \frac{1}{2}m_2 v_{2i,n}^2 = \frac{1}{2}m_1 v_{1f,n}^2 + \frac{1}{2}m_2 v_{2f,n}^2$$

$$m_1\vec{v}_{1i,n} + m_2\vec{v}_{2i,n} = m_1\vec{v}_{1f,n} + m_2\vec{v}_{2f,n}$$

A key observation of Figure 18.14 is that the impulse vectors are parallel to the x-axis. Consequently, the y-components of the velocities are untouched by the impulse vectors (there is no impulse component on the y-axis). What this means is that the y-component of the velocities are going to remain the same before and after the collision, and therefore we do not need to consider the y-component in either of our conservation equations (It does not change! This is the reason why we originally picked this coordinate system.) Consequently, we will drop the arrowheads and only look at the x-components:

(22) $$\frac{1}{2}m_1 v_{1i,nx}^2 + \frac{1}{2}m_2 v_{2i,nx}^2 = \frac{1}{2}m_1 v_{1f,nx}^2 + \frac{1}{2}m_2 v_{2f,nx}^2$$

(23) $$m_1 v_{1i,nx} + m_2 v_{2i,nx} = m_1 v_{1f,nx} + m_2 v_{2f,nx}$$

After doing some algebra we can rewrite (22) as:

$$m_1\left(v_{1i,nx}^2 - v_{1f,nx}^2\right) = -m_2\left(v_{2i,nx}^2 - v_{2f,nx}^2\right)$$

(24) $$m_1\left(v_{1i,nx} + v_{1f,nx}\right)\left(v_{1i,nx} - v_{1f,nx}\right) = -m_2\left(v_{2i,nx} + v_{2f,nx}\right)\left(v_{2i,nx} - v_{2f,nx}\right)$$

Similarly, we can rewrite (23) as:

(25) $$m_1\left(v_{1i,nx} - v_{1f,nx}\right) = -m_2\left(v_{2i,nx} - v_{2f,nx}\right)$$

If we divide equation (24) by equation (25) we get:

$$\frac{m_1\left(v_{1i,nx} + v_{1f,nx}\right)\left(v_{1i,nx} - v_{1f,nx}\right)}{m_1\left(v_{1i,nx} - v_{1f,nx}\right)} = \frac{-m_2\left(v_{2i,nx} + v_{2f,nx}\right)\left(v_{2i,nx} - v_{2f,nx}\right)}{-m_2\left(v_{2i,nx} - v_{2f,nx}\right)}$$

$$v_{1i,nx} + v_{1f,nx} = v_{2i,nx} + v_{2f,nx}$$

(26) $$\left(v_{1i,nx} - v_{2i,nx}\right) = -\left(v_{1f,nx} - v_{2f,nx}\right)$$

But equation (26) says that the initial relative velocity along the x-axis (i.e., normal vector $\hat{\vec{n}}$) is equal but opposite to the final relative velocity along the x-axis (i.e., normal vector $\hat{\vec{n}}$). But that is the same as:

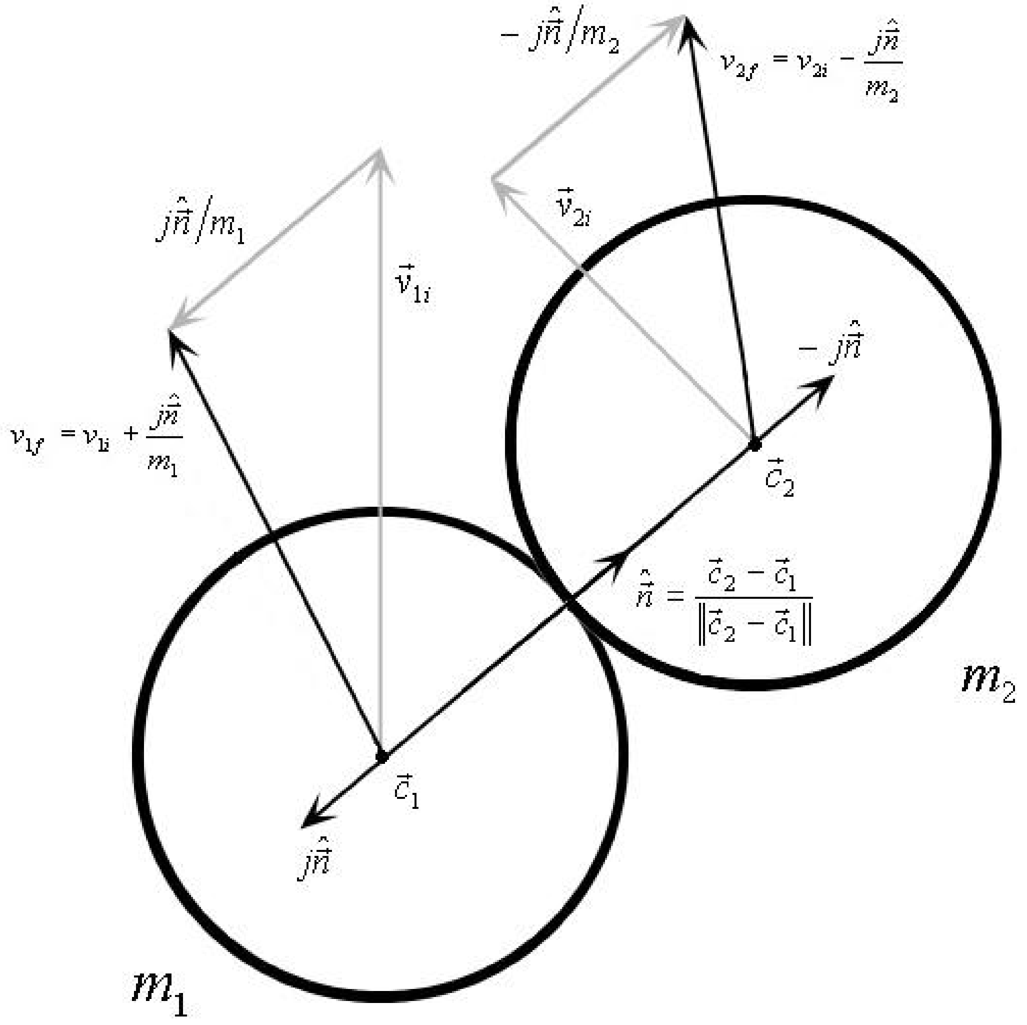
$$\left(\vec{v}_{1i} - \vec{v}_{2i}\right)\cdot\hat{\vec{n}} = -\left(\vec{v}_{1f} - \vec{v}_{2f}\right)\cdot\hat{\vec{n}}$$

