

# The Five AP Classes

Maria Litvin

Phillips Academy, Andover, Massachusetts

This document is available on the Internet:

**<http://www.skylit.com/apclass>**

Copyright © 1997, 1998 By Maria Litvin and Skylight Publishing  
All rights reserved.

For permission to reproduce please contact Maria Litvin  
Phillips Academy, Andover, MA 01810  
e-mail: [mlitvin@andover.edu](mailto:mlitvin@andover.edu)

Skylight Publishing  
9 Bartlet Street, Suite 70  
Andover, MA 01810  
(978) 475-1431  
e-mail: [support@skylit.com](mailto:support@skylit.com)  
web: <http://www.skylit.com>

## Acknowledgements

---

*The Five AP Classes* would not be possible without the help of Gary Litvin, who conceived and coded the Dictionary, Cookie Monster, and Madlibs examples and reviewed the text. I am grateful to Owen Astrachan for reading the draft and offering valuable comments, and to Mike Clancy for suggesting a chapter on how to use strings, vectors and matrices as members of structures and classes. As usual, I am very grateful to Margaret Litvin for her thorough and thoughtful editing.



# Contents

---

## **1. Introduction 7**

## **2. What is a C++ Class? 9**

- 2.1. Key Concepts 9
- 2.2. Syntax for Defining Classes and Accessing Member Functions 9
- 2.3. Constructors 10
- 2.4. Overloaded Operators and Functions 12

## **3. Templated Classes 13**

## **4. Integrating AP Classes into Your Projects 15**

- 4.1. Using the apstring Class 15
- 4.2. Example for Borland C++ 4.5 under Windows 16
- 4.3. Using the apvector, apmatrix, apstack, and apqueue Classes 17

## **5. The apvector Class 19**

- 5.1. Declarations 19
- 5.2. Other Member Functions 19
- 5.3. Subscripts 20
- 5.4. Passing apvector Arguments to Functions 21
- 5.5. The apvector Class vs. Built-in Arrays 23

## **6. The apmatrix Class 25**

- 6.1. Declarations 25
- 6.2. Other Member Functions 25
- 6.3. Subscripts 26
- 6.4. Example 26

## **7. The apstring Class 27**

- 7.1. Declarations 27
- 7.2. Other Member Functions 27
- 7.3. Subscripts 29
- 7.4. The Overloaded Assignment and Relational Operators 30
- 7.5. Concatenation 31
- 7.6. String Input and Output 31
- 7.7. Passing apstring Arguments to Functions 33
- 7.8. The apstring Class vs. Null-Terminated Strings 33

**8. An apstring/apvector Example: the Dictionary Program 35**

**9. AP Class Objects in Structures and Classes 39**

**10. The apstack Class 45**

**11. The apqueue Class 47**

**12. apstack and apqueue Examples 49**

12.1. The Cookie Monster Program 49

12.2. The Madlibs Program 53

**13. The AP Classes and Software Performance 57**

# 1. Introduction

This document describes the five C++ classes defined by the APCS Ad Hoc Committee on C++: `apvector`, `apstring`, `apmatrix`, `apstack` and `apqueue`. The five AP classes are not a part of standard C++, but they are fashioned as a subset of the Standard Template Library (STL) with a few changes and with the addition of the `apmatrix` class.

The `apvector` and `apmatrix` classes implement resizable one- and two-dimensional arrays, respectively. Unlike built-in arrays, these classes have safe subscripting: they verify that the values of the subscripts used in the program at run time fall within the legal range. The `apstring` class implements resizable character strings in a way that avoids cryptic C-style idioms based on pointers. The `apstring` class also checks the subscripts against the legal range. The `apstack` and `apqueue` classes implement the stack and queue Abstract Data Types.

The C++ Committee decided to provide these classes for several reasons. First, using the `apvector`, `apmatrix`, and `apstring` classes instead of built-in arrays and null-terminated strings eases software development, saves novices from subscript-out-of-bounds bugs, and fosters a programming style based on class libraries. Second, the five AP classes provide examples of well-behaved C++ classes with which all students taking the APCS exam will be familiar; exam questions and the Case Study will use or be based on the five AP classes. Third, the `apstack` and `apqueue` classes provide standard implementations of stacks and queues that may be used in programming examples and projects and tested on the APCS exam.

This document does not assume familiarity with C++ classes. The key concepts of C++ classes and the key syntax associated with their use are presented in Chapter 2. With the exception of the `apstring` class, the AP classes are *templated* classes: that is, classes designed to work with different data types. The concept of templates and the associated syntax are discussed briefly in Chapter 3. The integration of AP classes into programming projects is covered in Chapter 4. Chapters 5, 6 and 7 describe the `apvector`, `apmatrix`, and `apstring` classes respectively, and Chapter 8 provides a more extensive example of their use. Chapter 9 explains how to declare and initialize vector, string and matrix members of structures or classes. Chapters 10 and 11 describe the `apstack` and `apqueue` classes, and Chapter 12 provides more extensive examples of their use in two complete programs. In Chapter 13 we discuss some issues related to the proper use of the AP classes.

We do not discuss the implementation of the AP classes in this document. The details may change in the future and will not be tested on the A-level APCS exam; the official AB-level course outline in the 1998-99 "Acorn" booklet states that students should be able to "use and re-implement AP stack and queue classes."





## 2. What is a C++ Class?

### 2.1. Key Concepts

A class is an entity that combines data elements with functions that operate on them. Both data elements and functions are called *members* of the class. A class is akin to a `record` in Pascal or a `struct` in C, but, besides data elements, a class also includes member functions.

Once defined, a class introduces a new user-defined data type into the program. We will sometimes call instances of the class (i.e., variables or constants of the class data type) *class objects*. Classes are convenient for implementing ADTs and naturally lead to OOP (Object-Oriented programming)—a programming paradigm in which a program is designed as a set or hierarchy of active objects.

A class restricts a programmer's access to its members. All members of a class are divided into two groups: private and public. Private members (data and functions) are accessible (i.e., can be referred to or called) only within the member functions of the class. Public members are accessible anywhere in the program where the class is defined. Public member functions are used by the client program to interface to the class. It is common to hide all (or most) of the data elements and member functions that are used only inside the class as private members. This practice, known as *encapsulation*, makes software maintenance easier because one can make changes to the class implementation without changing the class interface or the rest of the program.

### 2.2. Syntax for Defining Classes and Accessing Member Functions

The syntax for defining a class is illustrated in the following example:

```
// apstring class implements a string of characters:

class apstring {

    public:

        apstring()           // Default constructor
        ...                 // Other constructors

                                // Other public member functions:
        int length();        // Returns the length of the string.
        int find(char ch);   // Returns the position of the
                                // first occurrence of ch in the string
        ...
}
```

```

private:
    // Private data members:
    int myLength;      // Current length of the string.
    int myCapacity;    // Space available for the string.
    char *myCString;   // Pointer to the buffer that contains
                       // the actual string characters.
};

```

Once a class is defined, class objects (constants or variables) can be declared the same way as constants and variables of built-in data types. For example:

```

const apstring msg = "Hello, World!";
apstring firstName, lastName;

```

Member functions are called using syntax similar to that used for accessing elements of a structure. For example:

```

int n, pos;
n = firstName.length(); // Set n to the length (number of chars)
                        // in firstName.
pos = lastName.find('a'); // Set pos to the position of the first
                        // 'a' in lastName.

```

### 2.3. Constructors

A class usually has one or more special public member functions, called *constructors*. All the constructors in a class have the same name as the class itself; they differ only in the number and types of arguments that they take. Constructors are never called explicitly; instead, a constructor is called automatically whenever a constant or variable of the class type is declared in the program or created using the `new` operator. A constructor is used to initialize the class object: set its data elements to default or specified values, allocate storage, and so on. A class also has a *destructor*: a complementary function that is called automatically when the class object goes out of scope or is deleted using the `delete` operator.

A well-behaved class has at least two constructors: the *default constructor* and the *copy constructor*. The default constructor is used to declare variables without arguments. For example, if we declare

```
apstring firstName;
```

then the default constructor is called. Here the variable `firstName` is initialized to an empty string.

