From a purely theoretical point of view, an array is also an abstract data type. The basic properties of an array are[1]:

1. we can create an array of a specific size with a specific data type,
2. we can store a value in the array by providing an index of the desired location, and
3. we can retrieve data stored at a specific index in the array by providing the index.

While using an array, the programmer only needs to deal with these properties and is not required to know how an array is actually represented in the system.

In addition to such built-in abstract data types, C++ also provides support for user defined abstract data types and that is elaborated in the next section.

1.3 ADTs and C++ Classes

In C++, classes are used to develop user defined Abstract Data Types. A class consists of data and functions that operate on that data called members of the class. A class in C++ has two parts – public and private (let's ignore the protected members for now). The members of the class that make up the public part capture the essential details of the class for the parts of the program, i.e. users, that use this class. Combined, they form the interface to the class. Data encapsulation is achieved by declaring all data members of a class as private. Such members can only be used by other members of the same class (let's also ignore the friends for now).

Users of the class can only access and manipulate the class state through the public members of the class.

[1] Arrays in C++ are not really first class ADT's as implementation details are not truly hidden. Rather, in many situations, knowledge of implementation details is essential for proper use of arrays.

```
class Point {
public:

   Point(float x = 0.0, float y = 0.0);

   float getX();
   float getY();
   float distance(Point);

   void moveRelative(float ,float );
   void moveAbsolute(float ,float );
   void print(ostream &);

private:

   float xCoord, yCoord;  //members to hold Cartesian

                          //coordinates of the point object
```

Figure 5 – Specification of Point ADT

```
class Point {
public:

   Point(float x = 0.0, float y = 0.0);

   float getX();
   float getY();
   float distance(Point);

   void moveRelative(float ,float );
   void moveAbsolute(float ,float );
   void print(ostream &);

private:

   float r, theta; //polar coordinates this time

};
```

Figure 6 – Specification of Point ADT with polar coordinates

As an example consider the specification in C++ of an ADT for a Point in a 2-dimensional Cartesian system shown in figure 6.

As mentioned earlier, the public part specifies the interface or behavior and the implementation is encapsulated in the private part. For example, in this case the xCoord and yCoord are declared as private. The user of the Point ADT is only concerned with the public interface and cannot directly access these private members that are used in the implementation of the ADT. Thus, any change in the implementation would not affect the user program as long as the interface stays the same. For example, the designer of the point ADT could decide to change the representation of point from Cartesian to Polar without requiring any change in the user program. Figure 6 shows the implementation of the Point ADT, which is usually kept separate from the specification[2].

[2] It is good programming practice to keep the specification of a class in .h file and the implementation in the .cpp file.

```
Point::Point(float x, float y) {
   xCoord = x;
   yCoord = y;
}

float Point::getX() { return xCoord; }
float Point::getY() { return yCoord; }

float Point::distance(Point other) {
   float x = xCoord - other.xCoord;
   float y = yCoord - other.yCoord;
   return sqrt(x*x + y*y);
}

void Point::moveRelative(float x, float y)
{
   xCoord = xCoord + x;
   yCoord = yCoord + y;
}

void Point::moveAbsolute(float x, float y)
{
   xCoord = x;
   yCoord = y;
}

void Point::print(ostream &str)
{
   str << "(" << xCoord << "," << yCoord << ")";
}
```

Figure 7 - Implementation of Point ADT

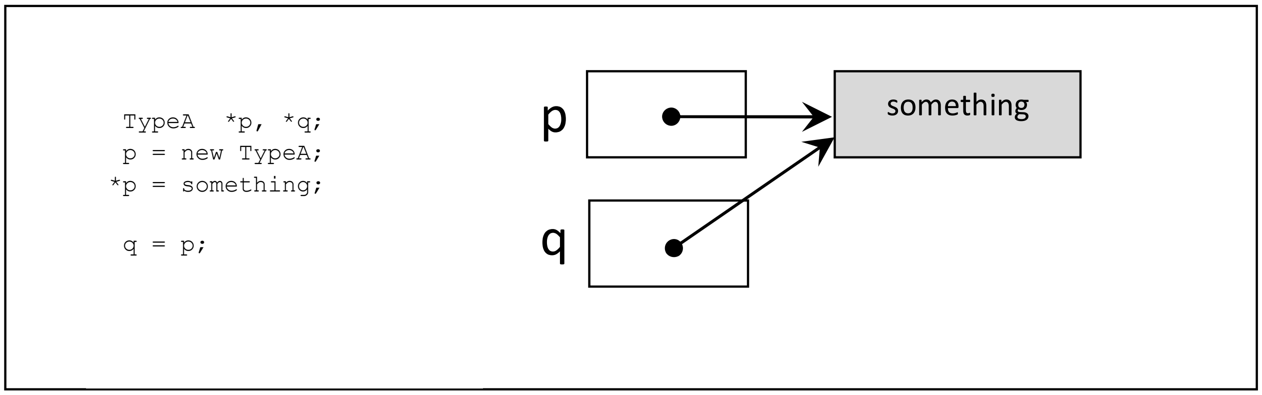
1.4 Classes with dynamically allocated memory

In general, the basic characteristics of the problem at hand are known. However, at the time of writing the code, the exact size and shape of the particular instance of the problem that is being solved may be unknown. This requires a mechanism through which the structure that is used to organize data can grow dynamically according to the need of the problem. In C++, pointers provide the facility to dynamically allocate memory on-demand and then manipulate this memory area. They give programmers a direct access to the memory and hence provide perhaps the most powerful programming tool. This most powerful tool is also the most dangerous one. Pointers, if not handled with utmost care, introduce the nastiest of errors that are very hard to debug and fix. The three most important nuances that ought to be avoided while dealing with pointers are: (a) shallow copy as opposed to a deep copy, (b) garbage creation or memory leaks, and (c) dangling references. Following is a brief discussion of these topics.

