A class library for safe string manipulation

The interface of a class is its set of member-function prototypes. Client code, written by the application programmer, declares instances of the class and has these instances, or objects, call member functions of the class.

The ANSI/ISO (American National Standards Institute / International Standards Organization) draft standard for C++ includes the specification for a class called string that provides a convenient and safe standard data type for character strings. In the C++ standard, unlike with C-style strings, a string is an object, with member functions and associated operators, not an array. Its identifier is not a pointer and operations on strings are checked automatically for validity. The ANSI/ISO standard string class is available if you write #include <string> in your program. It is supported by Microsoft Visual C++ Version 5.0.

In this course we will use a class called strings, similar to the draft standard string. It is declared in the header file strings.h and its member functions are defined in the file strings.cpp, available at m:\Resources\Faculty\dkeil\63259\01 stream

An instance of strings looks like this in memory:

```
content: [ ] [ ] [ ] [ ] [ ] [ ] ... [ ]
```

Part of the interface of class strings:

```cpp
strings::strings() // Default constructor
    Example: strings s1;

strings::strings(char* s) // Conversion constructor from char*.
    Example: strings s1("string1");

boolean strings::assign(char* s) // Copies s content to content
    Example: s1.assign("hello");

boolean strings::append(char* s) // Like strcat.
    Example: s1.append("ing");

boolean strings::append(strings s) // Like strcat.
```

```cpp
Example: s1.append(s2);
int strings::compare(char* s) // Like strcmp.
    Example: s1.compare(s2);

unsigned int strings::length() // Like strlen.
    Example: int len = s1.length();

char strings::at(unsigned int n) // Returns nth character in content.
    Example: char ch = s1.at(3);

char strings::set_at(unsigned int n, char ch) // Assigns ch as nth element of content.
    Example: s1.set_at(0,'P');

boolean strings::insert(unsigned int loc, char* s) // If s is not too long, inserts s into content at location loc.
    Example: s1.insert(2,"XXX");

boolean strings::delete(unsigned int loc, int num_chars) // Deletes num_chars from content, starting at location loc.
    Example: s1.delete(2,4);

char* strings::substr(unsigned int loc, int num_chars) // Returns pointer to a static string containing substring of content starting at loc that contains num_chars characters.
    Example: cout << s1.substr(3,2);

boolean strings::replace(unsigned int loc, char* s) // Replaces text in content at location loc with contents of s.
    Example: s1.replace(2,"QQQ");

void strings::input(istream& is) // Stream input.
    Example: s1.input(cin);

void strings::display(ostream& os) // Stream output.
    Example: s1.display(cout);
```
Example of client code that uses *strings*

```
// strgdemo.cpp
// Demonstrates safe-string library
// class <strings>.
#include "strings.cpp"

void main()
{
    // Declare, initialize, display two strings:
    strings string1,
        string2("string2");
    string1.assign("string1");
    cout << endl;
    string2.display(cout);
    cout << endl;

    // Get two from user:
    cout << "Enter a string: ";
    string1.input(cin);
    cout << "Another: ";
    string2.input(cin);
    string1.append(string2);
    cout << "Length of your strings together is "
         << string1.length() << ".\n";
    string1.display(cout);
    cout << endl;
}
```

Sample input/output:
```
string1
string2
Enter a string: Howdy
Another: Bye
The length of your strings together is 8.
HowdyBye
```

The following operators are defined for class *strings*:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Left operand</th>
<th>Right operand</th>
<th>Operation</th>
<th>Example (given strings s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>output stream</td>
<td>strings</td>
<td>stream output</td>
<td>cout &lt;&lt; s;</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>input stream</td>
<td>strings</td>
<td>stream input</td>
<td>cin &gt;&gt; s;</td>
</tr>
</tbody>
</table>

The following operators take a *strings* object as their left operands:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Right operand</th>
<th>Operation</th>
<th>Example (given strings s1,s2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>strings; char*; char</td>
<td>assignment</td>
<td>s1 = &quot;schoolbus&quot;;;</td>
</tr>
<tr>
<td>[]</td>
<td>int</td>
<td>access to element[subscript]</td>
<td>char c = s1[0];</td>
</tr>
<tr>
<td>()</td>
<td>two ints (start, length)</td>
<td>substring</td>
<td>cout &lt;&lt; s1(0,6);</td>
</tr>
<tr>
<td>&gt;, &gt;=, &lt;, &lt;=, ==, !=</td>
<td>strings; char*; char</td>
<td>string comparison</td>
<td>if (s1 &lt; s2) cout &lt;&lt; s1 &lt;&lt; &quot; comes first&quot;;</td>
</tr>
<tr>
<td>+</td>
<td>strings; char*; char</td>
<td>concatenation</td>
<td>cout &lt;&lt; s1 + s2;</td>
</tr>
<tr>
<td>+=</td>
<td>strings; char*; char</td>
<td>concatenation to self</td>
<td>s1 += s2;</td>
</tr>
</tbody>
</table>

A set of operators similar to the set shown above is defined for the ANSI/ISO *string* class.

Overloaded operators provide an even more convenient interface for string classes

In the C language, all operators, such as =, +, <, are as defined by the C compiler. But in C++, we may also extend, or overload, these operators to apply to instances of a class that we define. The ANSI/ISO *string* class and the instructor’s predefined *strings* class provide a number of overloaded operators for use with string objects, so that they application programmer may work with strings in a similar way to how the standard types are manipulated. For example, the overloaded << operator for *strings* objects makes this sequence of statements possible:
```
strings greeting = "Hello";
cout << greeting << endl;
```
so that “Hello” is displayed.

The program at left could be coded in the following way:
```
strings string1,
    string2("string2");
string1 = "string1";
cout << string1 << endl << string2 << endl;

// Get two from user:
cout << "Enter a string: ";
cin >> string1;
cout << "Another: ";
cin >> string2;
string1 += string2;
cout << "Length of your strings together is "
     << string1.length() << endl << string1
     << endl;
```
A program to manage employee objects

// employee.cpp
// Initializes and displays
// employee objects.
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
const int NAME_LEN = 30;

class employees
{
 public:
   employees();
   employees(char* nm,int I,float sal);
   char* get_name() const;
   void set_name(char* nm);
   void display() const;
 private:
   char name[NAME_LEN];
   int ID;
   float salary;
};

void main()
{
   employees emp1("Jones",1078,48000.0),
   emp2("Smith",2392,61034.0),
   emp3;
   emp1.display();
   emp2.display();
   cout << "Emp2's name is "
   << emp2.get_name() << endl;
   emp2.set_name("Smythe");
   emp2.display();
}

employees::employees(char* nm,int I,
   float sal)
// Constructor.
{
    strcpy(name,nm);
    ID = I;
    salary = sal;
}

void employees::display()
// Shows all member values.
{
    cout << setiosflags
         << setprecision(2)
<< setw(20) << name
<< setiosflags(ios::right | ios::fixed)
<< setw(8) << ID
<< setw(12) << setprecision(2)
<< salary << endl;
}

char* employees::get_name()
// Selector access function.
{
    return name;
}

void employees::set_name(char* nm)
// Mutator access function.
{
    if (strlen(nm) < NAME_LEN)
        strcpy(name,nm);
}

Output:
Jones       1078   48000.00
Smith       2392   61034.00
Emp2's name is Smith
Smythe      2392   61034.00
A program to manage a collection of integers

The program below stores an array safely inside a class. It checks before appending a new element to the array elements, to see if the array is already full. Similar bounds checks could be implemented to filter any attempt to access an array element.

```cpp
// score.cpp
// Prompts for and displays a series of test or game scores.
#include <iostream.h>
typedef int bool;
const bool FALSE = 0, TRUE = 1;

// A score list is a collection of integers:
class score_lists
{
  enum {MAX_SCORES = 6};
public:
  score_lists() { size = 0; };
  bool append(int new_element);
  void input();
  void display();
private:
  int element[MAX_SCORES];
  int size;
};

void main()
{
  score_lists list;
  list.input();
  list.display();
}

void score_lists::input()
// Keyboard-input member function.
{
  int input_value;
  do {
    cout << "Enter score (-1 to quit): ";
    cin >> input_value;
  } while (input_value >= 0 &&
    append(input_value));
}

bool score_lists::append(int new_element)
// Adds <new_element> to collection.
{
  if (size < MAX_SCORES)
  {
    element[size++] = new_element;
    return TRUE;
  }
  else
  {
    cout << "score_lists::append: "
    "collection already full" << endl;
    return FALSE;
  }
}

void score_lists::display()
// Displays values.
{
  cout << endl;
  for (int i = 0; i < size; ++i)
    cout << element[i] << endl;
}
```

