CS 2 Lecture Topic 5: Pointer Applications

Author: Satish Singhal Ph. D

Version 1.0

Having learned some basic of pointers, we proceed further to learn about higher-level pointer applications. The topics in this chapter are categorized in the following headings:

- **Address of objects and their data members.**
- **Pointer to user defined classes and member selection**
- **this pointer**
- **Memory allocation using pointers**
- **Dynamic arrays of primitives**
- **Memory de-allocation**
- **Dynamic arrays of objects**
- **Pointers as function arguments and return values from functions**
- **Assignment operator, copy constructor, and destructors**
- **Pointers as class members**
- **Rule of three**
- **Pointer to base class and its children classes**

Address of objects and their data members

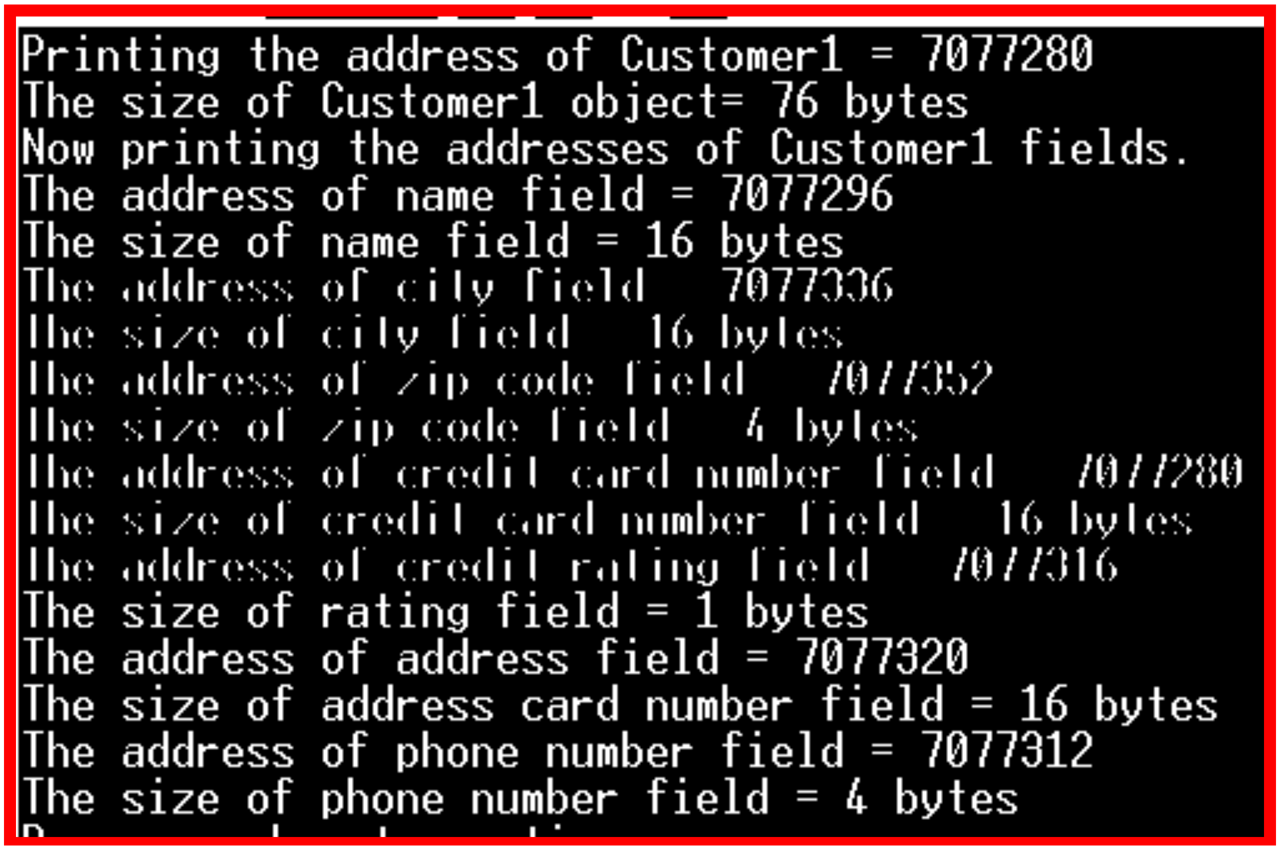
For the sake of simplicity we take the Customer class we discussed in chapter on classes and make all its members as public members. The Customer class thus modified is shown in Listing 5.1A.

```
//*Listing 5.1A
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
const string password = "FinManager101";
class Customer
{
public:
        string credit_card_num;
        string name;
        long phone_num;
        char credit_rating;
        string address;
        string city;
        long zip_code;

        //member functions
        string getCreditCardNumber( ) const
        {
                return credit_card_num;
        }
        //************************************************
        void setCreditCardNumber(string credit_card_num1)
        {
                credit_card_num = credit_card_num1;
        }
        //***********************************************
        string getName() const
        {
                return name;
        }
        //***********************************************
        void setName(string name1)
        {
                name = name1;
        }
        //***********************************************
        long getPhoneNumber() const
        {
                return phone_num;
        }
        //***********************************************
        void setPhoneNumber(long phone_num1)
        {
                phone_num = phone_num1;
        }
        //***********************************************
        string getAddress() const
        {
                return address;
        }
        //***********************************************
        void setAddress(string address1)
        {
                address = address1;
        }
        //***********************************************
        char getCreditRating() const
        {
                return credit_rating;
        }
        //***********************************************
        void setCreditRating(char credit_rating1)
        {
                credit_rating = credit_rating1;
        }
        //**********************************************
        string getCity() const
        {
                return city;
        }
        //**********************************************
        void setCity(string city1)
        {
                city = city1;
        }
        //**********************************************
        long getZipCode() const
        {
                return zip_code;
        }
        //**********************************************
        void setZipCode(long zip_code1)
        {
                zip_code = zip_code1;
        }
        //**********************************************
        void printCustomer(ostream& out) const
        {
                string input = "";
               out<<"The customer name = "<<name<<endl;
    out<<"The customer phone number is: "<<phone_num<<endl;
out<<"The customer credit rating is: "<<credit_rating<<endl;
out<<"The customer address is : "<<address<<endl;
             out<<"The customer city is : "<< city<<endl;
            out<<"The customer zip code is: "<<zip_code<<endl;
cout<<"To print cutomer's credit card number, please enter your”
<<” password : ";
                cin>>input;
                if(input == password)
                out<<"The customer credit card number is: "
                        <<credit_card_num<<endl;
                else
                        cout<<"The entered password did not
match the autorized password.\n"
                        <<"Please seek an authorization
from the billing department.\n";
        }
        //**********************************************
        bool isEqual(Customer Customer1) const
        {
                if(name == Customer1.name && address == Customer1.address &&
                    phone_num == Customer1.phone_num && city == Customer1.city &&
                    credit_card_num == Customer1.credit_card_num &&
                    credit_rating == Customer1.credit_rating &&
                    zip_code == Customer1.zip_code)
                    return true;
                else
                    return false;
        }
//**************************************************************
        //Constructors
        Customer()
        {
                name = "";
                phone_num = 0;
                credit_rating = ' ';
                address = "";
                city = "";
                zip_code = 0;
                credit_card_num = "";
        }
        Customer(string name1, long phone_num1, char credit_rating1,string address1,
                string city1, long zip_code1, string credit_card_num1)
        {
                name = name1;
                phone_num = phone_num1;
                credit_rating = credit_rating1;
                address = address1;
                city = city1;
                zip_code = zip_code1;
                credit_card_num = credit_card_num1;
        }
 };
//Listing 5.1A
```

Now it is certain that once an object of customer class is created, such object and all of its members have addresses in the memory. We illustrate the existence of such memory addresses by running the code in main function (Listing 5.1B).

```
#include "Customer.cpp"

int main()
{
                Customer Customer1("John Doe", 6604759,'A',"16007 Crenshaw Blvd.",
                "Torrance",90506,"47219984423");
        cout<<"Printing the address of Customer1 = "<<(long)&Customer1<<endl;
        cout<<"The size of Customer1 object= "<<sizeof(Customer1)<<" bytes"<<endl;
        cout<<"Now printing the addresses of Customer1 fields.\n";
        cout<<"The address of name field = "<<(long)&Customer1.name<<endl;
        cout<<"The size of name field = "<<sizeof(Customer1.name)<<" bytes"<<endl;
        cout<<"The address of city field = "<<(long)&Customer1.city<<endl;
        cout<<"The size of city field = "<<sizeof(Customer1.city)<<" bytes"<<endl;
        cout<<"The address of zip code field = "<<(long)&Customer1.zip_code<<endl;
        cout<<"The size of zip code field = "<<sizeof(Customer1.zip_code)<<" bytes"<<endl;
        cout<<"The address of credit card number field =
"<<(long)&Customer1.credit_card_num<<endl;
        cout<<"The size of credit card number field =
"<<sizeof(Customer1.credit_card_num)<<" bytes"<<endl;
        cout<<"The address of credit rating field =
"<<(long)&Customer1.credit_rating<<endl;
cout<<"The size of rating field = "<<sizeof(Customer1.credit_rating)<<" bytes"<<endl;
        cout<<"The address of address field = "<<(long)&Customer1.address<<endl;
cout<<"The size of address card number field = "<<sizeof(Customer1.address)
<<" bytes"<<endl;
        cout<<"The address of phone number field =
"<<(long)&Customer1.phone_num<<endl;
        cout<<"The size of phone number field = "<<sizeof(Customer1.phone_num)
<<" bytes"<<endl;
        return 0;
}//Listing 5.1B
```

The Figure 5.1 shows the output from Listing 5.1B.



```
Printing the address of Customer1 = 7077280
The size of Customer1 object= 76 bytes
Now printing the addresses of Customer1 fields.
The address of name field = 7077296
The size of name field = 16 bytes
The address of city field = 7077336
The size of city field = 16 bytes
The address of zip code field = 7077352
The size of zip code field = 4 bytes
The address of credit card number field = 7077280
The size of credit card number field = 16 bytes
The address of credit rating field = 7077316
The size of rating field = 1 bytes
The address of address field = 7077320
The size of address card number field = 16 bytes
The address of phone number field = 7077312
The size of phone number field = 4 bytes
```

FIG. 5.1

First thing we notice from this output is that the object Customer1 and all its data members are assigned memory addresses when the program is running. We summarize the data for the memory addresses for the object and its members and their sizes in bytes in the table 5.1 below.

Name (object or member)	Data type	Address in memory	Size in bytes
Customer1	User defined	7077280	76
credit_card_num	string	7077280	16
name	string	7077296	16
phone_num	long	7077312	4
credit_rating	char	7077316	1
address	string	7077320	16
city	string	7077336	16
zip_code	long	7077352	4
Total bytes for data members			65

Number of bytes for member function addresses	**= Size of object – sum of sizes of data members =76-65 = 11**

Table 5.2

The Figure 5.2 aptly shows the picture of object in the memory. Notice that the width of horizontal bars is not in proportion to the size of data members. Therefore vertical axis is just a schematic presentation of picture of data members inside the object Customer1. The horizontal length of bars shows the memory size of various data Customer1 object members.

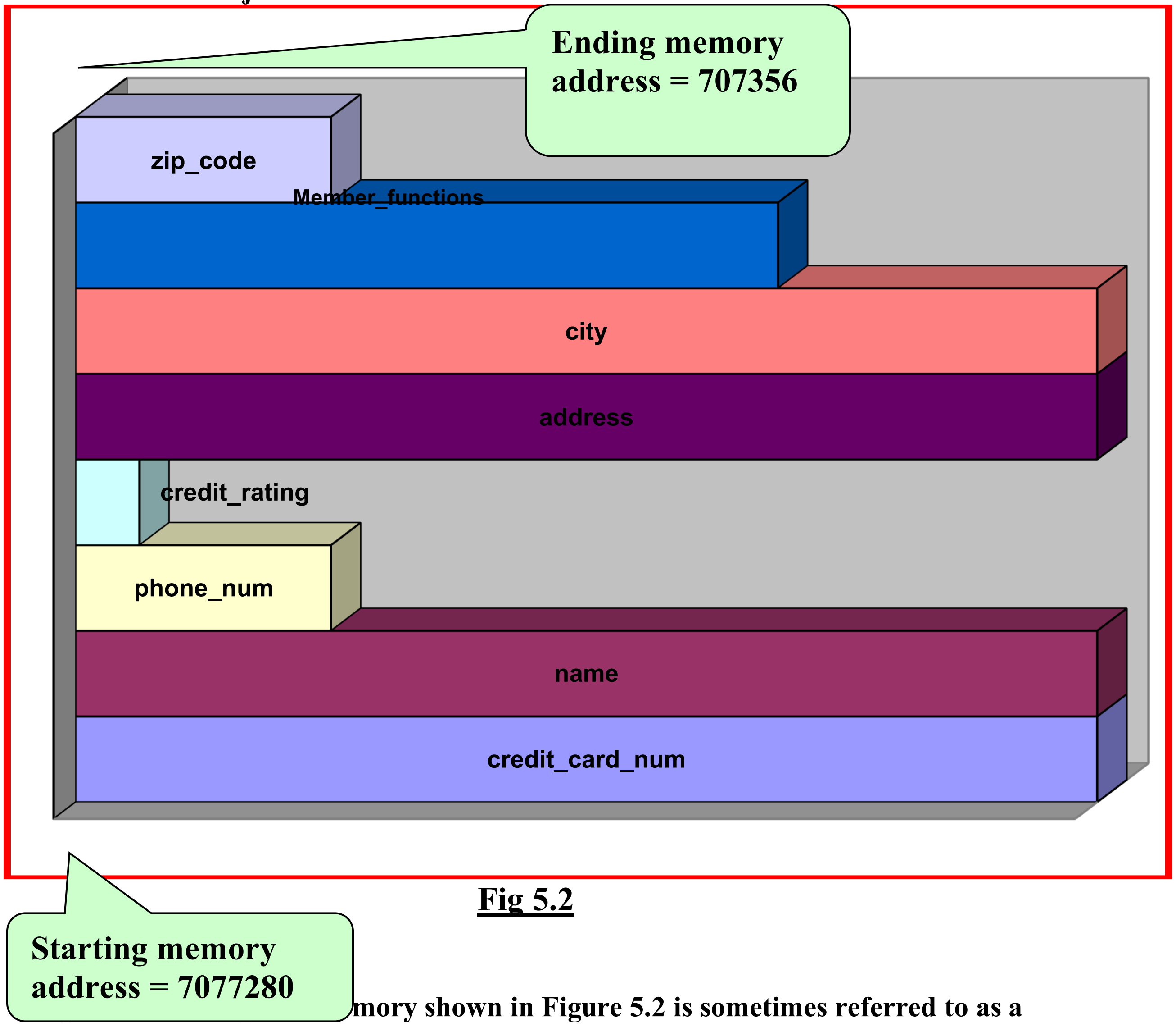


Fig 5.2

mory shown in Figure 5.2 is sometimes referred to as a stack. The program variables and function addresses are created on the stack in the sequence they are born in the program.

Pointers to Class Objects

One can assign pointers to store the addresses of user-defined objects and to their publicly accessible members. The syntax is identical to defining the pointers to primitive data types. For example the pointer to class Customer can be defined and assigned to a Customer type pointee as follows:

```
Customer Customer1 ("John Doe", 6604759,'A',"16007
Crenshaw Blvd.",
          "Torrance", 90506,"47219984423");
Customer * Cptr = &Customer1;
```

Pointer Cptr will now store the address of object Customer1. Important point about using pointers for object lies in using them for public access member selection. As you would recall that de-referenced value of a pointer gave us essentially the pointee it is pointing to. Same holds here. After de-referencing the object pointer we can use it for the member selection using the dot operator. For example using the Customer declaration above we have the access to name and city public members as follows:

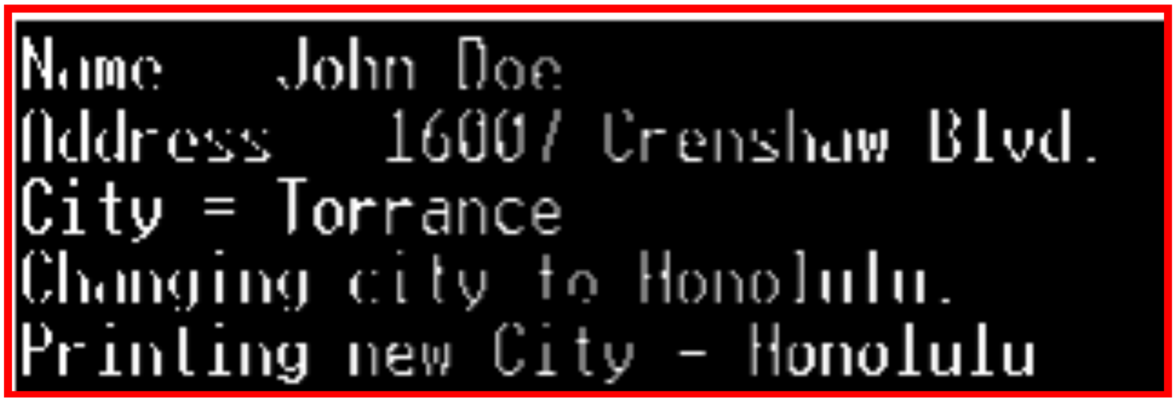
```
(*Cptr).name  ← Accesses the name field
(*Cptr).city ← Accesses the city field
(*Cptr).getCity( ) ← invokes getCity member function
(*Cptr).setCity( " Honolulu") ← invokes setCity member
function
```

Understand that syntax like below is a compile error.

```
*Cptr.name //Compile error
```

The reason is because the dot operator has a higher precedence compared to de-reference operator (*). Therefore the above syntax treats Cptr like a class or struct object and tries to de-reference it assuming it to be a pointer type. Therefore use of dot operator for object member access always requires parenthesis to force the de-referencing first followed by member selection. The function printByPointer( ) shown in Listing 5.1C below shows the application of member selection using the pointer to an instance of class Customer. Figure 5.3 shows the result when the function printByPointer is called in the main function.

```
void printByPointer( )
{
		Customer Customer1("John Doe", 6604759,'A',"16007 Crenshaw Blvd.",
		"Torrance",90506,"47219984423");
		Customer * Cptr = &Customer1;
		cout<<"Name = "<<(*Cptr).name<<endl;
		cout<<"Address = "<<(*Cptr).address<<endl;
		cout<<"City = "<<(*Cptr).city<<endl;
		cout<<"Changing city to Honolulu.\n";
		(*Cptr).setCity("Honolulu");
		cout<<"Printing new City = "<<(*Cptr).city<<endl;
	}//Listing 5.1C
```



```
Name = John Doe
Address = 16007 Crenshaw Blvd.
City = Torrance
Changing city to Honolulu.
Printing new City = Honolulu
```

Fig. 5.3

Simplified member selection using a pointer and arrow operator

The use of parentheses to force de-referencing first and the using dot operator for member selection becomes harder to type. Therefore, C++ allows another mechanism of member selection using the arrow operator as follows:

Customer Customer1 ("John Doe", 6604759,'A',"16007 Crenshaw Blvd.",
"Torrance", 90506,"47219984423");
Customer * Cptr = &Customer1;
Cptr->name; ← **Accesses the name field**
Cptr->city; ← **Accesses the city field**
Cptr->getCity( ); ← **invokes getCity member function**
Cptr->setCity(" Honolulu") ; ← **invokes setCity member function**

Notice that the use of arrow operator performs the task of de-referencing and member selection in one step. Therefore the indirection operator (*) is not needed. The following will be a compile error unless the name field is also a pointer.

```
*Cptr->name;// Compile error unless name is also a pointer
```

The following syntax is legal, but unnecessary.

```
(Cptr)->name;// Legal but parentheses are unnecessary
```

```
void printByPointer2( )
{
                Customer Customer1("John Doe", 6604759,'A',"16007
Crenshaw Blvd.",
                "Torrance",90506,"47219984423");
                Customer * Cptr = &Customer1;
                cout<<"Name = "<<Cptr->name<<endl;
                cout<<"Address = "<<Cptr->address<<endl;
                cout<<"City = "<<Cptr->city<<endl;
                cout<<"Changing city to Honolulu.\n";
                Cptr->setCity("Honolulu");
                cout<<"Printing new City = "<<Cptr->city<<endl;
        }//Listing 5.1D
```

Listing 5.1D shows the function printByPointer2 ( ) where the arrow operator is used for member selection. The results are identical to Figure 5.3.

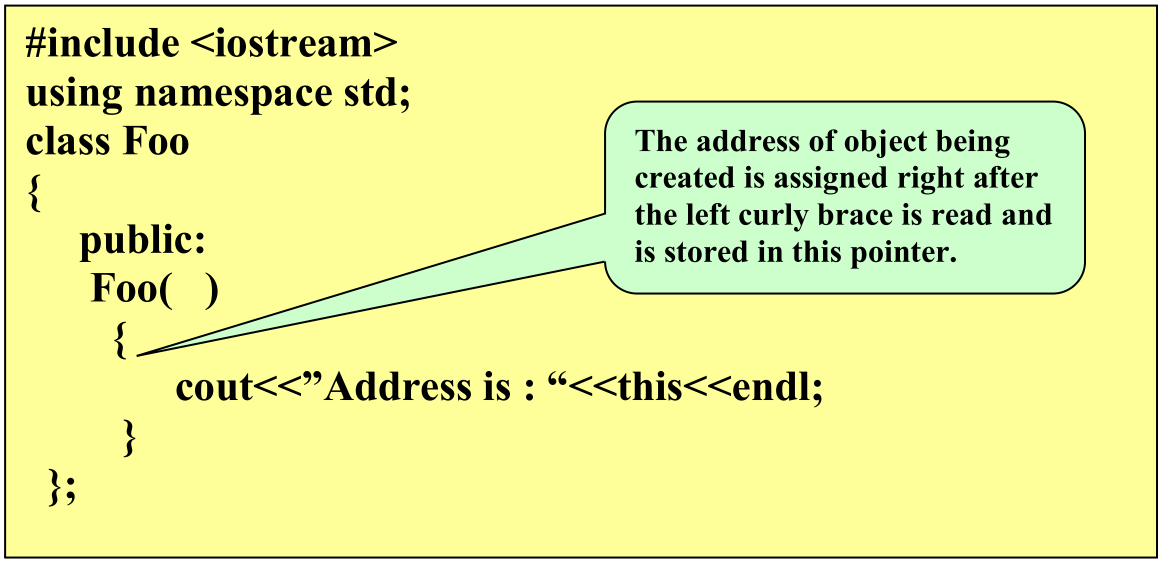
Assigning a reference to the object pointer and using the dot operator

Following technique of member selection through a pointer may look trivial to some, but it may be useful for people who do not like to use the arrow operator as a matter of personal preference. Here we assign a reference to alias the de-referenced value of the object pointer and then use the dot operator with the reference for member selection. This technique may look a bit silly in present context, but could be useful later when one allocates memory dynamically and deals with "no-name" objects. The example is shown below.

```
Customer Customer1 ("John Doe", 6604759,'A',"16007
Crenshaw Blvd.",
                "Torrance", 90506,"47219984423");
Customer * Cptr = &Customer1;
Customer & Cref = *Cptr;
Cref .name;              🡐 Accesses the name field
Cref .city;              🡐 Accesses the city field
Cref .getCity( );         🡐 invokes getCity member function
Cref.setCity( " Honolulu") ;🡐 invokes setCity member function
```

this Pointer for classes and structures

The classes and structures in C++ carry a pointer called this, which is a reserved word in C++. this is a pointer to the object itself and it can be used for member selection, inside the non-static member functions, constructors and destructors for a class, struct, and union. Corollary to this is that this pointer is also not available inside the friend functions. One can also think of this pointer as a read and write enabled but non-assignable pointer. Additionally this pointer can be thought of as a hidden private data member, which is not available to friend functions. this pointer is not a class data member in conventional sense because the sizeof operator will not include four bytes allotted to this pointer in the overall size of the object. this pointer to an object is created as soon as constructor call initiates. Figure 5.4 shows the time of birth of this pointer when constructor call begins for a hypothetical class Foo.



```
#include <iostream>
using namespace std;
class Foo
{
    public:
     Foo(   )
       {
            cout<<”Address is : “<<this<<endl;
        }
  };
```

The address of object being created is assigned right after the left curly brace is read and is stored in this pointer.

FIG. 5.4

The output line (Fig. 5.4) added to any constructor will be print out address of the object being created, each time a constructor call is made, as this pointer stores the address of the object being created[1].

One trivial use of this pointer is to use it for member selection inside the non-static class member functions. For example we may rewrite the printCustomer function for Customer class as given in Listing 5.1E, where member selection is done by invoking the this pointer.

[1] Some textbooks say that this pointer is a hidden argument passed to all non-static methods and constructors. We disagree with that characterization. For example in Figure 5.4, how can constructor for Foo have an address when constructor call has not even begun? At the very best one can say that when constructor call initiates after the left curly brace then an address is assigned to the object being created, and value stored in it can be printed.

```
void printCustomer(ostream& out) const
        {
                string input = "";
                out<<"The customer name = "<<this->name<<endl;
        out<<"The customer phone number is: "<< this->phone_num<<endl;
        out<<"The customer credit rating is: "<< this->credit_rating<<endl;
                out<<"The customer address is : "<< this->address<<endl;
                out<<"The customer city is : "<< this->city<<endl;
                out<<"The customer zip code is: "<< this->zip_code<<endl;
cout<<"To print customer's credit card number, please enter your password : ";
                cin>>input;
                if(input == password)
                        out<<"The customer credit card number is: "
                        << this->credit_card_num<<endl;
                else
        cout<<"The entered password did not match the authorized password.\n"
                <<"Please seek an authorization from the billing department.\n";
        }//Listing 5.1E
```

Such use of this pointer however is not very elegant. Shortly we will discuss that there are situations when we have to return the object itself as a return value from the function. This is done when we overload the equal operator for a class. The de-referenced value of this pointer is then used as the return value for the function. In the meanwhile we show another important use of this pointer by re-writing the postfix operator code for class Complex for complex number, that we discussed in the chapter on operator overloading.