1.4.1 Deep versus shallow copy

Many a times it is required that a copy of the dynamically allocated memory area is made. As shown in Figure 8, simply copying the pointer into another pointer variable does not do the job as it only copies the address of the memory area and both the pointers point to the same physical memory location. This is called shallow copy because the copy of the address rather than the contents of the memory is made.

When the copy is made by first allocating a separate memory area and then copying the contents, it is called deep copy. This is depicted in

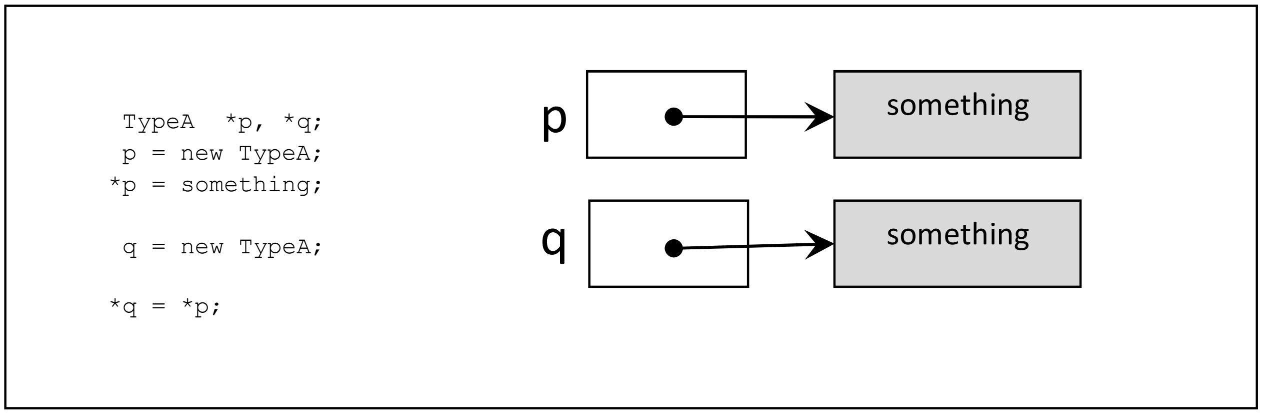


```
 TypeA  *p, *q;
 p = new TypeA;
*p = something;

 q = p;
```

Figure 8 - Shallow copy

Figure 9.



```
 TypeA  *p, *q;
 p = new TypeA;
*p = something;

 q = new TypeA;

*q = *p;
```

Figure 9 - Deep copy

1.4.2 Dangling references and garbage

Space may be acquired at run-time on request from free store[3] using *new* and the address of the memory area so acquired can be stored in a pointer type variable for future references. At a point in time when a memory area pointed to by a pointer is no longer required, it can be returned back to free store using *delete*. We should always return storage after we no longer need it. Once an area is freed, it is improper and quite dangerous to use it.

A memory area that is no longer required if not returned back to the free store, may eventually become inaccessible. Such memory area is called garbage or memory leak. On the other hand, a pointer that has an address of a memory area which is not under its control (it may have been returned back to the free store or it is not initialized and has junk value in it), then it is called a dangling reference. In other words, a dangling reference is an access path that continues to exist after the lifetime of the associated data object. A very common programming error that creates dangling references is outlined below. In order to understand the concept, let us refer back to Figure 8.

[3] The terms *heap* and *free-store* are used interchangeably when referring to dynamically allocated objects. However, in the context of C++, the two are similar but they are not the same and using "free store" is preferable as it is more conformant with the C++ standard.

At the moment, both p and q point to the same memory area and they can be used quite legitimately to manipulate data stored in the memory area pointed to by these pointers. But situation changes if something like the following is done.

```
    delete q;
    q = NULL;
```

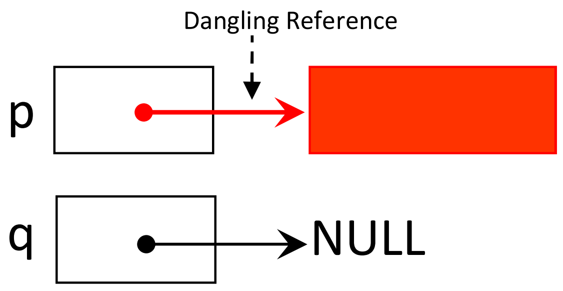


Figure 10 - A Dangling Reference

The programmer seems to have done the proper thing by setting the value of q to NULL after the memory has been returned to the free store. This very innocent looking piece of code, which seems to be correct at a first glance, has however created a dangling reference! In order to understand the problem, we have to remember that p was also pointing to the same memory area. When the memrory was deleted, what happened to p? As depicted in Figure 10, `p` now points to a memory area which has been returned back to the free store and it does not anymore have a legitimate right to use it. It's a dangling reference! Now an attempt to access the memory area whose address is present in p may result in an undesirable situation, which, at times, can be quite nasty.