Sample I/O (2 runs):
Enter score (-1 to quit): 90
Enter score (-1 to quit): 42
Enter score (-1 to quit): 64
Enter score (-1 to quit): -1
90
42
64
Enter score (-1 to quit): 90
Enter score (-1 to quit): 86
Enter score (-1 to quit): 98
Enter score (-1 to quit): 75
Enter score (-1 to quit): 92
Enter score (-1 to quit): 84
Enter score (-1 to quit): 95
score_lists::append: collection already full
Enter score (-1 to quit): -1
90
86
98
75
92
84
95
A rational-numbers class with the addition operator

The following program defines a class to implement rational numbers, each having a numerator and a denominator. The + operator, ordinarily not defined for structure types or classes, is implemented. The class has a friend inserter operator as well.

```cpp
// ration1.cpp
// Initializes, adds 2 rational numbers.
#include <iostream.h>
class rationals
{
  int numerator,
   denominator;
public:
  rationals(int num,int den):
    numerator(num),denominator(den) {}
  rationals operator+(rationals term2);
  friend ostream& operator<<(ostream& os,rationals r);
};
rationals rationals::operator+(rationals term2)
// Returns rational object expressing sum of
// calling object and <term2>.
{
  int sum_numerator =
    numerator * term2.denominator +
    denominator * term2.numerator;
  int sum_denominator =
    denominator * term2.denominator;
  rationals sum(sum_numerator,sum_denominator);
  return sum;
}
ostream& operator<<(ostream& os,rationals r)
// Overloaded inserter.
{
  cout << r.numerator << "/" << r.denominator;
  return os;
}
void main()
{
  // Add one-half and two-thirds:
  rationals a(1,2),b(2,3);
  rationals sum = a + b;
  cout << a << " + " << b
       << " = " << sum << endl;
}
Output:
1/2 + 2/3 = 7/6
```

Overloading relational operator for a class of employees

The program below defines a class, `parts`, that has an overloaded operator, `>`, as a member function. The expression in `main`, `(part1 > part2)`, uses this operator function.

The program also declares a global operator function, `<<`, that outputs a `parts` instance to a stream. This function is a friend of `parts` because it uses data members of a `parts` object.

```cpp
// employe2.cpp
// Initializes and displays an employee object,
// using some overloaded operators.
#include <iostream.h>
#include <string.h>
typedef int bool;
class employees
{
  public:
    employees(char* n,int i,double sal):
      ID(i),salary(sal) { strcpy(name,n); }     
    char* get_name() { return name; }
    bool operator<(employees emp2);
  private:
    char name[40];
    int ID;
    double salary;
};
bool employees::operator<(employees emp2)
// Less-than relational operator.
{
  return (strcmp(name,emp2.get_name()) < 0);
}
void main()
{
  employees emp1("Smith",1078,48000.0),
      emp2("Jones",2392,61034.0);
  if (emp1 < emp2)
    cout << emp1.get_name();
  else
    cout << emp2.get_name();
  cout << " comes first" << endl;
}
Output:
Jones comes first
```
Assignment and concatenation to strings via overloading

// strovlod.cpp
// Demonstrates two overloaded assignment operators, overloaded "+".
#include <iostream.h>
#include <string.h>

class strings
{
    char* s;
public:
    strings() { s = new char[1]; s[0] = '\0'; }
    void operator=(const strings& str)
    { delete[] s; s = new char[strlen(str.s)+1]; strcpy(s,str.s); }
    void operator=(const char* p)
    { delete[] s; s = new char[strlen(p)+1]; strcpy(s,p); }
    ~strings() { delete[] s; }
    strings operator+(strings source);
    void display() { cout << s; }
};

strings strings::operator+(strings source)
// Returns concatenation of self and <source>.
{
    strings retval;
    retval.s = new char[strlen(s)+2*strlen(source.s)+1];
    if (!retval.s) cout << "Out of memory\n";
    strcpy(retval.s,s);
    strcat(retval.s,source.s);
    return retval;
}

void main()
{
    strings s1,s2,s3;
    s1 = "The rain in Spain ";
    s2 = "is on the plain";
    s3 = s1 + s2;
    s3.display();
}
Overloaded assignment and subscript operators for strings

The class below uses a dynamically allocated character array to store the character content of a string. Accordingly, a copy constructor and a destructor are present, with new and delete used.

Access to single character elements of a string is provided by the subscript (bracket) operators, one for storage and the other for retrieval. The second function \( \text{operator[]} \) returns char\&, an lvalue, so that it may be used to assign a value to a character of a string, as in \( \text{main: } s2[0] = \text{'P'}; \).

```cpp
// strovlod.cpp
// Demonstrates a partial implementation of
// a string class using dynamically allocated
// character array.
#include <iostream.h>
#include <string.h>
class strings
{
public:
    strings();
    strings(strings& s);
    ~strings();
    void operator=(const strings& str);
    void operator=(const char* p);
    char& strings::operator[](int subscript);
    char strings::operator[](int subscript) const;
    friend ostream& operator<<(ostream& os, strings s);
private:
    char* content;
};
strings::strings()
// Default constructor.
{
    content = new char[1];
    content[0] = '\0';
}
strings::strings(strings& s)
// Copy constructor.
{
    content = new char[strlen(s.content) + 1];
    strcpy(content, s.content);
}
strings::~strings()
// Destructor.
{
    if (content) delete[] content;
}
void strings::operator=(const strings& str)
// Assignment operator accepting <strings>
// object.
{
    if (content) delete[] content;
    content = new char[strlen(str.content) + 1];
    strcpy(content, str.content);
}
void strings::operator=(const char* p)
// Assignment operator to accept character-
// array parameter.
{
    if (content) delete[] content;
    content = new char[strlen(p) + 1];
    strcpy(content, p);
}
char& strings::operator[](int subscript)
// Subscript operator for assignment.
{
    char& c = content[subscript];
    return c;
}
char strings::operator[](int subscript) const
// Subscript operator for retrieval.
{
    return content[subscript];
}
ostream& operator<<(ostream& os, strings s)
// Inserter.
{
    os << s.content;
    return os;
}
void main()
{
    strings s1, s2;
    s1 = "Cup";
    s2 = s1;
    cout << "First letter=" << s2[0] << endl;
    s2[0] = \'P\';
    cout << "s1=" << s1 << endl;
    cout << "s2=" << s2 << endl;
}
```

Output:
First letter=C
s1=Cup
s2=Pup
Header file for a string class library

The class declared below implements string objects using an array of characters. Notice that most member functions are overloaded operators. See handout on ANSI/ISO safe strings for usage.

```c
// strings.h
// Safe string manipulation class library.
// (c) David Keil, Framingham State College, 6/98.
#ifndef STRINGS_H
#define STRINGS_H

#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

const int STR_MAX = 1000;

typedef int bool;
const bool false=0,true=1;

class strings
{
  char content[STR_MAX];
public:
  strings();
  strings(char c);
  strings(char* s);
  strings(int qty,char ch);
  char* get_content();
  char* c_str();
  char* data();
  char get_content(int i);
  bool empty();
  unsigned int size();
  unsigned int length();
  int max_size();
  bool assign(int loc,char c);
  bool assign(char* s);
  bool assign(strings s,int start,int len);
  strings operator=(char ch);
  strings operator=(char* s);
  strings operator=(strings s);
  char& operator[](int subscript);
  const char& operator[](int subscript) const;
  strings& operator[](int start,int len);
  bool append(char c);
  bool append(char* s);
  bool append(strings s);
  bool append(strings s,int start,int len);
  bool concat(char c);
  bool concat(char* s);
  bool concat(strings s);
  bool operator==(char* s);
  bool operator!=(char* s);
  bool operator>=(char* s);
  bool operator>=(strings s);
  bool operator>(char* s);
  bool operator>(strings s);
  bool operator<=(char* s);
  bool operator<=(strings s);
  bool operator<=(char* s);
  bool operator<=(strings s);
  bool operator<(char* s);
  bool operator<(strings s);
  strings operator+(char c);
  strings operator+(char* s);
  strings operator+(strings s);
  void operator+=(char c);
  void operator+=(char* s);
  void operator+=(strings s);
  int compare(char* s);
  int compare(strings s);
  int compare(int start,int len,strings s2,int start2,int len2);
  int compare(int start,int len,strings s2);
  char at(unsigned int n);
  char set_at(unsigned int n,char ch);
  bool insert(unsigned int loc,char* s);
  bool erase(int start);
  bool delete(unsigned int loc,int num_chars);
  strings substr(unsigned int loc,int num_chars);
  void copy(char* s,int len,int);
  bool replace(unsigned int loc, char* s);
  bool replace(unsigned int loc, char* s,int start2,int len2);
  int find(char* s);
  int rfind(char* s);
  int find_first_of(char* s);
  int find_last_of(char* s);
  int find_first_not_of(char* s);
  int find_last_not_of(char* s);
  void swap(strings& s);
  void input(istream& is);
  void display(ostream& os);
};

ostream& operator<<(ostream& os,strings s);
istream& operator>>(istream& is,strings& s);

#endif
```
A file-maintenance program for a collection of employee objects

The program below, in five source files, implements a collection of objects of class employees using an array of employees. Class employees is defined in files employee.h and employee.cpp. The collection class is payrolls, defined in payroll.h and payroll.cpp.

