The art of programming is the art of managing complexity.

With processing power increased by a factor of thousand over the last ten to fifteen years, Man has become considerably more ambitious in selecting problems that now should be "technically feasible." Size, complexity and sophistication of programs one should like to make have exploded and over the past years it has become patently clear that on the whole our programming ability has not kept pace with these exploding demands made on it.

Edsger W. Dijkstra, circa 1972

4.1 Introduction

One of the C language shortcomings is that it lacks sophisticated string handling capabilities. Strings are very useful since nearly every program manipulates text data of one type or another.

The mystring class contains two data members

1. len - which holds the actual number of characters
2. ptr - the char pointer, points to the location on the heap of the buffer containing the string's text data.

There are lots of member functions which help you access and manipulate string objects.

4.1 Code listing - mystring class definition (mystring.h)

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

class  mystring
{
  char *ptr;               // pointer to allocated space
  int len;                 // current length of string
   public:
      mystring();          // zero argument constructor
      mystring(char *);    // one argument constructor
      mystring(char, int);              // two argument constructor
      mystring(const mystring &);       // copy constructor
      ~mystring();                      // destructor
      mystring & operator = (const mystring &);  // assignment operator
      operator const char *();               // conversion function

      // Concatenation functions
      friend mystring operator + (const mystring &, const mystring &);
      void operator += (const mystring &);
      friend mystring operator + (const mystring &, char);
      void operator += (char);

      // Access operator
      char & operator [] (int);

      // Output and input operator
      friend ostream & operator << (ostream &, mystring &);
      friend istream & operator >> (istream &, mystring &);
```

```
      // Comparison function
      int operator == (const mystring &) const;
      int operator != (const mystring &) const;
      int length() const;                   // value return function

      // Case conversion function
      void tolower();
      void toupper();
};
```

```
4.2 Code listing - String member functions implementation (mystring.cpp)
```

```
#include <string.h>
#include <mystring.h>
#include <iostream>
using namespace std;

// No argument constructor
mystring:: mystring()
{
   len = 0;
   ptr = new char[len + 1];
   ptr[0] = '\0';
}

// One argument constructor
mystring :: mystring(char *p)
{
   len = strlen(p);
   ptr = new char[len + 1];
   strcpy(ptr, p);
}

// Two argument constructor
mystring :: mystring(char filler, int count)
{
   len = count;
   ptr = new char [len + 1];
   memset(ptr, filler, count);
}

// Copy constructor
mystring :: mystring(const mystring & x)
{
   len = x.len;
   ptr = new char [len + 1];
   strcpy(ptr, x.ptr);
}

// Destructor
mystring :: ~mystring()
{
   delete ptr;
}
```

/* Overloaded assignment operator. We return references only when the object we reference exists even before the function is called. When we say a = b, a is invoking the assignment operator, so it already exists and we are returning a reference to the object invoking = (object a).

In this case we return (*this) to enable assignments of the type

```
      a = b = c = d = 20;
```

Always check for self-assignment. There are two reasons for doing this. It results in efficient code. If we detect an assignment to self at the top of our assignment, we can return right away, possibly saving a lot of work. Another reason is that of correctness. The assignment operator must typically free the resources allocated to an object (i.e., get rid of its old value) before it can allocate the new resources corresponding to its new value. When assigning to self, this freeing of resources can be disastrous, because the old resources might be needed during the process of allocating the new ones. Always assign all the data members in operator = (Refer to 4.4.1) */

```
mystring & mystring :: operator = (const mystring & x)
{
   if(this != &x)
   {
      len = x.len;
      delete ptr;
      ptr = new char [len + 1];
      strcpy(ptr, x.ptr);
   }
   return (*this);
}
```

// Conversion function - The casting operator function should satisfy the following conditions

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

```
mystring :: operator const char *()
{
   return ptr;
}

// Overloaded mystring addition operator
mystring operator + (const mystring & x, const mystring & y)
{
   int totallen;
   char *t;
   totallen = x.len + y.len;
   t = new char [totallen + 1];
   strcpy(t, x.ptr);
   strcat(t, y.ptr);
   mystring temp(t);
   delete t;
   return temp;
}

// Overloaded mystring addition operator +=
void mystring :: operator += (const mystring & x)
{
   int totallen;
   char *t;
   totallen = len + x.len;
   t = new char [totallen + 1];
   strcpy(t, ptr);
   strcat(t, x.ptr);
   delete ptr;
   ptr = t;
```

```
   len = totallen;
}

// Overloaded mystring addition operator
mystring operator + (const mystring & x, char ch)
{
   int totallen;
   char *t;
   totallen = x.len + 1;
   t = new char [totallen + 1];
   strcpy(t, x.ptr);
   t[x.len] = ch;
   t[x.len + 1] = '\0';
   mystring temp(t);
   delete t;
   return temp;
}