Let us now look at the problem of memory leak.

```
TypeA  *p, *q;
p = new TypeA;
q = new TypeA;
```

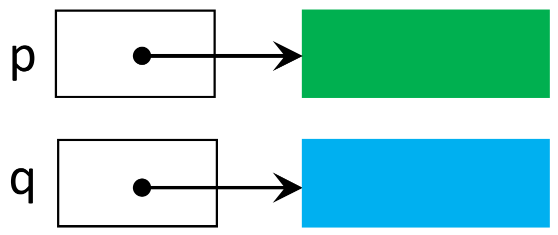


Figure 11

When all access paths to a data object are destroyed but the data object continues to exist, the data object is said to be garbage. Let us consider the code and corresponding diagram of Figure 11.

Let us now execute the following statement:

```
q = p;
```

The new situation is shown in Figure 12. What happens to the space that was pointed to by q?

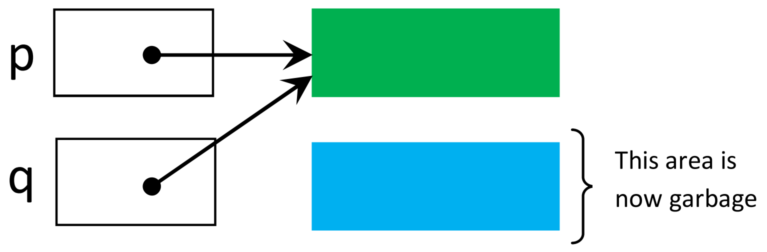


Figure 12 - Garbage or memory leak

This memory area has not been returned back to the free store but, at the same time, it is no longer accessible. As far as the run-time environment is concerned, this memory area is still in use and hence cannot be assigned against another request for memory allocation. It has virtually "leaked" from the system and is called garbage. Now, for all practical purposes, the system's memory has been reduced by that much.

In C++, garbage is only reclaimed by the operating system after the termination of the program.

Garbage slowly eats up the memory and perpetual accumulation of garbage slows down the system to an extent that no effective work can be done and the corresponding program needs to be terminated.

1.4.3 Classes and dynamic memory

If not created by the programmer, C++ compiler automatically creates four member functions for a class. These are: (1) the default constructor, (2) the destructor, (3) the copy constructor, and (4) the operator=.

When dynamically allocated memory is used inside classes, the last three (the destructor, the copy constructor, and assignment operator) become extremely important. In that case the programmer is required to explicitly write these three methods or face the pointer related problems mentioned earlier. Let us try to understand these issues with the help of a simple example of a class given in Figure 13.

```
class Student {
public:
        Student(char *sName,  int credits = 0);
          // other methods to access and manipulate data
          // assume that initially the destructor, copy constructor, and operator= have
          // not been written for this class.
private:
        char *studentName;
        int creditsEarned;
        void makeCopy(char *sName, int credits); // private auxiliary function
};

void Student::makeCopy(char *sName, int credits)
{
        creditsEarned  = credits;
        studentName    = new char[strlen(sName)+1];
        if (studentName != NULL)
                strcpy(studentName, sName);
        else
                throw OUT_OF_MEMORY_EXCEPTION;
}

Student::Student(char *sName, int credits)
{
        makeCopy(sName, credits);
}
```

Figure 13 - Class using dynamically allocated memory to store data

As can be easily seen, in this example, the name of the student is stored in a dynamically allocated memory area pointed to be the `studentName`. Let us now see what happens when the destructor, copy constructor, and `operator=` are not implemented by the programmer.

1.4.3.1 The Destructor

A destructor is a member function with the same name as its class prefixed by a ~ (tilde). The destructor of a class is called when an object of that type is destroyed. There are two basic scenarios: (1) when an object goes out of scope, or (2) when an object pointed to by a pointer is destroyed explicitly by using the `delete` operator.

For most cases, the default destructor is enough. However, when the object has some dynamically allocated memory, the default destructor is not sufficient. It does not return the dynamically allocated area back to the free store. As a result, as depicted in Figure 14, garbage is created in the process. Therefore, when dynamically allocated memory is used inside

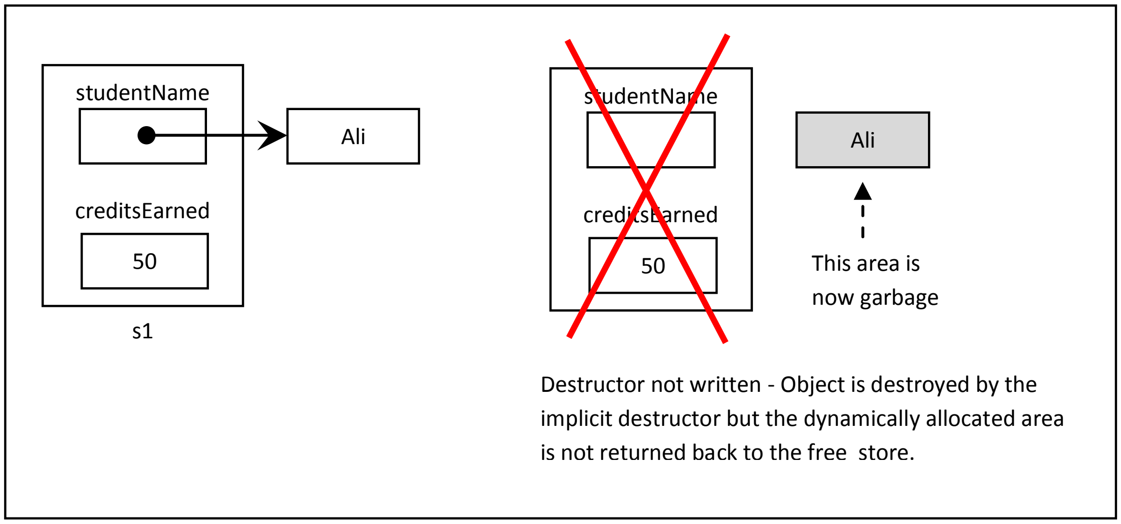

studentName
Ali
creditsEarned
50
s1
studentName
creditsEarned
50
Ali
This area is now garbage
Destructor not written - Object is destroyed by the implicit destructor but the dynamically allocated area is not returned back to the free store.