Alternative implementations of payrolls could use the same employees source files. Implementations in the example code subdirectory 04 Containers include an array of pointers to employees (payroll2.h, payroll2.cpp, roster2.cpp), a dynamically allocated array of employees (payroll3.h, payroll3.cpp, roster3.cpp), a linked list (emplist.cpp), and a stack (empstack.cpp).

// payroll.h
// Declares container class <payrolls>, organized as array of employee objects. // David Keil, Framingham State College, 6/98
#ifndef PAYROLL_H
#define PAYROLL_H
#include "employee.h"
const int MAX_RECS = 100;
class payrolls
{
public:
        payrolls();
        payrolls(char nm[]);
~payrolls();
        void display();
        void add_rec();
        void delete_rec();
        void edit_rec();
        bool retrieve();
        void save();
private:
        char file_name[40];
        employees emp[MAX_RECS];
        int num_recs;
};
#endif // #ifndef PAYROLL_H

// roster.cpp
// Maintains a payroll roster as a collection of employee objects. Options: save to disk, retrieve, edit, insert, delete employee.
#include "payroll.cpp"
int run_menu(char *option, ...);
void main()
{
        enum options {opQuit,opAdd,opDelete,opEdit};
        payrolls roster("payroll.dat");
if (roster.retrieve())
{
                int option;
        do {
                roster.display();
                cout << "\nChoose an operation: \n"; option = run_menu("Quit","Add rec", "Delete rec","Edit rec","" );
                switch (option)
                {
                case opAdd:
                        roster.add_rec(); break;
                case opDelete:
                        roster.delete_rec(); break;
                case opEdit:
                        roster.edit_rec(); break;
                }
} while (option != opQuit);
}
// Destructor automatically prompts to save data as <roster> goes out of scope.
} // Global functions:
int run_menu(char *option, ...)
// Displays a series of <option> arguments, gets integer from user, returns ordinal value of option chosen. Last argument must be "."
{
        // Display variable number of arguments:
        va_list ap; // declared in <stdarg.h>
        va_start(ap,option);
        for (int num_args=0; *option; ++num_args)
        {
                cout << num_args << ". " << option << endl;
                option = va_arg(ap,char *);
        }
        va_end(ap);
        // Get user choice:
        int retval;
        cin >> retval;
        if (retval >= 0 && retval <= num_args)
return retval;
else
return 0;
}
// payroll.cpp
// Member function definitions for class <payrolls>, employee list organized as array of objects.
// David Keil, Framingham State College, 6/98
#include "payroll.h"
#include "employee.cpp"

payrolls::payrolls(char nm[]) // Constructor.
{ 
    num_recs = 0;
    strcpy(file_name,nm);
}

payrolls::~payrolls() // Destructor. Prompts to save all records.
{ 
    cout << "Save file? ";
    char ch;
    cin >> ch;
    if (ch == 'y') 
        save();
}

void payrolls::edit_rec() // Prompts for record #, new value.
{ 
    cout << "Record # to edit: ";
    int rec_num;
    cin >> rec_num;
    if (rec_num > 0 && rec_num <= num_recs) 
    { 
        cout << "Record: ";
        emp[rec_num-1].display();
        cout << ".  New value: ";
        emp[rec_num-1].input_from_user();
    }
}

bool payrolls::retrieve() // Retrieves data file from disk into array.
{ 
    ifstream infile(file_name,ios::nocreate);
    if (infile.fail()) 
    { 
        cout << file_name << " not found\n";
        return FALSE;
    } else 
    { 
        while (emp[num_recs].retrieve(infile) 
            && num_recs < MAX_RECS)
            ++num_recs;
        infile.close();
        return TRUE;
    }
}

void payrolls::save() // Prompts user for permission to save data to disk, does so.
{ 
    ofstream outfile("payroll.dat");
    for (int i=0; i < num_recs; ++i) 
    { 
        emp[i].save(outfile);
    }
    outfile.close();
}

void payrolls::display() // Presents array on screen with items numbered.
{ 
    cout << \nFile " << file_name << ":\n";
    for (int i=0; i < num_recs; ++i) 
    { 
        cout << i + 1 << ". ";
        emp[i].display();
    }
}

void payrolls::add_rec() // Prompts for part name, appends it to array of records.
{ 
    if (num_recs < MAX_RECS) 
    { 
        emp[num_recs++].input_from_user();
    }
}

void payrolls::delete_rec() // Prompts for record number, deletes record.
{ 
    cout << \n# of record to delete: ";
    int rec_num;
    cin >> rec_num;
    if (rec_num > 0 && rec_num <= num_recs) 
    { 
        for (int i=rec_num-1; i <= num_recs; ++i)
            emp[i] = emp[i+1];
        --num_recs;
    }
}
// employee.h
// Declares a class to encapsulate an employee.
// David Keil, Framingham State College, 6/98
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <iostream.h>
#include <string.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdarg.h>

typedef int bool;
const bool FALSE = 0, TRUE = 1;

class employees
{
    char ID[12];
    char name[80];
    long int salary;
public:
    employees();
    employees(char* id, char* nm, long int sal);
    void display();
    void input_from_user();
    bool retrieve(ifstream& is);
    void save(ofstream& os);
    friend ostream& operator<<(ostream& os, employees e);
};
#endif // define EMPLOYEE_H

// employee.cpp
// Defines member functions of class employees.
// David Keil, Framingham State College, 6/98
#include "employee.h"

employees::employees() // Default constructor.
{
    *ID = *name = 0;
    salary = 0;
}

employees::employees(char* id, char* nm, long int sal) // Constructor.
{
    strcpy(ID, id);
    strcpy(name, nm);
    salary = sal;
}
Defining and using an iterator class

```cpp
// scoriter.cpp
// Defines and inputs collection of scores,
// uses iterator to display it.
// David Keil, Framingham State College, 10/98
#include <iostream.h>

typedef int bool;
const bool false=0,true=1;

class int_collections
{
public:
    int_collections() { num_scores = 0; };
    void input();
    void display();
friend class IC_iterators;
private:
    enum {MAX_INTS = 50};
    int element[MAX_INTS];
    int num_scores;
};
class IC_iterators
{
public:
    IC_iterators(int_collections* p_arg)
    { p_collection = p_arg; };
void first() { current = 0; }
bool next();
bool operator!=(int subscript);
int operator*();
private:
    int_collections* p_collection;
    int current;
};

// Class <int_collections>:
void int_collections::input()
// Prompts for scores until user enters negative value.
{ int input;
for( ; ;) 
    { cout << "Enter an integer (-1 to quit): ";
    cin >> input;
if (input >= 0 && num_scores < MAX_INTS)
    element[num_scores++] = input;
else
    break;
}

// Class <IC_iterators>:
bool IC_iterators::next()
{ int ok = (current < int_collections::MAX_INTS-1);
    if (ok)
        ++current;
    return ok;
}

bool IC_iterators::operator!=(int subscript)
{ if (subscript == NULL)
    return (current < p_collection->num_scores);
else
    return (current != subscript);
}

int IC_iterators::operator*()
{ return p_collection->element[current];
}

// MAIN
void main()
{ int_collections score_history;
    IC_iterators I(&score_history);
    // Apply input and output routines to
    // each item in the collection:
    score_history.input();
for (I.first(); I != NULL; I.next())
    cout << *I << " ";
    cout << endl;
}

Sample I/O:
Enter an integer (-1 to quit): 3
Enter an integer (-1 to quit): 5
Enter an integer (-1 to quit): 7
Enter an integer (-1 to quit): 9
Enter an integer (-1 to quit): -1
3 5 7 9
```
Linked list, stack, and binary search tree classes

The three programs below show how some classic data structures are implemented and how they support the abstract concept of a collection.