The algorithm for overloading the postfix operator is as follows:

1. Save the current state of the object.
2. Increment or decrement the data member that the postfix operator must change.
3. Return the saved object.

For the postfix increment for the Complex class we wrote the code for overloaded operator as a member function as follows: (Listing 5.2A)

```
const Complex Complex::operator ++(int marker) //post-increment
{
        double temp_real = Real;
        double temp_imag = Imag;
        Real++;
        return Complex(temp_real, temp_imag);
}//Listing 5.2A
```

In above code we had to save the current state of the object in temporary variables temp_real and temp_imag. This procedure may become cumbersome if the class has large number of variables. Since the this pointer already contains the current address of the object, its de-referenced value can give us the current object at any time. Therefore the code in Listing 5.2A is vastly modified using a this pointer and is given in Listing 5.2B.

```
const Complex Complex::operator ++(int marker) //post-increment
{
        Complex Temp = *this;
        Real++;
        return Temp;
}//Listing 5.2B
```

De-referenced value of this pointer gives the current object before post increment is done.

Another elegant use of this pointer is made in writing the explicit constructor. In large software projects every attempt is made to minimize number of new "names" to be used for variables, function names etc. This is done to maintain simplicity and avoid name conflicts. A rule is then followed that in explicit constructor the arguments will have same name as class data members. However, the this pointer is used to access the class data members, thus differentiating them from the arguments to the constructor. In Listing 5.2C, we re-write the constructor for Customer class using the this pointer.

```
Customer (string name1, long phone_num, char credit_rating, string
address, string city, long zip_code, string credit_card_num)
        {
                this->name = name;
                this->phone_num = phone_num;
                this->credit_rating = credit_rating;
                this->address = address;
                this->city = city;
                this->zip_code = zip_code;
                this->credit_card_num = credit_card_num;
        }//Listing 5.2C
```

Name field of the object.

name string passed to the constructor.

In Listing 5.2C, each field of the object being constructed is qualified by using the this pointer. Therefore compiler can correctly differentiate between the constructor arguments and data member names as latter are bound to the this pointer. Notice that in this scheme it will be a serious error to forget to bind the class data members with this pointer. For example a code line like below, which skips the this pointer will compile.

```
name = name;
```

What that means that the memory location holding the constructor argument called name is being set equal to itself and the value of the name field in Customer object being constructed is set to some unknown value! We will show more use of this pointer shortly.

Dynamic Memory Allocation

Compared to Fortran and Cobol one excellence of C programming language that made it dear to programmer very quickly was its facility to create variables dynamically at run time and delete them when they are no longer needed. This allowed for highly memory efficient programs. Many small device programs, even today cannot be written in languages other than C or C++. C++ has similar facilities for dynamic memory allocation. As we saw in Figure 5.2, when objects are create statically, they are given memory addresses in a part of program memory called stack. In fact in chapter on Fixed Stacks we gave a demonstration of how the function call stacks are built up during program execution. There is another part of program memory called "heap"[2] which is used to create the program variables at the run time. Primitives as well as the objects both can be created through the mechanism of dynamic memory allocation on heap. C++ provides an operator called new, which is used to assign the memory to create variables at run-time. The syntax is as follows.

```
int * iptr = new int( 0);//Initializes the pointee value to zero
int * iptr = new (int)( 0);//alternate form
```

Figure 5.5 shows the integer variable thus created on heap and the value stored in it. In terms of pointer and pointee relationship, the iptr stores the address of pointee created on the heap.

[2] There is a data structure also called 'heap" which has nothing to do with the part of program memory that is called heap. We will cover the heap data structure in later chapters.

Dynamic creation of variable

int * iptr = new int(0);

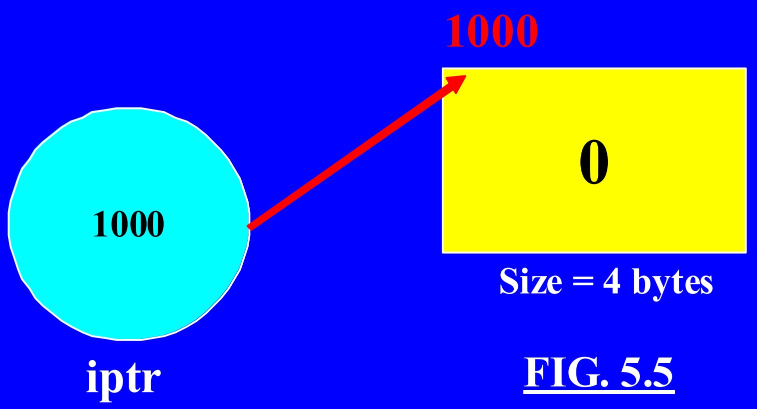


FIG. 5.5

Notice that the variable created in Figure 5.5 has no name! The variables created on heap have no names, and they are only accessible through their pointers! It is also possible to assign some value to the variable being created on heap by passing the value as an argument to the constructor call being made after the word new. Example of this is shown below.

```
double * dptr = new double (5.5);
```

The above dynamic assignment will create a double type variable whose address will be stored in pointer dptr, and the value stored in the variable will be 5.5. Understand that C++ will allow you to skip parenthesis after the data type following new. But my recommendation is to never do this, at least for primitive data types, as in that case the memory location created will have some garbage value in it. Therefore following is legal but not recommended!

```
float   * fptr  = new float;//Legal, but not recommended
float *fptr = new float (0.0f );//Always use this form for
//primitives
```

The second form will always put the value in parenthesis in the memory location created at runtime. You may have surmised that the use of data-type and parenthesis after the word **new** appears to be like a constructor call. That is what it is. C++ allows one to make constructor calls for primitive and user defined data, in an identical manner. The program variables created by use of new come into existence at runtime and unlike like the variables you may have been used so far, these variables are nameless. Their only connection to the program is through the pointer, which hold their memory address. One can create objects also in the similar

manner. For example to create an object of type Customer one would invoke the operator new in the following manner.

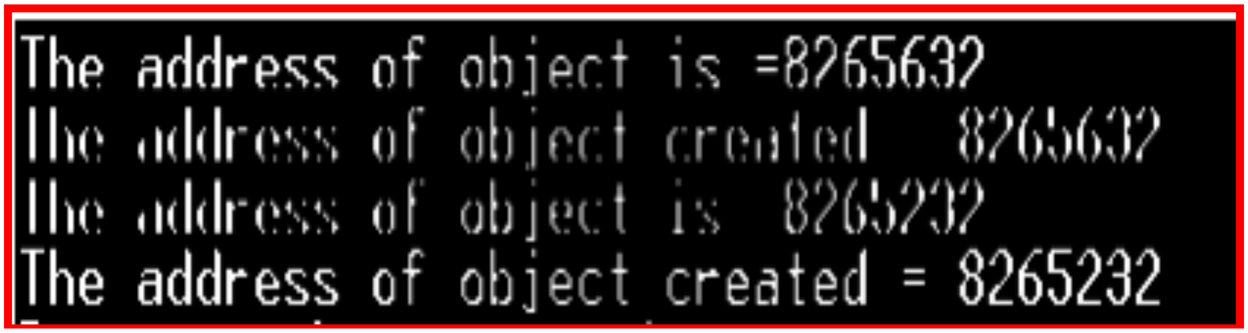
```
Customer * Cptr =
new Customer ("John Doe", 6604759,'A',"16007 Crenshaw Blvd.",
        "Torrance", 90506,"47219984423");//Explicit constructor invoked
```

In creating the Customer object being pointed by the pointer Cptr in this case we used the explicit constructor. A Customer object created by calling the default constructor will have the following syntax.

```
Customer * Cptr1 = new Customer( );//Default constructor called
```

In Listing 5.3 we create a Customer object dynamically by calling the explicit constructor. We, however modify the constructor to print the address of the object being created by using the this pointer. We also print the address of object using the pointer assigned to it during the call to operator new. The function createDynamically ( ) called in main and the results from it are shown in Listing 5.3 below.

```
void createDynamically()
{
        Customer * Cptr1 = new Customer( );
        cout<<"The address of object created = "<<(long)Cptr1<<endl;
Customer * Cptr2 = new Customer ("John Doe", 6604759,'A',
"16007 Crenshaw Blvd.", "Torrance", 90506,"47219984423");
        cout<<"The address of object created = "<<(long)Cptr2<<endl;
}
```



```
The address of object is =8265632
The address of object created   8265632
The address of object is  8265232
The address of object created = 8265232
```

Listing 5.3

In above results, the first address is printed by using the this pointer inside the constructor call, and second one is printed using the pointer pointing to the no-name customer object. In Listing 5.3, the pointers Cptr1 and Cptr2 can perform the member selection using the arrow or dot operator explained earlier in this chapter.

What does new do if computer is out of memory?

When you compile a C++ program using newer compiler, in those executable files if new asks for the memory and operating system or device is out of memory, then new will simply shut down the program. In executables compiled with older compilers, however, that may not happen. If compiling using older compilers, a check may be needed if new returned a pointer with a memory or not. If new cannot get the memory, then it will return a null pointer. For example, the following code type is safe for all compilers:

```
int * iptr = new int(0);
if (!iptr)
{
        cerr<<"Out of memory error.\n";
        exit (1); // will require inclusion of header file cstdlib or similar
}
```

Dynamic arrays of Primitives

Until now we were stuck with the arrays of fixed length. For example in declaration of an array
int arr [MAX] = {0};
we had to ascertain that MAX evaluates to a constant expression at "compile time". Therefore the MAX could only be:

- **A named constant**
- **A literal constant**
- **Or an expression containing above.**

This could be very wasteful of memory in programming, because we would need to estimate the size of array needed before we run the program. If we assign too large an array, we end up wasting lot of unused array length, and memory assigned to it. If we choose too small an array, we run the risk of data overflow or not are able to do all the computation we may need to do. Dynamic memory allocation allows us to assign the array length at run time. We show an example of this in Listing 5.4.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
void main()
{
	int max = int();
	cout<<"Please enter the length of array you wish to create : ";
	cin>>max;
	if(max<=0)
	{
		cout<<"Array of length zero or minus is not created.\n";
		exit(1);
	}
	int * iptr_arr = new int [max];
	if(!iptr_arr)
	{
		cerr<<"Failed to allocate memory.\n";
		exit(1);
	}
	cout<<"Filling your array with random numbers.\n";
	srand( (unsigned)time( NULL ) );
	for(int index=0; index<max; index++)
		iptr_arr[index] = rand()%100;

	cout<<"Printing your array with random numbers.\n";
	for( index=0; index<max; index++)
	{
		cout<<iptr_arr[index]<<"  ";

		if(index%10 == 0)
			cout<<"\n";
	}
	cout<<"\n";
//
}//Listing 5.4: Note no de-allocation of memory done in this example. We
will show de-allocation shortly
```

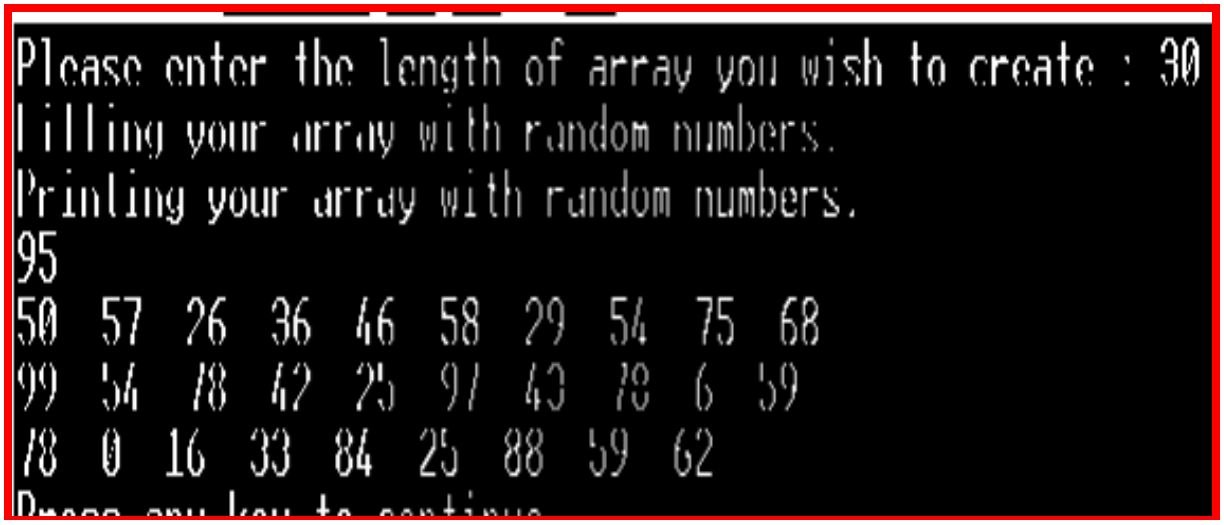
New creates an array of length entered by user. This array is not initialized.

Array filled with random numbers 0 to 99

Array is printed.

Memory allocated by new must be de-allocated. We show de-allocation shortly.

The results of Listing 5.4 are shown below in Figure 5.6.



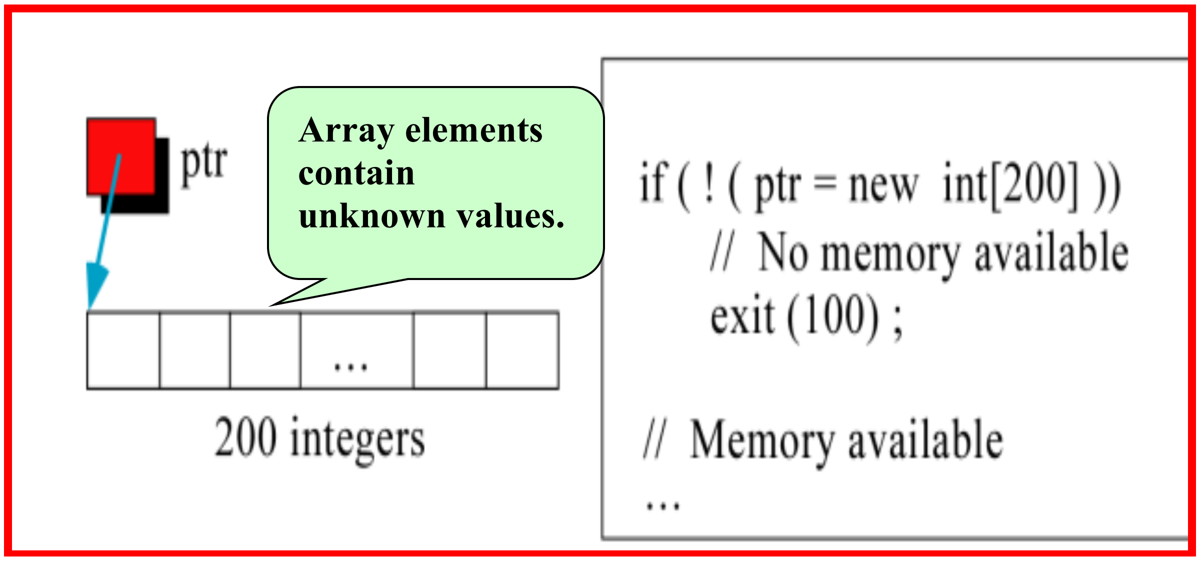
```
Please enter the length of array you wish to create : 30
Filling your array with random numbers.
Printing your array with random numbers.
95
50  57  26  36  46  58  29  54  75  68
99  54  78  42  25  97  43  78  6  59
78  0  16  33  84  25  88  59  62
```

Fig. 5.6

Referring to Listing 5.4, the syntax for creating an array of length n (n>0) is as follows:

int * iptr_arr = new int [n];
int * iptr_arr = new int [n]( );//Alternate form

The above statement returns a pointer to the first element of an integer array of physical length n. Note that the n elements of array are not initialized. Thus it is programmer's responsibility that array elements are initialized before use! Figure 5.7 shows the picture in the program memory when the statement similar to above is executed.



```
if ( ! ( ptr = new  int[200] ))
    //  No memory available
    exit (100) ;

//  Memory available
...
```

FIG. 5.7

Note that the statements similar to below will compile, but your program will crash at runtime.

```
int * iptr1 = new int [0];
int * iptr1 = new int [-2];
```

Will compile, but program will crash at runtime!

De-allocation of memory created by new

The variables created on heap are not deleted automatically the way variables created on stack are. For example if we declare two variables in a block as follows:

```
{
        int x = 5;
        int * y = new int (10);
}
//x and y are automatically deleted here
//Pointee of y is not deleted, it is still alive on heap
```

By the time we exit the block in which x and y are declared, the x is deleted automatically because the stack built in block containing x is popped. But y is on heap, and the program does not pop heap. The condition where a variable is created dynamically and is not deleted explicitly leads to a situation called "memory leak". The program memory gets tied up in variables, which are no longer being used. In some machines and devices, the operating system may delete these heap variables and free the memory, but as a programmer one should never count on it. Therefore. C++ provides a keyword called delete, which deletes the dynamically allocated memory and returns the memory to operating system. The syntax for use of delete for a single variable is shown below, where we re-write the above block of code.

```
{
        int x = 5;
        int * y = new int (10);
        delete y;
}
//x and y are automatically deleted here
```

Deletes the memory location pointed to by pointer y. Does not delete pointer y, which is still in scope.

Deletion of memory allocated for arrays follow the same pattern, except the subscript operator follows the delete operator. Example of freeing the memory allocated for an array is shown below.

```
{
        int * iptr_arr = new int [ 10];
//……………………
//……………………
        delete [ ] iptr_arr;//deletes the memory allocated
//for the array.
}
//iptr_arr is automatically deleted here
```

Figure 5.8 shows the situation in memory on heap after the delete has been called for an array and individual variable.

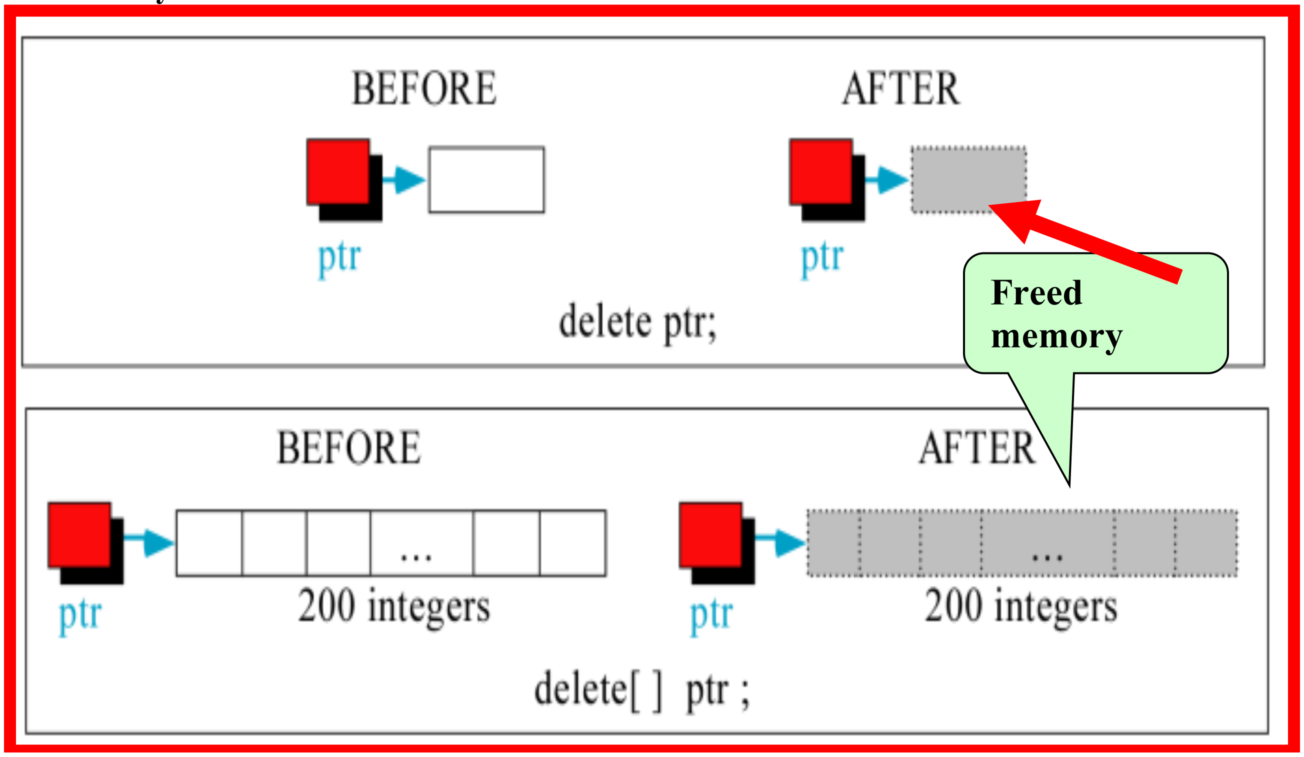
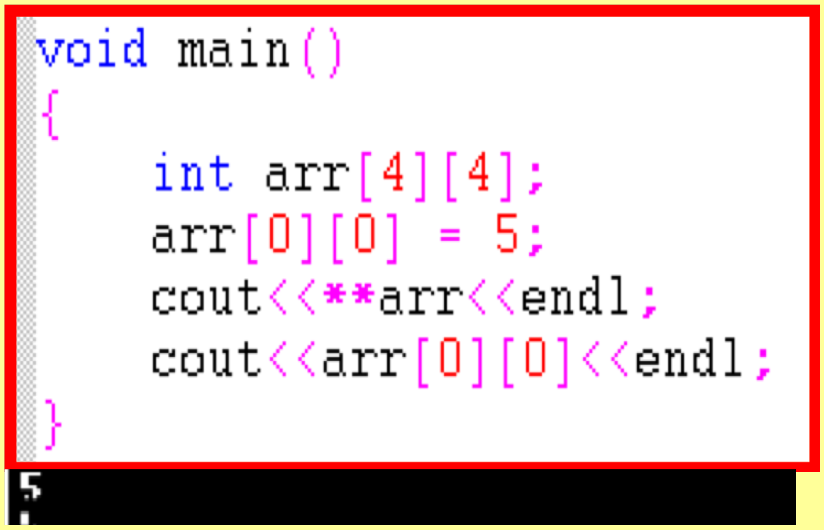


FIG. 5.8

For class objects, the delete calls the class destructor to delete any dynamically created class members. **delete must only be called once for each new call. Extra delete calls will crash program by debug assertion failure at runtime.**

Array of pointers or two dimensional arrays

An array of pointers is very useful for dynamically allocating the memory for a two dimensional array. First we must become clear that while the name of a one-dimensional array is the constant pointer to its first element, the name of a two dimensional array is a constant double pointer. Short Listing 5.5A shows the code and output where we print the first element of a two-dimensional array by double de-referencing its name and using the double subscript operator.

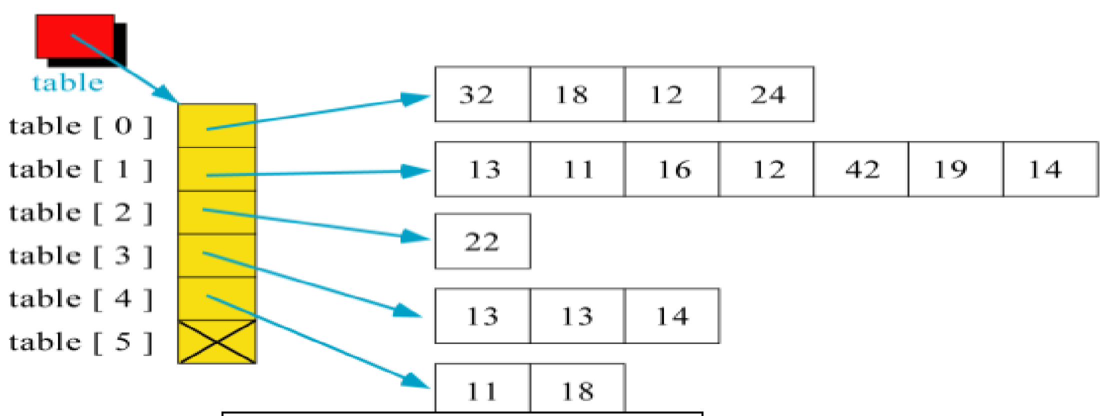


```
void main()
{
     int arr[4][4];
     arr[0][0] = 5;
     cout<<**arr<<endl;
     cout<<arr[0][0]<<endl;
}
```

```
5
5
```

Listing 5.5A

row and first column. Both print a value of 5. This proves that the name of a two dimensional array is a constant double pointer. A double pointer can be used to create the two dimensional array where each row does not need to have same number of columns. For example examine Figure 5.9.



int ** table ; int rowNum=4;