Figure 14 – Accumulation of garbage when the destructor is not written

a class, the programmer must write a destructor or garbage would be accumulated. Writing a destructor in this case is easy – all you have to do is explicitly delete the dynamically allocated memory as shown below:

```
Student::~Student()
{
        delete [] studentName;
}
```

1.4.3.2 The Copy Constructor

The copy constructor is a special constructor that is used to make a copy of the object. It is called implicitly to make a hidden temporary copy of the object in three situations listed below:

a) It is called when an object is initialized by another object at the time of declaration. For example, the copy constructor will be implicitly called in the statement shown below:

```
TypeA    a = b;
```

In this case a temporary copy of `b` will be made to initialize `a` and that copy will be destroyed at the completion of the job.

It may be noted that the copy constructor will not be called in the following scenario:

```
TypeA    a;
```

```
a = b;
```

In this case the assignment operator (operator=) will be used. Assignment operator is similar to the copy constructor but there are certain differences that we will discuss later.

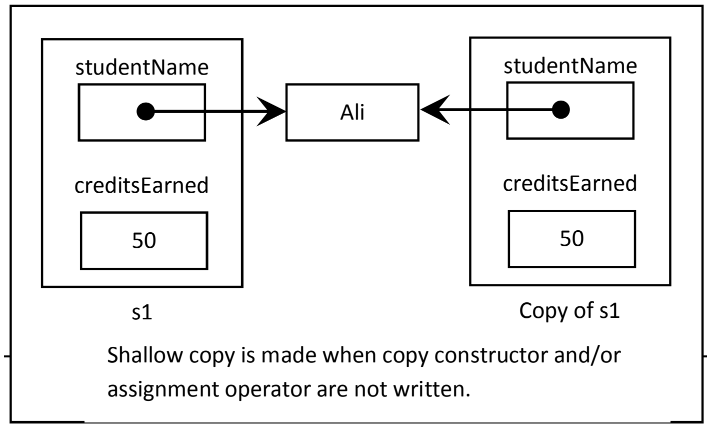
b) The second place where the copy constructor is implicitly called is when an object is returned by value from a function. As an example, consider the following code:

```
TypeA foo(){
  TypeA x;
  // something is done here
  return x;
}
```

Just before returning from the function, a copy of x will be made. That copy will be used to copy the data in the target. Once again this implicitly created temporary object (in this case copy of x) will be destroyed when the job is complete and the function terminates.

c) The third place for the use of copy constructor is when an object is passed by value as a parameter to a function.

As mentioned above, the life time of the implicitly created temporary object by the copy constructor is limited to the life time of the corresponding operation. At the completion of the required job, this object is destroyed. This creates problems when dynamic memory is involved.


studentName
Ali
studentName
creditsEarned
50
creditsEarned
50
s1
Copy of s1
Shallow copy is made when copy constructor and/or assignment operator are not written.
Figure 15 - Implicit copy constructor makes a shallow copy

The default copy constructor generated by the compiler simply copies the class data members. That is, if a pointer was involved, it would make a shallow copy whereas, in most cases, a deep copy would be required. This scenario is shown in Figure 15.

But that is not all – the destructor, if implemented, plays a nasty role here and joins hands with the default copy constructor in creating dangling references.

When the destructor destroys the implicitly created copy, it also disposes-off the dynamically allocated memory area. But if a shallow copy of the object was made, destroying the copy would destroy the original because the same memory area was shared by both. As shown in Figure 16, this creates a dangling reference.

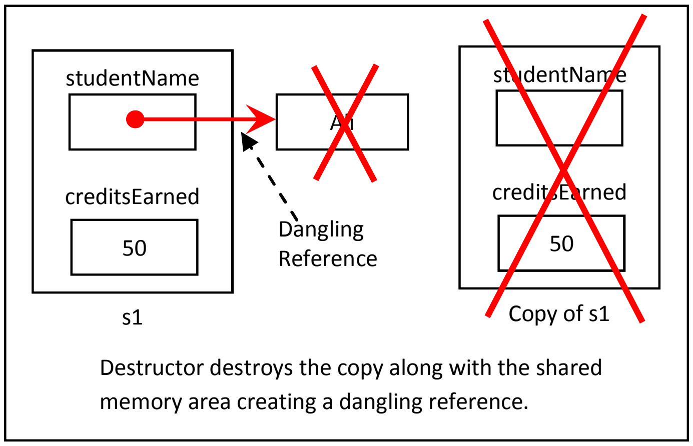

studentName
Ali
creditsEarned
50
s1
Dangling
Reference
studentName
creditsEarned
50
Copy of s1
Destructor destroys the copy along with the shared memory area creating a dangling reference.