// emplist.cpp
// Builds a linked list of employees from file, // displays it.
#include <iostream.h>
#include <string.h>
#include <ctype.h>
#include "employee.cpp"

class list_nodes
{
public:
    list_nodes() { next = NULL; };
    list_nodes(employees e)
    { emp = e; next = NULL; };
    employees get_value() const { return emp; };
    list_nodes *get_next() { return next; };
friend class employee_lists;
private:
    employees emp;
    list_nodes *next;
};
class employee_lists
{
public:
    employee_lists() { header.next = NULL; };
    bool retrieve(char* file_name);
    void prepend(employees e);
    void display();
    list_nodes *first() { return header.next; };
    list_nodes *get_header() { return &header; };
private:
    list_nodes header;
};

bool employee_lists::retrieve(char* file_name)
// Reads list of employees from file.
{
    ifstream infile(file_name,ios::nocreate);
    if (infile.fail())
    {   cout << file_name << " not found\n";
        return FALSE;
    }else
    {
        employees input;
        while (input.retrieve(infile))
            prepend(input);
        infile.close();
        return TRUE;
    }
}
void employee_lists::prepend(employees e)
// Inserts a node containing data value <s> at // the beginning of a linked list.
{
    list_nodes *new_node = new list_nodes(e);
    if (header.next == NULL)
    
        header.next = new_node;
    else

void employee_lists::display()
// Outputs data values of all nodes of list.
{
    list_nodes *p_node = header.next;
    while (p_node != NULL)
    {   p_node->emp.display();
        p_node = p_node->next;
    }
}

void main()
{
    // Build list from user input, and display:
    employee_lists list;
    list.retrieve("payroll.dat");
    list.display();
}
The following program shows how simple a collection may be to build if a class is already defined for the contained object (employees) and if a class template for the data structure (stacks here) is defined.

// empstack.cpp
// Builds a linked list of employees from file, // displays it.
#include "employee.cpp"
#include "liststac.h"

void main()
{
    // Build list from user input, and display:
    stacks<employees> stack;
    stack.push(employees("1234","Yang",30000));
    stack.push(employees("5678","Rosen",28000));
    while(! stack.is_empty())
    {   employees emp = stack.pop();
        emp.display();
    }
}
A binary search tree is composed of nodes, ordered by a key, with two links each, one to a left and one to a right subtree. The tree's advantage is quick access to a given node.

// treesort.cpp
// Reads all words in a text file, stores // in binary tree, displays sorted list.
#include <iostream.h>
#include <string.h>
#include <fstream.h>
#include <stdlib.h>
class words
{
    char spelling[40];
    words *left,
    *right;
public:
    words()
    { *spelling = '\0'; left = right = NULL; }
    friend class word_trees;
};
class word_trees
{
    words *root;
public:
    word_trees() { root = NULL; }
    words *get_root() { return root; }
    void retrieve(char *filenm); // Reads text file word by word, stores in tree using <insert>.
    void display(words *subtree); // Prints subtree thru inorder traversal.
    void insert(char *s,words *&nd); // Creates node with string <s>, // inserts it into tree below node <nd>.
    words *new_node(char *s); // Allocates node, assigns <s> to it.
};

void main()
{
    word_trees dictionary;
    cout << "\nText of file DECL_IND.DAT: \n";
    dictionary.retrieve("decl_ind.dat");
    cout << "\nText of same file, "
    << sorted alphabetically: \n";
    dictionary.display(dictionary.get_root());
}

void word_trees::display(words *subtree)
// Prints subtree thru inorder traversal.
{
    if (subtree != NULL)
    {
        if (subtree->left)
            display(subtree->left);
        cout << subtree->spelling " ";
        if (subtree->right)
            display(subtree->right);
    }
}

void word_trees::retrieve(char *filenm)
// Reads text file word by word, stores in tree using <insert>.
{
    ifstream infile(filenm);
    if (infile.bad())
    {
        cout << filenm << " not found.\n"
        exit(1); // in <stdlib.h>
    }
    char word[80];
    do
    {
        infile >> word;
        cout << word " ";
        insert(word,root);
    }
    while (*word);
    infile.close();
    cout << endl;
}

void word_trees::insert(char *s,words *nd)
// Creates node with string <s>, // inserts it into tree below node <nd>.
{
    if (nd)
    {
        int strdiff = strcmp(s,nd->spelling);
        if (strdiff > 0)
            if (nd->right)
                insert(s,nd->right);
            else
                nd->right = new_node(s);
        else
            if (strdiff < 0)
                if (nd->left)
                    insert(s,nd->left);
                else
                    nd->left = new_node(s);
    }
    else
        nd = new_node(s);
}

words *word_trees::new_node(char *s)
// Allocates node, assigns <s> to it.
{
    words *node = new words;
    strcpy(node->spelling,s);
    return node;
}
A sample of how the STL is used

The program below uses a container class, an iterator class, and an algorithm to store data of two different kinds in collections and then to search these collections. All three are generic: the vectors class stores items of any type; the iterators class may be associated with any type of collection; and the find function may be run on any of a broad category of collections, including C-style strings as in the demo program below. This assembling of generic containers, iterators, and algorithms is the basic design architecture of the Standard Template Library.

```
#include "stldemo.h"

void main()
{
    // Use search algorithm with C-style array:
    const int MAX = 6;
    int A1[MAX] = {2,5,3,9,4};
    cout << "Your array is ";
    for (int i=0; i < MAX; ++i)
        cout << A1[i] << " ";
    cout << endl << "Enter an integer 
" << "to search for: ";
    int search_key;
    cin >> search_key;
    int* match = find(A1,A1+MAX,search_key);
    if (match < A1+MAX)
        cout << "Found " << *match << " at 
" << "location " << match - A1 << endl;
    else
        cout << search_key << " not found" << endl;

    // Use search algorithm with C++ vector:
    char A2[] = "iterators!";
    int A2size = strlen(A2);
    vectors<char> score_list(A2,A2size);
    cout << "Your vector:" << score_list << endl;
    cout << "Enter a character to search for: ";
    char search_key2;
    cin >> search_key2;
    iterators<vectors<char>,char> match2
        (&score_list,0);
    match2 = find(score_list.begin(),
                  score_list.end(),search_key2);
    if (match2 < score_list.end())
        cout << "Found " << *match2
            " at location 
            " << match2 - score_list.begin() << endl;
    else
        cout << search_key2 << " not found" << endl;
}
```

```
Output:
Your array is 2 8 5 3 9 4
Enter an integer to search for: 5
Found 5 at location 2
Your vector is i t e r a t o r s !
Enter a character to search for: a
Found a at location 4
```
Sample I/O of an elevator simulator

The listing below shows part of one sample run of the Deitel/Deitel solution to the elevator simulation problem. This listing may help you perform an object-oriented analysis of the problem by showing what some reasonable specifications for the problem are.

The only user input is the number of seconds of duration for the elevator simulation. Note that the arrival of persons is randomized in the application to simulate how an elevator is actually used.