```
table = new int* [rowNum + 1] ;
...
table[0] = new  int[4] ;
table[1] = new  int[7] ;
table[2] = new  int[1] ;
table[3] = new  int[3] ;
table[4] = new  int[2] ;
table[5] = NULL ;
```

FIG. 5.9

In first line we declare a double pointer of integer type called table. The double pointer table will point to an array of pointers, and each of them will in turn point

to the first element of the one dimensional array. The line two of the code allocates the memory for the array of pointers of length 5. Then we dynamically allocate the one-dimensional array for each row, and have different number of columns for each row. After the values are placed in each row and column (corresponding code not shown in the above Figure) we create what we call a "jagged array" where each row may have unequal number of columns. Such jagged arrays are memory efficient, as just enough memory is allocated for each row to hold the requisite number of elements. Therefore understand the following facts about two dimensional arrays:

- **In a two dimensional array, the name of the array is a double pointer, which is pointer to the array of pointers for each row. Name of the array stores the address of the pointer to first row.**
- **The name of each row like row [0], row [1] etc. are pointers to the first element in the each row array.**

Therefore in dynamic creation of two-dimensional array, first we allocate the memory for an array of pointers and then we allocate memory for each row of the array. Listing 5.5B shows the creation of a dynamic two-dimensional rectangular array and its memory management.

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

void main()
{
      int max1 = int();
cout<<"Please enter the row length of array you wish to create : ";
      cin>>max1;

      if(max1<=0)
      {
            cout<<"Array of length zero or minus is not created.\n";
            exit(1);
      }

      int max2 = int();

cout<<"Please enter the column length of array you wish to create : ";
      cin>>max2;

      if(max2<=0)
      {
            cout<<"Array of length zero or minus is not created.\n";
            exit(1);
      }

      int ** grid = new int * [max1];

      if(!grid)
      {
```

1. Get new to allocate memory for an array of pointers of length max1.

```
                cerr<<"Failed to allocate memory.\n";
                exit(1);
        }

        for(int row =0; row<max1; row++)
        {
        grid [row] = new int [max2];

                if(!grid[row])
                {
                        cerr<<"Failed to allocate memory.\n";
                        exit(1);
                }
        }
        cout<<"Filling your 2 dimensional array with random numbers.\n";

        srand( (unsigned)time( NULL ) );

        for(row = 0; row<max1; row++)
        {
                for(int col=0; col<max2; col++)
                        grid[row][col] = rand()%100;
        }

        cout<<"Printing your 2 dimensional array with random numbers.\n";
        for(row = 0; row<max1; row++)
        {
                for( int col=0; col<max2; col++)
                {
                        cout<< grid[row][col]<<"\t" ;
                }
                cout<<endl;
        }

        cout<<"\n";
        //deletion of dynamically allocated memory
        //First delete each row
        for(row =0; row<max1; row++)
        delete [] grid [row];
        //now delete the double pointer
        delete [] grid;
}
//Listing 5.5B
```

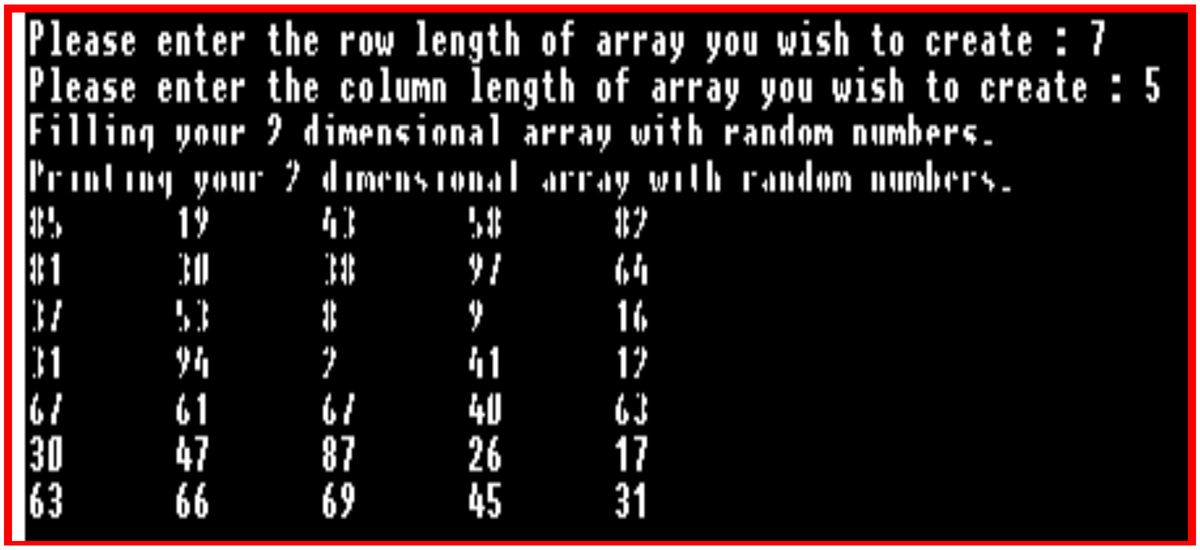
2. Get new to allocate memory for the each row with number of elements being equal to the max2 (columns).

3. Delete the memory assigned to each row.

4. Delete the memory assigned to double pointer for the array of pointers.

//

The output of Listing 5.5B is shown in Figure 5.10 below.



```
Please enter the row length of array you wish to create : 7
Please enter the column length of array you wish to create : 5
Filling your 2 dimensional array with random numbers.
Printing your 2 dimensional array with random numbers.
85      19      43      58      82
81      30      38      97      64
37      53      8       9       16
31      94      2       41      12
67      61      67      40      63
30      47      87      26      17
63      66      69      45      31
```

FIG. 5.10

In this Listing, first we prompt user to enter the legal values of rows and columns for the two dimensional array to be created. Assume that the number of rows is max1 and number of columns is max2. Since the address of each row of a two-dimensional array is to be stored in an array of pointers, first we create this array of pointers. The statement to which bubble #1 is pointing does this.

```
int ** grid = new int * [max1];
```

The above statement creates a one-dimensional array of integer pointers. Now we must run a for loop to assign memory for each row of the two dimensional array. The statement to which bubble #2 is pointing does this.

```
for (int row =0; row<max1; row++)
        grid [row] = new int [max2];
```

The above construct creates max1 number of rows, each having max2 number of columns. The name of double pointer grid can now be used like regular 2-dimensional array name. For example grid [0][0] will be the first element of first row. grid [0][1] will be the second element of the first row and so on. The array can be filled and displayed by using the double for loop construct.

The deletion of dynamically allocated memory must be in exact reverse order of the memory allocation. First we de-allocate the memory allocated for each row of the array. The for loop construct pointed to by bubble #3 does this.

```
for (row =0; row<max1; row++)
    delete [] grid [row];
```

Second square brackets are needed to specify the location of the pointer to each row.

Notice that the second square is needed, because it is needed to specify the correct location of the pointer to each row.

Finally the memory allocated for the array of pointers is deleted. The statement pointed by bubble #4 does this.

delete [ ] grid;

Understand that the objects are destroyed or de-allocated in the sequence, which is exact reverse of their creation. In creating a 2-dimensioanl array, the array of pointers to hold the address of each row is created first and then the memory is allocated for each row. In de-allocating this array, first the memory allocated for each row is de-allocated, and then finally the memory for the array of pointers is de-allocated.

Pointers as the arguments to functions

Like other data types, the pointers can also be passed to functions as function arguments. One dominant example of such applications is a swap function, which takes two pointers as an argument and swaps the contents of their pointees. The swap function, which takes two, void pointers as argument is written in Listing 5.6 and has the following characteristics:

Function swap will work for any data type as it takes two void pointers as arguments. A flag is passed to the function y value to indicate the data type of the pointer. For example, we set the following flag values

*1 = int
*2 = float
*3 = char
*4 = string
*5 = double

Swap function takes two void pointers and a flag. Value of flag controls the identity of void pointer.

```
////////////////////////////////////////////////////////////////////////////////
#include <iostream>
#include <string>
using namespace std;
void swap(void* ptr1, void* ptr2, int flag);



int main()
{
	int val1 = 5;
	int val2 = 10;
	int flag = 1;
	cout<<"Swapping two integers .\n";
	cout<<"Before swap val1 = "<<val1<<endl;
	cout<<"Before swap val2 = "<<val2<<endl;
	swap(&val1,&val2, flag);
	cout<<"After swap val1 = "<<val1<<endl;
	cout<<"After swap val2 = "<<val2<<endl;

	cout<<"Swapping two strings .\n";
	string name1 = "John";
	string name2 = "Mary";
```

Swap function is called by passing to it the address of two integers.

```
        cout<<"Before swap name1 = "<<name1<<endl;
        cout<<"Before swap name2 = "<<name2<<endl;
        flag = 4;
        swap(&name1,&name2, flag);
        cout<<"After swap name1 = "<<name1<<endl;
        cout<<"After swap name2 = "<<name2<<endl;

        return 0;

}
////////////////////////////////////////////////
void swap(void* ptr1, void* ptr2, int flag)
{
    if(flag == 1)
    {
        int  temp;
    temp = *((int*)ptr1);


*((int*)ptr1)= *((int*)ptr2);


*((int*)ptr2) = temp;
    }
    else if(flag == 4)
    {
        string  temp;
        temp = *((string*)ptr1);
        *((string*)ptr1)= *((string*)ptr2);
        *((string*)ptr2) = temp;
    }

}//Listing 5.6
////////////////////////////////////////////////////////
```

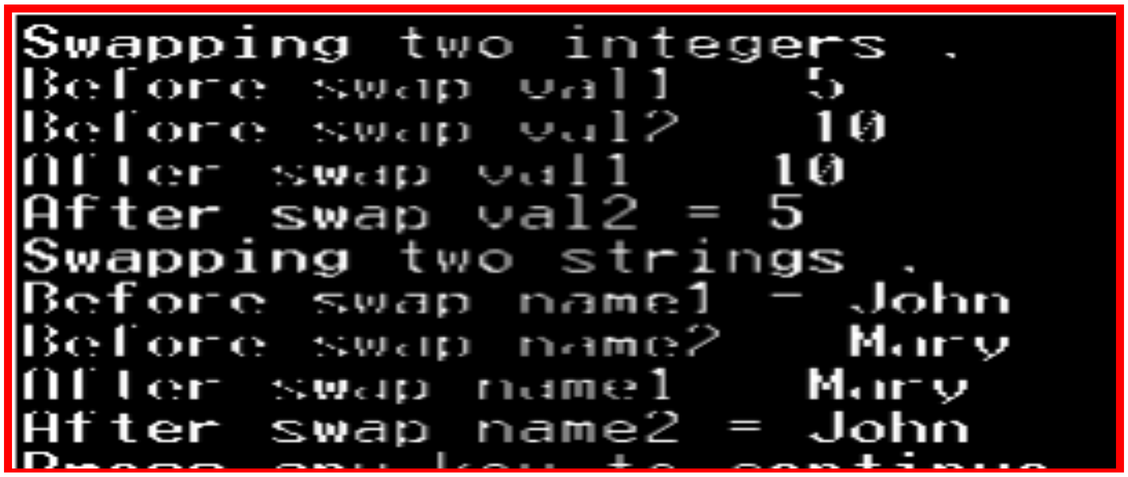
Swap function is called again, this time we pass to it address of two strings.

1. Save the value of pointee of ptr1. Since ptr1 is a void pointer, first it must be cast into an int* type pointer and then be de-referenced.

2. Assign the value stored in the pointee of ptr2 to the pointee of ptr1.

3. Assign the value saved in temp to the pointee of ptr2.

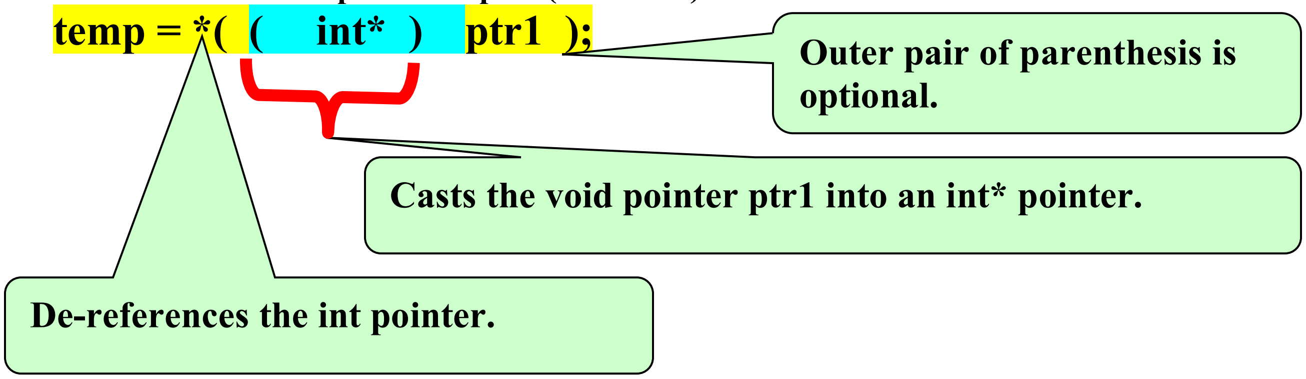
Figure 5.11 shows the results of Listing 5.6.



```
Swapping two integers .
Before swap val1 = 5
Before swap val2 = 10
After swap val1 = 10
After swap val2 = 5
Swapping two strings .
Before swap name1 = John
Before swap name2 = Mary
After swap name1 = Mary
After swap name2 = John
```

FIG. 5.11

When a function takes a pointer as an argument then the address of a variable may be passed to it using address of (&) operator. There is a good reason for using a void pointer because it may take the address of any data type as an actual argument. However, in order to de-reference a void pointer it must be cast into a pointer of the original data type. Therefore the function swap takes an integer flag as an argument, and the value of flag controls the correct casting of its void argument pointers. One must understand the procedure of casting a void pointer precisely. We explain this by the example pointed to by bubble #1. If value of the flag passed to function swap is one then ptr1 and ptr2 are the pointers to two integers. First we save the value of the pointee of ptr1 (bubble #1).



```
temp = *( (  int*  )  ptr1  );
```

Outer pair of parenthesis is optional.
Casts the void pointer ptr1 into an int* pointer.
De-references the int pointer.

After saving the value of the pointee of ptr1 in variable temp, the value of pointee of ptr1 is set equal to that of pointee of ptr2 (bubble #2). Swap is completed by setting the value of the pointee of ptr2 equal to temp, which is saved value of pointee od ptr1 (bubble #3). The string swap follows the same logic.

Returning Pointers as a returned value from a function

You would recall that C++ does not allow arrays to be returned as the return value from functions. That limitation may be relieved somewhat as the pointer to an array can be returned as a return value from a function. As we know that the arrays created dynamically are not initialized by C++. One good application of pointer to

an array being returned as a return value from a function is to initialize an array through such function. The example is shown in Listing 5.7.

```
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <iostream>
#include "counter.h"
#include <ctime>
using namespace std;
```

1. Initializes the array to zero values and returns a pointer to initialized array.

```
int * initializeArray (int * const arr, int len);
```

2. Fills the array with random numbers and returns a pointer to the filled array.

```
int * fillArray (int * const arr, int len);
```

3. Prints the array passed as an argument arr

```
void printArray(int * const arr, int len, ostream &
out);

void main()
{
      int len = 10;
int *  ptr = new int[len];
```

4. Array declared using new

```
       mem_couter++;
       cout<<"Printing the initialized array.\n";
```

5. Pointer to initialized array returned.

```
ptr = initializeArray(ptr,len);
      printArray (ptr, len, cout);
      cout<<"Printing the array filled with random numbers.\n";
ptr = fillArray(ptr, len);
```

6. Pointer to filled array returned

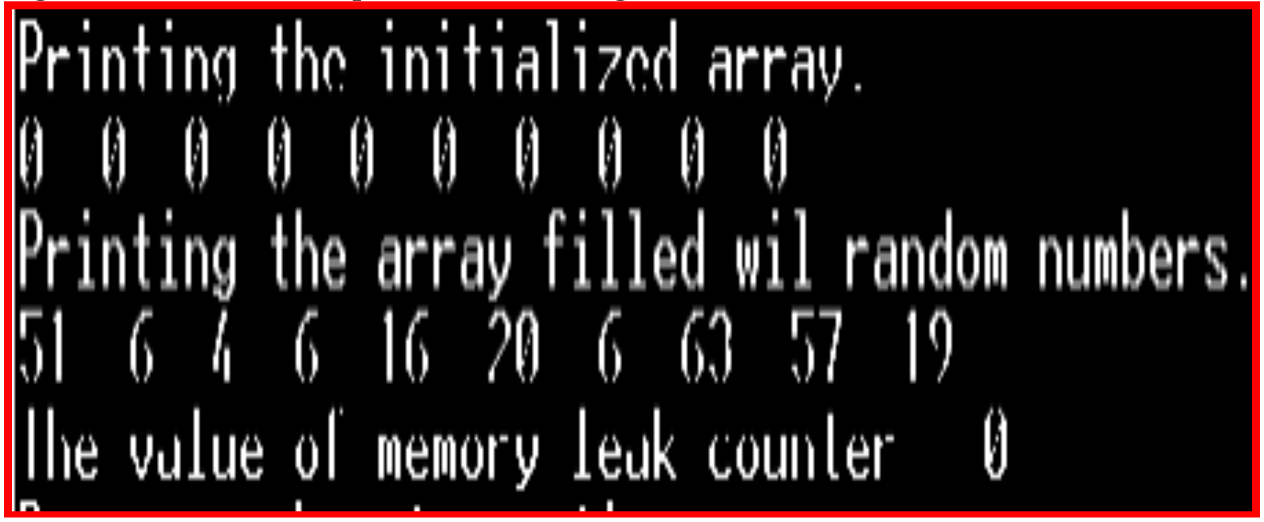
```
printArray(ptr,len,cout);
```

7. The printArray function prints the array to console.

```
delete [] ptr;
```

8. The memory allocated for the array released by calling the delete.

```
      mem_couter--;
      cout<<"The value of memory leak counter = "
            <<mem_couter<<endl;
}


//////////////////////////////////////////////////////
 int * initializeArray(int * const arr, int len)
{
      for(int index = 0; index<len; index++)
            arr[index] = 0;
      return arr;
}
/////////////////////////////////////////////////////////
 void printArray(int * const arr, int len, ostream& out)
 {
       for(int index = 0; index<len; index++)
             cout<<arr[index]<<"  ";
       cout<<"\n";
 }
 /////////////////////////////////////////////////////////
 int * fillArray(int * const arr, int len)
 {
       srand( (unsigned)time( NULL ) );

       for(int index=0; index<len; index++)
             arr[index] = rand()%100;
       return arr;
 }//Listing 5.7
 /////////////////////////////////////////////////////////////
```
Figure 5.12 shows the output from the Listing 5.7.



```
Printing the initialized array.
0  0  0  0  0  0  0  0  0  0
Printing the array filled wil random numbers.
51  6  4  6  16  20  6  63  57  19
The value of memory leak counter   0
```

FIG. 5.12

The Listing 5.7 has three helper functions whose details are described in the table 5.3 below.

int * fillArray (int *const *arr,* int *len*)
Fills the array of length len with random numbers from 0 to 99.

Parameters:
arr is the pointer to the integer array.
len is the logical length of the array.
Returns:
the pointer to the array filled with random numbers 0 to 99.

int * initializeArray (int *const *arr,* int *len*)

Initializes the array of length len to all zeros.

Parameters:
arr is the pointer to the integer array.
len is the logical length of the array.
Returns:
the pointer to the initialized array.

void printArray (int *const *arr,* int *len,* ostream & *out*)

Prints the array arr, of logical length len to user specified device.

Parameters:
arr is the pointer to the integer array.
len is the logical length of the array.

Table 5.3

The function initializeArray (bubble #1) takes a constant non-reassignable pointer as an argument and fills all logical length members with zero. The function fillArray fills all logical length members with random numbers from zero to 99 and returns a pointer to its first cell (bubble #2). The function printArray takes the pointer to array, its logical length, and ostream object to print the logical length elements of the array to medium dictated by the ostream object(bubble #3). In the main function the array of length len is declared using operator new (bubble #4). After

the array is initialized (bubble #5), filled (bubble #6), and printed (bubble #7), the delete operator deletes the memory allocated for the array (bubble #8).

Copy constructor

Let us write a following simple class (so simple that we do not even give it a Listing number).

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <iostream>
using namespace std;
class SimpleClass
{
private:
	int data;
public:
	SimpleClass (int data1=0): data (data1){}
	/*{

	}*/
	friend ostream& operator <<(ostream& out, const
SimpleClass & S)
	{
		out<<"The value of data = "<<S.data<<endl;
		return out;
	}};
void main()
{
	SimpleClass S1(10);
	cout<<"Printing S1.\n";
	cout<<S1;
	SimpleClass S2 (S1);  //Line #4
//Alternate way to write line #4 is below
//	SimpleClass S2 = SimpleClass (S1);
	cout<<"Printing S2.\n";
	cout<<S2;

}
//Listing SimpleClass.cpp
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

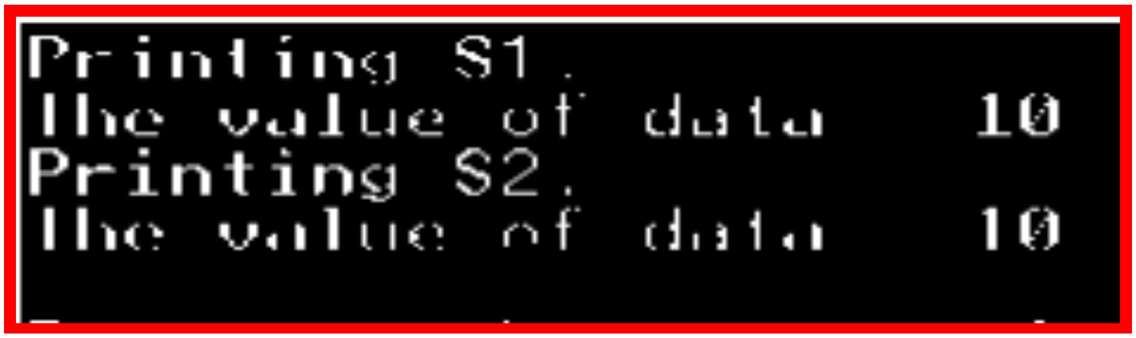
1. Place holder for Clone constructor or copy constructor.

This simple class called SimpleClass has one integer data member called data, an explicit/default constructor, and an overloaded insertion operator (<<). The first three line of main function simply create an object S1 of type SimpleClass and print

it. But the line #4 may shock you. In line 4, we are passing the pre-created instance of S1 as an argument to construct S2. Well your first reaction may be that this will not compile because there is no SimpleClass constructor whose possible signature, according to code in line #4, must be:

```
SimpleClass (SimpleClass Other);
SimpleClass (SimpleClass& Other);
```

But the truth is that this program does compile and the figure below shows the output.



```
Printing S1.
 The value of data    10
Printing S2.
 The value of data    10
```

FIG. Output From SimpleClass Listing

Well we cannot escape the conclusion that behind the scene C++ is helping us by providing a constructor, which can take an object of same type as an argument and creates a new object, which is a copy of the passed object. However, assumptions are not enough. It will be nice if we can prove it. Since this hidden constructor makes a clone or copy of the same type object passed as an argument, one can call it a clone constructor, but let us call it with a more commonly used term - a copy constructor. How can we prove that this hidden constructor really is there or it is another one of hidden C++ mystery? The answer is simple. Using our imagination we write the full body of this hidden clone/copy constructor and place it at the location shown by bubble#1 in the class SimpleClass. With some experimenting you will find out that the copy constructor with signatures below does not even compile.

```
SimpleClass (SimpleClass Other);//Compile error
```

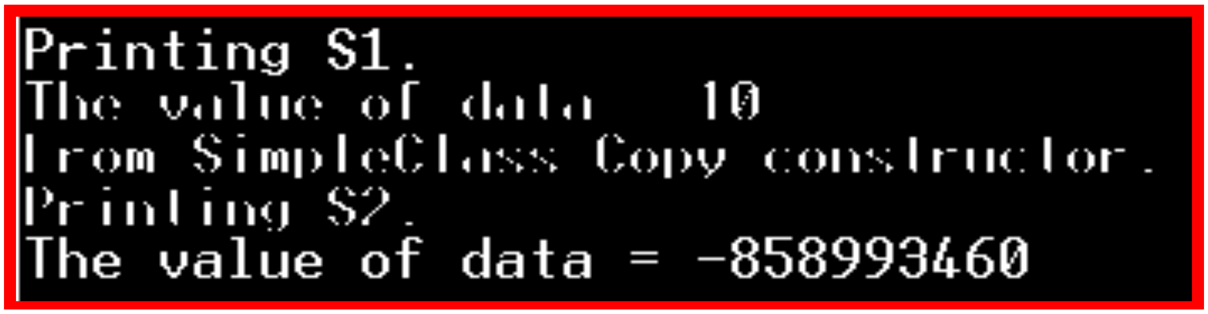
Since objects can only be passed by value, reference, and pointer, we try the second possible form, which is:

```
SimpleClass (SimpleClass& Other);
```

The above form will compile. We try the simple code first. We write the full body of this clone/copy constructor as follows and place it at the location held by bubble #1 in SimpleClass listing.

```
SimpleClass (SimpleClass& Other)
{
        cout<<"From SimpleClass Copy constructor.\n";
}
```

Now the above does compile, indicating that perhaps that is the correct proto-type of copy constructor. The results after we include this constructor in the SimpleClass are shown in the figure below.