Figure -16 – Implicit copy constructor and explicit destructor – a dangerous combination

Therefore, in order to avoid such situations it is required to make a deep-copy of the object. Hence, the programmer must write a copy constructor when dynamically allocated memory is involved. Once again, writing the copy constructor is easy – just make sure to allocate memory explicitly and then copy the data. The copy constructor for the Student is shown below:

```
Student::Student(const Student& other)
{
   makeCopy(other.studentName, other.creditsEarned);
}
```

The only other point to note here is that the reference of the object whose copy is to be made is passed as parameter in the copy constructor. It is important to do that otherwise if the object is passed by value, its copy will be made which will call the copy constructor again and hence will result in an infinite recursive call of the copy constructor.

1.4.3.3 The assignment operator (operator=)

The assignment operator comes into action when value of one object is assigned to another object. In this case, just like the copy constructor, the default assignment operator simply makes a shallow copy. Therefore, once again the programmer must write an overloaded assignment operator when dynamically allocated memory is involved. Overloading the assignment operator is not much different from the copy constructor and is written below:

```
const Student& Student::operator=(const Student& rhs)
{
   if (this != &rhs) {
        delete[] this->studentName;
        makeCopy(rhs.studentName,rhs.creditsEarned);
   }
   return *this; // return self-reference so
                 // cascaded assignment works
}
```

The important points to note are:

(a) The memory already allocated to the object on the left hand side of the assignment operator must be deleted to avoid memory leaks.

(b) Before deleting memory it is important to check for self assignment (like a = a;). This is done by comparing the *this* pointer with the address of the object on the right hand side. Not doing so will create dangling references when an object is assigned to itself.

(c) Reference to self is returned back to allow cascading (e.g. a = b = c;) of assignments.

Figure 17 shows the scenario when all three methods mentioned above are properly written.

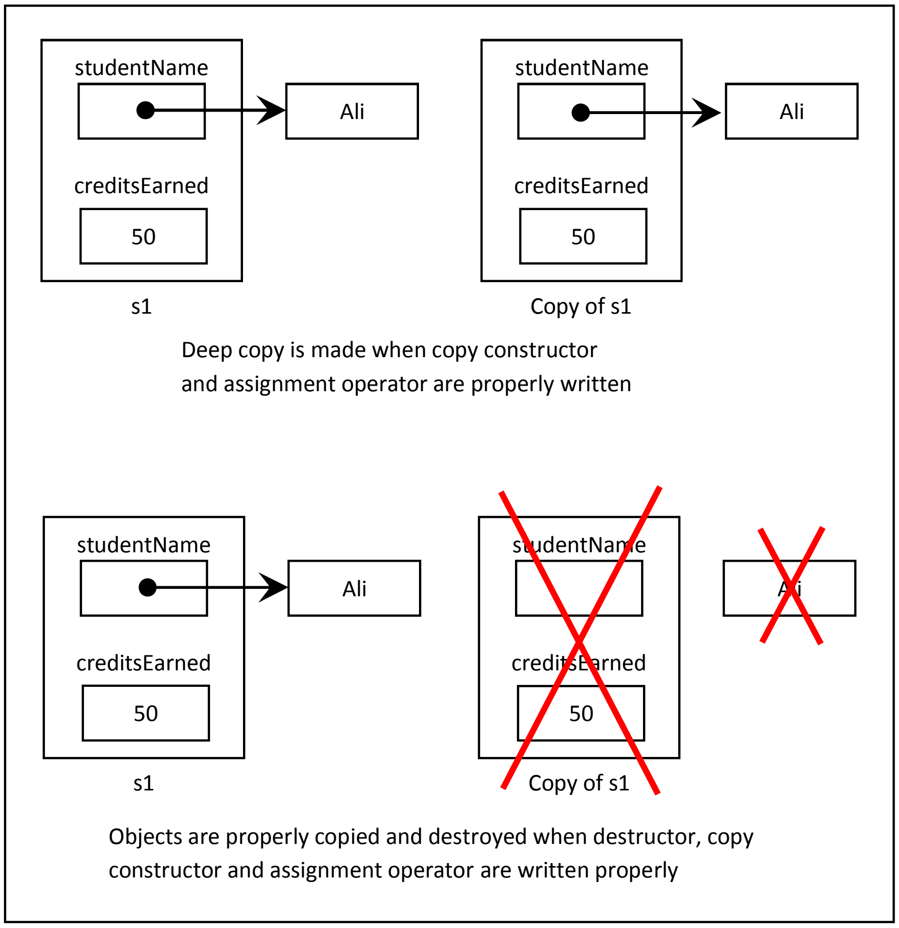

studentName
Ali
creditsEarned
50
s1
studentName
Ali
creditsEarned
50
Copy of s1
Deep copy is made when copy constructor and assignment operator are properly written
studentName
Ali
creditsEarned
50
s1
studentName
Ali
creditsEarned
50
Copy of s1
Objects are properly copied and destroyed when destructor, copy constructor and assignment operator are written properly

Figure 17 - Destructor, copy constructor, and assignment operator properly implemented

1.5 Error Handling

Except for very simple programs, it is common for a program to fail because of some rare condition occurring at run-time. This could be the result of a logical programming error, bad input data, unavailability of a resource, or some kind of a hardware failure. Simply terminating the program in such cases may result in the loss of some critical data or even a catastrophic disaster. Therefore, a programmer is required to design programs that can either recover from such error and then continue after taking the corrective measure or at least shut down gracefully to minimize the amount of damage. Proper handling of run-time errors is thus considered a key distinction between high quality professionally written programs and a poor amateurishly written code. To handle run-time errors two basic techniques are used. These are error code testing and exception handling. A brief discussion of these follows.