Enter length of elevator simulation: 20
First person scheduled to arrive on floor 1 at time 15
First person scheduled to arrive on floor 2 at time 7
STARTING ELEVATOR SIMULATION
Elapsed time:  0
Elevator waiting on floor floor 1 for passengers
Elapsed time:  1
Elevator waiting on floor floor 1 for passengers
Elapsed time:  2
Elevator waiting on floor floor 1 for passengers
Elapsed time:  3
Elevator waiting on floor floor 1 for passengers
Elapsed time:  4
Elevator waiting on floor floor 1 for passengers
Elapsed time:  5
Elevator waiting on floor floor 1 for passengers
Elapsed time:  6
Elevator waiting on floor floor 1 for passengers
Elapsed time:  7
Person 1 created
Elevator called from floor 2
Person 1 walked onto floor 2 and pressed the down button
Next person scheduled to arrive on floor 2 at time 18
Bell reset
Elevator stopped
floor door opened
Call button reset
elevator door opened
Bell sounded
Light turned on
Person 1 entered the elevator
Person leaving floor 2
Bell reset
Light turned off
elevator door closed
floor door closed
Elevator starting to move down
Elapsed time:  13
Elevator moving down
Elapsed time:  14
Elevator moving down
Elapsed time:  15
Elevator moving down
Person 2 created
Elevator called from floor 1
Person 2 walked onto floor 1 and pressed the up button
Next person scheduled to arrive on floor 1 at time 22
Elapsed time:  16
Elevator moving down
Elapsed time:  17
Elevator stopped
floor door opened
Call button reset
elevator door opened
Person 1 exited the elevator
Light turned on
Person 2 entered the elevator
Person leaving floor 1
Bell reset
Light turned off
elevator door closed
floor door closed
Elevator starting to move up
Elapsed time:  18
Elevator moving up
Elapsed time:  19
Elevator moving up
Person 3 created
Elevator called from floor 2
Person 3 walked onto floor 2 and pressed the down button
Next person scheduled to arrive on floor 2 at time 31
Elapsed time:  20
Elevator moving up
Elapsed time:  21
Elevator moving up
Elapsed time:  22
Elevator moving up
The program below uses a function template to exchange two values of any type that supports the assignment operator: int, double, and a programmer-defined employees class.

```cpp
// swaptplt.cpp
// Shows swapping of values of various types
// using function template.
// David Keil, Framingham State College, 6/98
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class employees
{
 public:
  employees(char* nm, int sal)
  { strcpy(name,nm); salary = sal; }

  friend ostream& operator<<(ostream& os, employees emp); 

 private:
  char name[80];
  int salary;
};

ostream& operator<<(ostream& os, employees emp)
// Overloaded inserter operator for <employees>.
{ os << emp.name << setw(15-strlen(emp.name)) << emp.salary << endl;
  return os;
}

template <class T>
void swap(T& a, T& b)
// Swaps <a> and <b>.
{
  T old_a = a;
  a = b;
  b = old_a;
}

void main()
{
  // Initialize variables, display values:
  int I1 = 2, I2 = 4;
  double F1 = 0.5, F2 = 0.8;
  employees emp1("Carl",20000),
             emp2("Valerie",30000);
  cout << "I1 = " << I1 
       << " I2 = " << I2 << endl 
       << "F1 = " << F1 
       << " F2 = " << F2 << endl 
       << "emp1 = " << emp1 
       << "emp2 = " << emp2 << endl;

  // Swap:
  swap(I1,I2);
  swap(F1,F2);
  swap(emp1,emp2);

  // Redisplay:
  cout << "I1 = " << I1 
       << " I2 = " << I2 << endl 
       << "F1 = " << F1 
       << " F2 = " << F2 << endl 
       << "emp1 = " << emp1 
       << "emp2 = " << emp2 << endl;
}
```

Output:

I1 = 2  I2 = 4  
F1 = 0.5  F2 = 0.8  
emp1 = Carl  20000 
emp2 = Valerie 30000 

Swapping  
I1 = 4  I2 = 2  
F1 = 0.8  F2 = 0.5  
emp1 = Valerie 30000  
emp2 = Carl 20000
A class template for collections

// colxtplt.cpp
// Implements collection class template.
// Prompts for, echoes collections of <int>
// and characters.
#include <iostream.h>

const int MAX_ITEMS = 100;

template <class T>
class collections
{
    public:
    collections()
    { num_items = 0; }
    void insert(T new_item)
    { item[num_items++] = new_item; }
    void display();

    private:
    T item[MAX_ITEMS];
    int num_items;
};

template <class T>
void collections<T>::display()
// Displays all items.
{
    for (int i=0; i < num_items; ++i)
        cout << item[i] << " ";
    cout << endl;
}

void main()
{
    // Define, populate two collections:
    collections<int> int_colx;
    int_colx.insert(1);
    int_colx.insert(2);
    int_colx.insert(3);
    int_colx.insert(4);
    collections<char> char_colx;
    char_colx.insert('t');
    char_colx.insert('e');
    char_colx.insert('m');
    char_colx.insert('p');
    char_colx.insert('l');
    char_colx.insert('a');
    char_colx.insert('t');
    char_colx.insert('e');
    char_colx.insert('s');
    char_colx.insert('!');
    // Display contents of both collections:
    int_colx.display();
    char_colx.display();
}

// Class <collections>:

template <class T>
void collections<T>:::display()
// Displays all items.
{
    for (int i=0; i < num_items; ++i)
        cout << item[i] << " ";
    cout << endl;

With the class template defined in the program to the left, a collection class is available for any contained type: numbers, characters, objects, etc. 

Output: 

```
1 2 3 4 templates!
```
A collection of points on a coordinate axis

A collection may be implemented in many ways: using an array of objects, a pointer to an array of objects, an array of pointers to objects, a linked list, a stack, a queue. The program below uses a class template to store a collection of points objects in a stack. The stack is implemented as a linked list.

```cpp
#include "liststac.h"

class points
{
public:
    points() { x = y = 0; }
    points(int x_init, int y_init) { x = x_init; y = y_init; }
friend ostream& operator<<(ostream& os, points p);  // Overloaded inserter.
private:
    int x, y;
};

ostream& operator<<(ostream& os, points p)  // Overloaded inserter.
{
    cout << "(" << p.x << ", " << p.y << ") \n";
    return os;
}

void main()
{
    stacks<points> path;
    path.push(points(0, 0));
    path.push(points(1, 1));
    path.push(points(2, 4));
    path.push(points(3, 9));
    path.push(points(4, 16));
    while (!path.is_empty())
    {
        cout << path.pop() << endl;
    }
}
```

Output:

```
(4, 16)
(3, 9)
(2, 4)
(1, 1)
(0, 0)
```
The library `stldemo.h`, below, is based on the container/iterator/algorithm architecture used in the Standard Template Library. The earlier handout, “A sample of how the STL is used,” presented a program, `stldemo.cpp`, that used these templates. These three generic components are implemented with templates: the `vectors` class template stores items of any type; the `iterators` class template may be associated with any type of collection; and the `find` function template may be run on any of a broad category of collections, including C-style strings.

```cpp
// stldemo.h
// Container, iterator, algorithm modelled
// on STL library.
// David Keil, Framingham State College, 6/98
#ifndef STLDEMO_H
#define STLDEMO_H
#include <iostream.h>
#include <string.h>
typedef int bool;
const bool FALSE=0,TRUE=1;
typedef char* cstring;

// Generic array-based sequence container:
template<class T>
class vectors
{
  public:
    enum { MAX_ITEMS = 50 }; 
    vectors() { size = 0; }
    vectors(T A[],int SZ)
    { size = SZ;
      for (int i=0; i < size; ++i)
        item[i] = A[i]; }
    void insert(T new_item)
    { item[size++] = new_item; }
    T at(T* p) { return *p; }
    iterators<vectors<T>,T> begin()
    { return iterators<vectors<T>,T>(this,item); }
    iterators<vectors<T>,T> end()
    { return iterators<vectors<T>,T>(this,item+size); }
    vectors<T> operator=(vectors<T> V)
    { size = V.size;
      for (int i=0; i < size; ++i)
        item[i] = V.item[i];
      return *this; }
    friend ostream& operator<<(ostream& os,vectors<T> v);
    template <class T>
    friend iterators<vectors<T>,T> begin()
    { return <(ostream& os,vectors<T> v)
    { for (int i=0; i < v.size; ++i)
      cout << v.item[i] << " ";
      return os;
```
// Iterator class template:
template <class Tcollection, class Telement>
class iterators {
    public:
        iterators(Tcollection* pC, Telement* ref)
            { pcollection = pC; p_element = ref; }
        Telement operator*() { return pcollection->at(p_element); }
        Telement* operator++()
            { p_element += 1; return p_element; }
        bool operator<(iterators<Tcollection, Telement> I2)
            { return (p_element < I2.p_element); }
        int operator-(iterators<Tcollection, Telement> I2)
            { return p_element - I2.p_element; }
        bool operator!=(iterators<Tcollection, Telement> I2)
            { return p_element != I2.p_element; }
    private:
        Tcollection* pcollection;
        Telement* p_element;
};

// Generic search algorithm:
template<class I, class T>
I find(I first, I last, T key) {
    I i = first;
    while (i != last && *i != key)
        ++i;
    return i;
}
#endif // STLDEMO_H

The advantage of the style of programming used above is that a single standard library module provides a wide variety of built-in functionality for a generic collection. A programmer using the library does not need to “reinvent the wheel” but rather may use this built-in standard type and set of operations. Furthermore, the STL defines a whole set of collections, each with special performance advantages, such as linked lists, trees, and priority queues. In addition, a single iterator class template like find (called an algorithm in STL terminology) can search a wide variety of different container types. The library thus provides a set of somewhat interchangeable components from which may be chosen the most appropriate tools for a task.