The copy constructor is used to declare a class object and at the same time initialize it to a previously declared constant or variable of the same type. For example, if we declare

```
apstring name = firstName;
```

then the copy constructor is called and `name` is initialized to a string equal to the current value of `firstName`.

The copy constructor is also called when a class object is passed to a function by value and when a class object is returned from a function.

In addition to the default and copy constructors, a class may have other useful constructors. For example, in the case of the `apstring` class, we want to be able to declare a string object equal to a specified literal string using declarations such as:

```
apstring msg = "Hello, Sunshine!";
```

To do this, we need a constructor that takes one argument: a constant literal string.

A declaration-with-initialization syntax using the equal sign can be used only if the corresponding constructor takes one argument. Constructors may also take two or more arguments. In that case the arguments are listed in parentheses, as in a function call. For example, the class `apmatrix` has a constructor that takes two arguments:

```
class apmatrix {  
    public:  
  
    ...  
    // Constructor: takes two arguments:  
    //   the number of rows and the number of columns in the matrix  
    apmatrix (int nrows, int ncols);  
    ...  
};
```

This constructor allows us to declare a matrix with given dimensions. For example:

```
apmatrix<int> table(5,3);  
    // Declare a matrix with 5 rows and 3 cols.
```

Some C++ purists prefer always to initialize constants and variables using parentheses instead of using the equal sign. They would write:

```
int i(0); // Instead of: int i = 0;
```

They would also write

```
apstring msg("Hello, World!");
```

The meaning is exactly the same as:

```
apstring msg = "Hello, World!";
```

We find the latter form more readable.

## 2.4. Overloaded Operators and Functions

C++ lets programmers redefine the meaning of standard operators for class objects. This technique is called *operator overloading*. It is common to overload the input operator, `>>`, and the output operator, `<<`, for class objects. The `apstring` class also overloads the `+` operator to signify concatenation of strings and overloads the relational operators (`>`, `>=`, etc.) to compare strings. The `apvector`, `apstring`, and `apmatrix` classes overload the subscript operator `[]` to support safe subscripting, which catches subscripts that are out of range.

The term *overloading* also applies to functions: several functions that have the same name but differ in the number or types of arguments they take are called overloaded functions. A class may have overloaded member functions. For example, the `apstring` class has two forms of the `find` member function:

```
// Find the first occurrence of the character ch in this string:
int find (char ch);

// Find the first occurrence of the string str in this string:
int find (const apstring &str);
```

All the different forms of constructors in a class are overloaded functions.

### 3. Templated Classes

C++ is a strongly typed language: every variable and constant has a specific data type, and every function takes arguments of specific types. However, it is often desirable to have exactly the same functionality with different data types. For example, a function that returns the maximum of two values may work the same way for two integers and for two real numbers:

```
int max(int a, int b)

{
    if (a >= b) return a;
    else return b;
}

double max(double a, double b)

{
    if (a >= b) return a;
    else return b;
}
```

To avoid duplicating code, C++ provides a mechanism called *templates*. A function or even a whole class can be programmed to operate on a *parameterized* data type—an abstract data type name that stands in for specific data types. For example, the two versions of the `max` function above may be rewritten as one *templated* function:

```
template <class someType>
someType max(someType a, someType b)
{
    if (a >= b) return a;
    else return b;
}
```

The line

```
template <class someType>
```

indicates that the function or class definition that follows is a templated function or class. It also assigns a name chosen by the programmer (in this case, `someType`) to the data type parameter.

When the compiler sees how the function is used, it substitutes the specific data type for `someType` and automatically generates the code for the appropriate version of the function. For instance, if we write

```
double a, b;  
cin >> a >> b;  
cout << max(a, b);
```

the compiler will generate the code for the `double max(double a, double b)` and call that function.

As we know, a class name introduces into our program a new user-defined data type used for declaring class objects. With a templated class, the data type parameter must be specified in each declaration of a class object. For example, the templated class `apvector` implements a dynamic array. When we declare an `apvector` object we must indicate whether we want a vector of integers, doubles, or some other type. We do this by placing the specific data type name in angular brackets after the class name:

```
apvector<int> a;           // Declare a vector of integers.  
apvector<double> x;       // Declare a vector of doubles.  
apvector<apstring> name;  // Declare a vector of character strings.
```

We cannot write just

```
apvector x;  // Syntax error!
```

`apvector<typeparameter>` becomes a new user-defined data type in the program and is used as one word. *typeparameter* can be any built-in or user-defined data type. For instance, a function that returns the sum of the elements in a vector of doubles may be declared as

```
double Sum(apvector<double> amt);  
    // The function Sum takes one argument amt  
    //   of the data type apvector<double>.
```

The `apvector`, `apmatrix`, `apstack`, and `apqueue` classes are templated classes that work with all built-in data types and with user-defined data types that support the copy constructor and the overloaded assignment operator. `apstring` is not a templated class but a regular class: it works only with character strings.

## 4. Integrating AP Classes into Your Projects

### 4.1. Using the `apstring` Class

Let us discuss the `apstring` class first. It is handled differently from the other four AP classes because it is a regular and not a templated class.

C++ is a modular language: it supports projects that consist of several modules. The process of building an executable program consists of two steps: (1) compile one or several modules that constitute a project separately to obtain *object* modules; (2) *link* the object modules together (using the linker program) into one executable program. It is considered bad style to lump together the source code from several modules if you can compile them separately.

A non-templated class may be implemented as an independent module. The code for a class is usually split between two files: a header file (which usually has the extension `.h`) and a `cpp` file (which usually has the same name and the extension `.cpp`). The header file serves as the interface between the class and the rest of the program; it contains the class definition and other definitions and constants—everything that the outside world needs to know about the class. The `cpp` file contains code for the class member functions—the details of implementation that the outside world does not need to know. The header file is included into the `cpp` file using the `#include` directive. The header file is also included into each client module that uses the class.

The `apstring` class is implemented in the `apstring.h` header file and the `apstring.cpp` implementation file.

Suppose your own program is in one module, but it needs to use the `apstring` class. First, you need to include the `apstring` header file into your code. Do this by placing the line

```
#include "apstring.h"
```

near the top of your program. When the compiler processes that line, it will replace it with the actual text of `apstring.h`.

It is appropriate to place all the files for AP classes, including `apstring.h` and `apstring.cpp`, into a separate directory or folder. However, the compiler must be able to find the `apstring.h` file when it tries to compile your module. To achieve this, you have to properly configure your IDE options to add the AP classes directory to the list of paths that the compiler searches for include files. Initially, that list contains only the path to the compiler's own include directory. Add a new path for the AP classes directory, separated by a semicolon.

The second step is to create a project in your IDE that includes two items: your program and `apstring.cpp`. When you build your executable file for the project, both your source and `apstring.cpp` will be compiled independently; then they will be linked together into one executable program. Once `apstring.cpp` is compiled, the compiler will create an `apstring.obj` file. As long as `apstring.obj` exists, the IDE will know not to recompile `apstring.cpp`.

## 4.2. Example for Borland C++ 4.5 under Windows

As an example, let us create and build the following program in Borland C++ 4.5.

```
#include <iostream.h>
#include "apstring.h"

int main()

{
    const apstring msg = "Hello, AP classes!";
    cout << msg << endl;
    return 0;
}
```

Suppose your working directory is `c:\work` and all your files for AP classes are stored in the `c:\apclass` directory. Suppose you want to call your program "hello."

First go to the *Options/Project...* menu item, click on *Directories*, and find the input field marked *Include*. It may show something like

```
c:\bc45\include
```

At the end of that line add

```
;c:\apclass
```

So you will end up with

```
c:\bc45\include;c:\apclass
```

This configures your include search paths. While you are at it, find the fields for the intermediate and final output directories and change them to `c:\work`. You have to configure your directories only once.

Now create a new project. Go to *Project/New project* menu item, enter the project name `c:\work\hello.ide`, select the target type *EasyWin*, click on the *Advanced* button and uncheck the *rc* and *def* boxes, click on *OK* on the "advanced" box then on *OK* on the



"project" box. This will create a project with one executable module, `hello.exe`, and one source module, `hello.cpp`. You can double-click on `hello.cpp` to bring up a new (or existing) source file and edit it.

