Chapter 4

Objective

- Reference Type
- Pointer Swapping using reference
- Pointer Swapping using Double pointers
- const Specifies
- Constant Array Elements
- Constant Objects
- Member initialization List
- Copy Constructor
- Allocate memory Using Copy constructor
- Static Members
- Static Member Functions
- Design Pattern Using Static Member Function

Unit 1

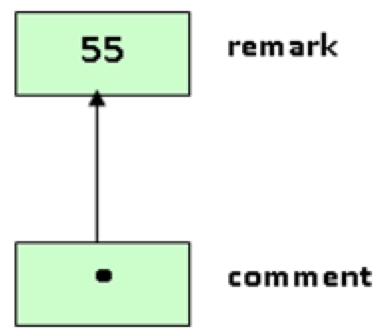
Reference Type

- & is used as a reference operator.
- A reference is an alias for an already existing variable.
- A reference must be initialized at declaration and cannot be changed.
- References work best when passing user-defined data types as function parameters.

```
#include <iostream>              // Example 4-1
using namespace std;
int main()
{
        int remark = 17;         // an integer remark is set to 17
        int &comment = remark;          // Reference comment is an alias of remark
        comment = 55;            // Set comment to 55, which also causes remark to be 55
        // comment points to the same location that remark has
        // comment does not need de-reference.
        // comment cannot be used for any other memory created by
        // other variables as a reference variable.
        cout << comment << ' ' << remark << "\n";
}
```


Unit 1

```
#include <iostream>            // Example 4-2
using namespace std;
int main()
{
        int remark = 17;       // an integer remark is set to 17
        int *comment = &remark;      // assign the address of remark to comment
        *comment = 55;         // Set comment to 55, which also causes
        // remark to be 55 comment is holding the
        // address of remark, therefore remark
        // also becomes 55 automatic de-reference
        // is not done, so use ( * ) comment can
        // hold the address of any integer
        // variable.
        cout << *comment << ' ' << remark << "\n";
}
```



Unit 2

Things Which Can Not Be Done With References

- Reference variable can not be modified as an alias for other variable.
- Reference variable can not be initialized with the address of other variable.

Things Which Can Be Done With References

- Reference variable can be passed as a parameter to other functions.
- Function may return the reference variable.

Advantages of Using Reference Operator

When function parameter is declared as reference it provides following benefits:

- Overhead caused by passing the large data structure as a parameter is eliminated.
- It prevents the unnecessary copying of parameter to the stack.
- Unnecessary copying of returning reference from a function to the stack is avoided.
- De-referencing is not needed when function parameter is used.
- Called function operates on the caller's copy of the data. (call-by-reference)

Unit 2

```
#include <iostream>                      // Example 4-3
using namespace std;
void using_pointer(int* fruit);          // function prototype
void using_reference(int &fruit);
int main()
{
        int apple = 25;
        cout << apple << "\n";           // output 25
        using_pointer(&apple);           // passing the address of apple
        cout << apple << "\n";           // output 17
        int orange = 25;
        using_reference(orange);
        cout << orange << "\n";          // output 17
}

```

Unit 2

```
// formal parameter declared as pointer
void using_pointer(int * fruit)
{
        * fruit = 17;
}
// argument is declared to be a reference variable
void using_reference(int & fruit)
{
        fruit = 17;
}
```