Associated with each collection should be an iterator. In the sample program, stldemo.cpp, it is declared in this way:

```
iterators<vectors<char>, char> match2(&score_list, 0);
```

This declares match2 as an iterator object associated with the character-vector collection named score_list.

An iterator has a pointer to a collection and a pointer to one element of the collection. It is like a bookmark or cursor.

find is an algorithm available in the STL for a wide variety of containers. I is an iterator class and T is the type of the contained objects in the container that find will search. The search will begin at the collection element indicated by first and end at the element just before last. If find returns last, then the search is unsuccessful.
Inheritance with employee classes

// emplinh.cpp
// Demonstrates inheritance with
employee classes.
#include <iostream.h>
#include <iomanip.h>

class employees
{
public:
  employees(int I) : ID(I) { };
  int get_id() { return ID; }
private:
  int ID;
};

class hourly : public employees
{
public:
  hourly(int I,double W,double H) :
    wage_rate(W),hours(H),employees(I) { };
  friend ostream& operator<<(ostream& os,hourly e)
    (ostream& os,hourly e);
private:
  double wage_rate,hours;
};

ostream& operator<<(ostream& os,hourly e)
{
  os << setw(6) << e.get_id()
     << setw(6) << e.wage_rate
     << setw(6) << e.hours;
  return os;
}

void main()
{
  hourly emp(1234,15.00,32);
  cout << emp << endl;
}

Problems:
1. Modify the program to define a class of salaried employees derived from employees.
2. Declare an array of hourly employees and an array of salaried ones. Input values into each and display those values, including a calculated weekly paycheck.
A class of strings of arbitrary length using inheritance

The class declared below uses inheritance to build on the functionality of the strings class, whose limit is 1000 characters. An lstring may have an unlimited size. It is implemented as a linked list.

Notice the header of the class declaration:

```cpp
class lstrings : public strings
```

As a class that uses public inheritance, lstrings makes public members of its base class part of its public interface.

Conceptually, lstrings are a kind of strings. The designer of a class must decide whether inheritance is appropriate conceptually as well as technically. (Is a circle a kind of point? A point has a location and a circle has a location. But we don’t think of a circle as a kind of point in day-to-day life. A long string, however, may be called a kind of string.)

```cpp
// lstrings.h
// Safe string manipulation class
// library.
#include "strings.h"
class lstrings : public strings
{
    lstrings* next;
public:
    lstrings();
lstrings(char* s);
lstrings(strings s);
    boolean concat(char* s);
    boolean concat(lstrings s);
    boolean insert(unsigned int loc, char* s);
    boolean delet(unsigned int loc, int num_chars);
    lstrings substring(unsigned int loc, int num_chars);
    boolean overwrite(unsigned int loc, char* s);
    void input();
    void display();
    friend ostream& operator<<(ostream& os, lstrings s);
};
```

The base class, strings (defined in an earlier example in the library file strings.h) has a single data member, a C-style string, content:

```cpp
class strings
{
    private:
        char content[STR_MAX];
    public:
        strings();
        strings(char* s);
        ...
};
```

Most of the functions for lstrings are currently implemented in library file, lstrings.cpp, only as stubs. An exception is the conversion constructor that takes a C-style string as a parameter:

```cpp
lstrings::lstrings(strings s)
// Conversion constructor from <char*>.
{
    assign(s.get_content());
}
```

This function must use a base-class access function, get_content, rather than directly passing the data member content as a parameter, because content is a private member of the base class strings. Derived classes do not have access to private members of their base classes. They do, however, have access to public and protected members. The difference between a protected member and a public one is that classes have access to protected members of their base classes, but to other code a protected member is inaccessible.

Challenge problem:

The class strings has a public access function, get_content, that returns a pointer to the content data member. Is it appropriate to make that public for lstrings, when it returns only part of the long string stored in an lstrings instance? Discuss two alternative implementations of the library that would change the access status of the base-class members.
General-purpose arrays using inheritance

Base-class declaration defines a generic array:

```cpp
// array.h
// Generic array base class, available for inheritance.
#include <iostream.h>
class arrays
{
protected:
    int max_size, size;
    void **element;
public:
    arrays()
    { max_size = size = 0; }
    void set_max_size(int sz)
    { max_size = sz; }
    void *get_element(int subscript)
    { if (subscript < size) return element[subscript]; else return NULL; }
    void *set_element(int subscript, void *value)
    { if (subscript < size) element[subscript] = value; }
    void insert(void *value)
    { if (size < max_size) element[size++] = value; }
};
```

The `arrays` base class may be used to create a derived class of string arrays:

```cpp
// strarray.cpp
// Reads strings from keyboard, displays them.
// Uses generic array class <arrays>.
#include "array.h"
#include <string.h>
typedef char *char_ptrs;
class string_arrays: public arrays
{
public:
    string_arrays(int sz);
    char *str_ele(int subscript);
    void insert(char *s);
    void get();
    void display();
};

string_arrays::string_arrays(int sz)
// Constructor.
{ 
    arrays::arrays();
    set_max_size(sz);
    element = (void **)(new char_ptrs[sz]);
}

char *string_arrays::str_ele(int subscript)
// Returns array element <subscript>, a string.
{ 
    return (char *)get_element(subscript);
}

void string_arrays::insert(char *s)
// Adds contents of <s> to array.
{ 
    char *str = new char[strlen(s)+1];
    strcpy(str,s);
    arrays::insert(str);
}

void string_arrays::get()
// Prompts for a list of strings from user.
{ 
    int i = size;
    while (i < max_size) {
        char input[80];
        cout << "Enter a string (blank to quit): ";
        cin.getline(input,80);
        if (strlen(input) > 0)
            insert(input);
        else
            break;
    ++i;
}
}

void string_arrays::display()
// Writes all strings to screen.
{ 
    for (int i=0; i < size; ++i)
        cout << str_ele(i) << endl;
}

void main()
{ 
    string_arrays A(10);
    A.get();
    A.display();
}
```
A virtual function's address is resolved at runtime

// animal.cpp
// Tells how animals move.
#include <iostream.h>

class animals
{
public:
    animals() {}
    void tell_motion() { cout << "\nAnimals move around.\n"; }
};

void main()
{
    animals a;
    a.tell_motion();
}

// bird1.cpp
// Tells how birds move.
#include <iostream.h>

class animals
{
public:
    animals() { }
    void tell_motion() { cout << "\nAnimals move around.\n"; }
};

class birds: animals
{
public:
    birds() { }
    void tell_motion()
    { cout << "\nBirds move on two legs and wings\n"; }
};

void main()
{
    birds b;
    b.tell_motion();
}
// bird2.cpp
// Tries to tell how birds move and reproduce.
#include <iostream.h>
class animals
{
public:
  animals() { };
  void tell_motion() { cout << "Animals move around.\n"; };
  void tell_reproduction() { cout << "Animals reproduce.\n"; };
  void tell_all() { tell_motion(); tell_reproduction(); };
};
class birds: public animals
{
public:
  birds() { };
  void tell_motion()
  { cout << "Birds move on two legs and fly.\n"; };
  void tell_reproduction()
  { cout << "Birds lay eggs.\n"; };
};
void main()
{
  cout << endl;
  birds b;
  b.tell_all();
}

// bird3.cpp
// Tells how birds move and reproduce. Uses virtual functions.
#include <iostream.h>
class animals
{
public:
  animals() { };
  void virtual tell_motion()
  { cout << "Animals move around.\n"; };
  void virtual tell_reproduction()
  { cout << "Animals reproduce.\n"; };
  void virtual tell_all()
  { tell_motion(); tell_reproduction(); };
};
class birds: public animals
{
public:
  birds() { };
  void virtual tell_motion()
  { cout << "Birds move on two legs and fly.\n"; };
  void virtual tell_reproduction()
  { cout << "Birds lay eggs.\n"; };
};
void main()
{
  cout << endl;
  birds b;
  b.tell_all();
}

Output:
Animals move around.
Animals reproduce.

Output:
Birds move around on two legs and fly.
Birds lay eggs.
Polymorphism in graphics

The program below illustrates how polymorphism and virtual functions permit us to create a collection of objects of mixed classes and run through this collection having each object call a function of the same name, *draw* here. Each object knows its own class, so it will call the virtual function *draw* of its own class to draw itself properly.