Now you need to add the `apstring.cpp` module to your project. Highlight the `hello.exe` item and then click on the *Add Item* button on the toolbar. In the dialog box that appears, go to the `c:\apclass` directory and chose `apstring.cpp`. This adds the `apstring.cpp` "node" to your project, at the same level as `hello.cpp`. (If, by accident, `apstring.cpp` ends up hanging under `hello.cpp`, you can drag and drop it into the right place, under `hello.exe`.)

Click on the *Make-and-Run* button to build and test your program.

### 4.3. Using the `apvector`, `apmatrix`, `apstack`, and `apqueue` Classes

A templated class cannot be compiled separately because the compiler needs to know how the client program uses it. Thus a templated class is usually implemented in one header file that includes both the class definition and the code for all member functions. That file is simply included at the top of the client program using the include directive. For example, a program that uses the `apmatrix` class will have the line

```
#include "apmatrix.h"
```

near the top of the file. As before, we assume that all files for AP classes are in a separate directory and that the path to that directory has been added to the list of paths that the compiler searches for include files.

The C++ Committee has chosen to split the `apvector`, `apmatrix`, `apstack`, and `apqueue` classes into `.h` and `.cpp` files even though they are templated classes and their entire code has to be included into the source of a client program. This way, a class interface can be presented to students in the header file and discussed separately, without the implementation details. The corresponding `cpp` file is attached at the bottom of the header file using `#include`. For example, the `apvector.h` file has the line

```
#include "apvector.cpp"
```

at the bottom. This is equivalent to placing all the code from `apvector.cpp` directly into `apvector.h`. Both `apvector.h` and `apvector.cpp` must be in a directory where the compiler will find them.

It is important to remember that `apvector.cpp`, `apmatrix.cpp`, `apstack.cpp`, and `apqueue.cpp` are not true source modules, but rather fragments of code split from the respective header files for cosmetic reasons. Do not add these `.cpp` files as separate items to your projects.

Many compilers allow you to create, compile, and run a single-module program without creating an explicit project for it. The `apvector`, `apmatrix`, `apstack`, and `apqueue` classes, being in header files, are not compiled separately and thus do not produce separate object modules. A program that uses any of these classes may be a single-module program.

## 5. The apvector Class

The apvector class implements a dynamic (resizable) array with subscript range checking. It is easy to change the length of the vector by calling the `resize(...)` member function.

### 5.1. Declarations

The apvector class has four constructors. Here are a few examples of the corresponding declarations of apvector objects:

```
apvector<int> a;
    // Empty (zero-length) vector of int.
    // Calls the default constructor apvector().

apvector<double> x(100);
    // Vector of doubles of length 100. Calls the constructor
    // apvector (int size);

apvector<char> stars(5, '*');
    // Vector of chars of length 5, filled with '*'.
    // Calls the constructor
    // apvector (int size, const someType &fillValue);

apvector<int> zeroCounts(26, 0);
apvector<int> counts = zeroCounts;
    // const apvector zeroCounts of length 26,
    //   filled with 0's.
    // apvector counts is set equal to zeroCounts.
    // The second line calls the copy constructor
    // apvector (const apvector &otherVector);
```

You can declare a constant vector that cannot be changed. For example:

```
const apvector<double> zeros(100, 0.0);
```

### 5.2. Other Member Functions

Besides constructors and the destructor, the apvector class has two member functions, `length()` and `resize(...)`.

The `length()` member function returns the current length of the vector. For example:

```
apvector<double> sample(100);
    // Declare a vector of 100 doubles.
```

```
...
int len = sample.length();
    // set len to the current length of sample.
```

The `resize(int newSize)` member function sets the length of the vector to `newSize`. If the new length is less than the initial length, the vector's tail is truncated; if the new length is bigger than the initial length, the vector is padded with default elements (i.e., elements initialized by the default constructor for `someType`; for the vector of characters, integers, or doubles, the default values are zeros).

For example:

```
apvector<double> sample; // Declare an empty vector of doubles
int len;
...
cin >> len;
sample.resize(len);      // Resize sample to the new length.
```

### 5.3. Subscripts

The overloaded `[ ]` operator lets us refer to the individual elements of a vector the same way we refer to the elements of built-in arrays. The subscript for the first element is 0; the subscripts range between 0 and `length-1`, where `length` is the current length of the vector. The `apvector` class checks at run time that each subscript value used falls within the legal range. (This is not so with built-in C++ arrays: subscript checking is not performed, and it is easy to overwrite memory area outside the array by mistake.)

In the following example, the function `DropNegatives(...)` eliminates all negative values from a vector of integers and resizes the vector:

```
void DropNegatives (apvector<int> &a)

// Eliminates all negative elements from a and resizes a.

{
    int i, count = 0, n = a.length();

    for (i = 0; i < n; i++) {
        if (a[i] >= 0) {
            a[count] = a[i];
            count++;
        }
    }
    a.resize(count);
}
```

### 5.4. Passing apvector Arguments to Functions

As with any other data type, an apvector constant or variable can be passed to a function as an argument, either by value or by reference. A function can also return a value of the apvector type. The programmer has to remember, however, that an apvector object may contain a large array, which will be copied each time an apvector is passed to a function by value or returned from a function. Passing an apvector argument by reference, where possible, is more efficient and appropriate. If the function changes the vector, passing by reference is the only way. (See the `DropNegatives(...)` example in the previous section.) If the function does not change the vector, it is still better to pass the vector by reference, adding the keyword `const` to document the fact that the function does not change the vector and to prevent the code from changing it by mistake.

The following example illustrates passing a vector to a function by reference:

```
double Average(const apvector<double> &sample)

// Returns the average of values in sample.
// Assumes that sample contains at least one element.

{
    double sum = 0.;
    int i, n;

    n = sample.length();
    for (i = 0; i < n; i++)
        sum += sample[i];

    return sum / n;
}
```

If declared as

```
double Average(apvector<double> sample)
```

(the `&` symbol dropped) the function will work with exactly the same code and return the correct value. However, the argument will be passed by value, and the whole vector will be copied unnecessarily. Students working with classes should take extra care to avoid forming bad habits like this, which lead to outrageously inefficient code.

Note that as opposed to built-in arrays, it is not necessary to pass the length of a vector to a function as a separate argument because the length is stored within the vector object itself and can be obtained by calling the `length(...)` member function.

Suppose we want to write a function that takes a vector and builds a new vector with the elements in reverse order. Let us consider two approaches. In the first approach, both the original and resulting vectors are passed to the function by reference:

```

void Reverse (const apvector<int> &a, apvector<int> &b)

// Places elements from a in reverse order into b.

{
    int i, n = a.length();

    b.resize(n);
    for (i = 0; i < n; i++)
        b[i] = a[n-1-i];
}

```

This function may be used as follows:

```

int main()
{
    const int SIZE = 100;
    apvector<int> a(SIZE), b(SIZE);

    ...
    Reverse(a, b);
    ...
}

```

Alternately, we can build the resulting array inside the function and return it to the calling program:

```

apvector<int> Reverse (const apvector<int> &a)

{
    int i, n = a.length();
    apvector<int> b(n);

    for (i = 0; i < n; i++)
        b[i] = a[n-1-i];

    return b;
}

```

This version of `Reverse(...)` will be used as follows:

```

int main()
{
    const int size = 100;
    apvector<int> a(size), b;

    ...
    b = Reverse(a);
    ...
}

```

The second version might seem a bit more elegant, but it is in fact less efficient because the `return` statement in `Reverse(...)` calls the copy constructor, which copies the whole vector. In the first version, unnecessary copying does not happen.

## 5.5. The apvector Class vs. Built-in Arrays

The `apvector` class has several important advantages over built-in C++ arrays. We have already mentioned that the `apvector` class checks subscripts at run time and makes sure that they fall within the legal range. This helps to quickly detect and correct subscript-out-of-bounds bugs, which can otherwise lead to unexpected results or to a memory protection fault.

The `apvector` class has other important advantages:

- The overloaded assignment operator lets us copy one vector into another:

```
apvector<double> a, b;
...
b = a;
```

Besides direct convenience, this also allows us to use `apvector` objects with other templated classes, such as `apstack` and `apqueue`: we can easily create a stack or a queue of vectors.