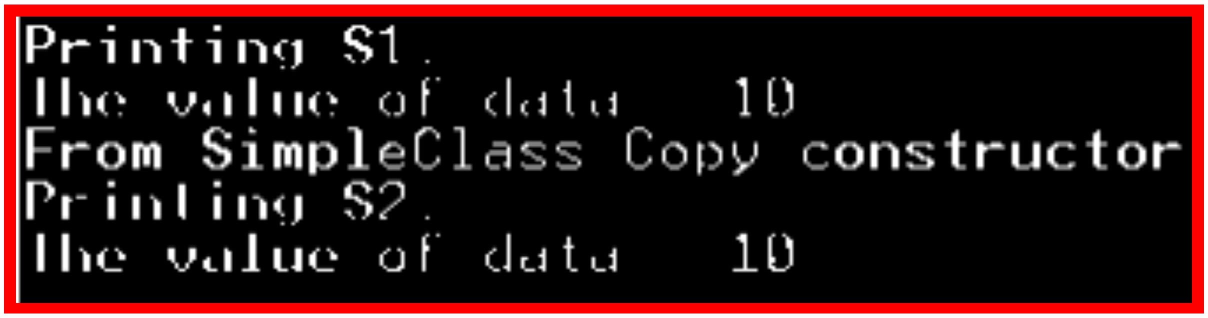


```
Printing S1.
The value of data = 10
From SimpleClass Copy constructor.
Printing S2.
The value of data = -858993460
```

The output shows that indeed the copy constructor we wrote was called since the output statement from it is displayed. However, this constructor does not set the data field of object S2, equal to the data field of object S1. Well this happens because of one C++ rule about which you are already aware. Whatever C++ giveth, it also taketh it away. Recall that if you put an explicit constructor in your class, then C++ takes away the default constructor it gives you. Therefore if you are going to provide your own explicit constructor, then you must provide your own default constructor as well. Same thing happens here. As soon as we provide our own copy constructor, C++ takes away the one it provided for us. And we can see that our constructor body does really nothing, except outputting a statement. We can fix this by adding the copy code in our copy constructor by upgrading it as given below.

```
SimpleClass (SimpleClass& Other)
{
        this->data = Other.data;
         cout<<"From SimpleClass Copy constructor.\n";
}
```

Once we use this constructor inside the Listing for SimpleClass, the output is shown below.



```
Printing S1.
The value of data = 10
From SimpleClass Copy constructor
Printing S2.
The value of data = 10
```

One can see that except for the output statement “From the SimpleClass Copy constructor”, this output is identical to the situation when we had no copy constructor present in the class SimpleClass. Therefore we can conclude that each C++ class/struct or union carries a hidden constructor inside itself whose signature will be of form

```
MyClass (MyClass& Other);
```

It turns out that we can use the const modifier for the formal argument in the copy constructor, which allows us to write its other acceptable form as follows:

```
MyClass (const MyClass& Other);
```

For SimpleClass this will become

```
SimpleClass (const SimpleClass& Other)
```

The above forms have the advantage that the object Other will not be accidentally altered by the constructor body.

Having concluded that C++ provides a constructor, called copy constructor, which can make member-wise copy of the passed object to generate its clone, we can move to the discussion of default assignment operator C++ provides.

Assignment Operator

When we discussed the class Complex in the chapter on operator overloading you may have noticed that we were able to write expressions such as these:

```
Complex Num1, Num2, Num3;
Num3 = Num1 + Num2;// Expression1
```

Perhaps you also remember from previous class in C++ that structures are assignable to each other. Which means that if we define a structure such as?

```
struct Student
{
        string  name;
        float   gpa;
};
```

then we can write expressions such as:

```
Student S1, S2;
S2 = S1; // Expression2
```

In expression 1 the overloaded plus (+) operator adds Num1, and Num2 and sets all data fields of Num3, equal to the Complex number returned by the overloaded + operator. Similarly in expression 2, all fields of object S2 will be set equal to object S1. Notice that in neither of above class (or struct) we needed to overload the assignment operator (=) to ascertain that data transfer from source object to the destination object is done accurately. Upon invocation of assignment operator (as in expressions1 and expression2), C++ guarantees member-wise copy as long as the followings are true:

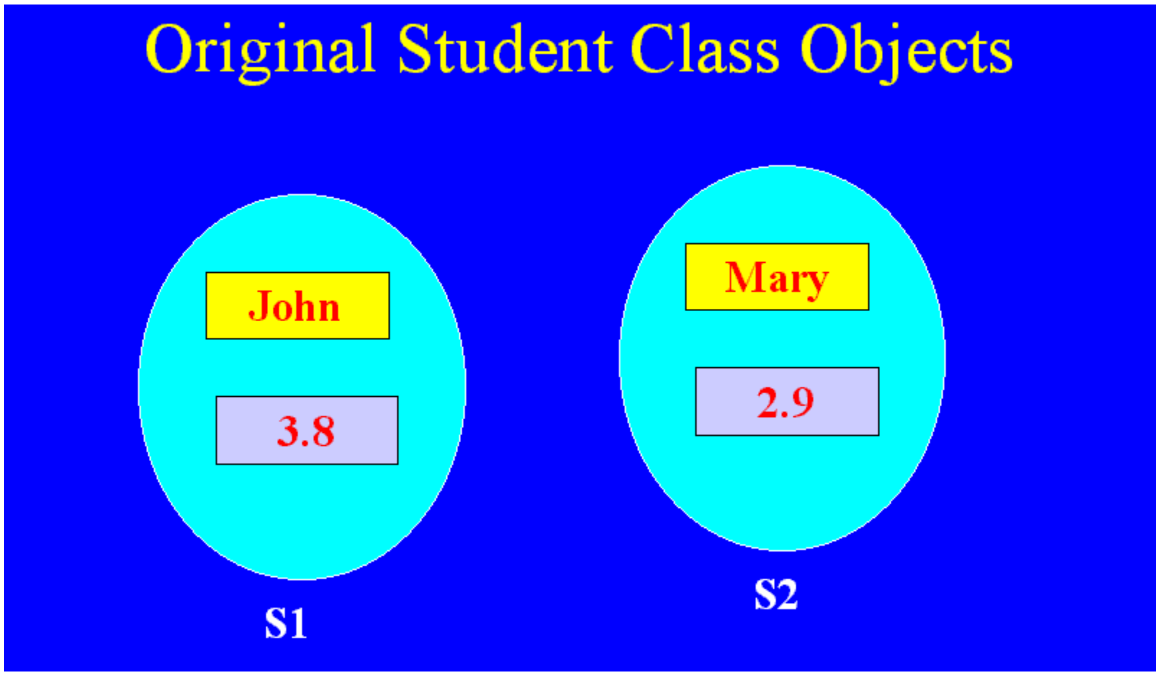
- The class/struct invoking assignment operator only has primitives or their static array as their data members.

- **Or, data member is a class/struct or its static array, for which assignment operator (=) has been overloaded.**

We show the following PowerPoint presentation to illustrate the mechanism of invocation of assignment operator. Here we make the assumption that both assignment operator and copy constructor are passed argument by value.

EqualityOperator1.ppt

Figure 5.12A shows the two objects of Student class.



<u>FIG. 5.12A</u>

Upon invocation of default assignment operator a no name clone of the object S1 is created. A C++ provided copy constructor creates this no-name clone of S1 (Figure 5.12B).

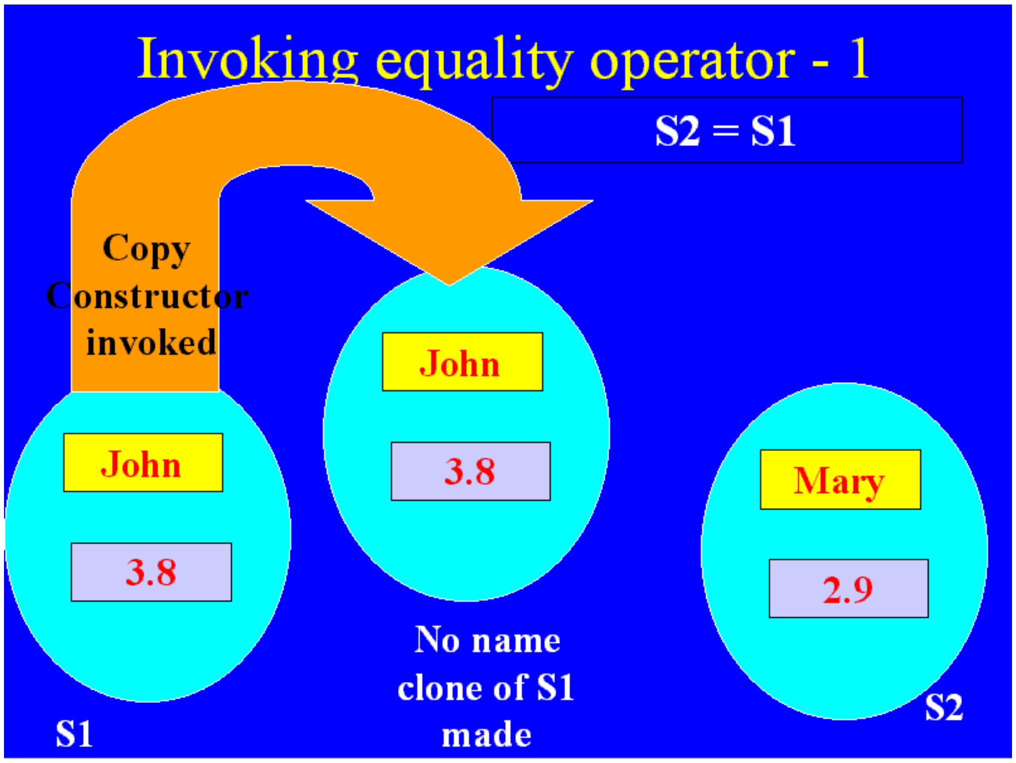


FIG. 5.12B

Then member-wise copy from the clone to the destination object (S2) is made, completing the assignment process. (Figure 5.12C).

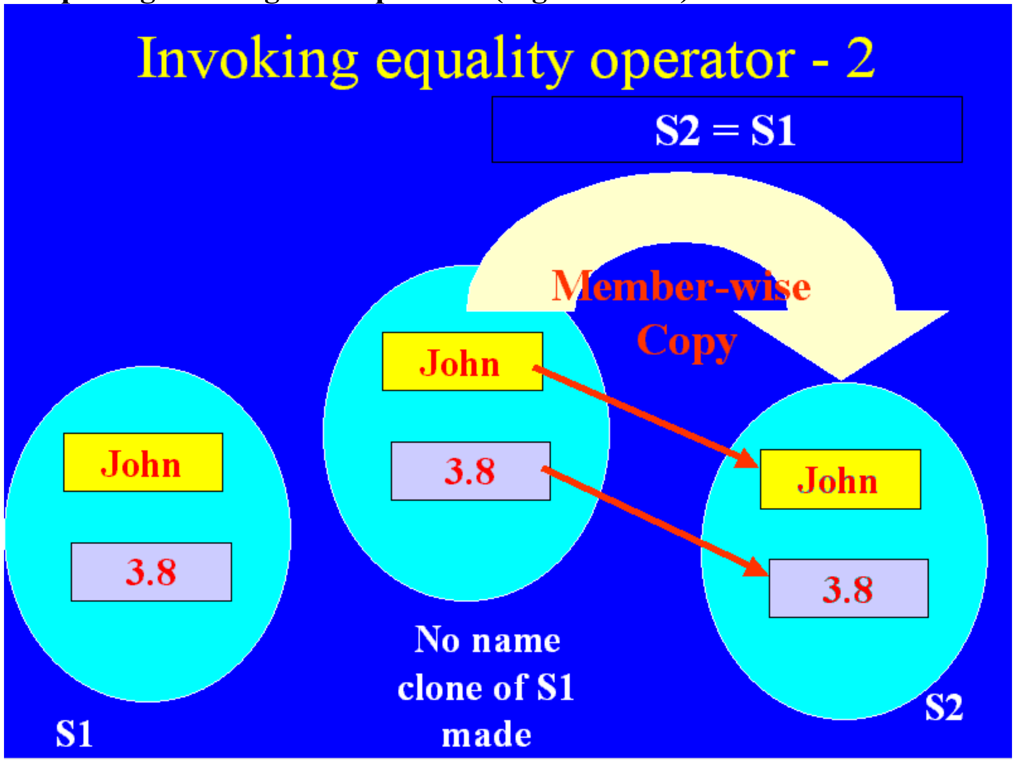
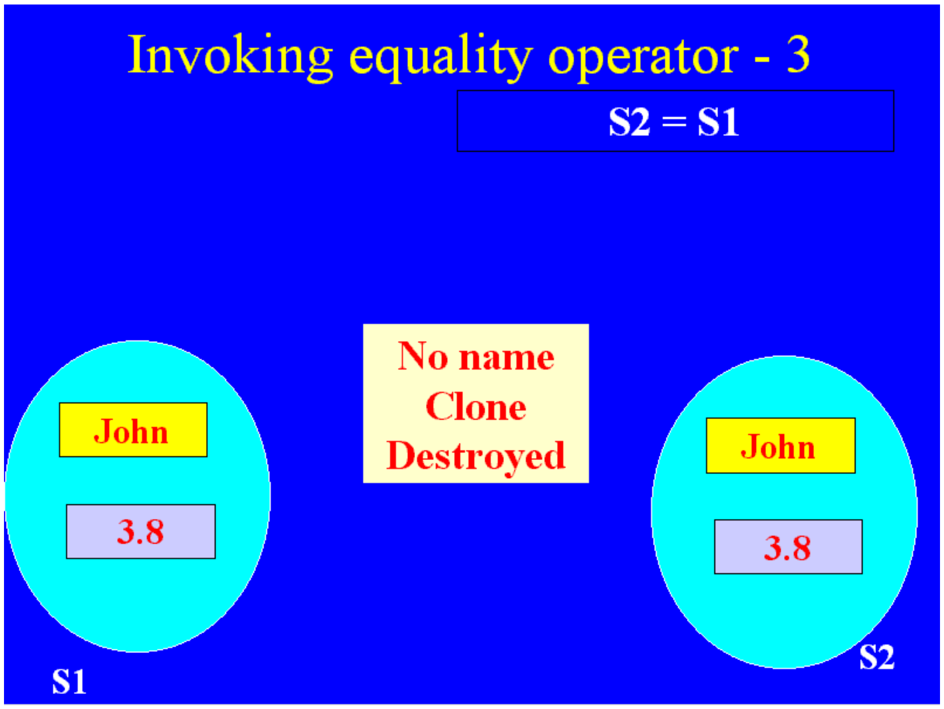


FIG. 5.12C

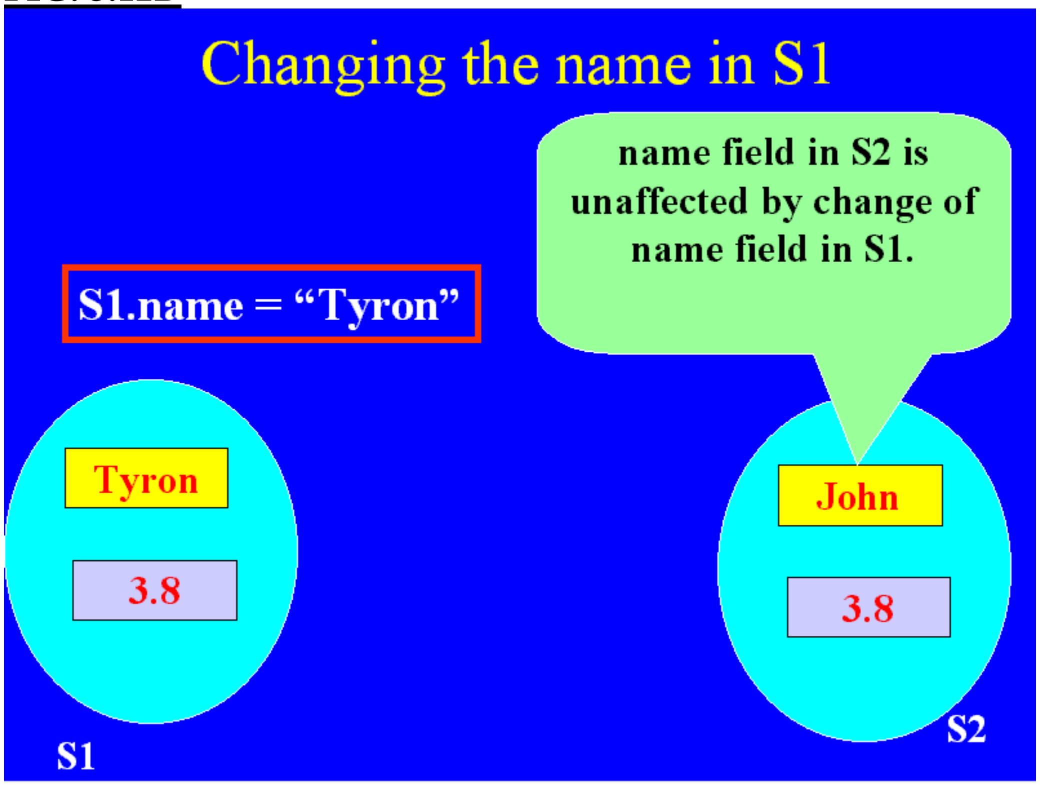
Finally the no-name clone of S1 is destroyed (Figure 5.12D).



Invoking equality operator - 3

S2 = S1

FIG. 5.12D



Changing the name in S1

S1.name = “Tyron”

name field in S2 is unaffected by change of name field in S1.
Tyron
3.8
S1
John
3.8
S2

FIG. 5.12E

Understand that if the name in object S1 is changed to “Tyron”, then name in S2 is unaffected by this change (Figure 5.12E). This is called making a deep copy, where each object owns independent copies of their data members.

For class/struct Student the signatures of C++ provided copy constructor and assignment operator could have the following signatures:
Student (const Student& Other_Student);
Student operator = (const Student& Other_Student);

The general prototype for the assignment operator for a class MyClass could be:
MyClass operator = (const MyClass&);
The following variations in return types from assignment operator are allowed.
MyClass operator = (MyClass);//Allowed, but unsafe as the returned object //may be altered. Also it is inefficient.
const MyClass operator = (const MyClass&);// safer but less efficient
const MyClass& operator = (const MyClass&);//Most efficient &safe
How can we prove that the above described assignment process is feasible? We can simply provide a blank body copy constructor and minimal copy code in assignment operator with an output statement inside them to see that these calls are indeed made. The Listing 5.9 below shows the code and the output (Figure 5.13).

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

struct Student
{
      string name;
      float gpa;

      Student(){}
      Student(const Student& Other_Student)
      {
cout<<"From  Student class copy constructor.\n";
      }

      Student operator = (const Student&
Other_Student)
      {
cout<<"From Student class overloaded =”
<<“Operator.\n";
            name = Other_Student.name;
            gpa=Other_Student.gpa;
            return *this;
```

1. A do nothing copy constructor

2. Overloaded assignment operator.

```
        }

        void printStudent()
        {
                cout<<"Name = "<<name<<endl;
                cout<<"GPA = "<<gpa<<endl;
        }
};

void main()
{
        Student S1;
        S1.name = "John";
        S1.gpa = 3.8f;
        cout<<"Printing S1\n";
        S1.printStudent();
```

Create and print object S1

```
        Student S2;
        S2.name = "Mary";
        S2.gpa = 2.9f;
        cout<<"Printing S2\n";
        S2.printStudent();
```

Create and print obiect S2

```
        cout<<"Setting S2 = S1.\n";
        S2 = S1;
```

Invoke assignment operator.

```
        cout<<"Printing S2\n";
        S2.printStudent();


        cout<<"Changing the name field of student S1 to \"Tyron\"\n";
        S1.name = "Tyron";
        cout<<"Printing S1\n";
        S1.printStudent();
```

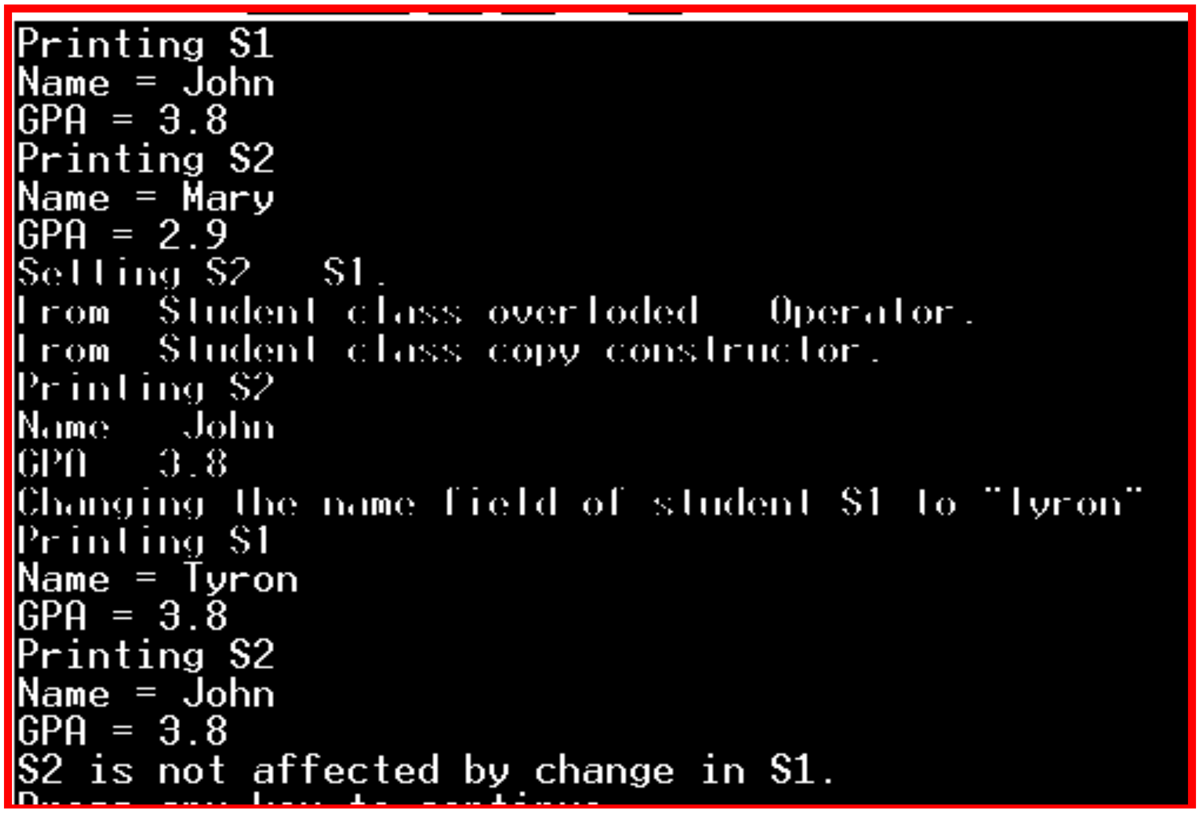
Change the name field of S1.

```
        cout<<"Printing S2\n";
        S2.printStudent();
        cout<<"S2 is not affected by change in S1.\n";
}
```

Listing 5.9[3]

//

[3] We need to provide a default constructor, because of C++ rule that once any constructor (be it a copy constructor) is provided the default constructor is taken away.