This was the relationship we wanted to derive from fundamental physics definitions, which we have just done.

18.4.6 Collision and Response

We now have enough physics vocabulary and formulas to solve the original section problem: how two circular masses (paddle and puck) respond physically when they collide. Note that we assume the collision is elastic, and this is a close enough approximation; the collision between an air hockey paddle and puck is almost elastic in practice. The fact that we assume the collision is elastic is important because, if we do not, then equation (33) does not hold.

Consider two circular masses m_1 and m_2 with initial linear momentum $m_1\vec{v}_{1i}$ and $m_2\vec{v}_{2i}$ colliding as Figure 18.15 shows (we have already seen this picture twice before).



$-j\hat{\vec{n}}/m_2$
$v_{2f} = v_{2i} - \frac{j\hat{\vec{n}}}{m_2}$
$\vec{v}_{2i}$
$j\hat{\vec{n}}/m_1$
$\vec{v}_{1i}$
$-j\hat{\vec{n}}$
$v_{1f} = v_{1i} + \frac{j\hat{\vec{n}}}{m_1}$
$\vec{c}_2$
$\hat{\vec{n}} = \frac{\vec{c}_2 - \vec{c}_1}{\|\vec{c}_2 - \vec{c}_1\|}$
m_2
$\vec{c}_1$
$j\hat{\vec{n}}$
m_1

Figure 18.15: A collision.

Clearly, the impulse is in the direction $\hat{\vec{n}}$ that is perpendicular to the point of contact. Let $\vec{J} = \vec{F} \cdot \Delta t = j\hat{\vec{n}}$, where $\hat{\vec{n}}$ gives the direction of the impulse and j gives its magnitude (which is unknown), and $\vec{F}$ is the average force exerted during the collision and Δt is the temporal duration of the collision.

We will know the objects' incoming initial velocities before the collision. What we want to find is the velocities after the collision $\vec{v}_{1f}$ and $\vec{v}_{2f}$; informally that is, "how do the circular masses bounce off of each other?" To begin, we observe that formula (9) gives the final velocity in terms of the initial velocity and impulse. Thus, we need only evaluate the right side of the following two equations to find the final velocities:

(27) $$\vec{v}_{1f} = \vec{v}_{1i} + \frac{\vec{J}}{m_1} = \vec{v}_{1i} + \frac{j\hat{\vec{n}}}{m_1}$$

(28) $$\vec{v}_{2f} = \vec{v}_{2i} - \frac{\vec{J}}{m_2} = \vec{v}_{2i} - \frac{j\hat{\vec{n}}}{m_2}$$

As an aside, let us confirm here that linear momentum is conserved. If we add the total system momentum—that is equations (27) and (28)—and do some algebra rearrangements then we find that:

$$m_1\vec{v}_{1i} + \vec{J} + m_2\vec{v}_{2i} - \vec{J} = m_1\vec{v}_{1f} + m_2\vec{v}_{2f}$$

The impulses cancel out, leaving:

$$m_1\vec{v}_{1i} + m_2\vec{v}_{2i} = m_1\vec{v}_{1f} + m_2\vec{v}_{2f}$$

This states that the total system momentum before the collision is the same as the total system momentum after the collision, and that satisfies the conservation law of linear momentum.

Recall that the $-\vec{J}$ quantity in (28) comes from our discussion of Newton's third law; that is, the two objects feel equal but opposite impulses.