- There is no need to pass the length of the vector as a separate argument to a function because it is stored within the vector object and available through the `length()` member function.
- If you declare a local array, the compiler allocates space for it on the program stack. A large array can easily overflow the stack. The `apvector` variable allocates only a small structure on the stack (the vector's length and a pointer to the actual array); the array itself is allocated dynamically from the free store, a large area in memory used for dynamic memory allocation.

The big disadvantage of the `apvector` class is the loss of simple initialization syntax for a short constant array, as in

```
const double price[3] = {.79, .99, 1.09};
```

Other disadvantages are the slight overhead associated with subscript range checking and the additional caution required to avoid code that works but is very inefficient.





## 6. The apmatrix Class

The apmatrix class implements resizable two-dimensional arrays with safe subscripting. It is similar to the apvector class with the following differences: the constructors and the `resize` function take two arguments, `rows` and `cols`, instead of one, `size`, and the `length()` function is replaced by two functions, `numrows()` and `numcols()`.

### 6.1. Declarations

The apmatrix class has four constructors. Here are a few examples of corresponding declarations of apmatrix objects:

```
apmatrix<int> m;
    // Empty (zero-size) matrix of int.
    // Calls the default constructor apmatrix().

apmatrix<double> x(100, 3);
    // Matrix of doubles with 100 rows and 3 cols.
    // Calls the constructor
    // apmatrix(int rows, int cols);

apmatrix<char> stars(5, 5, '*');
    // 5 by 5 matrix of chars filled with '*'.
    // Calls the constructor
    // apmatrix (int rows, int cols, const someType &fill);

apmatrix<int> zeroCounts(26, 2, 0);
apmatrix<int> counts = zeroCounts;
    // 26 rows by 2 cols apmatrix zeroCounts filled with 0's.
    // apmatrix counts is set equal to zeroCounts
    // (calls the copy constructor).
```

### 6.2. Other Member Functions

Besides the four constructors and the destructor, the apmatrix class has three member functions: `numrows()`, `numcols()`, and `resize(...)`.

The `numrows()` and `numcols()` member functions return the dimensions of the matrix. For example:

```

apmatrix<double> t(5, 1000);
// Declares a matrix t with 5 rows and 1000 cols
...
int rows = t.numrows();
int cols = t.numcols();
// rows is set to the current number of rows and
// cols to the number of columns in t.

```

The `resize(int newRows, int newCols)` member function sets the new dimensions of the matrix to `newRows`, `newCols`. Extra elements are truncated, and new elements are filled with default values (i.e., elements initialized by the default constructor for `someType`; for an `apmatrix` of characters, integers, or doubles, the default values are zeros).

### 6.3. Subscripts

The overloaded `[ ]` operators let us refer to the individual elements of a matrix the same way we refer to the elements in built-in 2-D arrays. For example:

```
t[r][c] = 0;
```

The first subscript refers to the row and the second to the column. As with built-in arrays, the subscripts for the first element are 0, 0. The `apmatrix` class checks at run time that all subscript values used fall within the legal range.

### 6.4. Example

The following function finds the number of non-zero elements in a matrix of integers:

```

int CountNonZeros(const apmatrix<int> &m)

// Returns the number of non-zero elements in m.

{
    int rows = m.numrows();
    int cols = m.numcols();
    int i, j, count = 0;

    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            if (m[i][j] != 0)
                count++;
    return count;
}

```

## 7. The apstring Class

The apstring class implements character strings. It is in many respects more convenient and safer than using C library functions for null-terminated strings. First, the apstring class checks the subscript range at run time. Second, the class maintains the current length of the string, so there is no need to worry about the terminating null (0 character) or reserve space for it. Third, the overloaded assignment operator and relational operators make apstring objects usable with templated classes or functions, such as templated implementations of sorting algorithms. Fourth, the apstring class eliminates cryptic C-style idioms involving pointers.

### 7.1. Declarations

The apstring class has three constructors. Here are a few examples of corresponding apstring declarations:

```
apstring s;  
    // Empty (zero-length) string.  
    // Calls the default constructor apstring().  
  
apstring day = "Friday";  
    // A string day initialized to the literal string "Friday".  
    // Calls the constructor  
    // apstring (const char *s);  
  
apstring otherDay = day;  
    // String otherDay is set equal to the value of day.  
    // Calls the copy constructor.
```

You can also declare a constant string that cannot be changed. For example:

```
const apstring msg = "Welcome";
```

### 7.2. Other Member Functions

Besides constructors and the destructor, the apstring class has five member functions:

```
int length();  
int find(char ch);  
int find(const apstring &str);  
apstring substr(int pos, int len);  
const char *c_str();
```

The `length()` member function returns the current length of the string. For example:

```
apstring msg = "Ciao";

int len = msg.length(); // sets len to 4.
```

The first form of the `find(...)` member function,

```
int find(char ch);
```

returns the position of the first occurrence of the character `ch` in the string. If the `ch` character is not found, the function returns a special constant value, `npos` (currently set to -1 in `apstring.cpp`). For example:

```
apstring name = "TEST.DAT";
int pos = name.find('.'); // Sets pos to 4
                        // (starts counting from 0).
int pos2 = name.find('X'); // Not found; sets pos2 to npos (-1).
```

The second overloaded form of `find(...)` takes an `apstring` as an argument and returns the position of its first occurrence as a substring. For example:

```
apstring msg = "Hello, Sunshine!";
apstring s = "Sun";

int i = msg.find(s); // Sets i to 7.
```

This form of `find(...)` also returns a value of `npos` when the substring is not found. For example:

```
if (msg.find("Moon") == npos) // If not found
    ...
```

Note that one of the `apstring` constructors builds an `apstring` object from a constant literal string. This constructor knows how to convert a literal string in double quotes into an `apstring` object. This makes it possible to pass a literal string to a function that expects an `apstring` argument, for instance to the `find(...)` function (as well as to your own functions). So you can write:

```
apstring msg = "Hello, Sunshine!";

int i = msg.find("Sun"); // Sets i to 7.
```

The

```
apstring substr(int pos, int len);
```

function builds and returns the substring that starts at position `pos` and has the length `len`. For example:

```
apstring fileName = "TEST.DAT", ext;

ext = fileName.substr(5, 3); // ext gets the value "DAT".
```

The `c_str()` member function returns the pointer to the actual null-terminated string associated with the apstring object. This function is needed to convert an apstring object into a `const char *` for use with classes and functions that do not understand apstrings. For example:

```
apstring fileName;
...
ifstream file(fileName.c_str());
// The ifstream constructor here does not understand
// the argument of the apstring type, so you can't write simply
//     ifstream file(fileName);
// This constructor expects a char * argument, and c_str()
// performs the conversion.
```

### 7.3. Subscripts

The overloaded `[ ]` operator lets us refer to the individual characters in a string. As with built-in arrays, the subscript for the first element is 0 and the subscripts range between 0 and `length-1`, where `length` is the length of the string. The apstring class checks at run time that each subscript value used falls within the legal range.

In the following example, the function `UpperCase(...)` converts a string to the upper case:

```
#include <ctype.h> // Declares the toupper(ch) function.

void UpperCase (apstring &s)

// Converts all letters in s to upper case.

{
    int i, n = s.length();

    for (i = 0; i < n; i++)
        s[i] = toupper(s[i]);
}
```

### 7.4. The Overloaded Assignment and Relational Operators

The `apstring` class provides three forms of the overloaded assignment operator. The first form lets us copy one `apstring` object into another. Two other forms help us convert a literal string and a single character, respectively, into an `apstring`. For example:

```
apstring str, str2;

str = "Sunshine"; // Copies "Sunshine" into str.
str2 = str;       // Copies str into str2.
str = '$';       // Creates an apstring that consists of the
                  // char '$' and copies it into str.
```

The `apstring` class overloads the relational operators `==`, `!=`, `<`, `>`, `<=`, and `>=` for comparing two strings. For example:

```
apstring name1, name2;
...
if (name1 <= name2)
    ...

apstring cmd;
...
if (cmd == "quit")
    ...
```

You can use a literal string instead of one of the arguments because the class will automatically convert it into an `apstring`.