```
Printing S1
Name = John
GPA = 3.8
Printing S2
Name = Mary
GPA = 2.9
Setting S2 = S1.
From Student class overloded = Operator.
From Student class copy constructor.
Printing S2
Name = John
GPA = 3.8
Changing the name field of student S1 to "Tyron"
Printing S1
Name = Tyron
GPA = 3.8
Printing S2
Name = John
GPA = 3.8
S2 is not affected by change in S1.
```

<u>FIG. 5.13</u>

The output shows that both copy constructor and assignment operator are called. Moreover any change in the name field of S1 does not affect object S2, which is a deep copy of S1. In other words the picture in Figure 5.12D is accurate.

What is most instructive, however is that we can comment out the highlighted code for constructors and assignment operator, and the object is still copied correctly. And output remains identical to the Figure 5.13.

This would prove that behind the scene C++ provides a copy constructor and default assignment operator, which does member-wise copy for all objects of struct, class and union. C++ provided copy constructor and assignment operator work fine as long as the following two conditions are true:

1. **The class/struct/union members are atomic data types.**
2. **Or the class, which is member of the other class, provides the copy constructor and assignment operator, which perform the task, which similar C++, provided functions would.**

We can alter the Listing 5.9 to show, as to how we run into trouble, as soon as we alter the name data type to be a dynamically allocated string pointer, instead of a string. This is shown in re-written Student struct in Listing 5.10.

//

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

struct Student
{
      string * name;
      float gpa;
```

1. Field name is a pointer to a string now.

```
      void printStudent()
      {
            cout<<"Name = "<<*name<<endl;
            cout<<"GPA = "<<gpa<<endl;
      }
};

void main()
{
      Student S1;
      S1.name = new string ("John");
```

2. Allocate the memory for the name field of S1

```
      S1.gpa = 3.8f;
      Student S2;
      S2.name = new string ("Mary");
```

3. Allocate the memory for the name field of S2.

```
      S2.gpa = 2.9f;
      cout<<"Printing student S1.\n";
      S1.printStudent();
      cout<<"Printing student S2.\n";
      S2.printStudent();
      cout<<"Setting student S2=S1.\n";
      S2 = S1;
```

4. Set S2 = S1.

```
      cout<<"Printing student S2 again.\n";
      S2.printStudent();
      cout<<"Changing name field of student S1 to \"Tyron\".\n";

      *S1.name = "Tyron";
```

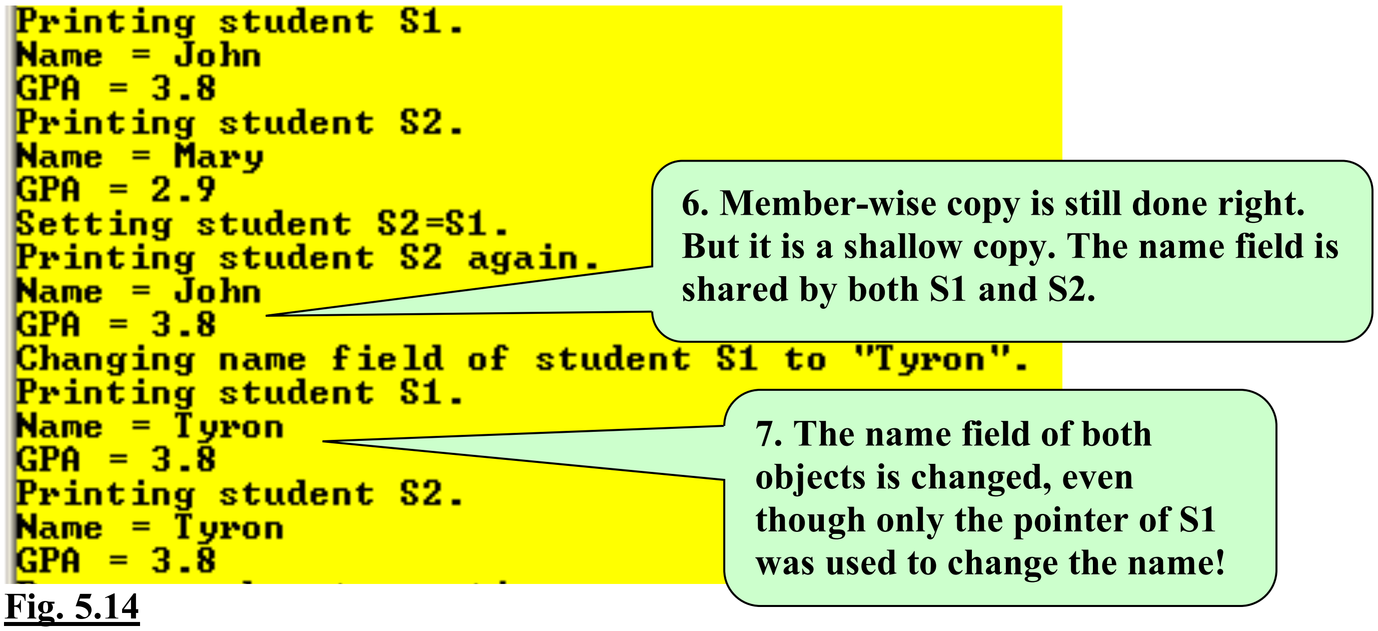
5. Change the name field of S1 through its pointer.

```
      cout<<"Printing student S1.\n";
      S1.printStudent();
      cout<<"Printing student S2.\n";
      S2.printStudent();
}
```

Listing 5.10

//

The output of Listing 5.10 is shown in Figure 5.14.



```
Printing student S1.
Name = John
GPA = 3.8
Printing student S2.
Name = Mary
GPA = 2.9
Setting student S2=S1.
Printing student S2 again.
Name = John
GPA = 3.8
Changing name field of student S1 to "Tyron".
Printing student S1.
Name = Tyron
GPA = 3.8
Printing student S2.
Name = Tyron
GPA = 3.8
```

6. Member-wise copy is still done right. But it is a shallow copy. The name field is shared by both S1 and S2.

7. The name field of both objects is changed, even though only the pointer of S1 was used to change the name!

Fig. 5.14

In main we set the name fields of both objects S1 and S2 by dynamically allocating memory for names (bubbles #1 and #2). Printing both objects gives results similar to Listing 5.9. We set the object S2 equal to S1 (bubble #4). Printing S2 also give expected results. Then we do what we did in Listing 5.9, i. e. change the name in object S1 through its pointer (bubble #5). Now, however the results are totally different from Listing 5.9. The name field of both objects (S1 and S2) is changed to new name Tyron (bubble #7).

Why changing the name through the pointer of S1 also changes the name field of object S2? We show the modified form of our earlier PowerPoint presentation to explain this.

ShallowCopy.ppt

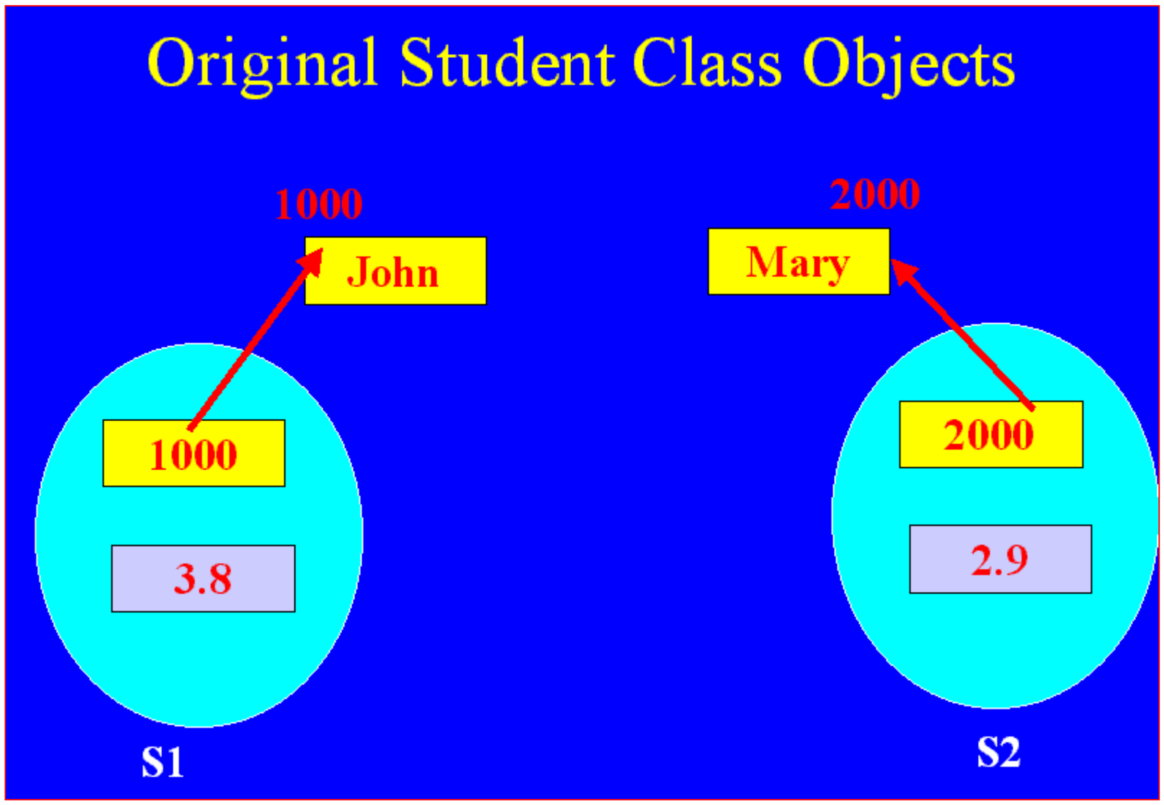


Fig. 5.15A

Since the Student class now stores a string pointer name, the pointer stores the address of string object created on heap. Pointer name in objects S1 and S2 store different addresses, thus different names. (Figure 5.15A).

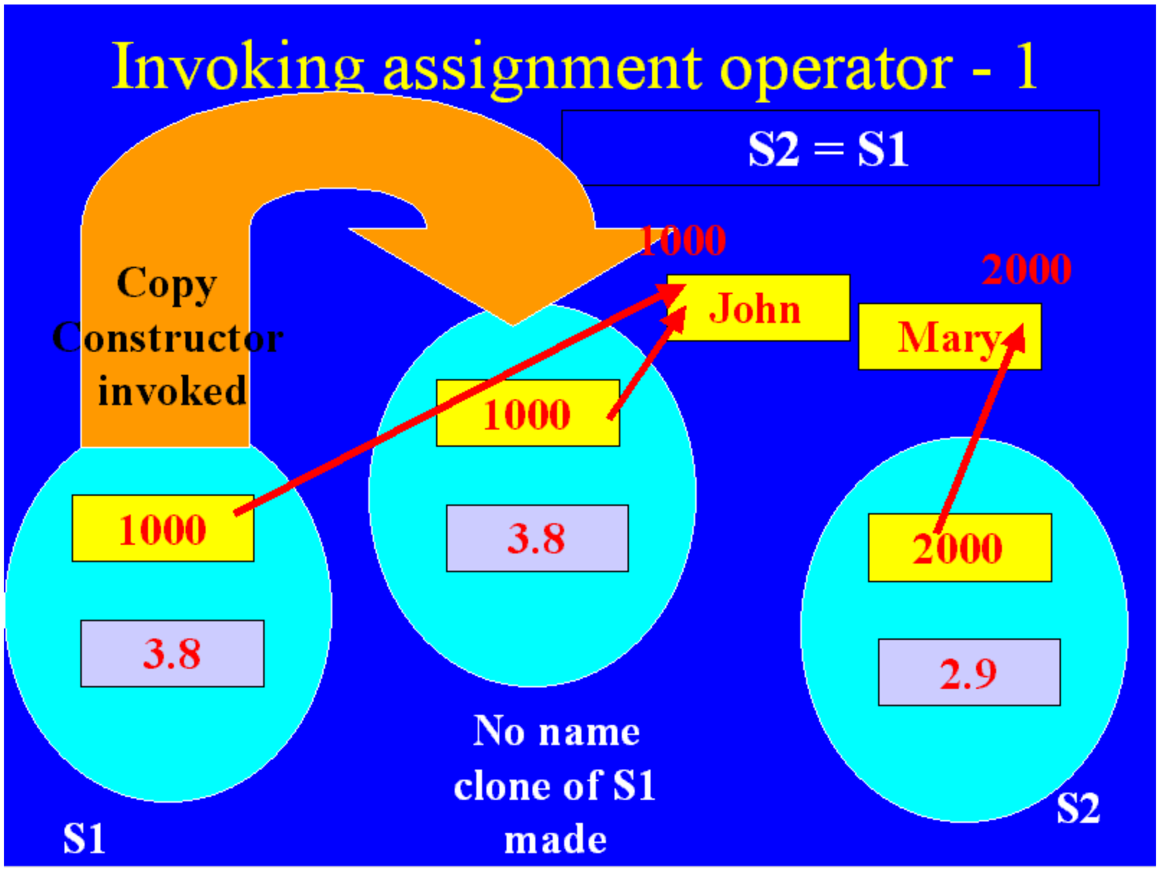


Fig. 5.15B

When we perform assignment
S2 = S1;
Now, the default copy constructor creates a no-name clone of S1, and the pointer value 1000 is also stored in the name pointer of the clone (Figure 5.15B).

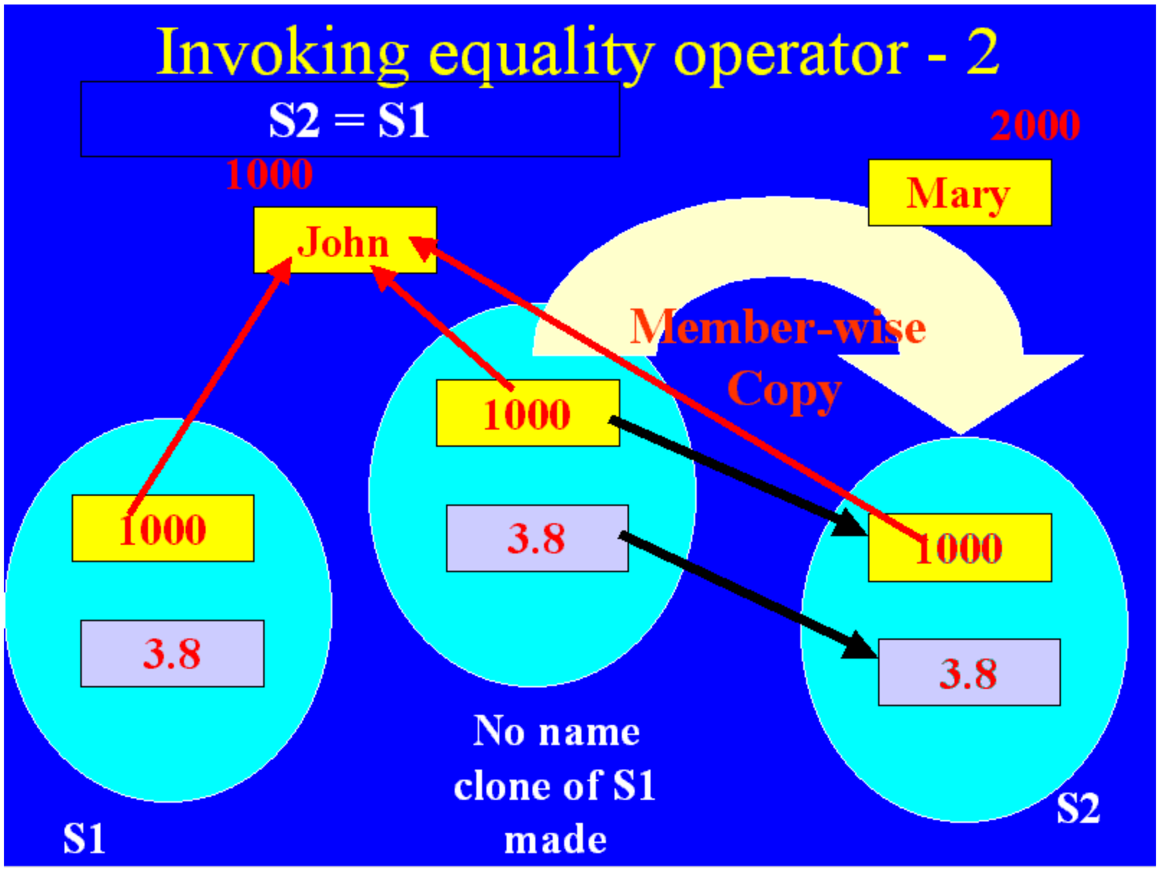


Fig. 5.15C

Then default assignment operator makes the member-wise copy from no-name clone to the respective fields of S2. Now name pointers of all objects (S1, S2, and no-name clone) all share the same pointee (Fig. 5.15C).

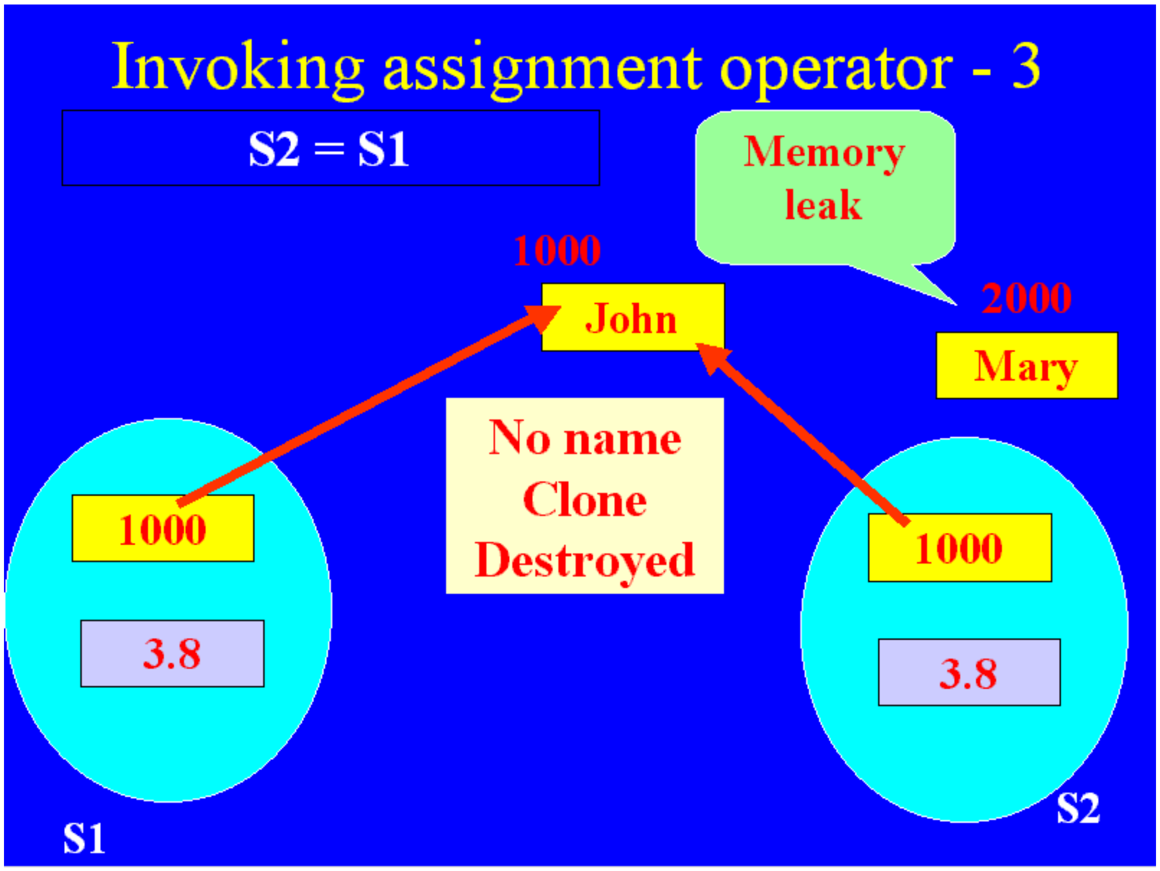
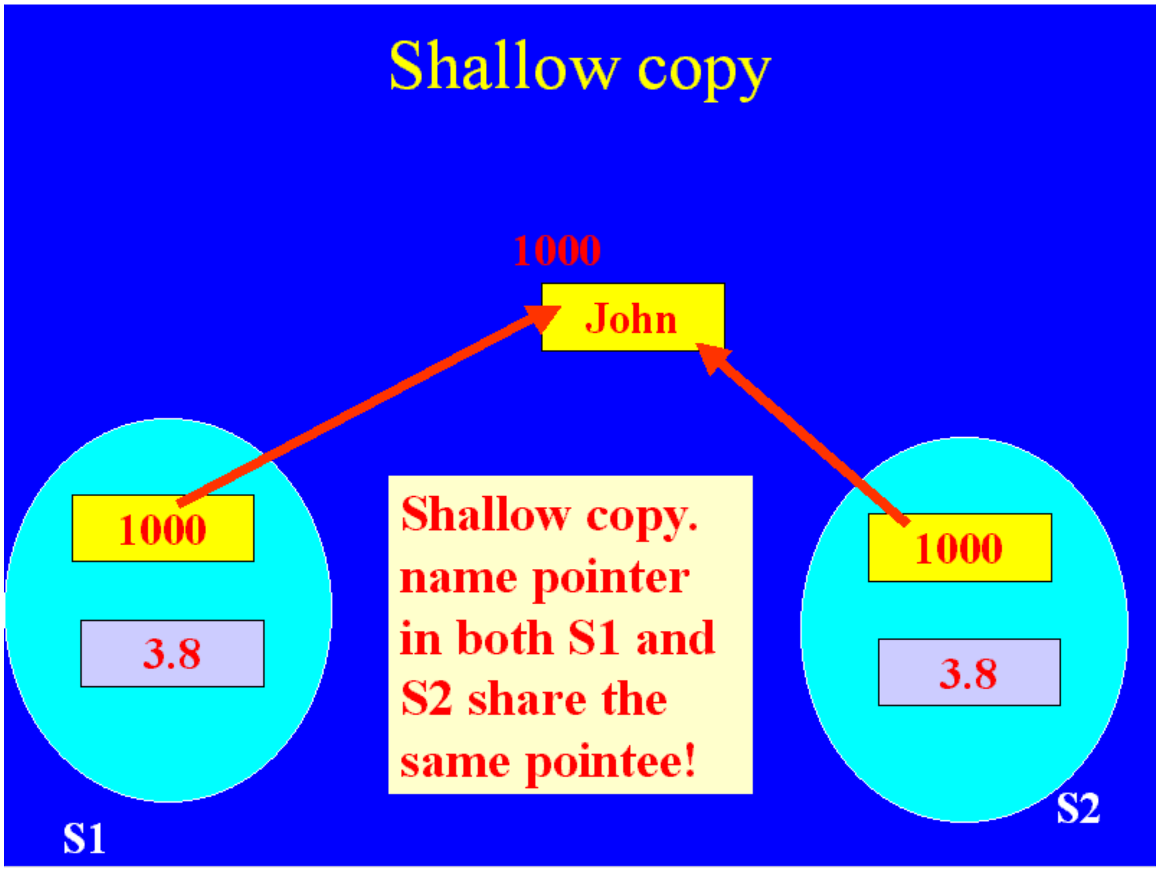
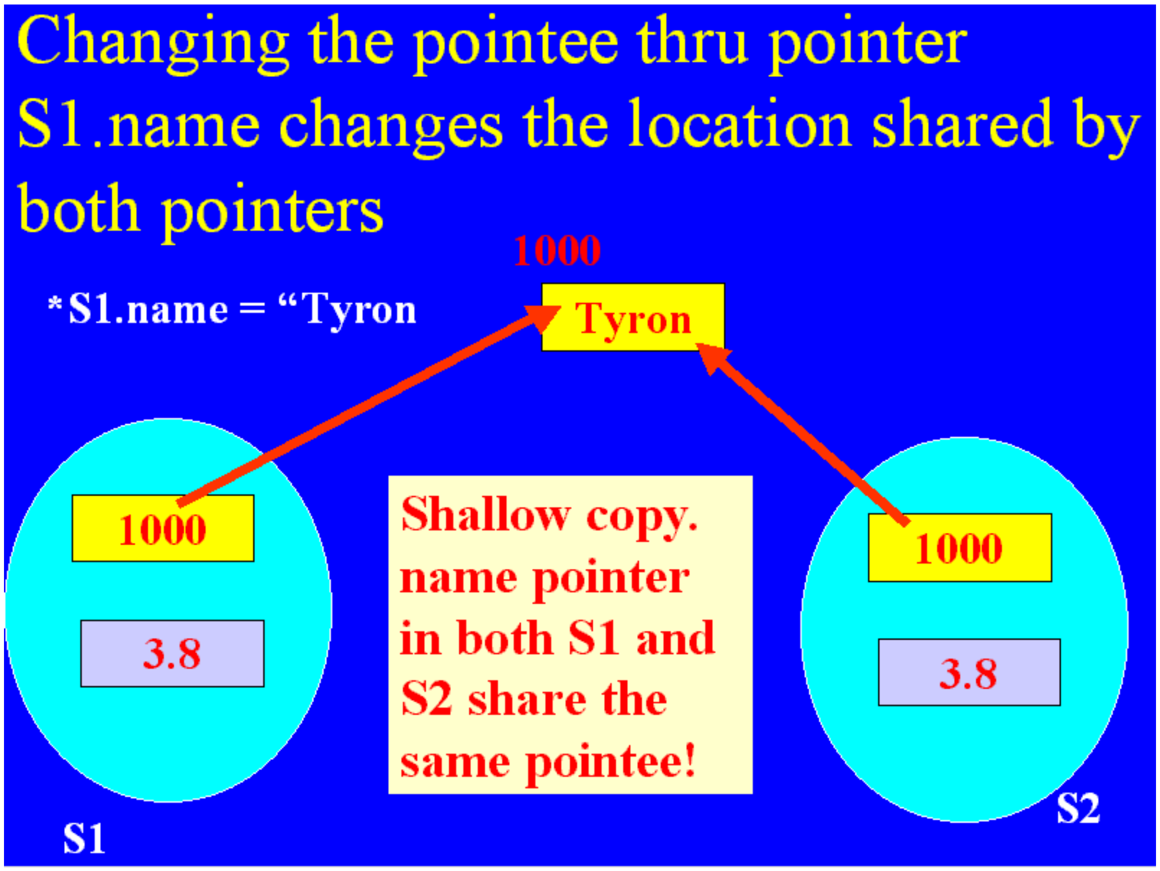


Fig. 5.15D
No-name clone is destroyed. Location in memory storing Mary becomes a memory leak.


Shallow copy
1000
John
1000
3.8
Shallow copy. name pointer in both S1 and S2 share the same pointee!
1000
3.8
S1
S2

<u>Fig. 5.15E</u>
This is called shallow Copy as name pointers in both S1 and S2 have same pointee. This is also called sharing. (Figure 5.15E). The default assignment operator and copy constructor provided by C++ can only make shallow copies as for pointer class members; only the values of pointers are copied. No copies are made of the pointee! Hazards of this are shown in Figure 5.15F.


Changing the pointee thru pointer S1.name changes the location shared by both pointers
*S1.name = "Tyron
1000
Tyron
1000
3.8
S1
Shallow copy. name pointer in both S1 and S2 share the same pointee!
1000
3.8
S2

<u>Fig. 5.15F</u>

Now changing the name field in S1 through its name pointer by program statement;
***S1.name = "Tyron";**
changes the shared pointee, and in effect the name field for both objects changed.

Therefore when classes have pointer data as their members C++ provided copy constructor and assignment operator only make shallow copies, where the copied objects share the memory location with the source object. Thus programmer must provide their own version of copy constructor and assignment operator, so that the overall process makes "deep " copies!