The program is based on the following inheritance hierarchy:

```cpp
// shapes.cpp
// Defines an array of three drawable items
// (two lines, two dots), draws them on
// text screen.
// Declares <shapes> class and derived
// <dots> and <horiz_lines> classes.
#include <iostream.h>
#include <iomanip.h>

// The base class, an abstract class, has a pure
// virtual function <draw>, whose derived-class
// versions are called to draw each item.
class shapes
{
public:
    shapes() {};
    void set_x(int x) { x_origin = x; };
    int get_x() { return x_origin; }
    virtual void draw() = 0;
    // may not be called
private:
    int x_origin;
};

class dots : public shapes
{
public:
    dots(int x) { set_x(x); }
    virtual void draw()
    { cout << setw(get_x()) << "X" << endl; }
};

class horiz_lines : public shapes
{
public:
    horiz_lines(int x, int len)
    { set_x(x); length = len; }
    virtual void draw();
private:
    int length;
};

// Class <horiz_lines>:

void horiz_lines::draw()
// Puts rule of <length> at origin.
{ 
    cout << setw(get_x()) << ";
    for (int i=0; i < length; ++i)
        cout << "X"
    cout << endl;
}

void main()
{
    // Spec an array of items:
    shapes *item[] =
    { 
        new horiz_lines(3,10),
        new dots(4),
        new dots(5),
        new horiz_lines(6,20),
    }
    // Loop to draw each item:
    for (int i=0; i < (sizeof item) /
        (sizeof item[0]); ++i)
        item[i]->draw();
}
```

The output of this program is a line of 10 X’s at horizontal position 3, a dot at position 4, a dot at position 5, and a line of 20 X’s at position 6:

```
xxxxxxxxxxxx
X
X
xxxxxxxxxxxxxxxxxxxxxx
```
An application framework uses polymorphism

The application wincalc.cpp below uses an application framework defined in the library files winfrmwk.h and winfrmwk.cpp. The library defines a base class, applications, among others. The application wincalc uses an application class, calculators, derived from applications. The

```cpp
// wincalc.cpp
// Repeatedly prompts for 2 integers,
// displays sum.
// Uses application framework defined in
// <winfrmwk.h>.
#include "winfrmwk.h"

class calculators : public applications
{
public:
    calculators() { };  
    virtual void init_prompt()
    { set_prompt("Choose an operation"); };  
    virtual void init_menu()
    { menu.set("+ Add","Q Quit",""__); };  
    virtual void handle_event(events event);

void main()
{
    // Declare an application, <calc>,
    // an instance of this program's class
    // <calculators> that inherits from
    // <applications> library class:
    calculators calc;
    // Run the application instance:
    calc.run();
}

void calculators::handle_event(events event)
// Virtual event handler for descendant of
// <applications>.
{
    // Handle base-class events, e.g.,
    // input of 'Q' to quit:
    applications::handle_event(event);
    // Handle events defined in this
default:
    switch(event.get_text())
    {
    case '+':
        cout << "Enter 2 integers: ";
        int input_1, input_2, sum;
        cin >> input_1 >> input_2;
        sum = input_1 + input_2;
        cout << input_1 << " + " << input_2
            << " = " << sum << endl;
        break;
    }
}
```

Sample I/O:
```
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Choose an operation</td>
</tr>
<tr>
<td>------------------------------------</td>
</tr>
<tr>
<td>+ Add     Q Quit</td>
</tr>
</tbody>
</table>
```

An application framework

```cpp
// winfrmwk.h
// Declares text-mode library classes
// <events>, <windows>, <menus>,
// <applications>.
// David Keil, Framingham State College, 7/98
#ifndef WINFRMWK_H
#define WINFRMWK_H
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <stdlib.h>
const int MAX_WIN_HT = 20,  // in rows
        MAX_WIN_WIDTH = 80; // in columns
enum event_types
{ev_nothing, ev_escape, ev_menu_selection};

class events
{
public:
    events() { category = ev_nothing; };
    char get_text() { return text; };
    void fetch();
private:
    event_types category;
    char text;
};

class models
// Data model for storage in a window.
{
public:
    models() { num_lines = 0; }
    void set_line(char* s)
    { strcpy(line[0],s); num_lines++; }
    char* get_line(int line_num)
    { return line[line_num]; }
    void display(int line_num)
    { cout << line[line_num]; }
private:
    char line[MAX_WIN_HT-2][MAX_WIN_WIDTH-2];
    int num_lines;
```

```cpp
const int MAX_WIN_HT = 20,  // in rows
        MAX_WIN_WIDTH = 80; // in columns
enum event_types
{ev_nothing, ev_escape, ev_menu_selection};

class events
{
public:
    events() { category = ev_nothing; };
    char get_text() { return text; };
    void fetch();
private:
    event_types category;
    char text;
};

class models
// Data model for storage in a window.
{
public:
    models() { num_lines = 0; }
    void set_line(char* s)
    { strcpy(line[0],s); num_lines++; }
    char* get_line(int line_num)
    { return line[line_num]; }
    void display(int line_num)
    { cout << line[line_num]; }
private:
    char line[MAX_WIN_HT-2][MAX_WIN_WIDTH-2];
    int num_lines;
```
class windows
{
public:
    windows();
    windows(int x0, int y0, int w, int h, models* text);
    void set_line(char* s);
    int get_x() { return x; }
    int get_y() { return y; }
    int get_width() { return width; }
    int get_height() { return height; }
    void set(int x0, int y0, int w, int h);
    void draw_horiz();
    void draw_vert();
    virtual draw();
    void move();
private:
    int x, y, // screen pos of upper-left corner
    width, height; // size of window
    models* content;
};

const int MAX_MENU_OPTIONS = 6,
    MENU_OPT_WID = 12; // Width of single
label in columns
class menus: windows
{
public:
    menus() { num_options = 0; }
    void set(char opt1[],...);
    void draw();
private:
    char option[MAX_MENU_OPTIONS][40];
    int num_options;
};
class applications
{
public:
    applications();
    void virtual init_prompt() { };
    void virtual init_menu() { menu.set(" "); }
    void virtual handle_event(events event) {};
    void display_prompt();
    void set_prompt(char *s) { prompt_line.set_line(s); };
    void run();
private:
    windows prompt_line;
protected:
    menus menu;
};
#endif

Member function definitions for an
application framework

// winfrmwk.cpp
// Linkable application-framework library.
// Defines member functions for windows,
// menus, application class.
// David Keil, Framingham State College, 7/98
#include "winfrmwk.h"

// Class <windows>:

void windows::set_line(char* s)
// Sets content.
{
    if (!content)
        content = new models;
    content->set_line(s);
}

void windows::set(int x0, int y0, int w, int h)
// Access function.
{
    x = x0; y = y0; width = w; height = h;
}

void windows::draw_horiz()
// Draws row of graphic characters across
// top.
{
    cout << "|";
    for (int col = 2; col < width-2; ++col)
        cout << "-";
    cout << "|" << endl;
}

void windows::draw_vert()
// Draws column of graphic characters
// on left.
{
    for (int row = 1; row < height-1; ++row)
        if (content)
        {
cout << "| ";
content->display(row-1);
int len = strlen(content->get_line(row-1));
cout << setw(width - len - 3) << "|\n";
}
}
}

void windows::draw()
// Displays window.
{
    draw_horiz();
draw_vert();
draw_horiz();
}

// Class <menus>:

void menus::set(char opt1[], ...)
// Assigns parameters to option string array elements. Last arg must be null string.
{
    va_list ap; // declared in <stdarg.h>
    va_start(ap,opt1);
    for (num_options=0; opt1[0] != '\0'; ++num_options)
    {
        strcpy(option[num_options],opt1);
        opt1 = va_arg(ap,char *);
    }
    va_end(ap);
}

void menus::draw()
// Draws labels at top of screen. 4/96
{
    for (int i=0; i < num_options; ++i)
    {
        cout << setw(12) << option[i];
    }
    cout << endl;
}

// Class <applications>:

applications::applications()
// Default constructor.
{
    prompt_line.set_line("\"");
};

void applications::display_prompt()
// Shows prompt line in window.
{
    prompt_line.set(0,0,40,3);
prompt_line.draw();
};

void applications::run()
// Executes event loop, terminates on "Q" for Quit.
{
    init_prompt();
    init_menu();
    events event;
    char event_text;
    do {
        display_prompt();
        menu.draw();
        event.fetch();
        event_text = toupper(event.get_text());
        if (event_text != 'Q')
        {
            handle_event(event);
        }
    } while (event_text != 'Q');
cout << "\nGoodbye\n";
Examples related to exception handling