The implementation of these operators is based on the `strcmp(...)` library function for null-terminated strings. On personal computers, this function compares individual characters based on their ASCII codes; it is case-sensitive for letters. The function compares strings character by character, starting from the beginning, for as long as the corresponding characters are equal. If the function encounters a character in the first string that is smaller than the corresponding character in the second string or if the first string ends, the function decides that the first string is smaller. For example, if

```
apstring str1 = "AP-COMPSCI";
apstring str2 = "ap-compSci";
apstring str3 = "ap-compSci@ets.org";
```

then the following conditions are true:

```
if (str1 < str2)... // True: 'A' < 'a' in ASCII code.
if (str2 < str3)... // True: str2 is shorter.
```

## 7.5. Concatenation

The `apstring` class overloads the `+` operator for concatenating two strings. For example:

```
apstring name = "MYPROG", ext = ".CPP";
apstring fileName;

fileName = name + ext;
fileName = name + ".CPP";
fileName = "MYPROG" + ext;
    // In all three statements fileName gets the value
    // "MYPROG.CPP".
```

Note that you cannot write

```
fileName = "MYPROG" + ".CPP";    // Error!
```

because this form of overloaded `+` operator is not provided. The compiler will think that you are adding two pointers and report an error.

You can also concatenate a character and a string. For example:

```
apstring word = "AT";
apstring word2 = 'B' + word;    // word 2 gets the value "BAT";
apstring word3 = word2 + 'H';    // word3 gets the value "BATH";
```

The overloaded `+=` member operator appends a string or a character to a string object. For example:

```
apstring fileName = "MYPROG";
fileName += '.';
fileName += "CPP";    // Now fileName gets the value "MYPROG.CPP".
```

## 7.6. String Input and Output

The `apstring` class overloads the `<<` operator so that it writes the string to the output stream (to `cout` or to a file). For example:

```
apstring hello = "Hello, World!";
cout << hello << endl;    // outputs Hello, World! and newline.
```

The overloaded `>>` operator reads one word from an input stream (from `cin` or from a file). For example:

```
apstring fileName;
cout << "Enter a file name: ";
cin >> fileName;    // Skips white space, reads one word
```

The apstring module also includes a free-standing (non-member) function

```
istream &getline(istream &is, apstring &str);
```

This function reads a whole line of characters (up to MAX\_LENGTH characters, currently set to 1024) from the input stream `is` (which can be `cin` or a file). `getline(...)` returns the reference to the stream and can be tested in conditional statements: an error or end of file tests as false. For example, to print all lines from a file, you can write:

```
while (getline(file, str))
    cout << str << endl;
```

The following program prints a file with line numbers added:

```
// LINENUM.CPP

// This program prints a file with line numbers.

#include <fstream.h>
#include <iomanip.h>
#include "apstring.h"

int main()
{
    apstring line, fileName;
    int lineNo = 0;

    cout << "File name: ";
    cin >> fileName;

    // Open input file fileName
    // (c_str() returns a pointer to the actual null-terminated
    // string associated with apstring fileName):

    ifstream file(fileName.c_str());
    if (!file) {
        cout << "Cannot open " << fileName << ".\n";
        return 1;
    }

    // Read and display lines, incrementing lineNo:
    while (getline(file, line)) {
        lineNo++;
        cout << setw(6) << lineNo << ' ' << line << endl;
    }

    return 0;
}
```



### 7.7. Passing apstring Arguments to Functions

As with `apvectors` and `apmatrices`, it is more efficient to pass `apstring` arguments to functions by reference. If the function does not change the string, we add the keyword `const` to document that fact and to prevent our code from changing the string by mistake.

The functions that return an `apstring` object may lose some efficiency. For example, the `apstring` class member function `substr(...)` first copies the substring characters into a new buffer, then copies the resulting substring again in the return statement.

### 7.8. The apstring Class vs. Null-Terminated Strings

The `apstring` class has several important advantages over direct manipulation of pointers and null-terminated strings:

- `apstrings` catch subscript-out-of-bounds errors.
- There is no need to keep track of terminating null characters.
- `apstrings` eliminate the confusion between declaring pointers (with no allocated space) and declaring character arrays (with allocated space).
- `apstrings` eliminate cryptic C-style idioms.
- The overloaded assignment and relational operators allow us to use `apstrings` with templated classes and functions: we can easily create a vector, a stack, or a queue of `apstrings` and apply templated versions of searching and sorting algorithms to `apstrings`.
- Local `apstring` variables take only a few bytes on the stack; the actual array of characters is allocated dynamically from the free store.

One potential inconvenience of the `apstring` class is that some useful standard functions and classes work with null-terminated strings and do not understand the `apstring` data type, requiring that the `c_str()` function be used for conversion. Also, if one is not careful, it is easy to write working but inefficient code using the `apstring` class. For example:

```
// Read a line from a file and pad with blanks
//   to the length of 100:
apstring line;
getline (file, line);
while (line.length() < 100)
    line += ' ';
```

The above code looks good, but it may unnecessarily reallocate and copy the string several times.



## 8. An apstring/apvector Example: the Dictionary Program

The following program reads a foreign-language dictionary from a file and then finds words entered by a user in the dictionary and displays their translations. The first line in the dictionary file is a heading that identifies the dictionary. The subsequent lines contain words and their translations. The dictionary file may look like this:

ENGLISH-ITALIAN DICTIONARY	
want	volere
a	un
I	io
program	programma
this	questo
today	oggi
love	amare

The program loads the dictionary file into a matrix of strings. The matrix has up to 1000 rows and two columns. The first column of each row (column 0) holds a word, and the second column (column 1) its translation. The program prompts the user to enter a word, finds it in the dictionary, and displays its translation (or a message that the word is not in the dictionary). A short dialog with the program may look as follows:

```
ENGLISH-ITALIAN DICTIONARY
(7 words)

Enter a word or 'q' to quit ==> I
io

Enter a word or 'q' to quit ==> this
questo

Enter a word or 'q' to quit ==> program
programma

Enter a word or 'q' to quit ==> hate
hate -- not in the dictionary.

Enter a word or 'q' to quit ==> love
amare

Enter a word or 'q' to quit ==> q
Thanks for using Dictionary.
```

The program uses the apstring and apvector classes:

```
/* DICT.CPP

    This program works as an on-line dictionary. It reads a word
    and shows its translation. The program loads the dictionary
    data from the DICT.DAT file.

    Author: M. Webster
    Rev. 1.0ap 09/20/98
*/

#include <iostream.h>
#include <fstream.h>
#include "apvector.h"
#include "apstring.h"

struct ENTRY {
    apstring word;
    apstring translation;
};

const int MAXWORDS = 1000; // Max number of words
                          // in the dictionary

// Function prototypes:
bool LoadDictionary(apstring fileName, apvector<ENTRY> &dict);
bool FoundWord(const apvector<ENTRY> &dict,
               const apstring &word, apstring &translation);

/*****
/*****                               Main Program                               *****/
/*****/

int main()
{
    apvector<ENTRY> dict(MAXWORDS);
    apstring word, translation;
    bool ok, quit;

    // Load the dictionary from the file

    ok = LoadDictionary("DICT.DAT", dict);
    if (!ok) {
        cout << "*** Cannot load dictionary ***\n";
        return 1;
    }
}
```

Continued 

```

// Translate words:

quit = false;
while (!quit) {
    cout << "Enter a word or 'q' to quit ==> ";
    cin >> word;           // Read one word and
    cin.ignore(80, '\n');   // skip the rest of the line
    if (word == "q")
        quit = true;
    else if (FoundWord(dict, word, translation))
        cout << translation << "\n\n";
    else
        cout << word << " -- not in the dictionary.\n\n";
}
return 0;
}

/*****
/*****                      Functions                      *****/
/*****/

bool LoadDictionary(apstring fileName, apvector<ENTRY> &dict)

// Reads dictionary entries from a file.
// Returns true if successful, false if cannot open the file.


{
    int cnt = 0;
    apstring line;

    // Open dictionary file:
    ifstream inpFile(fileName.c_str());
    if (!inpFile)
        return false;

    // Read and display the header line:
    getline(inpFile, line);
    cout << line << endl;

    // Read words and translations into the dictionary array:
    while (cnt < MAXWORDS &&
           inpFile >> dict[cnt].word >> dict[cnt].translation) {
        inpFile.ignore(80, '\n'); // Skip the rest of the line
        cnt++;
    }
}

```

Continued 

```
    // Report the number of entries:
    cout << " (" << cnt << " words)\n\n";
    dict.resize(cnt);

    return true;
}

/*****/

bool FoundWord(const apvector<ENTRY> &dict,
               const apstring &word, apstring &translation)

// Finds a word in the dictionary.
// dict -- the dictionary array
// word -- word to translate
// translation -- returned translation from the dictionary.
// Returns true if the word has been found, false otherwise.

{
    bool found = false;
    int i;

    for (i = 0; !found && i < dict.length(); i++) {
        if (dict[i].word == word) {
            translation = dict[i].translation;
            found = true;
        }
    }
    return found;
}
```

## 9. AP Class Objects in Structures and Classes

Often it is convenient to use structures or classes whose members are vectors, matrices, or strings. To do this, we must be comfortable with a C++ feature called "constructors with initializer lists." Although this topic is fairly technical, it allows us to use the AP classes in more meaningful ways; therefore, it has been included in both the A- and AB-level curricula.