<u>Writing Copy constructor and overloading assignment operator</u>

Extending the example of Student class/struct discussed in Listing 5.10 we write the copy constructor and assignment operators, which will facilitate making deep copies. Listing 5.11 shows this.

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

```
static int leak_counter = 0;

struct Student
{
	string * name;
	float gpa;

	Student(){name = NULL; gpa=0.0f;}
	Student(string name1, float gpa1)
	{
		name = new string(name1);
		leak_counter++;
		gpa = gpa1;
	}

	Student (const Student& Other)
	{
		if(this==&other)
		{
			exit(1);
		}
		name = new string (*Other.name);
		leak_counter++;
		gpa = Other.gpa;
		cout<<"From Copy constructor.\n";
	}


	const Student & operator = (const Student&
Other)
	{
		if(this == &Other)
			return *this;
		delete name;
		leak_counter--;
		name = new string(*Other.name);
		leak_counter++;
		gpa = Other.gpa;
		return *this;
	}

	virtual ~Student()
	{
```

1. Leak_counter is incremented, each time new operator is invoked and decremented, each time delete operator is called.

2. Copy constructor

Check for self-copy construction.

3. Assignment operator overloaded.

4. Check for self-assignment

5. Virtual destructor provided. The destructor deletes the dynamically allocated memory.

```
            delete name;
            leak_counter--;
      }



      void printStudent()
      {
```

6. Overloaded assignment operator is called when return value is assigned to self. Copy constructor is called when return value is assigned to some other object of type Student. Return type can be Student or Student &

```
static const Student& increaseGpa (Student & Std)
      {
            if (Std.gpa<=3.9)
             Std.gpa+=0.1f;

            return Std;
      }
};
```

7. increaseGpa static member function increases the gpa of Student object passed to it by 0.1 with out going over 4.0

```
void main()
{
      {
            Student S1;
            S1.name = new string("John");
            leak_counter++;
            S1.gpa = 3.8f;
            Student S2;
            S2.name = new string("Mary");
            leak_counter++;
            S2.gpa = 2.9f;
            cout<<"Printing student S1.\n";
            S1.printStudent();
            cout<<"Printing student S2.\n";
            S2.printStudent();
            cout<<"Setting student S1=S2.\n";
            S2 = S1;
            cout<<"Printing student S2 again.\n";
            S2.printStudent();
            cout<<"Changing name field of student S1 to \"Tyron\".\n";
            *S1.name = "Tyron";
            cout<<"Printing student S1.\n";
            S1.printStudent();
            cout<<"Printing student S2.\n";
            S2.printStudent();
            S2 = Student::increaseGpa(S2);
            cout<<"After increasing GPA of student S2.\n";
            S2.printStudent();
      }
      cout<<"The value of leak counter = "<<leak_counter<<endl;
}//Listing 5.11
```

//

In above code we use a global static integer variable called leak_counter (bubble #1) to ascertain that our program has no memory leaks. We use leak_counter in the following way. Each time we invoke operator new, we increment the leak_counter by one, whereas it is decremented by one, each time we invoke the operator delete. We put the code in the main function inside a block (a pair of curly braces) so that all new and delete calls as well as the call to destructor are forced to be completed with in that block. Then we print the value of leak_counter outside the code block, just before the return statement. If value of leak_counter is zero then all the dynamically allocated memory has been de-allocated safely, indicating that program has no memory leaks.

Copy constructor must make the deep copies of the data members, which are pointed to by class members, which are pointers. We write the copy constructor (bubble #2) using the following algorithm.

1. **Check for self-copying. If object passed is same then exit. (use exit(1) or assert.**
2. **Assign the memory for the pointer class data members using new.**
3. **Increment the leak_counter as many times as new is invoked.**
4. **Copy the contents of the pointee of name pointer of object Other to the name pointer (bubble #2). This step copies the content and not the pointer address. Therefore deep copy pointee of the Other name pointer is made.**
5. **Copy the other data members in usual manner.**

The output statement shown in copy constructor is optional.

The assignment operator (bubble #3) must make deep copies whether the object is being assigned to some other object or to itself. With respect to class Student, the assignment operator must make deep copies whether we make self-assignment as in:

Student S1;
S1 = S1;

Or assignment is made to other objects of same type as in:

Student S2;
S2 = S1;

The algorithm for writing the assignment operator is as follows (bubbles #4 and code after that):

1. Check for self-assignment. If object passed as an argument has the same address as the current object then return the current object.
2. De-allocate the previously allocated memory to pointer data members by calling delete.
3. Decrement the variable leak_counter number of times the delete was invoked in step two.
4. Make the deep copies of pointees of object Other by freshly allocating the memory for pointer data members by calling new.
5. Increment the variable leak_counter as many times as new was called in the step five.
6. Make the copies of non-pointer data.
7. Return the current object (*this).

In classes, which have pointers as data members, writing a destructor becomes extremely important. Recall that C++ provided default destructor would not delete the memory dynamically allocated to the class pointers, causing a memory leak. Once we provide our own destructor, the C++ calls it when the objects need to be de-constructed or destroyed. The destructor simply calls the delete operator on the members on which new were called in the constructor. It is important to make the destructor virtual as it facilitates the object identification using C++ typeid operator for the derived classes by turning on, what is called RTTI (run time type identification). This brings us to something called Rule of Three.

Rule of three

For all your classes, which have a pointer data member, always provide the following three!

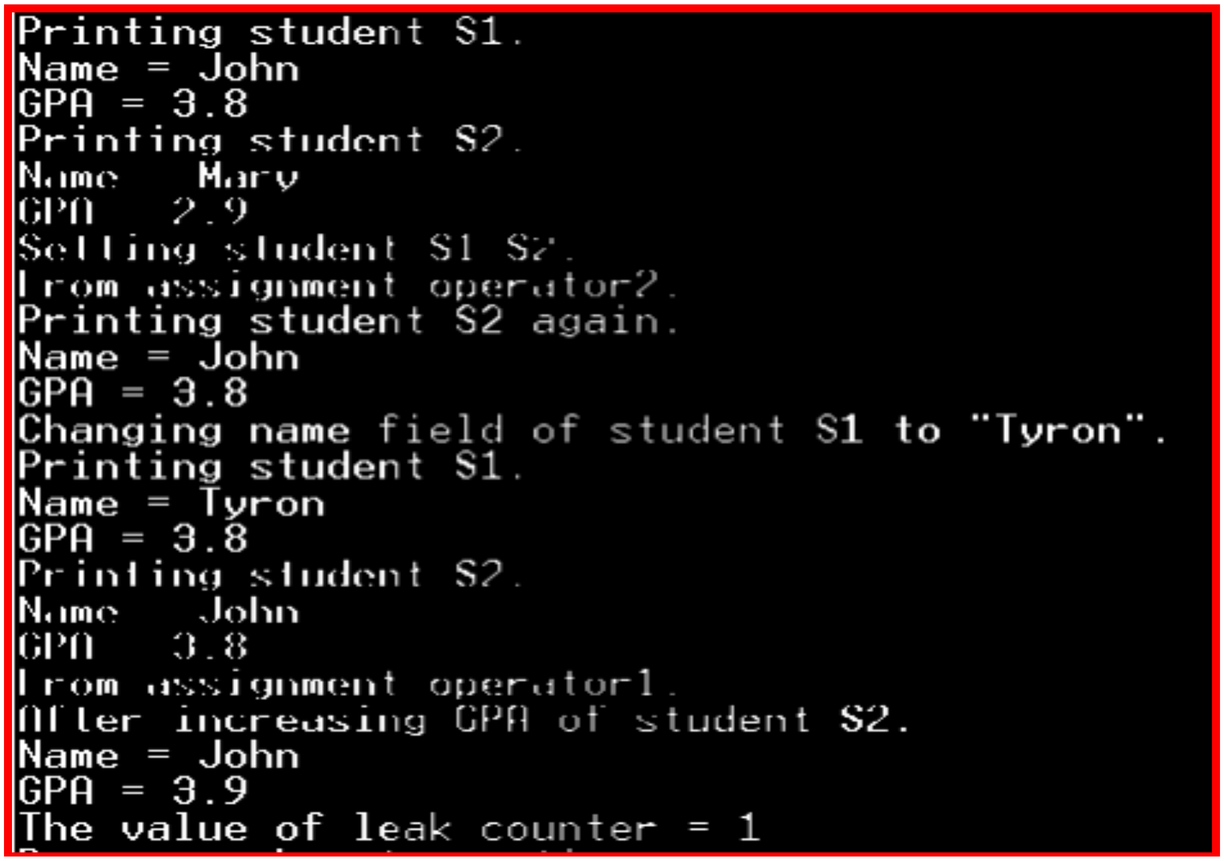
1. A copy constructor
2. An assignment operator
3. A virtual destructor

Remembering the rule of three will avoid lot of subtle bugs in your C++ programs!

We write an extra static member function increaseGpa, in the class Student in Listing 5.11 to better illustrate the functioning of assignment operator (bubbles 6 & 7). The function increaseGpa takes an object of type Student as an argument by const reference, and returns this object as a const reference after students GPA has been increased by 0.1, with out going over 4.0.

The main function in Listing 5.11 is similar to the one in Listing 5.10, except that we make a call to increaseGpa function and we print the

value of leak_counter. The results of Listing 5.11 are shown in Figure 5.16 below.

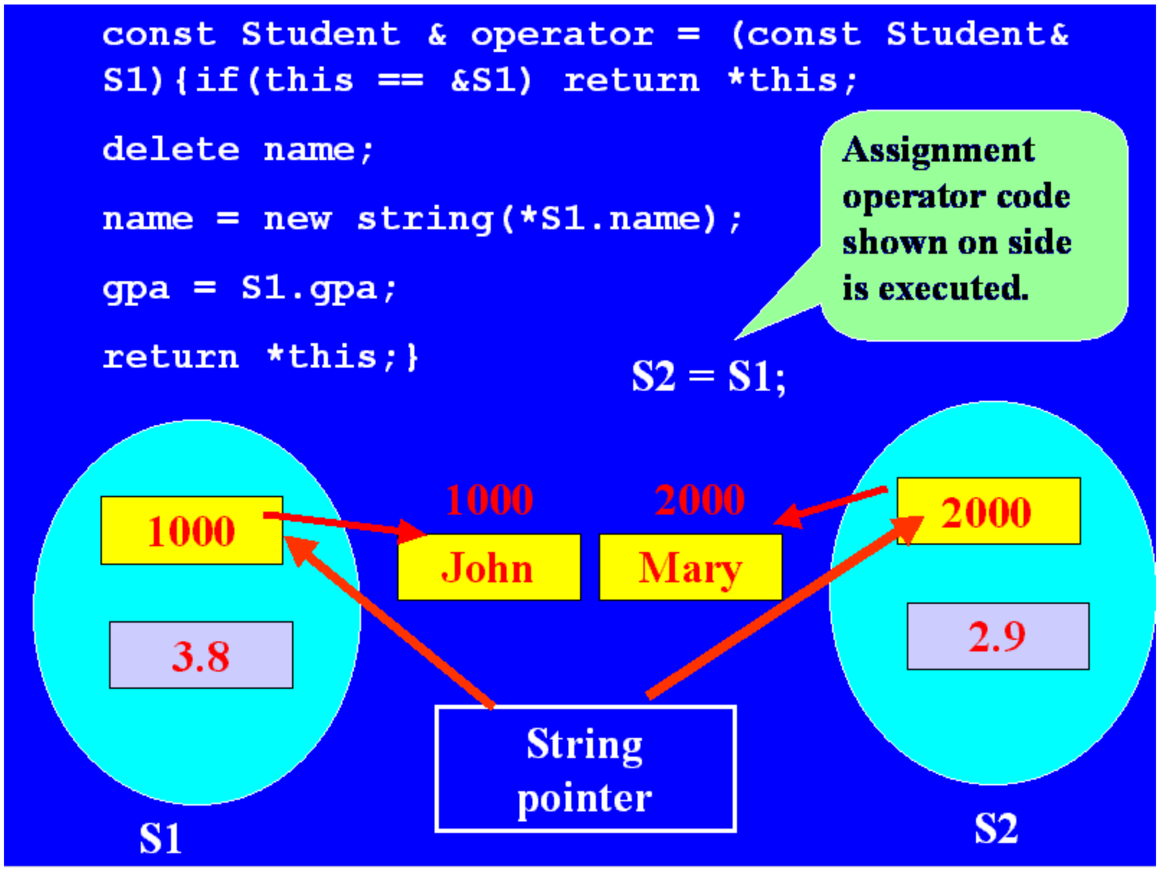


```
Printing student S1.
Name = John
GPA = 3.8
Printing student S2.
Name = Mary
GPA = 2.9
Setting student S1 = S2.
From assignment operator2.
Printing student S2 again.
Name = John
GPA = 3.8
Changing name field of student S1 to "Tyron".
Printing student S1.
Name = Tyron
GPA = 3.8
Printing student S2.
Name = John
GPA = 3.8
From assignment operator1.
After increasing GPA of student S2.
Name = John
GPA = 3.9
The value of leak counter = 1
```

<u>FIG. 5.16</u>

We immediately see that after object S2 has been set equal to object S1, compared to Figure 5.14, changing the name field to "Tyron" in object S1 does not change the name field in object S2, which means S2 is a deep copy of S1. In setting S2=S1, the assignment operator written by us is called, which ascertains that not only the pointers, but also their pointees are copied correctly. We show the following PowerPoint presentation to show as to how the code in assignment operator is executed.

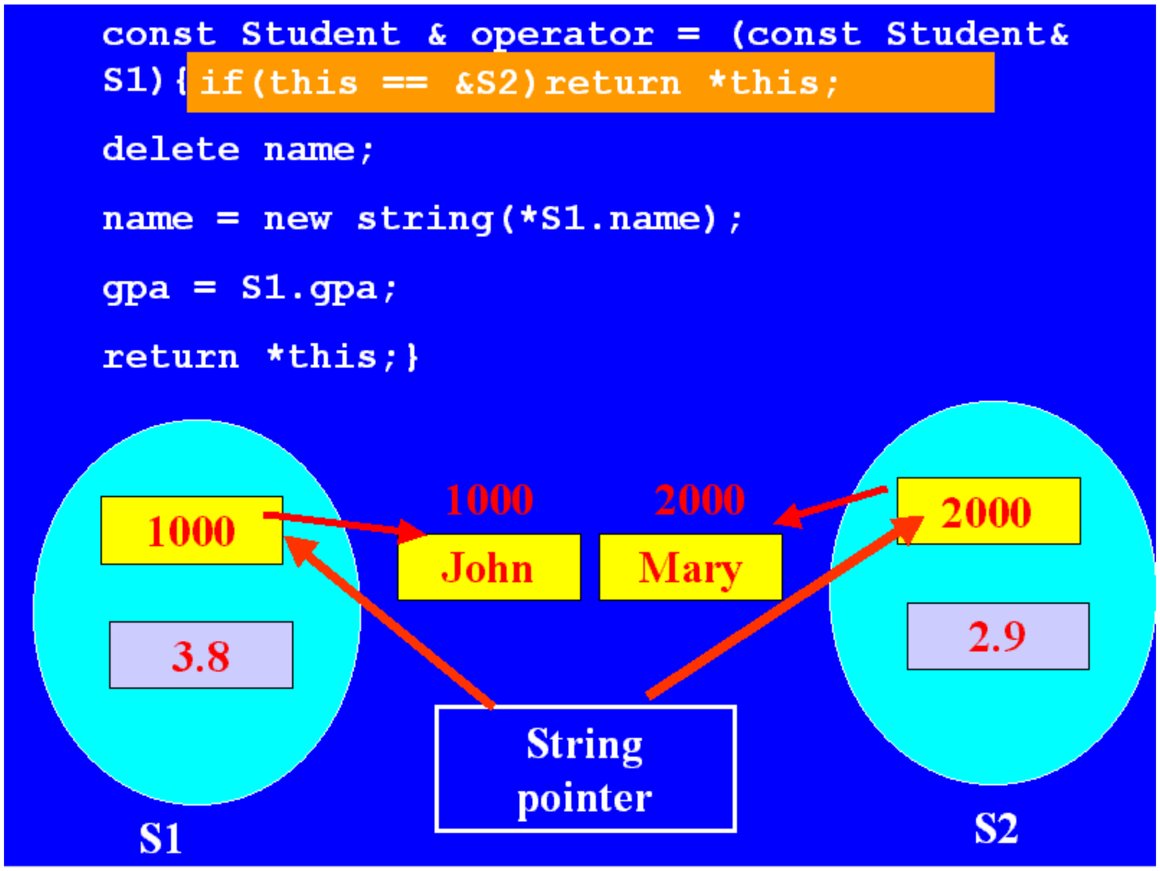
DeepCopy.ppt



```
const Student & operator = (const Student&
S1){if(this == &S1) return *this;

delete name;

name = new string(*S1.name);

gpa = S1.gpa;

return *this;}
```

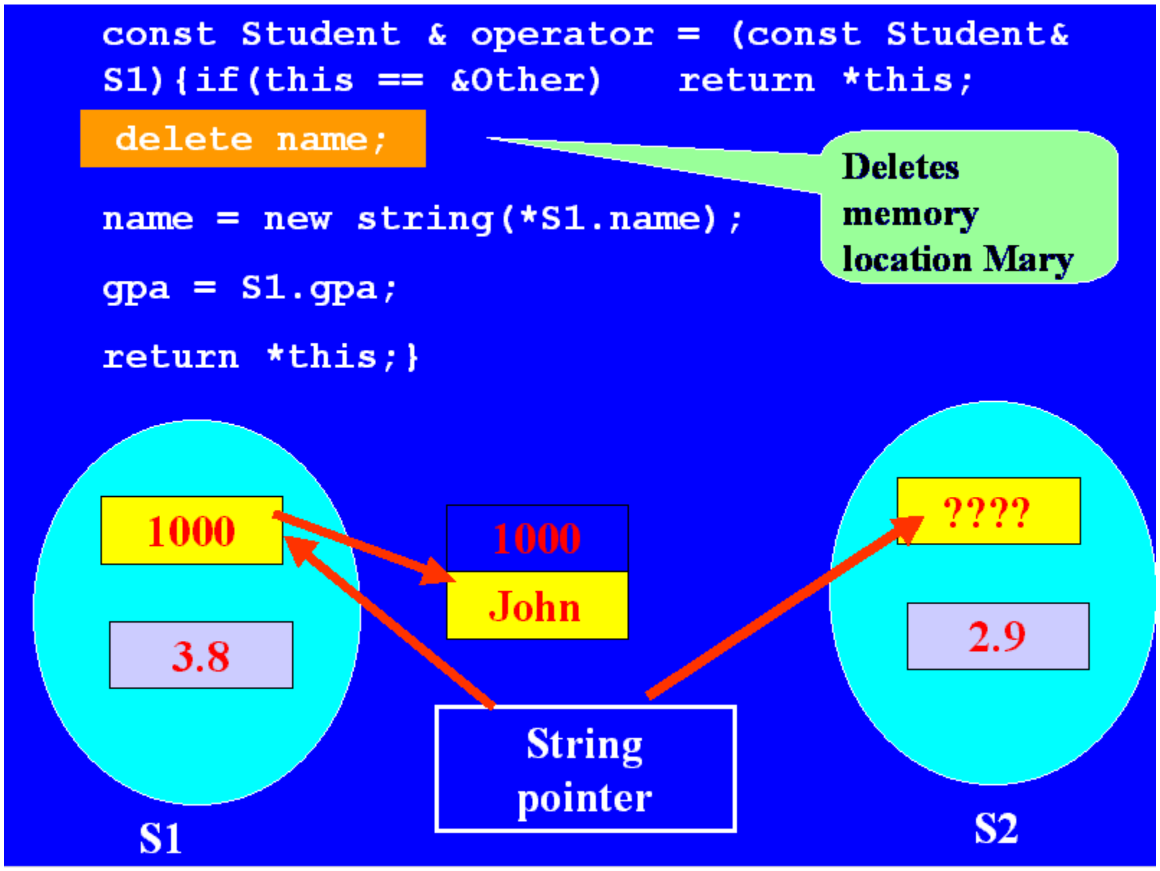
Fig. 5.17A

As soon as we set S2=S1 the code in the assignment operator is executed (Figure 5.17A). This means that object S2 should copy all fields from S1.



```
const Student & operator = (const Student&
S1){if(this == &S2)return *this;
delete name;
name = new string(*S1.name);
gpa = S1.gpa;
return *this;}
```

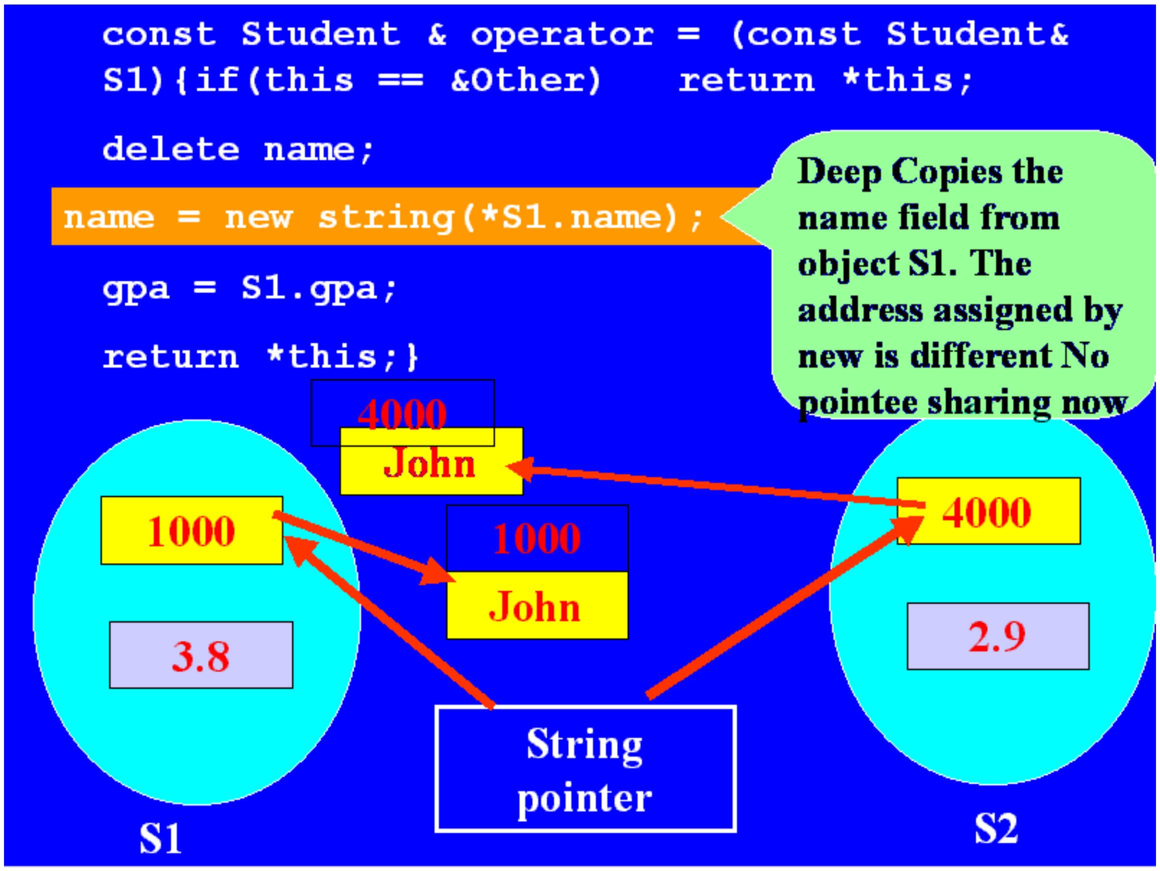
<u>Fig. 5.17B</u>
Understand that call S2=S1 is resolved as:
S2 = S2.operator = (S1);
Therefore the object S1 is passed as the function argument. Upon entering the function for assignment operator, for S2=S1, the fist line (Figure 5.17B) is not executed because S2 is not S1 in this case. Therefore control moves to next line.



```
const Student & operator = (const Student&
S1){if(this == &Other)   return *this;
delete name;
name = new string(*S1.name);
gpa = S1.gpa;
return *this;}
```

<u>Fig. 5.17C</u>

The statement
delete name;
deletes the memory allocated to string "Mary"(Figure 5.17C). Program execution proceeds to next statement.