Standard C-style error handling

// div0.cpp
// Performs division on input values, trapping
// divide-by-zero errors locally.
#include <iostream.h>

float quotient(int a, int b);

void main()
{
    cout << "Enter two integers: ";
    int a, b;
    cin >> a >> b;
    cout << a << " / " << b << " = " << quotient(a, b) << endl;
}

float quotient(int a, int b)
// Returns (a/b). On divide-by-zero, shows message, returns 0.
{
    if (b != 0)
        return (float)a / b;
    else
    {
        cout << "Division by 0 is not permitted.\n";
        return 0;
    }
}

Error handling with assert terminates program prematurely

// div1.cpp
// Performs division on input values, using <assert> to
// trap divide-by-zero errors.
#include <iostream.h>
#include <assert.h>

float quotient(int a, int b);
void main()
{
    cout << "Enter two integers: ";
    int a, b;
    cin >> a >> b;
    cout << a << " / " << b << " = " << quotient(a, b) << endl;
}

float quotient(int a, int b)
// Returns (a/b). Terminates program on divide-by-zero.
{
    assert(b != 0);
    return (float)a / b;
}

C++-style exception handling separates
error detection from a program’s response

// div2.cpp
// Performs division on input values, using exception
// handling to manage divide-by-zero errors.
#include <iostream.h>
#include <string.h>

Programmer may define
class such as this to use
for throwing errors
float quotient(int a, int b);

class math_errors
{
public:
    char message[80];
    math_errors(char *msg) { strcpy(message, msg); }
};

void main()
{
    cout << "Enter two integers: ";
    int a, b;
    cin >> a >> b;
    try
    {
        cout << a << " / " << b << " = " << quotient(a, b) << endl;
    } catch (math_errors error)
    {
        cout << error.message << " is not permitted.\n";
    }
}

float quotient(int a, int b)
// Returns (a/b). Terminates program on divide-by-zero.
{
    if (b == 0)
        throw math_errors("Division by 0");
    return (float)a / b;
}
The ‘=’ operator is automatically overloaded as a constructor

In the program below, the conversion constructor declared $\text{strings}(\text{char}^* s)$, which converts from a C-style string data item to an instance of class $\text{strings}$, is called twice: once by passing a string literal (“string1”) to the constructor as a parameter, once by using the equal sign. The compiler treats the equal sign automatically as an overloaded operator function equivalent to the constructor.

```cpp
// strdemo2.cpp
// Demonstrates two ways to invoke conversion constructor for class <strings>.
#include "strings.h"
#include "strings.cpp"

void main()
{
    // Declare, initialize, display two strings:
    strings string1("string1"); // Conversion constructor
    string2 = "string2"; // '=' is automatically overloaded as copy/conversion constructor
    cout << string1 << endl << string2 << endl;
}
Output:
string1
string2
```

The declaration of class $\text{strings}$ appears below. The overloaded insertion operator is declared as a friend function. Notice the protected access function $\text{get_content}$. It is accessible only to classes publicly derived from $\text{strings}$.

```cpp
class strings
{
    char content[STR_MAX];
    protected:
        char* get_content() { return content; }
    public:
        strings();
        strings(char* s);
        boolean assign(char* s);
        boolean concat(char* s);
        boolean concat(strings s);
        int compare(char* s);
        unsigned int length();
        char nth_char(unsigned int n);
        char set_nth_char(unsigned int n, char ch);
        boolean insert(unsigned int loc, char* s);
        boolean delet(unsigned int loc, int num_chars);
        char* substring(unsigned int loc, int num_chars);
        boolean overwrite(unsigned int loc, char* s);
        void input();
        void display();
    friend ostream& operator<<(ostream& os,strings s);
};
```

Below is the definition of the overloaded inserter for class $\text{strings}$:

```cpp
ostream& operator<<(ostream& os,strings s)
    // Inserter.
    {
        os << s.content;
        return os;
    }
```
An iterator class for an integer collection

The program below declares a class, `int_collections`, to store an array of integers and its size. It is associated with a friend class, `int_iterators`. The iterator performs looping operations on a collection.

The `main` function of the program declares one instance of the collection and one of the iterator. The iterator retrieves a file of integers into the collection and displays the collection. It does so using two versions of a member function, `apply`, that accepts a pointer to a function and executes that function on each element of the collection.

```cpp
// scorscan.cpp
// Defines array of scores, uses iterator class
// to retrieve and display it.
// David Keil, Framingham State College, 6/98
#include <iostream.h>
#include <fstream.h>
typedef int bool;
const bool false=0,true=1;
// Pointer-to-function data types:
typedef void (*functions_taking_int)(int);
typedef bool (*functions_taking_file)(int&, ifstream& infile);
// Utility functions:
bool retrieve(int& n, ifstream& infile);
void display(int n);
const int MAX_INTS = 10;
class int_collections
{
public:
    int_collections() { num_ints = 0; }
    friend class int_iterators;
private:
    int element[MAX_INTS];
    int num_ints;
};

class int_iterators
{
public:
    int_iterators(int_collections* p_arg);
    void first();
    bool next();
    void apply(functions_taking_int func);
    bool apply(functions_taking_file func, char* file_name);
private:
    int_collections* p_collection;
    int current;
};

void main()
{
    int_collections score_history;
    int_iterators scanner(&score_history);
    // Apply input and output routines to
    // each item in the collection:
    if (scanner.apply(retrieve,"scores.dat"))
    {
        cout << "Contents of SCORES.DAT:\n";
        scanner.apply(display);
        cout << endl;
    }
}

// Global functions:
bool retrieve(int& n, ifstream& infile)
// Reads from <infile> into <n>.
{
    if (infile.eof())
        return false;
    else
    {
        infile >> n;
        return true;
    }
}

void display(int n)
// Displays integer parameter.
{
    cout << n << " ";
}

// Class <int_iterators>:
int_iterators::int_iterators(int_collections* p_arg)
// Constructor.
{
    p_collection = p_arg;
    first();
}

void int_iterators::first()
// Sets index to point to first
// element in collection.
{
    current = 0;
}

bool int_iterators::next()
// Advances <current> if possible,
// reports success or failure.
{
    bool ok = (current < MAX_INTS-1);
    if (ok) ++current;
    return ok;
}

void int_iterators::apply
(functions_taking_int func)
// Scans <db> array, applying <func>
// to each element.
{
    first();
    do
    {
        func(p_collection->element[current]);
    }while (next());
}

bool int_iterators::apply
(functions_taking_file func, char* file_name)
// Opens file <file_name> and scans array,
// applying function <func> to each element
// of collection.
```
ifstream infile(file_name, ios::nocreate);
if (! infile)
{
    cout << "File " << file_name
         << " not found" << endl;
    return false;
}
else
{
    first();
    do {
        func(p_collection->element[current],
             infile);
    } while (next());
    return true;
}

boolean int_iterators::next()
// Advances <current> if possible,
// reports success or failure.
{
    int ok = (current < MAX_INTS-1);
    if (ok)
    {
        ++current;
        return ok;
    }
}

void int_iterators::apply
(functions_taking_int func)
// Scans <db> array, applying <func>
// to each element.
{
    first();
    do {
        func(p_collection->element[current]);
    } while (next());
}

Output:
Contents of SCORES.DAT:
1 2 3 4 5 6 7 8 9 10

Each iterator class is closely associated with a collection class. The iterator handles traversals of the collection or operations that must be performed on each item in the collection. An iterator has some features in common with a cursor or bookmark. The iterator class encapsulates a control aspect of a collection: looping. Separating the iterator class from the collection class has advantages; one collection may have two iterators scanning it at one time, for example.

The iterator is declared as a friend because it may be convenient to give the iterator access to private members.

The iterator object must know about both the collection it is associated with and about the item in the collection that is the iterator’s current position in the collection; p_collection and current fill these requirements in the class int_iterators.

The code that uses the iterator must have a way to set the iterator to the first item in the collection; this is done by the function first in the example. The iterator steps to the next item using the member function named next.

An apply function, a member of the iterator class, enables the programmer to scan the entire collection with one command, performing a particular operation on each item. The operation is specified by a pointer to the function that will carry it out, a global function. The iterator’s member

    void apply(functions_taking_int func);

for example, scans through the collection and causes the function func to be executed once for each item in the collection, with that item passed to func as a parameter. The programmer who uses the iterator determines what function will be passed.

The mechanism for passing functions a parameters is the pointer-to-function data type. One such type, function_taking_int, is declared in the example as follows:

    typedef void (*function_taking_int)(int);

This typedef declares a type identifier, function_taking_int, and says that an instance of this type is a pointer to a void function that takes an int parameter.