As we know, members of structures and classes in C++ may have any built-in or user-defined data types. In particular, they may have data types defined by the AP classes. For example:

```
struct GAME {
    apstring mName;           // Game name
    apmatrix<char> mBoard;    // Game board
};
```

Something is missing in the above declaration, though. How do we tell the `mBoard` member what the dimensions of the matrix are? And how do we initialize the `mName` string with a particular name? If we used built-in arrays, we could write:

```
struct GAME {
    char mName[30];           // Game name
    char mBoard[3][3];        // Game board 3 by 3
};

GAME game = {"Tic-Tac-Toe"}; // Syntax with braces to initialize
                               // struct members in declarations.
```

But if we try something similar with AP classes, the compiler will report syntax errors:

```
struct GAME {
    apstring mName;
    apmatrix<char> mBoard(3,3); // Syntax error 1.
};

GAME tictactoe = {"Tic-Tac-Toe"}; // Syntax error 2.
```

The first error message will tell you that you cannot specify arguments in member declarations. The second error message will tell you that initialization with braces is not allowed for the structure `GAME` (because it contains members with types such as `apstring` or `apvector`).

But unless we specify the matrix dimensions our `GAME` structure is useless: the matrix dimensions are set to zero by default. What to do? Our first impulse might be to initialize

each instance of `GAME` "manually" resizing the `mBoard` matrix as needed using the `resize(...)` function. We may think of writing a separate initialization function, or several functions, for doing this. For example:

```
void SetForTicTacToe(GAME &game)
// Initializes a GAME structure for Tic-Tac-Toe.
{
    game.mName = "Tic-Tac-Toe";
    game.mBoard.resize(3,3);
}

void SetForGo(GAME &game, int size)
// Initializes a GAME structure for Go.
{
    game.mName = "GO";
    game.mBoard.resize(size, size);
}
```

This approach is dangerous: we would have to remember to initialize every `GAME` object before we used it. Also, it would be impossible to declare `const GAME` objects. But more importantly, this approach would fail completely if `GAME` were not a `struct` but a `class` with encapsulated data members:

```
class GAME {
    private:
        apstring mName;           // Game name
        apmatrix<char> mBoard;    // Game board
        ...
};
```

Then members of `GAME` would not be accessible to our initialization functions or any other code outside the `GAME` class.

As usual, whenever C++ creates a problem, it soon provides us with a reasonable solution. In this case the solution is called "constructors with initializer lists." The same solution works for both structures and classes. Recall that in C++ the `struct` and `class` keywords have almost the same meaning: the only difference is that in `struct` the first group of members is by default public, and in `class`, private. In particular, we can use constructors for structures as well as classes.

As we know, your program calls `GAME`'s constructor whenever it declares or creates an object of the `GAME` type. If you do not specify any constructors for your structure or class, the "automatic" constructor provided by the compiler will be called. This automatic constructor leaves the members of built-in types uninitialized, but it calls the default constructor (the constructor with no arguments) for each member that has a user-defined class type. In our example, if we do not provide any constructors in `GAME`, the default



constructors for `apstring` and `apmatrix` will be called. These constructors will create an empty string `mName` and a matrix `mBoard` with zero dimensions, respectively.

We can provide a constructor for `GAME` that will get around this default behavior. Our constructor must somehow pass the appropriate arguments to the `apstring` and `apmatrix` constructors. It cannot just call these constructors, because constructors are never called explicitly. But we can use an *initializer list*. The syntax is as follows:

```
struct GAME {
    apstring mName;           // Game name
    apmatrix<char> mBoard;    // Game board

    GAME();                  // Constructor: initializes GAME
                             //   for Tic-Tac-Toe.
};

// Constructor for struct GAME
GAME::GAME()
    : mName("Tic-Tac-Toe"), mBoard(3,3)    // Initializer list
{
    ... // Any additional code if necessary
}
```

The initializer list simply provides arguments for the `mName` and `mBoard` members, prompting the compiler to invoke the appropriate constructors. The elements of the list are separated by commas, and there is no semicolon before the opening brace. The initializer list basically reminds the compiler that in the process of building an object of the type `GAME`, it must first build the objects `mName` and `mBoard` with certain parameters.

Quite often the initializer list does all the work and the constructor code has nothing left to do. Then you can just put two braces with nothing between them:

```
GAME::GAME() // Constructor for Tic-Tac-Toe
    : mName("Tic-Tac-Toe"), mBoard(3,3)    // Initializer list
{ }
```

The initializer list, together with any short and simple code for the constructor, may also be placed inside the struct or class definition. For example:

```
struct GAME {

    apstring mName;           // Game name
    apmatrix<char> mBoard;    // Game board

    GAME()                   // Constructor for Tic-Tac-Toe
        : mName("Tic-Tac-Toe"), mBoard(3,3)
        { }
};
```

Or, for a class:

```
class GAME {  
  
    public:  
  
        GAME()                // Constructor for Tic-Tac-Toe  
            : mName("Tic-Tac-Toe"), mBoard(3,3)  
        {}  
  
        ...                    // Other public members  
  
    private:  
  
        apstring mName;        // Game name  
        apmatrix<char> mBoard; // Game board  
};
```

Now if you say in your program

```
GAME game;
```

the object `game` will be created with the name "Tic-Tac-Toe" and a Tic-Tac-Toe board.

You can provide several constructors with different sets of arguments for your structure or class. Some of these arguments may be passed to the constructors of members that are strings, vectors, or matrices. For example:

```
class GAME {  
  
    public:  
  
        GAME()                // Constructor for Tic-Tac-Toe  
                                // (default --no arguments)  
            : mName("Tic-Tac-Toe"), mBoard(3,3)  
        {}  
  
        GAME(int size)         // Constructor for GO  
                                // (one argument--size)  
            : mName("GO"), mBoard(size, size)  
        {}  
  
        GAME(const apstring &name, int size)  
                                // Constructor for other games  
            : mName(name), mBoard(size, size, ' ')  
        {}  
  
    private:  
  
        apstring mName;        // Game name  
        apmatrix<char> mBoard; // Game board  
};
```

Now if your program declares a `GAME` object with no arguments, the default constructor will be used and the object created will be a Tic-Tac-Toe board:

```
GAME game;    // Declare a game object for Tic-Tac-Toe.
```

If one integer argument is provided, the appropriate constructor will initialize `mName` to "GO" and create a square board of the specified size:

```
GAME game(19); // Declare a game object for GO with
               // the name "GO" and a 19 by 19 board
```

Given two arguments, a string and an `int`, the last constructor will be used. It will set the name to the specified string and build a square matrix `mBoard` of the specified size. It will also fill `mBoard` with spaces, because we specified the ' ' (the space char) fill value for the `mBoard` constructor in the initializer list. For example:

```
GAME game1("Reversi", 8);
           // Declare a game object for Reversi with
           // the name "Reversi" and an 8 by 8 board,
           // initially filled with spaces.
GAME game2("Hasami Shogi", 9);
```



## 10. The apstack Class

The `apstack` class implements the stack ADT. It is a templated class, so you can easily implement a stack of integers, doubles, strings, vectors—or data elements of any other data type. The `apstack` class has a default constructor and a copy constructor. To declare an empty stack of doubles named `stk` you can write:

```
apstack<double> stk;
```