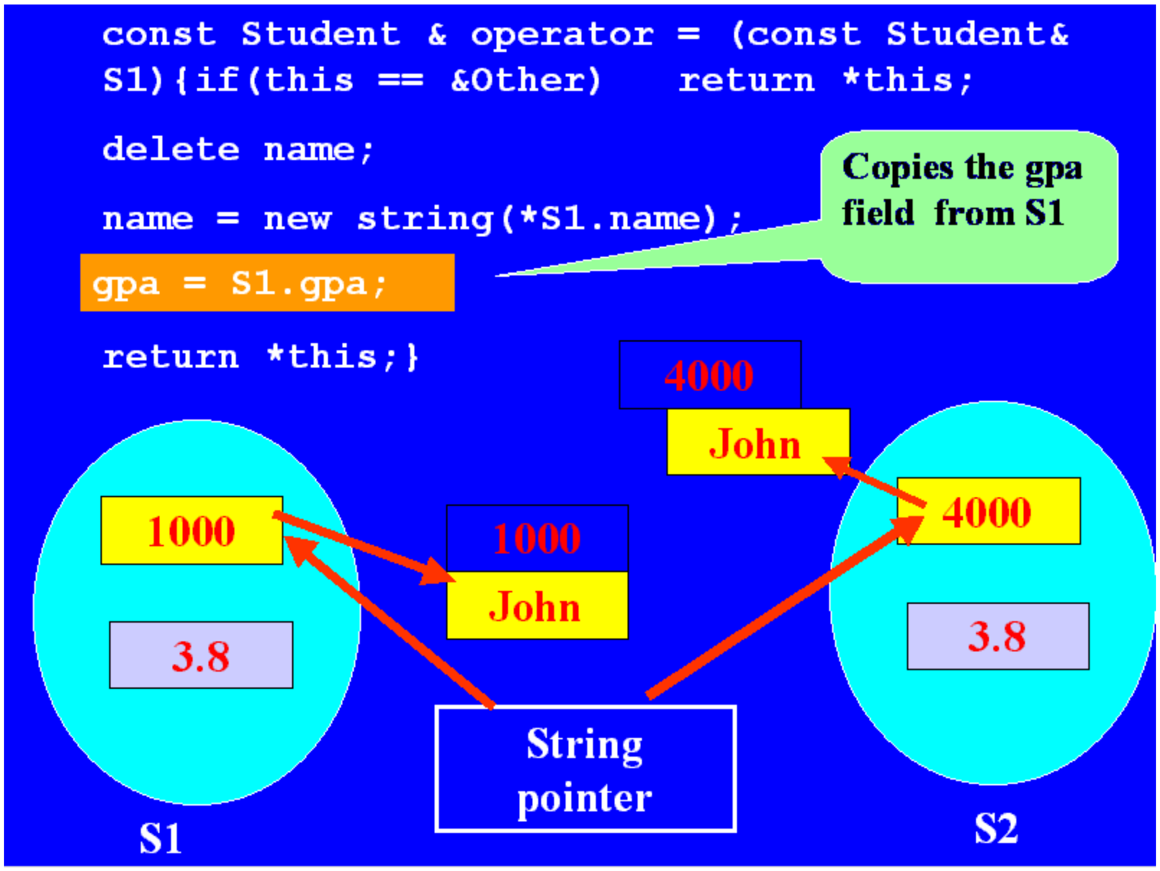


```
const Student & operator = (const Student&
S1){if(this == &Other)   return *this;
delete name;
name = new string(*S1.name);
gpa = S1.gpa;
return *this;}
```

Deep Copies the name field from object S1. The address assigned by new is different No pointee sharing now
4000
John
1000
3.8
1000
John
4000
2.9
String pointer
S1
S2

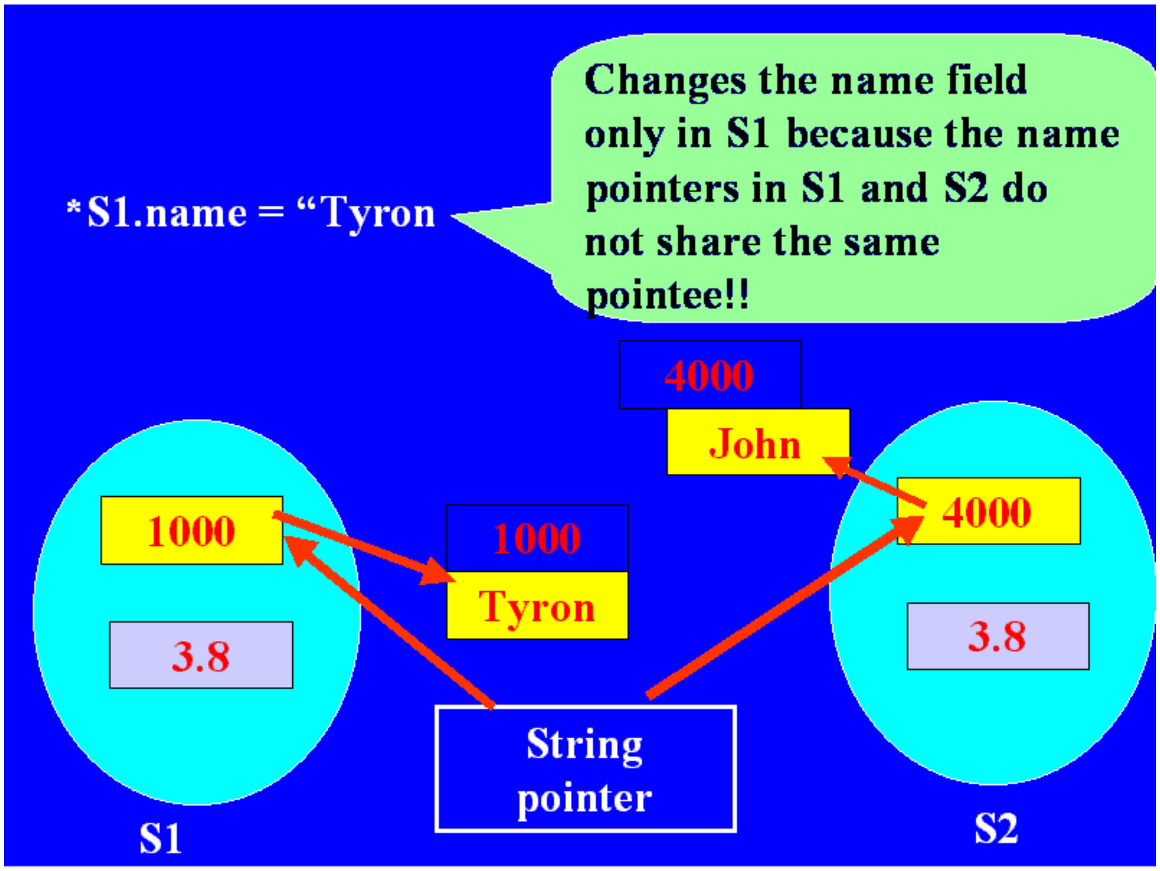
<u>**Fig. 5.17D**</u>

The program statement
name = new string (*S1.name);
creates a new memory location (assume 4000) and copies the name pointed to by pointer in S1 (John) (Figure 5.17D). The name string attached to two objects is same content wise, but two distinct copies of name John exist. There is no sharing of pointee by the pointers in S1 and S2. This is called Deep Copying. In Deep copy, each pointer has its own pointee and no pointee sharing is allowed. The control of program then moves to next statement.



```
const Student & operator = (const Student&
S1){if(this == &Other)   return *this;
delete name;
name = new string(*S1.name);
gpa = S1.gpa;
return *this;}
```

Fig. 5.17E
Now the gpa field from S1 is copied into S2, which is a simple member-wise copy (5.17E). The assignment is complete when the return statement is executed.


*S1.name = "Tyron
Changes the name field only in S1 because the name pointers in S1 and S2 do not share the same pointee!!
4000
John
1000
3.8
1000
Tyron
4000
3.8
String pointer
S1
S2

<u>Fig. 5.17F</u>

Now if the statement such as
***S1.name = "Tyron";**
is executed, it only alters the name attached to object S1. Since S2 is no longer sharing the pointee with S1, the name string attached to it remains unaffected.

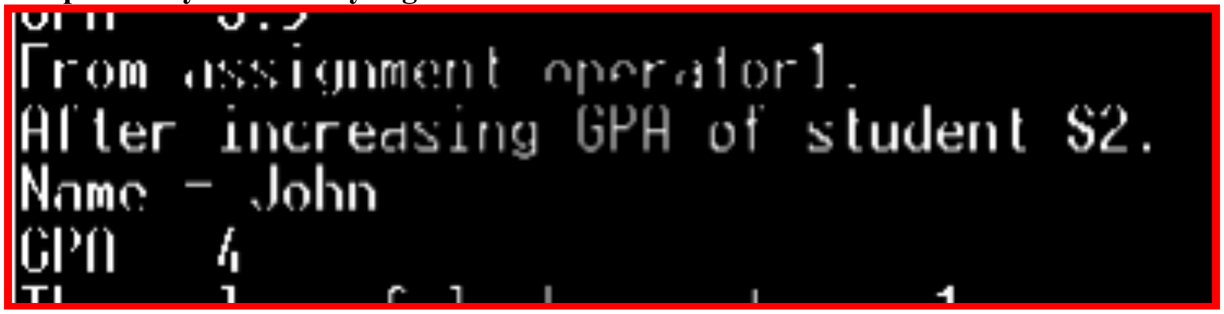
<u>Call to Copy constructor vs to assignment operator</u>

We show the workings of function increaseGpa to illustrate as to when the call is made to copy constructor and when to the assignment operator. The bubble # 6 in Listing 5.11 summarizes this. The overloaded assignment operator is called if the returned object is assigned to itself. For example, that will result in a call such as:
S2 = Student::increaseGpa(S2);
On the other hand the copy constructor is called if return value from the function increaseGpa is assigned to object other than itself. For example in call such as below the assignment operator is not called. Rather the copy constructor alone is called.
S3 = Student::increaseGpa(S2);

We write two small functions in Listing 5.11 to illustrate when the calls to copy constructor or to the assignment operator are made. The body of two functions is shown below.

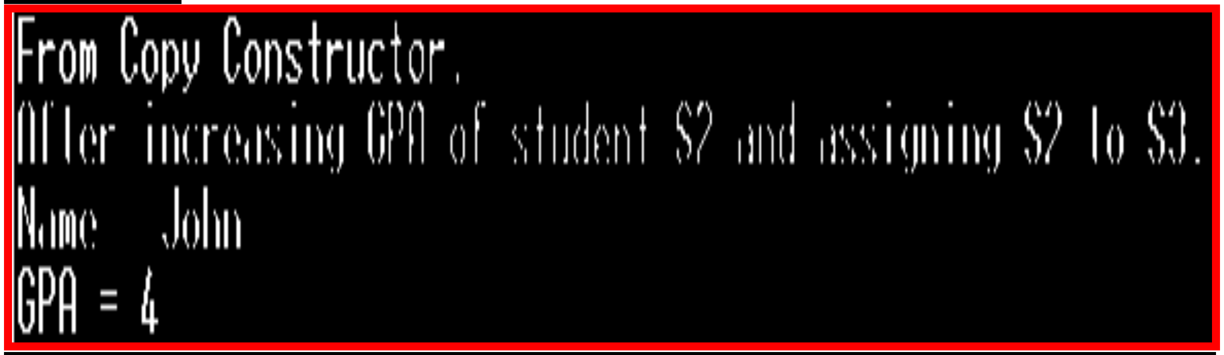
```
void assignToSelf(Student& S2)
{
      S2 = Student::increaseGpa(S2);
      cout<<"After increasing GPA of student S2.\n";
      S2.printStudent();
}

void assignToOther(Student& S2)
{
        Student S3 = Student::increaseGpa(S2);
        cout<<"After increasing GPA of student S2 and assigning "
<<"S2 to S3.\n";
       S3.printStudent();
}
```

In function assignToSelf, the object passed as a function argument is passed to the function increaseGpa, but the return value is assigned to the object itself. On the other hand in the function assignToOther, the return value from the function increaseGpa is assigned to a new object. The output after invoking each function independently is shown by Figures 5.18A and B.



```
From assignment operator1.
After increasing GPA of student S2.
Name = John
GPA = 4
```

FIG. 5.18A



```
From Copy Constructor.
After increasing GPA of student S2 and assigning S2 to S3.
Name = John
GPA = 4
```

FIG. 5.18B

Notice that the self-assignment expressions involving the return value from the function increaseGpa call the assignment operator, but the corresponding call where a different object is on the left hand side of function call expression, invokes the copy constructor. It is as if in the function assignToOther the call to increaseGpa is being made with the syntax given by either of the two statements below.

Student S3 = Student (Student::increaseGpa(S2));
Student S3(Student::increaseGpa(S2));
As a matter of fact changing the code as above does not affect the output at all. Now this may be a bit confusing because earlier we had shown that when an expression such as
S2 = S1;
is executed, both the copy constructor and the assignment operator are called. Understand that when a function takes an object as an argument and its return value is assigned to same object, that task cannot be done by the copy constructor because constructors do not return anything. Therefore by default the assignment operator is called. But when the return value is assigned to another object, the copy constructor is adequate to do the job, as it can make member-wise or deep copies depending upon how it is written. For example a call to copy constructor, made in the following two ways is identical and the second way does not invoke the assignment operator.
Student S3 (S2);
Student S3 = Student (S2);
This is in concordance with our earlier emphasis that constructors can be invoked on the right hand side of an expression, even though they do not return anything. In case of function assignToOther the call to increaseGpa does not invoke assignment operator either.

Summary of what C++ provides for its classes

C++ provides the following five hidden members for all its classes, structs, and unions.

1. **A default constructor**
2. **A default copy constructor**
3. **A default destructor**
4. **A default assignment operator**
5. **A this pointer**

The last one is not modifiable by the user of the class, unless user overloads the star (*) de-reference operator for the class so that *this will have some other meaning. The user can write versions of the first four as they choose. However the rule of thumb is that there is no need to write the copy constructor and assignment operators unless, one of the data members (either directly or by inheritance) is pointer type. For such classes the C++ provided copy constructor and assignment operators work fine. However, we emphasize that it is always important to provide a virtual destructor for every class, as that becomes the basis for run time type identification of objects of a class and of the classes derived from it by using typeid operator. Such type identification at run time allows one to write polymorphic functions, which can implement generic algorithms for a class and its descendents. (Templates are another mechanism to accomplish the same goal). Therefore, we summarize our recommendations as follows:

1. **Always provide your own virtual destructor.**

2. **Follow the rule of three. Provide the destructor, copy constructor, and the overloaded assignment operator when one of the class members is of type pointer.**
3. **Always provide a default constructor or the one which acts as default and explicit by virtue of carrying the default arguments. This rule may be relaxed in some situations.**
4. **Use the member initialization list whenever possible.**
5. **Only return the reference to objects passed to the function as one of the arguments. Violating this recommendation may either cause a memory leak or in a code that will have unpredictable results.**
6. **Use a leak counter to make sure that new and delete have equal number of invocations.**

Finally we show an example of a class NameList, which uses a pointer to store names such that the list can grow dynamically as names are added to it. This model of list will be emulated further to write list classes with improved functionality.

Pointers as Class data members and List class

Classes may have dynamic arrays like the one we discussed earlier as class members. This is useful when the class needs to maintain a list of objects and the list length may change from one invocation of the application software to another. We summarize the details of a class NameList, which maintains a list of names in a data member, which is a dynamic array of strings[4].

Class NameList will store the names read from keyboard/data file, and be able to print the list to screen/file. The list may be searched for a name and sorted in alphabetical or reverse alphabetical order. The names can be added or deleted from the list using simple operators such as plus (+) or minus (-). The list will grow dynamically as data are added to it. Two or more lists can be merged into one list.

Design Details

Our design has to have following components:

1. **Class data member design.**
2. **Pointer data necessitated member functions.**
3. **Functions relating to**
    a. **Adding members to list**
    b. **Searching the list**

[4] **Almost all lists can be built similar to this model.**

c. Merging two or more lists into one
d. Deleting a name from the list
e. Sorting the list in alphabetically or reverse alphabetically.
f. Printing the list to a file or to console.

We discuss the design details for each component.

Data Member design

The Table 5.4 presents the design of data members of class NameList.

enum	{ MAX = 50 } *Default length of the List member of class NameList.*
enum	{ GROW = 50 } *Default number by which the array will grow, once filled.*
string *	List *Pointer to the first member of the List as the list is a dynamic array of strings.* C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListr0
int	count *Number of elements in the List.* C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListr1
int	growby *The size by which the list will grow , once it is filled and more elements are added.* C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListr2
int	size *Current physical size of the List.* C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListr3

Table 5.4

The class needs to carry two constants. One is the default length of the array and other the default length by which the array will grow, each time it is filled. Since it is somewhat cumbersome to define constants inside the class and for the sake of integrity of encapsulation, we would like to carry the constants as class members, a technique to carry initialized class constants is needed. For integer data type, the enum provides this technique. A declaration such as

```
enum {MAX = 50};
```

allows us to define an int constant MAX and initialize it to a value 50. We define the default size by which the filled array must grow, GROW, the same way. Client of class NameList does not need both of these class members. The declaration such as

```
enum {GROW = 50};
```

is possible because in C++ enum is considered a symbolic name for int data type. Understand that declaring array length in this manner, limits us from declaring arrays that would be bigger than the upper limit of int data type. Apart from two constants, MAX and GROW, we need to carry the following size related parameters for our dynamic array: size, which is the actual physical size of the array, growby, the number by which array will grow, once it is filled and more elements are added, and count, which gives us the current number of elements in the list. The data member size is needed because in C/C++ arrays do not remember their lengths. Finally the pointer data member List is a pointer to a string array, where each member in the array will store a string (preferably a name). This model can be used to write a list class of any type if we just replace the string pointer to the pointer ItemType, latter being a generic object used in earlier discussions of stack and queues.

Design details related to pointer data member

Since our class NameList has a string pointer to store name, we need to follow the rule of three discussed earlier in this chapter. Therefore we would need to write a copy constructor, overload an assignment operator, and provide a virtual destructor. Very often the code in copy constructor and assignment operator is similar. Therefore we write a helper function called copy to condense the code to be used in copy constructor and assignment operator into the function copy. C++ software designers' advise to not make explicit calls to destructor inside member functions or elsewhere. Yet, at times (for example inside the assignment operator) we first need to delete the dynamically allocated memory in the existing object, before we can copy fields of source object into it. Therefore we write a helper function called destroy, which de-allocates the dynamically allocated memory, and we call this function inside the destructor. In designing constructor, it is important that client should not have to pass an argument to initialize the pointer data member of the class. This is so that client does not have to perform the delicate task of allocating and de-allocating memory. (One must accomplish that even at the risk of writing slightly bulky programs). Therefore we do not provide a constructor, where client will need to initialize the pointer data member of the class – the List. The design philosophy here is that pointer manipulation is better left to the designer of the class, rather than on its client. Table 5.5 shows the summary of class constructors and destructor.

	NameList (int growby1=GROW) ***Default/Explicit constructor creates a string array of length MAX.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProct5_8html" l - NameLista0
virtual	~NameList ( ) ***Destroys the dynamically allocated memory assigned by new in the constructor.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista1
	NameList (const NameList &NameList1) ***Copy constructor: Deep copies the object NameList1 into the caller object.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista2
	NameList (string name) ***Constructor, which facilitates the name to be added to the list by using the plus (+) operator.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista4

Required by rule of three

Table 5.5

Design of Utility and other member Functions

A list of any kind needs the functionality described above, such as searching, merging, sorting, printing, etc. The Table 5.6 gives the summary of member functions, which do some of that. The table also includes the summary of assignment operator required by rule of three.

Assignment operator, required by rule of three

const NameList &	operator= (const NameList &NL) ***Overloaded Equal operator, which makes the assignment from source list to destination list.*** **C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista3**

void	add (string Name) ***Adds the new name to the NameList.*** **C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista5**
void	getList (istream &in, int flag) ***Enters the data to List from a file or from keyboard.*** **C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista6**
int	getCount ( ) const ***returns the current number of elements in the NameList.*** **C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista7**
int	getSize () const ***Returns the current physical size of list.*** **C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista8**
bool	find (string name, int &index) const ***Returns true if finds the name in the list , else returns false. Also returns the first index in the array where the name is found. Returns a negative index if name is not found.*****C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista9**
void	sort (int flag) ***Sorts the list in alphabetical order or reverse order based on the value of the flag passed to the function sort. Performs case-insensitive sorting.*****C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameLista10**

Table 5.6

In the Table 5.6, the function getList deserves a special discussion. If we simply overload the extraction operator >>, the problem remains of deciphering as how to read data from the keyboard, because that needs user prompt. Therefore the member function getList takes a flag and an istream object, such that the flag value determines whether the data are entered from the keyboard or read from a file. Friend function overloaded operator >> is called from with in getList. Client can also call it directly. Table 5.7 gives summary of protected member functions.

Protected Member Functions

void	destroyList ( ) ***Destroy is called internally either by the destructor or by assignment operator to destroy the list.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListb0
void	copy (const NameList &NameList1) ***copy is called internally to copy the List as argument to current list.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListb1
Table 5.7	

The reason for making the copy and destroy functions protected is because the client would not need to use them. Table 5.8 describes the friend and helper member functions.

ostream &	operator<< (ostream &out, const NameList &NameList1) ***Prints the NameList to user defined output media.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListn0
istream &	operator>> (istream &in, NameList &NameList1) ***Overloaded extraction operator is used when data are read from the file automatically.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListn1

const NameList	operator+ (const NameList &NL1, const NameList &NL2) ***Merges two NameList objects and returns the combined List.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListn2
const NameList	operator- (NameList &NL, const string name) ***Removes the first occurrence from lower index of name string from the list if found.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListn3
const NameList	operator- (const string name, NameList &NL) ***Removes the first occurrence from lower index of name string from the list if found.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListn4
bool	checkNullRef (const void *ptr) ***Function checks whether the reference or pointer passed to it is a NULL or not.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListn5
int	compare (const void *ptr1, const void *ptr2) ***Function sort uses compare when alphabetical sorting is needed.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListn6
int	compare1 (const void *ptr1, const void *ptr2) ***Function sort uses Compare1 when reverse alphabetical order sorting is needed.*** C:\My Documents\CoursesCS2LectureNotesTopic05PointerApplicationsProject5_8html" l - NameListn7

Table 5.8

The client can use the insertion operator << directly to output the list to a file or standard output. The extraction operator will work to read data from a file

directly, but getList can be used for data input from the keyboard or file both. The overloaded plus (+) operator will merge two or more lists or add a single string to the list. The automatic type conversion from string to type NameList is also performed. The overloaded minus (-) operator can delete the first occurrence of a name or string from the list, if it found. The function chekNullRef is used internally to filter out any null reference or pointer passed to any member of friend function of class NameList. The functions compare and cmpare1 are used internally by the sort function, which calls qsort. The data files for NameList.h and NameList.cpp are shown in Listing 5.8 below[5].

```
#ifndef NAMELIST_H
#define NAMELIST_H
#include "counter.h"
#include <iostream>
#include <fstream>
#include <string>
#include <typeinfo>
using namespace std;

class NameList
{
private:
        enum {MAX = 50};
        enum {GROW = 50};
        string *  List;
        int count;
        int growby;
        int size;
public:
        NameList(int len = MAX, int growby1=GROW );

        virtual ~NameList();
        NameList(const NameList& NameList1);
        const NameList& operator = (const NameList& NL);
        NameList(string name);
         void add(string Name);
        friend ostream& operator <<(ostream& out, const NameList& NameList1);
        void getList(istream& in, int flag);
        friend istream& operator>>(istream& in, NameList& NameList1);
        friend const NameList  operator +(const NameList& NL1, const NameList& NL2);
```

[5] **Our listing numbers are out of order – a flaw to be fixed in next version of this topic.**

```cpp
        int getCount()const;
        int getSize() const;
        friend const NameList operator - ( NameList& NL, const string name);
        friend const NameList operator - (const string name, NameList& NL);
        bool find(string name, int& index)const;
        void sort(int flag);
  friend bool checkNullRef(const void * ptr);
  friend int compare(const void * ptr1, const void * ptr2);
 friend int compare1(const void * ptr1, const void * ptr2);
protected:
        void destroyList();
        void copy(const NameList& NameList1);
};

#endif
```