Unfortunately, we cannot evaluate the right hand side (27) and (28) because we do not know *j*. In order to find *j* we need another equation. We recall that in Section 18.4.5 we did some work to show that the following relationship is true:

(29) $$\left(\vec{v}_{1i} - \vec{v}_{2i}\right)\cdot\hat{\vec{n}} = -\left(\vec{v}_{1f} - \vec{v}_{2f}\right)\cdot\hat{\vec{n}}$$

Substituting (27) and (28) into (29) yields:

$$\left(\vec{v}_{1i} - \vec{v}_{2i}\right)\cdot\hat{\vec{n}} = -\left(\left(\vec{v}_{1i} + \frac{j\hat{\vec{n}}}{m_1}\right) - \left(\vec{v}_{2i} - \frac{j\hat{\vec{n}}}{m_2}\right)\right)\cdot\hat{\vec{n}}$$

$$\vec{v}_{1i}\cdot\hat{\vec{n}} - \vec{v}_{2i}\cdot\hat{\vec{n}} = -\vec{v}_{1i}\cdot\hat{\vec{n}} - \frac{j\hat{\vec{n}}}{m_1}\cdot\hat{\vec{n}} + \vec{v}_{2i}\cdot\hat{\vec{n}} - \frac{j\hat{\vec{n}}}{m_2}\cdot\hat{\vec{n}}$$

$$-2\left(\vec{v}_{1i} - \vec{v}_{2i}\right)\cdot\hat{\vec{n}} = j\hat{\vec{n}}\cdot\hat{\vec{n}}\left(\frac{1}{m_1} + \frac{1}{m_2}\right)$$

Using the fact that $\hat{\vec{n}}\cdot\hat{\vec{n}} = 1$ and rearranging leads to:

(30) $$j = \frac{-2(\vec{v}_{1i} - \vec{v}_{2i}) \cdot \hat{\vec{n}}}{\left(\frac{1}{m_1} + \frac{1}{m_2}\right)}$$

Substituting (30) in for *j* in equations (27) and (28) gives:

(31) $$\vec{v}_{1f} = \vec{v}_{1i} + \left[\frac{-2(\vec{v}_{1i} - \vec{v}_{2i}) \cdot \hat{\vec{n}}}{\left(\frac{1}{m_1} + \frac{1}{m_2}\right)}\right]\frac{\hat{\vec{n}}}{m_1} = \vec{v}_{1i} + \frac{-2\left(\vec{v}_{12,i} \cdot \hat{\vec{n}}\right)\hat{\vec{n}}}{\left(\frac{m_1}{m_1} + \frac{m_1}{m_2}\right)}$$

(32) $$\vec{v}_{2f} = \vec{v}_{2i} - \left[\frac{-2(\vec{v}_{1i} - \vec{v}_{2i}) \cdot \hat{\vec{n}}}{\left(\frac{1}{m_1} + \frac{1}{m_2}\right)}\right]\frac{\hat{\vec{n}}}{m_2} = \vec{v}_{2i} - \frac{-2\left(\vec{v}_{12,i} \cdot \hat{\vec{n}}\right)\hat{\vec{n}}}{\left(\frac{m_2}{m_1} + \frac{m_2}{m_2}\right)}$$

Equations (31) and (32) are the same as equations (4) and (5) given in Section 18.2.1.3, which at the time we gave without proof.

18.5 Closing Remarks

The Air Hockey game illustrated five subparts of the game loop. Specifically, each game loop cycle we do the following:

1. Check user input.
2. Execute Artificial Intelligence code.
3. Update physics (including collision detection)
4. Game logic.
5. Draw the updated frame of animation.

Each of these processes makes up a component of the overall game engine. Two other components missing from our game, but typically mandatory for most games, is sound and networking. These two additional components would also need to be updated frequently and be part of the main game loop. To summarize, a game engine can be thought of as having seven core components: Input, AI, Physics, Graphics, Sound, Networking, and Game Logic.

With the successful completion of this course, you have learned the C++ programming language, received some exposure and a basic introduction to Windows programming with the Win32 API, and most importantly, you have learned fundamental game programming concepts and graphics concepts such as animation, double buffering, sprites, the game loop, collision detection, checking user input, and basic artificial intelligence.

You have successfully climbed one mountain, but your journey is just getting started. We hope that you continue your education here at Game Institute. It is the school's recommendation that the next course you should take following this one is the *3D Graphics Programming with DirectX Module I* course. This course will teach you the fundamentals of hardware accelerated 3D graphics using the Direct3D API, which is mandatory study for all contemporary games programmers. Many other 3D programming related topics are also covered in that course, so it is an excellent next step. In addition, as you have already seen from the Air Hockey game, math, physics, input, and artificial intelligence also play a key role in game development. As a result, your training will soon lead you to the *Game Mathematics*, *Game Physics*, and *Introduction to Artificial Intelligence* courses.

Finally, even though there is still much to learn, it might be a good idea to review what you have learned here and try to implement your own classic 2D games using the basic sprite and backbuffer code we used for the Air Hockey game. Possible game ideas are *Asteroids*, *PacMan*, *Space Invaders*, *Commander Keen*, and so on. A search for "arcade games" on www.google.com will yield many more ideas. We wish you the very best of luck as you continue your game programming adventures!