// Overloaded mystring addition operator +=
void mystring ::operator += (char ch)
{
   int totallen;
   char *t;
   totallen = len + 1;
   t = new char [totallen + 1];
   strcpy(t, ptr);
   delete ptr;
   ptr = t;
   ptr[len] = ch;
   ptr[len + 1] = '\0';
   len = totallen;
}

// Overloaded mystring comparison operator
int mystring :: operator == (const mystring & x) const
{
   int a;
   a = strcmp(ptr, x.ptr);
   return(!a);
}

// Friend function to output objects of mystring class
ostream & operator << (ostream & strm, mystring & x)
{
   strm << x.ptr;
   return strm;
}

// Friend function to input objects of mystring class
istream & operator >> (istream & strm, mystring & x)
{
   char buffer[256];
   if(strm.get(buffer, 256))
      x = mystring(buffer);
   return strm;
}

// Returns the number of characters
int mystring :: length() const
{
```

```
   return len;
}

// Convert to lower case. If the function strlwr() doesn’t exist, write one
void mystring :: tolower()
{
   strlwr(ptr);
}

// Convert to upper case. If the function strupr() doesn’t exist, write one
void mystring :: toupper()
{
   strupr(ptr);
}

char & mystring :: operator[] (int pos)
{
   return ptr[pos];
}
```

4.3 Code Listing - mystringMain.cpp

```
#include <mystring.h>
#include <iostream>
using namespace std;

void main()
{
   mystring s1("Genesis");
   mystring s2("Insoft");
   cout << "s1 = " << sizeof(s1) << endl;
   cout << "s2 = " << sizeof(s2) << endl;
   s1 = s2;
   s1[0] = 'D';
   cout << s1 << endl;
   cout << s2 << endl;
}
```

4.2 Overload assignment and copy constructor

Always define an assignment and the copy constructor for a class which has dynamically allocated memory (class which contains a pointer data member).

Let us assume we had not declared the assignment and the copy constructor for the mystring class. If we make the declaration,

```
   mystring a(“Hello”);
   mystring b(“World”);
```

then the situation is as shown in Fig. 4.1.

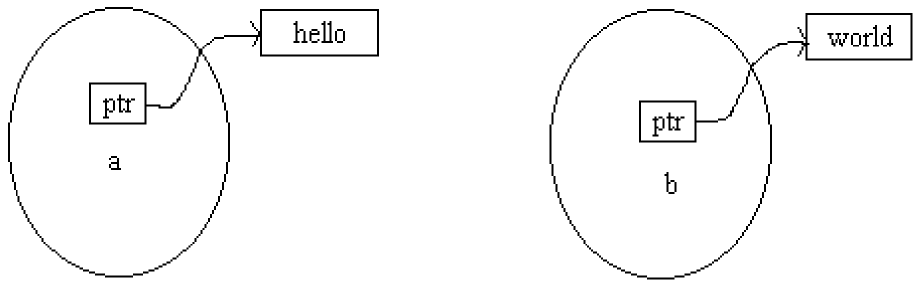

Fig. 4.1

Inside the object ‘a’ is a pointer to memory containing the character string “Hello”. Separate from that is an object ‘b’ containing a pointer to the character string “World”. If we now perform an assignment,

```
b = a;
```

there is no client defined operator = to call, so the C++ compiler generates and calls the default assignment operator instead. This default assignment operator performs member wise assignment from the members of ‘a’ to the members of ‘b’, which for pointers (a.ptr and b.ptr) is just a bitwise copy. The result of the assignment is as shown in Fig. 4.2.

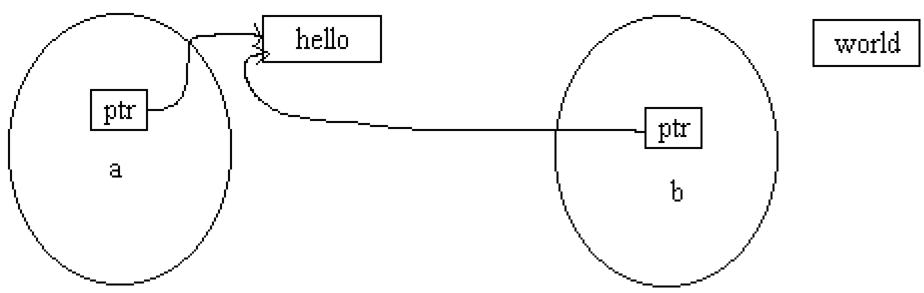

Fig. 4.2

There are atleast two problems with this. First, the memory that ‘b’ used to point to was never deleted; it’s lost forever. This is a classic example of how memory leaks can arise. Second, both ‘a’ and ‘b’ now contain pointers to the same character string. When one of them goes out of scope, its destructor will delete the memory still pointed to by the other. If one of the objects is modified, the other also gets modified. Because of the above problems, we have defined the assignment and the copy constructor for the mystring class, which solves the problem.

4.2.1 Assign to all data members in operator=

C++ will write an assignment operator for you if you don’t write one yourself. Item 4.2 describes why we write our own assignment operator. So you are probably wondering if we can have the best of both worlds, whereby you let C++ generate a default assignment operator and you selectively override those parts you don’t like. No such luck. If you want to take control of any part of the assignment process, you must do the entire thing yourself. In practice, it means that you need to assign to every data member of your object when you write your own assignment operator.

4.3 Prefer initialization to assignment in constructors

Consider the bank class with reference and const members.

4.2 Code listing