//Listing 5.8A

```cpp
#include "NameList.h"
#include <cassert>
#include <cstdlib>
bool checkNullRef(const void * ptr)
{
        if(ptr == NULL)
                return true;
        else
                return false;
}

int compare(const void * ptr1, const void * ptr2)
{
        string str1 = *(string*)ptr1;
        string str2 = *(string*)ptr2;
        string str3="";

        for(int index = 0; index<str1.length(); index++)
                str3+=tolower(str1[index]);
        str1 = str3;
        str3 = "";
        for( index = 0; index<str2.length(); index++)
                str3+=tolower(str2[index]);
        str2 = str3;
        if(str1 == str2)
                return 0;
        else if(str1>str2)
                return 1;
        else
                return -1;
}
int compare1(const void * ptr1, const void * ptr2)
{
        string str1 = *(string*)ptr1;
        string str2 = *(string*)ptr2;
        string str3="";

        for(int index = 0; index<str1.length(); index++)
                str3+=tolower(str1[index]);
        str1 = str3;
        str3 = "";
        for( index = 0; index<str2.length(); index++)
                str3+=tolower(str2[index]);
        str2 = str3;

        if(str1 == str2)
                return 0;
        else if(str1>str2)
                return -1;
        else
                return 1;
}

NameList::NameList(int len, int growby1):growby(growby1)
{
     if(len<=0 || growby1<=0)
     {
cerr<<"Zero or negative length array length or growth size "
<<" is not created. Default parameters used.\n";
      List = new string[MAX];
      count = 0;
      size = MAX;
      this->growby = this->GROW;
   }
   else
  {
      List = new string[len];
      count = 0;
  }
      mem_couter++;
}
```

```
void NameList::destroyList()
{
        delete [ ] List;
        mem_couter--;
}
NameList::~NameList()
{
        destroyList();
}
NameList::NameList(const NameList& NameList1)
{
        assert (this != &NameList1);
        assert(NameList1.size>=NameList1.count);

        if(checkNullRef(&NameList1))
        {
                cerr<<"Null reference passed.\n";
                exit(1);
        }
        copy(NameList1);
}
void NameList::copy(const NameList& NameList1)
{
        this->List = new string[NameList1.size];
        mem_couter++;
        for(int index=0; index<NameList1.count; index++)
                this->List[index] = NameList1.List[index];
        this->count = NameList1.count;
        this->growby = NameList1.growby;
        this->size = NameList1.size;
}
const NameList& NameList::operator = (const NameList&
NameList1)
{
        if(checkNullRef(&NameList1))
        {
                cerr<<"Null reference passed.\n";
                exit(1);
        }
        if(this == &NameList1)
                return *this;

        this->destroyList();
        copy(NameList1);
        return *this;
```

```
}
void NameList::add(string Name)
{
        if(Name.length() !=0)
        {
                if(count<size)
                {
                        List[count++] = Name;
                }
                else
                {
                        int len = count+growby;
                        string * Temp = new string[len];
                        mem_couter++;
             for(int index = 0; index<this->count; index++)
                                Temp[index] = List[index];
                        delete []List;
                        mem_couter--;
                        List = Temp;
                        Temp = NULL;
                        size = len;
                        List[count++] = Name;
                }
        }
        else
                        cerr<<"Zero length name entered.\n";
}
ostream& operator <<(ostream& out, const NameList& NameList1)
{
        if(checkNullRef(&NameList1))
        {
                cerr<<"Null reference passed.\n";
                exit(1);
        }

        for(int index = 0; index<NameList1.count; index++)
        {
                out<<NameList1.List[index]<<"    ";
                if(index%7 == 0)
                        out<<endl;
        }
        out<<endl;
        return out;
```

```
}
void NameList::getList(istream& in, int flag)
{
        if(flag==1)
        {
                bool done = false;

                while(!done)
                {
                        string temp ="";
       cout<<"Please enter the name to be added to list: ";
                        in>>temp;
                        this->add(temp);
cout<<"More data? enter zero to continue and 1 to stop : ";
                        in>>done;
                }
        }
        else if(flag==2)
        {
                in>>*this;
        }
        else
                cerr<<"Illegal flag value entered.\n";
}
istream& operator>>(istream& in, NameList& NameList1)
{
   if(checkNullRef(&NameList1))
   {
         cerr<<"Null reference passed.\n";
         exit(1);
   }
   string str="";
   in.peek();
   while(!in.eof())
   {
         in>>str;
         NameList1.add(str);
   }
     return in;
}
NameList::NameList(string name)
{
        List = new string[MAX];
        List[0] = name;
        count = 1;
        size = MAX;
        growby = 50;
        mem_couter++;
}
const NameList  operator +(const NameList& NL1, const
NameList& NL2)
{
        if(checkNullRef(&NL1) || checkNullRef(&NL2))
        {
                cerr<<"Null reference passed.\n";
                exit(1);
        }

        NameList Temp;
        for(int index=0; index<NL2.count; index++)
        {
                Temp.add(NL2.List[index]);
        }

        for( index=0; index<NL1.count; index++)
        {
                Temp.add(NL1.List[index]);
        }
        return Temp;
}
int NameList::getCount()const
{
        return this->count;
}
int NameList::getSize() const
{
        return this->size;
}
bool NameList::find(string name, int& index)const
{
        for( index=0; index<this->count; index++)
        {
                if(name==this->List[index])
                {
                        return true;
                }
        }
        index = -1;
        return false;
}

```

```
const NameList operator - ( NameList& NL, const string name)
{
        if(name.length() == 0)
          {
   cerr<<"Zero length name cannot be in List.\n";
                    return NL;
        }

        if(checkNullRef(&NL))
          {
               cerr<<"Null reference passed.\n";
                    exit(1);
          }

        int index = 0;
        if(!NL.find(name, index))
          {
    cerr<<"Name " << name << " not found in list.\n";
                    return NL;
          }
      for(int j=index; j<NL.count-1; j++)
      {
             NL.List[j] = NL.List[j+1];
      }
          NL.count = NL.count - 1;

        return NL;
}
```

1. If name is a string of zero length return the original list.

2. If list passed is a NULL reference then exit program.

3. If the string name is not found in the list then return the original list.

4. If find returns true then run a for loop from index to count-1 to shift all elements from index to count-1 to remove the element at index.

```
const NameList operator - (  const string name, NameList& NL)
{
        if(name.length() == 0)
        {
         cerr<<"Zero length name cannot be in List.\n";
                return NL;
        }

        if(checkNullRef(&NL))
        {
                cerr<<"Null reference passed.\n";
                exit(1);
        }

        NL = NL - name;
        return NL;
}

void NameList::sort(int flag)
{
        if(flag == 1)
        {
       qsort(List, this->count,sizeof(List[0]),compare);
        }
        else if(flag == 2)
        {
         qsort(List, this->count,sizeof(List[0]),compare1);
        }
}
```

//Listing 5.8B

```
#include "NameList.h"


int main()
{
        {
                NameList NL;
                ifstream in;
                in.open("Name.txt");
                NL.getList(in,2);
                cout<<NL<<endl;
       cout<<"The size of List = "<<NL.getSize()<<endl;
   cout<<"Number of names in List ="<<NL.getCount()<<endl;
                NL = string("Zack")+ NL;
                cout<<NL<<endl;
      cout<<"The size of List = "<<NL.getSize()<<endl;
    cout<<"Number of names in List ="<<NL.getCount()<<endl;
                NL =  NL+ string("Zelda");
                cout<<NL<<endl;
        cout<<"The size of List = "<<NL.getSize()<<endl;
    cout<<"Number of names in List ="<<NL.getCount()<<endl;
                in.close();
                NameList NL2;
                ifstream in1;
```

```
                      in1.open("Name3.txt");
                      if(!in1)
                      {
                              cerr<<"No file found.\n";
                      }
                      NL2.getList(in1,2);
                      NL = NL2+NL;
                      cout<<NL<<endl;
               cout<<"The size of List = "<<NL.getSize()<<endl;
          cout<<"Number of names in List ="<<NL.getCount()<<endl;
                      in.close();

                      NL = NL - string("Ahmed");
                      cout<<NL<<endl;
            cout<<"The size of List = "<<NL.getSize()<<endl;
         cout<<"Number of names in List ="<<NL.getCount()<<endl;
                      NL.sort(1);
                      cin.get();
                 cout<<"Printing the sorted list of names.\n";
                      cout<<NL<<endl;
                      cin.get();
                      NL.sort(2);
                      cin.get();
cout<<"Printing the sorted list of names in reverse order.\n";
                      cout<<NL<<endl;
                      cin.get();

                      NL = NL - string(",,,,,,,,");
              }
        cout<<"The value of memory counter = "<<mem_couter<<endl;

              return 0;
}//Listing 5.8C
```

Overloaded operator minus (-)

In file NameList.cpp we focus our attention on overloaded operator minus (-) which is used to remove the first occurrence of a string from the list. The algorithm for the function is as follows:

- **If attempt is made to remove a zero length string or the string to be removed is not in the list, then print appropriate message and return the original list. (Bubbles #1 and 3).**
- **If the list passed is a null reference then exit the program. (Bubble #2).**

- If the string name is found then run a for loop from the location index to the count-1, to shift each array element up by one and then adjust count downwards by one. (Bubble #4).

The operator plus (+) is used only to merge lists. Additional forms of operator plus (+) can be overloaded to add a string to list using it. For now we expect the client to invoke the add function to add a single string to the list.

Member function add

The function add, adds a single string to the list. If a zero length string is not passed to the add and the count is less than the size of list then the string name is added to the location count and count is incremented by one (Line 128, Listing 5.8B). But if count is equal to list size then the list is full, and must be grown in size (lines 132 to 142). The following algorithm accomplishes the addition of a string in that case.

1. Set the new length of the list = count + growby (line 132).
2. Create a temporary array of new size in step one (line 133).
3. Increase the memory allocation/de-allocation counter by one (line 134).
4. Copy the elements from array pointer List into the array Temp created in step 2(Line 135-136).
5. Delete the current List and decrease the memory counter by one. (Lines 137-138).
6. Set the pointer List to point to the pointee of Temp and set Temp to NULL (Lines 139-140).
7. Set the current size of NameList to the length set in step 1 and add the Name string to the list and increase the count by one (Lines 141-142).
8.

Merging two or more lists: Overloaded operator plus (+)

Of three overloaded operators plus (+), the one which merges the two or more lists together and returns a master list has the following proto-type: (Lines 219-238).

```
const NameList  operator +(const NameList& NL1,
const NameList& NL2)
```

The algorithm is as follows:

1. If either list reference is a null reference then exit (Lines 221-225).
2. Declare a NameList object Temp. (Line 227).
3. Add the all members of one of the List (NL2), member-by-member to the list created in step 2(Lines 228-231).

4. Repeat step3, once for the other list (NL1) (Lines 233-236).
5. Return the list created in step 2. (Line 237).

Helper Function compare

The function compare is used by qsort inside the member function sort when the List is to be sorted in alphabetical order. The proto-type of compare is given below.

```
int compare(const void * ptr1, const void *
ptr2)
```

The compare function (as needed by qsort) takes two constant void pointers as argument and returns an integer. The criterion of return values is as follows:

- If pointees of ptr1 and ptr2 have same contents then return zero.
- Else if pointee of ptr1 is "larger" than the pointee of ptr2, then return one.
- Else return minus one.

Since the incoming arguments to compare are void pointers, they must be cast into the type their pointee is, and then de-referenced to get the contents of their pointee. The mechanism of doing so is exactly the one we discussed for swap function in Listing 5.6. We get the strings attached to the pointers ptr1 and ptr2 (Lines 16-17). The algorithm to convert both strings to lower case, so that case insensitive comparison can be done is as follows:

1. Declare a temporary string (Line 18).
2. Copy string str1 character by character after converting each of latter's characters into lower case, to string declared in step1. (Lines 20-21).
3. Set str1 equal to string declared in step1.
4. Repeat step 1 to 3 once for the string str2 (Lines 24-26).

The if and else comparison algorithms described above is then implemented.

Driver Program

We write the driver program to test some of the member and friend functions of class NameList. When we actually run the program, we can reduce the size of maximum default array length and the size by which it will grow.

- First we create a NameList object NL and use getList function to read the names from file Name.txt and we print the list (Lines 7 to 11, Listing 5.8C). Output is shown in Figure 5.19. The file Name.txt has 17 names. The physical size of the array is 20. Since we set the original length of array to be five and growby parameter to five also, after adding 17 elements we get the physical size of array to be 20.
- Then we add "Zack" to list using the overloaded plus operator using the following syntax: NL = string("Zack")+ NL;

  Then we print this modified list and number of names in it (Lines 14-17). The results show that "Zack is added to the list and number of names increased to 18.

- **Then we add "Zelda" to list using the following syntax:**
  NL = NL+ string("Zelda"); **Printing the modified list shows that "Zelda" is added to the list and size of list is now 19 (Lines 18-21). Notice that because of the syntax we use to add Zelda and corresponding code for overloaded plus (+) operator, Zelda goes to the front of list.**
- **We create a second NameList object NL2, and read the data from file Name3.txt and add data from it to NL2. We then merge NL and NL2, using the following syntax: NL = NL2+NL; (Lines 23-31). We then print the merged list and its number of elements (Lines 32-34). The number of names in list is now 37 and physical size of list is 40.**
- **Then we remove "Ahmed" from the list using the syntax:**
  NL = NL - string("Ahmed"); **Then we print the modified list. (Lines 37-40).**
- **Then we sort the list in alphabetical order (case insensitive) and print it. (Lines 41-44).**
- **We then sort the list in reverse alphabetical order and print it (Lines 46-49).**
- **Finally we try to find a name that does not exist in the list by using the statement: NL = NL - string(",,,,,,,,");** **The result shows that this string is not in the list.**

This model of dynamic array or list can be used for any object or class.

```
Ron
Lamar    Tina    Mary    Joe    Apple    Candy    Ken
Sam    Satish    Nancy    Tyron    Jaime    Titu    Yates
Adam    Maria

The size of List = 20
Number of names in List =17
Ron
Lamar    Tina    Mary    Joe    Apple    Candy    Ken
Sam    Satish    Nancy    Tyron    Jaime    Titu    Yates
Adam    Maria    Zack

The size of List = 20
Number of names in List =18
Zelda
Ron    Lamar    Tina    Mary    Joe    Apple    Candy
Ken    Sam    Satish    Nancy    Tyron    Jaime    Titu
Yates    Adam    Maria    Zack

The size of List = 20
Number of names in List =19
Zelda
Ron    Lamar    Tina    Mary    Joe    Apple    Candy
Ken    Sam    Satish    Nancy    Tyron    Jaime    Titu
Yates    Adam    Maria    Zack    Kisung    Martin    Tania
Bruce    Papel    Ahmed    ellie    Quan    Paul    Joseph
Nanda    Bharti    Jason    Zander    Tomeline    Jill    Christian
Christina

The size of List = 40
Number of names in List =37
Zelda
Ron    Lamar    Tina    Mary    Joe    Apple    Candy
Ken    Sam    Satish    Nancy    Tyron    Jaime    Titu
Yates    Adam    Maria    Zack    Kisung    Martin    Tania
Bruce    Papel    ellie    Quan    Paul    Joseph    Nanda
Bharti    Jason    Zander    Tomeline    Jill    Christian    Christina

The size of List = 40
Number of names in List =36
Printing the sorted list of names.
Adam
Apple    Bharti    Bruce    Candy    Christian    Christina    ellie
Jaime    Jason    Jill    Joe    Joseph    Ken    Kisung
Lamar    Maria    Martin    Mary    Nancy    Nanda    Papel
Paul    Quan    Ron    Sam    Satish    Tania    Tina
Titu    Tomeline    Tyron    Yates    Zack    Zander    Zelda

Printing the sorted list of names in reverse order.
Zelda
Zander    Zack    Yates    Tyron    Tomeline    Titu    Tina
Tania    Satish    Sam    Ron    Quan    Paul    Papel
Nanda    Nancy    Mary    Martin    Maria    Lamar    Kisung
Ken    Joseph    Joe    Jill    Jason    Jaime    ellie
Christina    Christian    Candy    Bruce    Bharti    Apple    Adam
Name ,,,,,,,, not found in list.
The value of memory counter = 0
```

Original list

Zack added to list

Zelda added to List

The above list merged with the list from file Name3.txt

"Ahmed" removed from the list.

FIG. 5.19: The above output is taken with array length of 5 and growby = 5.

Description of C/C++ QSort Function

This part describes how you can use C/C++ library function qsort to facilitate sorting of any kind of arrays. There is some disagreement in literature about the algorithm used in qsort, but I am quite sure that it is a simplified version of quicksort algorithm. E-book describes quicksort algorithm in tremendous detail in chapter at the link below.

https://sites.google.com/site/cplusplussmc/10-c-data-structure/12---analysis-of-algorithms

The call to qsort requires understanding of function pointers. The web address below gives a tutorial on function pointers.

http://www.newty.de/fpt/index.html

The name of a function stores the address of a function. C++ allows name of a function passed as an argument to a function. We show an example below:

Table 4: Function Pointer Program Example 1

Source Code

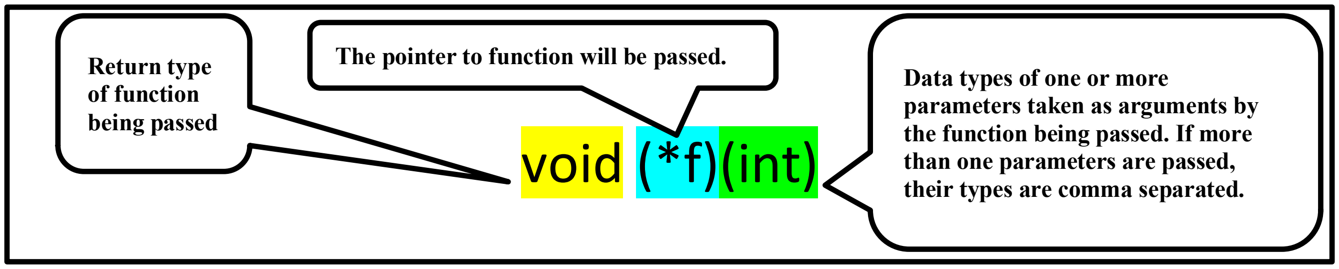
```
#include <iostream>
using namespace std;

void print(int val);
void printArrayUsingFunctionPointer(const int arr[], int len, void (*f)(int));

int main()
{
        int vals[10] = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};
        printArrayUsingFunctionPointer(vals, 10, print);

        return 0;
}

void print(int val)
{
        cout<<val<<endl;
}

void printArrayUsingFunctionPointer(const int arr[], int len, void (*f)(int))
{
        for (int i = 0; i < len; i++)
        {
                (*f)(arr[i]);
        }
}
```

output

```
10
1
9
2
8
3
7
```

34	4
35	6
36	5

On line 4 we show proto-type of regular print function that returns a void, takes an int as argument. Lines 15 to 18 show code for print function that prints its argument val to console. Line 5 shows the proto-type for a function called printArrayUsingFunctionPointer. The first two arguments are the array to be printed arr, and its logical length len. The last argument needs explanation, as provided in the figure below. This is the syntax used when pointer to a function that will be called in the body of printArrayUsingFunctionPointer is passed to it.


Return type of function being passed
The pointer to function will be passed.
Data types of one or more parameters taken as arguments by the function being passed. If more than one parameters are passed, their types are comma separated.
void (*f)(int)

The first word must be whatever type the function will be returning. Since print function returns a void that is the first word. The part (*f) is generic, only stating that a pointer to function, which is actually the name of the function will be passed as third argument when function printArrayUsingFunctionPointer is called. The last pair of parentheses encloses a comma separated list of data types that passed function takes as argument(s). If this syntax causes you too much consternation, then you can use typedef to simplify it. The typedef version is given below in Table 5.

Table 5: Function Pointer Program Example 2

Line #	Source Code
1	#include <iostream>
2	using namespace std;
3	typedef void (*anyFunc)(int) ;
4	void print(int val);
5	void printArrayUsingFunctionPointer(const int arr[], int len, anyFunc f);
6	
7	int main()
8	{
9	int vals[10] = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};
10	printArrayUsingFunctionPointer(vals, 10, print);
11	
12	return 0;
13	}
14	
15	void print(int val)
16	{
17	cout<<val<<endl;
18	}
19	
20	void printArrayUsingFunctionPointer(const int arr[], int len, anyFunc f)

```
21  {
22              for (int i = 0; i < len; i++)
23              {
24                          (*f)(arr[i]);
25              }
26  }
    output
27  10
28  1
29  9
30  2
31  8
32  3
33  7
34  4
35  6
36  5
```

After doing the typedef statement below the complexity of word assembly *void (*f)(int)* is simply replaced by the word anyFunc.

typedef void (*anyFunc)(int) ;

Then the last argument to function printArrayUsingFunctionPointer becomes (anyFunc f). The parameter name is now required because we abstracted rest of the syntax using the typedef or type definition. Line 10 in both tables 4 and 5 is quite important. It tells us how the function whose pointer is passed to another function is called. While calling, we just type the name of the function at location where function pointer is expected. Thus below line passes the pointer to function print as last argument.

printArrayUsingFunctionPointer(vals, 10, print);

Inside the body of printArrayUsingFunctionPointer the syntax

(*f)(arr[i]);

is equivalent to

print(arr[i]);

Why Use Function Pointers and Application of Qsort?

Function pointers make some generic programming possible, without use of C++ templates. The use of function pointers originated in C languages and continues in C++. For example the C/C++ qsort function allows sorting of any type of array, regardless the data type, because it takes a user defined compare function pointer as argument to facilitate array sorting. The compare function can be tailored to sort any kind of array. [Name compare is an industry accepted name, but you can use any name you like]. Table 7 will show the use of Qsort, which can be used without adding any includes in Visual Studio 2012.

The header of qsort function is below: [Full code is available if you search your hard drive with qsort, provided Visual Studio is installed on your computer.

void qsort (void *base,   size_t numElements,   size_t SizeInBytes,

int(*compar)(const void *, const void *));

Table 6 below describes various arguments passed to qsort.

Table 6		
Argument	**Description**	**Example**
void *base	**The pointer to first element of the array, which is essentially array name. Since a void pointer is being passed, that makes qsort generic because when a function accepts a void pointer, pointer to any real type can be passed at runtime.**	**If array is int arr[5] = {1,2,3,4,5}; then first qsort argument will simply be arr**
size_t numElements	**This is the number of elements in the logical length of the array, if you wish to sort entire array. Otherwise it is some length less than the logical length. Then only a smaller portion of the array will get sorted. size_t is an alias for unsigned integral type.**	**numElements for arr in above row is 5.**
size_t SizeInBytes	**This parameter simply passes the size in bytes for one array element. The sizeof operator simply does that for you.**	**Since above array is int type, the parameter SizeInBytes is simply sizeof(int)**
int(*compare)(const void *, const void *)	**This is the function pointer to the compare function we will write to facilitate sorting of array we pass to qsort.**	**We will show compare function in code example soon. Generally in actual call to qsort you need not be concerned the function pointer syntax, as very simply the name of function is used when call is made.**

The program showing use of qsort is shown below in Table 7. The details of the functions shown in previous tables are skipped.

	Table 7
Line #	
1	**#include <iostream>**
2	**using namespace std;**
3	**#include <iostream>**

```
using namespace std;
typedef  void (*anyFunc)(int) ;
void print(int val);
//void printArrayUsingFunctionPointer(const int arr[], int len, void (*f)(int));
void printArrayUsingFunctionPointer(const int arr[], int len, anyFunc f);
int compareInt(const void * ptr1, const void * ptr2);
int main()
{
        int vals[10] = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};
        cout<<"Printing unsorted array.\n";
        printArrayUsingFunctionPointer(vals, 10, print);
        cout<<"Printing sorted array.\n";
        qsort(vals,10,sizeof(int),compareInt);//Call to qsort
        printArrayUsingFunctionPointer(vals, 10, print);
        return 0;
}
///////////////////////////////////////
int compareInt(const void * ptr1, const void * ptr2)
{
        int left = *(int *)(ptr1);
        int right = *(int*)(ptr2);

        if(left>right){
                return 1;
        }else if(left<right){
                return -1;
        }else{
                return 0;
        }
}
void print(int val)
{
        cout<<val<<endl;
}
//void printArrayUsingFunctionPointer(const int arr[], int len, void (*f)(int))
void printArrayUsingFunctionPointer(const int arr[], int len, anyFunc f)
{
//See full definition in previous tables.
}
```

Output

```
Printing unsorted array
10
1
9
2
8
3
7
4
6
5
Printing sorted array.
1
2
3
4
5
6
```

	7 **8** **9** **10**

Now we discuss the details of the compareInt function, which the function pointer passed to qsort call on line 16. The success of use of qsort depends upon the success of writing corresponding compare function, which in this case is CompareInt. As lines 9 and 21 show, this or any other compare function will take two void pointers as arguments. Once again void pointers are passed to impart generic characteristic to qsort. When void pointers are passed to a function, at run time any concrete data pointers can be passed as function arguments.

When qsort is executed, two neighboring data are to be compared and based on the comparison results; an in-place swap is done if data at lower index is larger than the neighbor at higher index. The compareInt function returns a positive 1 if the data at lower index is higher, a -1, if two neighbors are actually in ascending order, and a zero if their values are same. However, since compareInt gets two void pointers, first to compute values of their pointees, the void pointers are cast to concrete pointers. This is done using the syntax such as below:

```
int left = *(int *)(ptr1);
int right = *(int*)(ptr2);
```

ptr1 is a void pointer carrying the address of the left neighbor being compared in the int array. The part below first casts the void pointer ptr1 into a concrete int pointer.

(int *)(ptr1);

Notice, this is standard C style casting. The item on right (ptr1) is being cast to the type, item on left (int *). Once the void pointer ptr1 is cast into concrete int pointer, the asterisk in the front dereferences it to get the value of array element and stores it in the variable left. If you are uncomfortable with this single line syntax, you can break it down into two lines as follows:

```
int * temp = (int *)(ptr1);

int left = *temp;
```

Once lines 23, and 24 in Table 7 are executed, the variables left and right have the concrete values of neighboring array elements, the left being the lower index neighbor and right being the higher index one. Code in lines 26 to 32 returns 1, -1, or 0, depending upon the relative values of the two neighbors. This code must return a 1 if left neighbor is higher in value, -1, if it is lower and 0 if they have same value. The returned value is used by qsort internally to make the swap if 1 was returned. If -1 or 0 were returned by compareInt function (or any other compare function) then two neighbors are already in correct order and no swap is done.

Review questions

1. What will the following code print?

```
struct    ex
{
        int * ptr;
};

int main()
{
        ex x;
        int val = 5;
        x.ptr = &val;
        ex * ptr2 = &x;
        cout<<*ptr2->ptr;
        return 0;
}
```

2. Describe the use of arrow operator for member selection in an object.
3. What is this pointer? Explain what does it store.
4. Can this pointer be used inside the static member functions and friend functions?
5. Describe how to use this pointer to simplify the postfix overloaded operators?
6. For the class Student discussed in this chapter will the following compile?

```
Student S5(S5);
```

Will the following compile and run fine if the copy constructor is written as given below and the default constructor has no code in it?

```
Student S5 = Student(S5);
Student(const Student& Other)
    {
        name = new string(*Other.name);
        leak_counter++;
        gpa = Other.gpa;
        cout<<"From Copy Constructor.\n";
    }
```

7. Will the code

```
Student S5 = Student(S5);
```

run fine if there is no copy constructor written by you and the default constructor has a blank body?