The `apstack` class has seven member functions. Frequently used functions are:

```
bool isEmpty() // Returns true if the stack is empty,
               // false otherwise.

void push(const someType &item); // Pushes item on stack.

void pop(itemType &item);
    // Pops the top element and stores it
    // in item.
```

Other functions:

```
const someType &top() // Returns the top element,
                    // but leaves it on stack.

void pop(); // Pops the top element from stack
            // (and throws it away).

int length(); // Returns the current number of elements on stack.

void makeEmpty( );
    // Removes all elements from stack.
```

The `apstack` class also has the overloaded assignment operator, so you can copy a whole stack with one statement:

```
stk2 = stk;
```

A typical program with a stack uses the default constructor and the `push(item)`, `pop(item)`, and `isEmpty()` functions. For example:

```
void Reverse (apvector<double> v)

// Reverse vector v using a temporary stack.

{
    apstack<double> stack; // Declares an empty stack of doubles.
    int i, len = v.length();

    for (i = 0;   i < len;   i++) // Push all elements on stack
        stack.push(v[i]);

    for (i = 0;   !stack.isEmpty();   i++)
        // Pop all elements from stack
        stack.pop(v[i]);
}
```

## 11. The apqueue Class

The apqueue class implements the queue ADT. Like apstack, apqueue is a templated class that lets us implement a queue of integers, doubles, strings, vectors—data elements of any built-in data type or user-defined data type that supports a copy constructor and assignment. The apqueue class functions are similar to the apstack functions. In fact, in the STL's (Standard Template Library) queue class, the functions that insert an element and remove an element are called push and pop. This may be confusing because these names are traditionally used for the LIFO (Last In First Out) access method of a stack. The apqueue class uses more conventional function names, *enqueue* and *dequeue*.

The apqueue class has a default constructor and a copy constructor. To declare an empty queue of doubles named q you can write:

```
apqueue<double> q;
```

The apqueue class has seven member functions. Frequently used functions are:

```
bool isEmpty() // Returns true if the queue is empty,
               // false otherwise.

void enqueue(const someType &item);
           // Inserts item at the end of the queue.

void dequeue (itemType &item);
           // Removes the element from the front of
           // the queue and stores it in item.
```

Other functions:

```
const someType &front() // Returns the front element,
                       // but leaves it in the queue.

void dequeue();         // Removes the front element from the queue
                       // (and throws it away).

int length(); // Returns the current number of elements in the
queue.

void makeEmpty( );
           // Removes all elements from the queue.
```

The apqueue class also has the overloaded assignment operator, so you can copy a whole queue with one statement:

```
q2 = q;
```

A typical program with a queue uses the default constructor, `enqueue(item)`, `dequeue(item)`, and `isEmpty()` functions.



## 12. apstack and apqueue Examples

This chapter presents two programs that use the `apstack` and `apqueue` classes. While the stack and queue ADTs themselves are conceptually quite simple and easy to code, applications that use them in a meaningful way are usually more involved. Stack, in particular, is usually used to process nested structures or branching processes—the kind of problems that may take some effort to grasp.

The first example below, `MONSTER.CPP`, uses a stack and a queue to find the path along a board on which Cookie Monster can collect the most cookies. The second example, `MADLIBS.CPP`, uses a queue of strings to implement the MADLIBS game.

### 12.1. The Cookie Monster Program

This program helps Cookie Monster find the optimal path from the upper left corner (0,0) to the lower right corner (SIZE-1, SIZE-1) of the board. The board is a square matrix. The elements of the matrix contain cookies (a non-negative number) or barrels (-1). On each step Cookie Monster can only go down or to the right. He is not allowed to step on barrels. The optimal path contains the largest number of cookies.

The program prompts the user for a file name, reads the board configuration from the file, and reports the optimal path and the number of cookies on it.

The program works as follows. At each intermediate point it examines the possible ways to proceed. If there is only one way to proceed from the current position, then it goes there and updates the total accumulated number of cookies and the path. If there are two ways to proceed, the program saves one of the two possible points on stack, together with the path leading to it and its total, and proceeds to the other point. When the program reaches the end of the path it verifies that it has reached the lower-right corner and updates the best total. If there is nowhere to go, the program examines the stack, pops a saved point, if any, and resumes from there.

The board is implemented as an `apmatrix<int>`, a path is implemented as an `apqueue<int>` (which stores first the row, then the col for each board position in the path). `BOARDPOS` is a structure that holds the current position (row, col), its total number of accumulated cookies, and the best path leading to it. The stack is implemented as an `apstack<BOARDPOS>`.

```

// MONSTER.CPP
//
// In this program Cookie Monster finds the optimal path from
// the upper left corner (0,0) to the lower right corner
// (SIZE-1,SIZE-1) of a cookie matrix. The elements of
// the matrix contain cookies (a non-negative number)
// or barrels (-1). On each step, Cookie Monster
// may only go down or to the right. He is not allowed
// to step on barrels. The optimal path contains the largest
// number of cookies.
//
// The program prompts the user for a file name,
// reads the cookie array from the file, and reports the
// optimal path and the number of cookies on it.

#include <fstream.h>
#include <iomanip.h>
#include "apstring.h"
#include "apmatrix.h"
#include "apstack.h"
#include "apqueue.h"

static bool LoadCookies (const apstring &filename,
                        apmatrix<int> &cookies);
static int OptimalPath(const apmatrix<int> &cookies);

int main()
{
    apstring filename;
    apmatrix<int> cookies;

    cout << "Cookies file name ==> ";
    cin >> filename;
    if (!LoadCookies(filename, cookies)) {
        cout << "Cannot open " << filename << ".\n";
        return 1;
    }

    cout << "The optimal path has " << OptimalPath(cookies)
        << " cookies.\n";
    return 0;
}

```

Continued 

```

//*****

static bool LoadCookies (const apstring &filename,
                        apmatrix<int> &cookies)

// Loads the data from a file into the "cookies" array.
// Returns true if successful, false if cannot open the file.

{
    const int SIZE = 12;          // For the sake of simplicity
                                   // assumes that the
                                   // cookie matrix is of fixed size.

    int row, col;

    ifstream file(filename.c_str());
    if (!file)
        return false;

    cookies.resize(SIZE, SIZE);
    for (row = 0; row < SIZE; row++)
        for (col = 0; col < SIZE; col++)
            file >> cookies[row][col];

    return true;
}

//*****

inline bool GoodPoint (const apmatrix<int> &cookies,
                      int row, int col)

// Returns true if (row, col) is within the matrix
// and does not contain a barrel (-1); false otherwise.

{
    return (row >= 0 && row < cookies.numrows() &&
            col >= 0 && col < cookies.numcols() &&
            cookies[row][col] >= 0);
}

//*****

static int OptimalPath(const apmatrix<int> &cookies)

// Displays the optimal path.
// Returns the largest number of cookies collected by Monster
// on a path from (0,0) to (SIZE-1, SIZE-1).

{
    int newRow1, newCol1, newRow2, newCol2;

```

Continued 

```

struct BOARDPOS {
    int row, col;    // position on the board
    int total;      // accumulated number of cookies
    apqueue<int> q;  // the best path leading to this point
};

BOARDPOS current, saved, best;

apstack<BOARDPOS> stack;

if (!GoodPoint(cookies, 0, 0))
    return 0;

best.total = 0;

current.row = 0;
current.col = 0;
current.total = cookies[0][0];
current.q.enqueue(0);
current.q.enqueue(0);

for (;;) {
    newRow1 = current.row + 1;
    newCol1 = current.col;
    newRow2 = current.row;
    newCol2 = current.col + 1;
    bool good1 = GoodPoint(cookies, newRow1, newCol1);
    bool good2 = GoodPoint(cookies, newRow2, newCol2);
    if (good1 && good2) {
        // If both are good: save point 2 on stack,
        // proceed to point 1:
        saved = current;
        saved.row = newRow2;
        saved.col = newCol2;
        saved.total += cookies[newRow2][newCol2];
        saved.q.enqueue(newRow2);
        saved.q.enqueue(newCol2);
        stack.push(saved);
        current.row = newRow1;
        current.col = newCol1;
        current.total += cookies[newRow1][newCol1];
        current.q.enqueue(newRow1);
        current.q.enqueue(newCol1);
    }
    else if (good1) {
        current.row = newRow1;
        current.col = newCol1;
        current.total += cookies[newRow1][newCol1];
        current.q.enqueue(newRow1);
        current.q.enqueue(newCol1);
    }
}