```
#include <string.h>
#include <mystring.h>
#include <iostream>
using namespace std;

class bank
{
   int & acctnum;
   mystring name;
   double balance;
   const float int_rate;
   public:
      bank(int &, mystring, double, float);
      void show();
};
```

When we write the bank constructor, you have to transfer the values of the parameters to the corresponding data members. There are two ways to do this. The first is to use the member initialization list:

```
bank :: bank(int & a, mystring acct_name, double new_bal, float roi)
:  acctnum(a), name(acct_name), int_rate(roi)
{
   balance = new_bal;
}
```

The second way is to make assignments in the constructor body:

```
bank :: bank(int & a, char * acct_name, double new_bal, float roi)
{
   acctnum = a;
   name =  acct_name;
   balance = new_bal;
   int_rate = roi;
}

void bank :: show()
{
   cout << "Name is " << name << endl;
   cout << "Acctnum is " << acctnum << endl;
   cout << "Balance is " << balance << endl;
   cout << "Interest rate is " << int_rate << endl;
}

void main()
{
   int id = 2;
   mystring cname("john");
   bank customer(id, cname, 5555.5, 12.5);
   customer.show();
}
```

There are two important differences to these two approaches.

Const and reference members can only be initialized, never assigned. So, if you decide that a bank object would never change its int_rate, you might declare it as a const.

This declaration requires that you use a member initialization list, because const members can only be initialized, never assigned.

There are times when we use the initialization list to performing assignments in the constructor, from the efficiency point of view. When the member initialization list is used, only a single mystring member function is called. To understand why, consider what happens when you declare a bank object.

Construction of objects proceeds in two phases:

1. Initialization of data members in the order of their declaration in the class
2. Execution of the body of the constructor that was called.

For the bank class, this means that a constructor for the mystring object will always be called before you ever get inside the body of the bank constructor. The only question, then, is this: which mystring constructor will be called?

That depends on the member initialization list in the bank class. If you fail to specify an initialization argument for name, then the default mystring constructor will be called. When you later perform an assignment to name inside the bank constructor, you will call operator = on name. That will make two calls to mystring member functions: one for the default constructor and one more for the assignment.

On the other hand, if you use the member initialization list to specify that name should be initialized with initname, then name will be initialized through the copy constructor, at a cost of only a single function call.

There is, however, one time when it makes sense to use assignment instead of initialization for the data members in a class, and that is when you have a large number of data members of built in types.

4.4 Exercise

Theory Questions

1. What problems do we have when we have a pointer data for a class? How can this be solved?
2. When and why do we prefer initialization to assignment?
3. What is done in a class constructor and destructor that have pointer data members?
4. When and why should the default assignment and the copy constructor be overloaded?

Problems

1. Implement and test the following constructor for the mystring class:
   ```
   mystring (const char* s, unsigned n, unsigned k = 0);
   ```
   It uses n characters from the string s, beginning with character s[k].

For example, the declarations

```
mystring x("ABCDEFGHIJKL", 3);
mystring y("ABCDEFGHIJKL", 3, 5);
```

would construct the object x representing the substring “ABC” and the object y representing the substring “FGH”.

2. Implement and test the following modification of the copy constructor for the mystring class:

```
mystring(const mystring& s, unsigned n, unsigned k = 0);
```

this uses n characters from object s, beginning with character s.buf[k].

For example, if x is a mystring object representing “ABCDEFGHIJKL”, then

```
mystring y(x, 3);
mystring z(x, 3, 5);
```

would construct the object y representing the substring “ABC” and the object z representing the substring “FGH”.

3. Implement and test the following member function for the mystring class:

```
int frequency(char c);
```

This returns the number of occurrences of the character c in the mystring.

For example, if x is the string “Mississippi”, then the call

```
frequency('i') would return 4.
```

4. Implement and test the following member function for the mystring class:

```
void remove(unsigned n, unsigned k = 0);
```

this removes n characters from the object, beginning with character buf[k].

For example, if x is the string “ABCDEFGHIJKL”, then the call

```
x.remove(3, 5);
```

would remove the substring "FGH" from the object x, changing it to "ABCDEIJKL".

5. Implement and test the following member function for the mystring class:

```
int palindrome();
```

This returns a value one if the string is a palindrome (a word which reads the same backwards, for example, liril, madam, Malayalam etc), and zero if it is not a palindrome.

Build the palindrome as both a friend and member function.