1.5.1.1 Error code testing

Error code testing is a very common error handling technique. In this approach, functions are written to return values to indicate success or failure. Values that indicate failure are called error codes. The programmer can then test for those return values and decide what to do next. This concept is elaborated with the help of the following example. In this example, the program would go into an infinite loop if, at any stage, the data entered by the user was non-numeric.

```
int i = 0;
while (i != -1)
{
  cout << "Enter a number (-1 to exit): ";
  cin >> i;
  if(i != -1) {
        cout << "The number is: " << i << endl;
  }
}
```

If the user enters something like "abc", the input stream will go into an error state and will not work properly until it is cleaned-up and flushed. In order to avoid such problems, the user should check for the validity of the input by testing the input stream for failure and then take the corrective measure as shown next:

```
int i = 0;
while (i != -1)
{
  cout << "Enter a number (-1 to exit): ";
```

```
        cin >> i;
        if (cin.fail()) {
                cout << "Not a valid number!" << endl;
                cin.clear();
                cin.ignore(1);
        }
        else if(i != -1) {
                cout << "The number is: " << i << endl;
        }
      }
```

If used properly, error code testing is a useful and effective mechanism to detect and recover from errors. However, its effectiveness is easily compromised if a programmer fails to check the error code. In many such cases, ignoring an error code results in nasty problems that appear to have occurred at a place which is far removed from the actual problem area and are thus extremely difficult to debug and fix. For example, failing to check for NULL value as a result of using the malloc function may consequently lead to an inappropriate use of the pointer, hence resulting in program crash and loss of data. In addition, if we use error code testing to handle errors, recovery from deeply nested function calls may also be a non-trivial exercise.

1.5.1.2 Exception handling

Modern programming languages offer an alternative approach, called exception handling, to handle run-time errors. In most situations, this provides a much simpler and cleaner strategy for error handling as compared to error code testing. In addition, exceptions are much harder to be ignored by the programmers and hence help in producing code which is more robust and is easier to debug.

Exception handling mechanism is briefly discussed next as a detailed discussion on the topic is beyond the scope of this book.

```
void exceptionExample() {
        int x;
        x = mightThrow();
        cout << "x = " << x << endl;
}

int mightThrow() {
        int i;
          cout << "Enter a number: ";
          cin >> i;
          if (cin.fail())
                    throw INPUT_ERROR;
          return i;
}

int main()
{
        try {
          exceptionExample();
        }
        catch(MyExceptionType e)
        {
          cout << "Exception " << e << "occurred" << endl;
          // some corrective measure may be taken
        }
        catch(…)
        {
          cout << "something else happened" << endl;
        }
        return 0;
}
```

Figure 18: A simple exception handling example

Exception handling framework in C++ is based upon three activities: (a) indicating an exception, (b) monitoring for exception condition, and (c) handling the exception. The language provides the support for these activities through throw, try, and catch statements respectively. The throw statement is used to indicate the exceptional condition that needs to be taken care of. To catch an exception, code must be written inside a try block and the code to handle any exceptions occurring in the try block is written in a corresponding catch block. To handle different types of errors arising from a single try block, there may be more than one corresponding catch blocks. All catch blocks related to a try block are written one by one immediately after the try block. In order to understand, let us consider the code segment of Figure 18.

In this example, the main program calls the function `exceptionExample` within its *try* block. `exceptionExample` calls `mightThrow` which might throw an exception if `cin` goes into an error state. If the exception is raised, the control goes back to a *catch* block which is equipped to handle that particular type of error. So, if `INPUT_ERROR` is thrown by `mightThrow`, then the control returns to `exceptionExample`. As `exceptionExample` does not have a *catch* block to handle `INPUT_ERROR`, the control goes back to the main program without printing the value of `x`. Associated with the *try* block, there are two *catch* blocks in the main function. The first one can handle exceptions of type `MyExceptionType` and the second one can catch any exception. Assuming that `INPUT_ERROR` is of type `MyExceptionType`, it will be handled by the first *catch* block and the necessary corrective measure can be taken at that point.

In addition, exception has another advantage over error codes. As a constructor cannot return a value, hence if it fails error codes cannot be used. Therefore throwing an exception is the only real option to report an error from a constructor.

For these reasons, rather than using error codes, we have taken the exception handling approach. Therefore, throughout this book, error conditions will be indicated by throwing an exception and it will be up to the user programs to handle these exceptions and use an appropriate strategy to recover from these error conditions.

Exception handling is an involved topic and, as stated earlier, a detailed discussion on this topic is beyond the scope of this book. An interested reader is therefore advised to look at some of the reference material mentioned at the end of the chapter.

1.6 Generic Programming

Let us revisit our Selection Sort function once again. It sorts an array of integers in ascending order. If we wanted to sort an array of floating point numbers, we would have to rewrite the function. The only difference in the two functions would be the data types of some variables and parameters. It may be noted that in the case of second version of the selection sort algorithm, all three functions would need to be rewritten with data types being the only changes required.

```
void swap (int &x, int &y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

```
void swap (float &x, float &y)
{
        float temp = x;
        x = y;
        y = temp;
}
```

Figure 19 – Duplicated code because of change in data type

As an illustration, the two swap functions are given in Figure 19.

It may be noted that function overloading allows us to use the same name for the function and hence that does not need to be modified.

It is obviously inconvenient to write the same code again simply because there was a change in the data type (though it is mostly copy-paste). From a maintenance point of view, it is a nightmare – every time there is some change required in the code for whatever reason, the same change has to be applied to all the different versions of the same code. It is not very unusual that some version of the code is missed in the process, hence making it inconsistent and faulty.

From this perspective, it would have been quite useful if, just like function parameters, data types could also be parameterized. Generic programming is a mechanism to do exactly that.

In C++, support of generic programming is provided through *templates*. In this case, the generic entity is declared as *template* and it takes special parameters through which types can be passed as arguments. These parameters are used inside the entity as if they were any other regular type. C++ supports two kinds of templates, function and class. Both of these kinds of templates are briefly discussed in the following sub-sections.

1.6.1 Function templates

A function template defines a function in which types of function parameters and return type are parameterized. Thus, with the help of templates, the swap function can be written generically as shown in Figure 20.

In this example we have created a template function with T as its template parameter. It is a placeholder for whatever type is specified when *swap* is called.