```

```

        else if (good2) {
            current.row = newRow2;
            current.col = newCol2;
            current.total += cookies[newRow2][newCol2];
            current.q.enqueue(newRow2);
            current.q.enqueue(newCol2);
        }
        else {
            // if last point -- update best path:
            if (current.row == cookies.numrows() - 1 &&
                current.col == cookies.numcols() - 1 &&
                current.total > best.total)
                best = current;

            if (stack.isEmpty()) break;
            stack.pop(current);
        }
    }

    if (best.total > 0) {
        // Display the best path:
        int count = 0;
        while (!best.q.isEmpty()) {
            best.q.dequeue(current.row);
            best.q.dequeue(current.col);
            cout << " -> (" << setw(2) << current.row << ', '
                    << setw(2) << current.col << ")";

            count++;
            if (count == 6) {
                cout << endl;
                count = 0;
            }
        }
        cout << endl;
    }

    return best.total;
}

```

## 12.2. The Madlibs Program

In the "Madlibs" party game, the leader has the text of a short story with a few missing words in it. The missing words are tagged by their function: [noun], [verb], [place], etc. For example:

It was a [adjective] summer day.  
 Jack was sitting in a [place].

The leader examines the text and prompts the players for the missing words:

Please give me an/a:  
 adjective  
 place  
 ...

He then reads the text with the supplied words inserted into their places.

The Madlibs program acts as the Madlibs leader. It prompts the user for a file name, reads the text from the chosen file, finds the tags for the missing words (in square brackets), prompts the user for the words, and, finally, displays the completed text.

The list of entered words is implemented as the `apqueue<apstring>`. The `CollectWords(...)` function opens the file and reads each line. When it encounters a '[' character it looks for the matching ']', takes the substring within brackets, displays it as a prompt to the user, then reads and enqueues the entered word. The `PlugInWords(...)` function opens the file again and reads each line. It displays each character, but when it encounters an opening bracket it plugs in a word from the queue and skips all characters until the closing bracket.

```
// MADLIBS.CPP

// This program plays the Madlibs game.

#include <fstream.h>
#include "apstring.h"
#include "apqueue.h"

//*****

bool CollectWords(const apstring &filename, apqueue<apstring> &q)

// Reads the file, prompts the user with clues in brackets and
// collects entered responses in a queue.
// Clues in brackets cannot be split between lines.
{
    int i, i0, len;
    apstring word, line;

    ifstream file(filename.c_str());
    if (!file)
        return false;
```

Continued 

```

    while (getline(file, line)) {
        len = line.length();
        i0 = 0;
        for (i = 0; i < len; i++) {
            if (line[i] == '[')
                i0 = i + 1;
            else if (line[i] == ']' && i0 > 0) {
                if (q.isEmpty())
                    cout << "Please give me a/an:\n";
                // Show substring within brackets:
                cout << line.substr(i0, i - i0) << ": ";
                getline (cin, word);
                q.enqueue(word);
                i0 = 0;
            }
        }
    }

    return (!q.isEmpty());
}

//*****


bool PlugInWords(const apstring &filename, apqueue<apstring> &q)

// Reads the file, plugs in words from the queue and shows
// the final text.

{
    int i, i0, len;
    apstring word, line;

    ifstream file(filename.c_str());
    if (!file)
        return false;

```

Continued 

```

while (getline(file, line)) {
    len = line.length();
    i0 = 0;
    for (i = 0; i < len; i++) {
        if (line[i] == '[')
            i0 = i + 1;
        else if (line[i] == ']' && i0 > 0) {
            q.dequeue(word);
            cout << word;
            i0 = 0;
        }
        else if (i0 == 0) // Currently not inside brackets
            cout << line[i];
    }
    cout << endl;
}
return (q.isEmpty());
}

//*****

int main()
{
    apstring filename;
    apqueue<apstring> q;

    cout << "Ready for Madlibs?\n\n";
    cout << "Enter a Madlib file name: ";
    cin >> filename;
    cin.ignore(100, '\n');

    if (!CollectWords(filename, q)) {
        cout << "Cannot open " << filename
             << " or the file does not contain any clues.\n";
        return 1;
    }

    cout << endl < endl;

    if (!PlugInWords(filename, q)) {
        cout << "Cannot open " << filename << ".\n";
        return 1;
    }

    return 0;
}

```



## 13. The AP Classes and Software Performance

The set of five AP classes is a useful educational tool that simplifies C++ programming and takes care of many implementation details. Similar (but more extensive) tools are used in professional software development. As with any such tools, it is easy for a novice to misuse them. The AP classes allow a programmer to easily manipulate and copy large structures (vectors, matrices, etc.): one innocent-looking statement or an implicit constructor can move a lot of bytes in memory. This may create significant overhead and sometimes makes a "working" solution impractical.

It is not our intention to discuss here all the details related to optimizing code with the AP classes. The classes themselves are implemented with an emphasis on readability and ease of use rather than optimization. We just want to mention a few general points that apply to dealing with class libraries of this type.

1. As a rule, class objects should be passed to functions by reference. This eliminates unnecessary copying of class objects. Use the keyword `const` to prevent the function code from changing the object and to document that fact.
2. Avoid writing functions that return a class object. In such functions the return statement calls the copy constructor that copies the object.
3. The overloaded assignment operator allows you to copy a class object in a single statement:

```
b = a;
```

This statement may hide quite a lot of activity. For example, if `a` and `b` are vectors of strings, the assignment operator first allocates a vector that contains string descriptors (the length of the string and the pointer to its character array), then allocates memory and copies each individual string.

Beginners may be tempted to use many temporary variables and copy them many times. For example, when asked to reverse a vector, a beginner may write:

```
void Reverse(apvector<apstring> &v)  
{  
    int i, n = v.length();  
    apvector<apstring> temp;  
  
    temp = v; // Copy v into temp;  
    for (i = 0; i < n; i++)  
        v[i] = temp[n-i-1];  
}
```

The above code works, of course, but a more economical algorithm would not require the `temp` vector.

4. A constructor for a class object may do considerable work. For instance,

```
apmatrix<double> sample(100, 100);
```

allocates and zeros out each row of the matrix. It is rather costly to place such a declaration inside a loop because the constructor and the destructor will be executed on each pass through the loop. It is better to declare this variable at the top of the function, even if it is used only inside the loop.

5. The `resize` function in the `apvector` and `apmatrix` classes may be costly, too. When resizing to a bigger size, it has to reallocate the storage for the vector or for each row of the matrix, copy the old contents into the new area, pad it with default values, then deallocate the old storage. So avoid using `resize` within a loop, if possible.
6. The overloaded `+` operator for concatenating strings hides a lot of activity. It allocates a new buffer for the combined string, copies each of the strings into its proper place, then returns (copies again) the resulting string. If you declare:

```
apstring firstName = "Alice", lastName = "Wonderland";
```

and compare

```
cout << firstName << ' ' << lastName;
```

with

```
cout << firstName + ' ' + lastName;
```

you will see that the latter statement copies strings several times (10 times in the current implementation of the `apstring` class) before displaying the result.

7. The `substr` function in the `apstring` class is not very efficient either, because it builds a new `apstring` object and copies the substring into it. For example, if you need to compare strings starting from the tenth position, you may write:

```
if (str1.substr(10, str1.length() - 10) >
    str2.substr(10, str2.length() - 10))
    ...
```

This involves building substrings by copying character buffers. Perhaps more cryptic but more economical code uses pointer arithmetic to avoid copying:

```
if (strcmp(str1.c_str() + 10,  
          str2.c_str() + 10) > 0)  
    ...
```

The above code calls the `strcmp` library function, the same function that is used in the implementation of the `apstring <` operator. The second version may run many times faster than the first, which would make a real difference in a large sorting job.

Considerations like these come into play whenever one uses a class library, such as the AP classes. In a teaching environment such subtleties may seem unimportant, as long as the code "works." However, we believe that students will benefit from even a cursory discussion of performance issues: it will help them acquire good programming habits and pay attention to the practicality of their code.