```
template <class T>
void swap (T &x, T &y)
{
        T temp = x;
        x = y;
        y = temp;
}
```

Figure 20 - Function Template

To use *swap* with a concrete type we can either specify the type explicitly, or let the compiler figure it out. So, assuming that x and y are two integers, we can either write

```
swap <int> (x, y);          // type explicitly specified
```

or

```
swap(x, y);                 // type to be determined by the complier
```

When the compiler encounters a call to a function template, it uses the template to automatically (and transparently) generate a function replacing each occurrence of the dummy type parameters by the type passed as the actual template parameter (`int` in this case) and then calls it.

1.6.2 Class templates

Like function, many classes are also intrinsically type independent. In fact, container classes used for implementing different data structures that we will study later are independent of the type of the data that is stored in these entities. A classical example of such an entity is an array. In C++, class templates facilitate the development of such classes.

```
template <class T>
class SmartArray
{
public:
        SmartArray(int n);                    // constructor

        SmartArray(const SmartArray &other); // copy constructor
        ~SmartArray();                        // destructor

        T& operator[] (int index);            // overloaded subscript operators
        const T& operator[] (int index) const;

private:
        int mSize;                            // size of the array
        T *mArray;                            // array to be created dynamically
        int *mRefCount;                       // to be used in the copy
                                              //constructor and destructor

        SmartArray& operator =(const SmartArray& other);
                                              // assignment operator is decalred
                                              // in the private part to disallow
                                              // array assignments
        void checkBounds(int index);
};
```

Figure 21 – Class template Specification

Following is an example of a generic container, `SmartArray`, which is essentially a one-dimensional array with support for bound checking in it. Since an array can store elements of any type, `SmartArray` needs to be made generic. Therefore, we will create a class template for `SmartArray` and then we can use it instead of a single dimensional array.

1.6.2.1 SmartArray Specification

Specification of the `SmartArray` is given in Figure 21. It shows that it is a class template where the data type to be stored in the array is parameterized. In the private part of the specification we have three data members. `mArray` is a pointer to the memory area used for storing the data in the `SmartArray`. `mSize` maintains the size of the array and will be used in checking the array bounds. The role of `mRefCount` will be discussed later. In addition, the `operator=` is made private to disallow assignment and `checkBounds` is a private method that will be used internally by other member functions of the class. Except for the differences enumerated below, a `SmartArray` will behave exactly like an array. The differences are:

a) It has bound checking in it.
b) Syntax for declaring an entity of `SmartArray` type is different.
c) An expression (to be evaluated at run-time) can be used in its declaration.

The implementation of the `SmartArray` class template requires some special attention.

In order to make it behave like a normal array, we need to take the following measures:

a) To disallow a declaration of the form
```
   SmartArray<int> x;
```
   the default constructor is not written.
b) As mentioned above, in order to disallow assignment of one `SmartArray` to another, the assignment operator is declared as private. This is sufficient to achieve the desired objective and no implementation of the `operator=` is required. It is important to note that if the assignment operator is not overloaded, default assignment operator will be used, which will yield undesirable results in this case.
c) The effect of passing a `SmartArray` to a function should be similar to a normal array. That is, it should behave as if it was passed by reference so that any changes made to any of its elements inside the function should be reflected in the original `SmartArray` that was passed as the argument to the function.

1.6.2.2 `SmartArray` implementation – A copy constructor that makes a shallow copy

Usually, the copy constructor is written to facilitate pass by value. However, as mentioned above, in our case we want it to support pass by reference. Therefore we need to write it differently.

In order to achieve this effect, a shallow copy of the object is made. The resultant copy constructor is shown next:

```
template<class T>
SmartArray<T>::SmartArray(const SmartArray& other)
{
      mSize       = other.mSize;
      mArray      = other.mArray;    // make shallow copy
      mRefCount   = other.mRefCount;// copy the pointer
      (*mRefCount)++;
}
```

On a cursory glance it appears that the default copy constructor would have done exactly the same therefore there really was no need to write our own copy constructor. However, there is some complication in this case that makes it necessary to write a copy constructor to make a shallow copy of the data. In order to understand the difficulty, the role of the destructor needs to be revisited. Implementation of the constructor and destructor and are discussed next.

1.6.2.3 `SmartArray` implementation – The destructor and the constructor

You may recall that the destructor is called when the object of a class is destroyed for any reason. As mentioned earlier, the temporary copy made at the time when an object is passed by value is destroyed when the function terminates. As we want to achieve the effect of pass by reference, the copy and the original object would share the same data. Therefore, when the function returns, invocation of the destructor will destroy the copy and hence the original object will also be destroyed. This will create a dangling reference.

In order to avoid that situation, we have used a reference count. The reference count, maintained in the dynamic variable pointed to by the pointer `mRefCount`, is initialized to 1 when an object of type `SmartArray` is declared. It is shared by all the copies of this object (a shallow copy of `mRefCount` is made) and is incremented by one when a copy of the object is made. When the destructor is called, it is decremented by one and the array is deleted only when it is zero. That is, the array containing the data is deleted when no other copy of the object exists. The resultant constructor and destructor are shown next:

```
template<class T>
SmartArray<T>::SmartArray(int n)
{
   if (n < 1) throw ILLEGAL_ARRAY_SIZE;
   else {
      mSize = n;
      mArray = new T[mSize];
      if (mArray == NULL)  throw OUT_OF_MEMORY;
   }
   mRefCount  = new int;
   if (mRefCount == NULL)  throw OUT_OF_MEMORY;
   *mRefCount = 1;
}

template<class T>
SmartArray<T>::~SmartArray()
{
      (*mRefCount)--;
      if ((*mRefCount) == 0)
            delete [] mArray;
}
```

1.6.2.4 `SmartArray` implementation – Overloading the subscript operator

Finally, in order to support the same syntax in SmartArray to access the elements directly just like regular arrays, we need to overload the subscript operator (`operator[]`). The subscript operator, `[]`, must be a member function.

In order to allow the use an array that has been passed to a function as a constant in an expression, we need to overload it twice – one for providing the read-only access and the other for read-write access so that it can also be used on the left hand side of the assignment. The former is implemented by returning a constant reference of the element at the specified index and the latter by returning the non-constant reference of the element at the given index. Except for the difference in the return type, these two versions of the overloaded subscript operator would look exactly the same. From the context, the compiler will automatically decide which particular function needs to be called and it is totally transparent to the user. These two overloaded functions are given below:

```
template<class T>
const T& SmartArray<T>::operator[] (int index)
const
{
      checkBounds(index);
      return mArray[index];
}

template<class T>
T& SmartArray<T>::operator[] (int index)
{
      checkBounds(index);
      return mArray[index];
}
```

The bound checking function is trivial and is shown below:

```
template<class T>
void SmartArray<T>::checkBounds (int index)
{
   if (index < 0 || index >= mSize)
      throw OUT_OF_BOUNDS;
}
```

1.7 Summary

In this chapter we discussed the importance of maintenance in the software development lifecycle and then looked at different tools and techniques available in C++ for developing data structures with high degree of maintainability. The main points discussed in the chapter are enumerated below:

1. After being put into use, long-lasting software goes through continuous change. Maintenance cost normally exceeds development cost by factors ranging from 2 to 3. Thus reduction in the cost associated with maintenance could significantly reduce the overall cost associated with the software system. Maintainability of the

system can improve if the design and code is understandable and the changes are, to a large extent, local in effect. The three basic principles that guide maintainability are: simplicity, clarity, and generality.

2. The principle of abstraction plays a major role in achieving these objectives. In C++, support for abstraction is provided at the code as well as at the data level. Code abstraction is mainly achieved by breaking a larger function into smaller subprograms. In C++, classes are used to achieve data abstraction through user defined Abstract Data Types (ADT's).
3. When working with dynamically allocated memory, one has to be careful about memory leaks and dangling pointers. When a class uses dynamic memory, the programmer must write the destructor, copy constructor, and operator=.
4. Error handling can be achieved by using error code checking or exception handling. Error codes can be ignored by the programmers whereas exceptions must be caught and processed. Exception handling thus provides a more robust error handling mechanism.
5. Generic programming is very powerful feature of the C++ programming language that allows the development of data type independent algorithms and data structures through the use of templates. This rids the programmer to write the same (similar) code again and again for different data types. Generic algorithms are achieved through function templates whereas data structures use class templates.

1.8 Exercises

1) Execute the following program and explain the output. You may need to include appropriate header files for the required libraries.

```
int main( )
{
      int     i, *pi;
      float   f, *pf;
      i = 1024;
      pi = &i;
      pf = (float *) pi;
      f = *pf;
      cout << i  << "   " << f << "\n";
      f = i;
      cout << i  << "   " << f << "\n";
      return 0;
}
```

2) Write a function in C++ to swap two entities of arbitrary type without using templates.
3) Write function template for binary search.
4) Convert the selection sort given in Figure 5 into a template based version.
5) Write a template based function to insert data into a sorted list.
6) Use the function developed in Q 4 to write a template based function for insertion sort.
7) Develop a class in C++ for complex numbers.
8) Develop a class in C++ to store a set of points in an array that is created dynamically in a constructor.
9) Develop a generic class for two-dimensional array with bound checking.
10) The C++ standard library includes a smart pointer called auto_ptr. These types of pointers are said to be safe from the usual pointer related issues. Study auto_ptr and compare it with C++ pointers.
11) Rewrite SmartArray (section 1.6.2) using auto_ptr.

12) Identify and correct errors in the following class

```
class X {
   private:
         const int size;
         int *storage;
         int count;
   public:
         X (int _size) : size(_size) {
               if (size < 1) throw ILLEGAL_SIZE;
               else {
                     storage = new int[size];
                     if (storage == NULL)
                           throw OUT_OF_MEMORY;
                     count = 0;
               }
         }

         void insert(int data) {
               if (count >= size) throw OUT_OF_SPACE;
               storage[count] = data;
               count++;
         }

         void print() {
               for (int i = 0; i < count; i++)
                     cout << storage[i] << " ";
               cout << endl;
         }
};
```