Unit 3

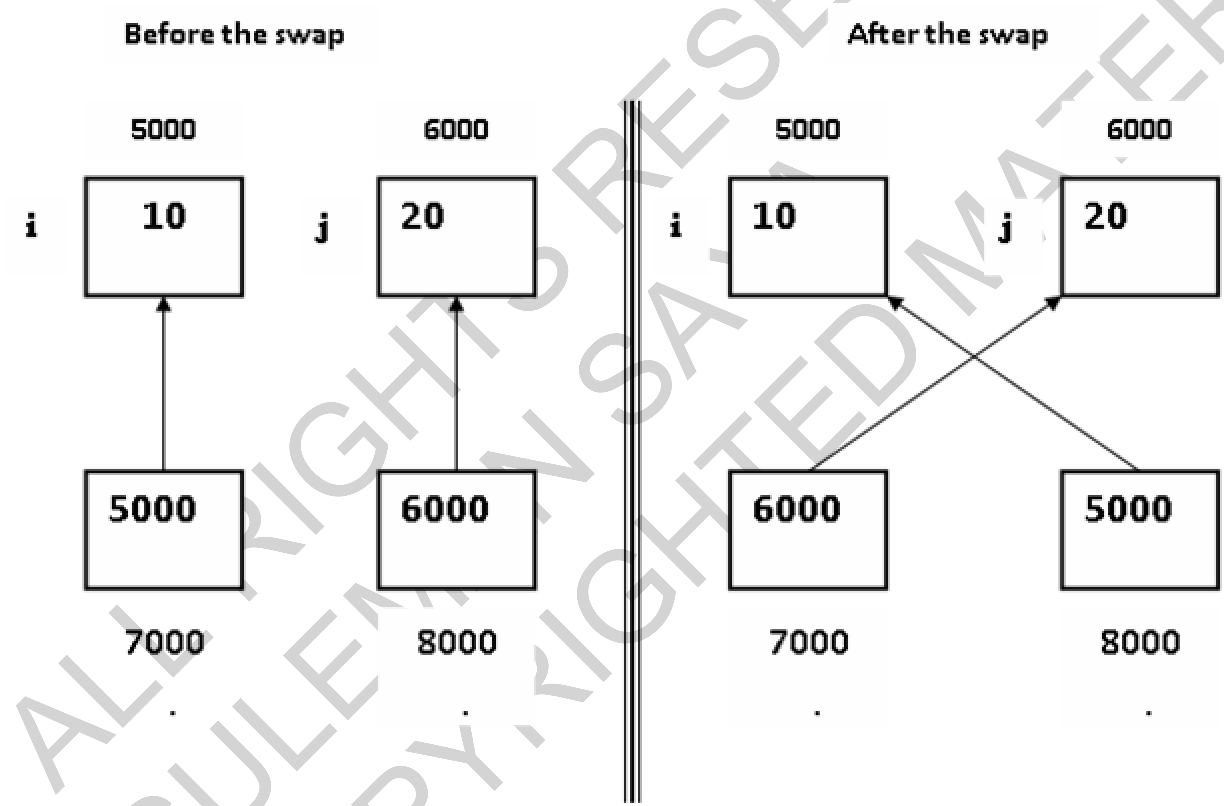
Pointer Swapping using reference

```
#include <iostream> // Example 4-4
using namespace std;
// Pointer argument declared as a reference, this will
// modify the pointer itself. num1 and num2 are the
// reference to a pointer to an object of type int
void swapPtr(int *&num1, int *&num2)
{
    int *temp = num2;
    num2 = num1;
    num1 = temp;
}

```

Unit 3

```
int main()          // Show pointer swapping
{
        int i = 10, j = 20;
        int *pi = &i, *pj = &j;

        cout << "Before swapping pointers:\tpi = ";
        cout << *pi << ";\tpj = " << *pj << "\n";
        swapPtr(pi,pj);
        cout << "After swapping pointers:\tpi = ";
        cout << *pi << ";\tpj = " << *pj << "\n";
}
```

Unit 3



Before the swap
5000
6000
i
10
j
20
5000
6000
7000
8000
After the swap
5000
6000
i
10
j
20
6000
5000
7000
8000

Unit 3

Pointer Swapping using Double pointers

```
#include <iostream>    // Example 4-5
using namespace std;
// Argument declared as a pointer to a pointer, this will
// modify the pointer itself num1 and num2 are the double
// indirect pointer to an int type
void swapPtr(int **num1, int **num2)
{
       int *temp = *num2;
       *num2 = *num1;
       *num1 = temp;
}
```

Unit 3

```
int main() // Show pointer swapping
{
    int i = 10, j = 20;
    int *pi = &i, *pj = &j;

    cout << "Before swapping pointers:\tpi = ";
    cout << *pi << ";\tpj = " << *pj << "\n";
    swapPtr(&pi,&pj);
    cout << "After swapping pointers:\tpi = ";
    cout << *pi << ";\tpj = " << *pj << "\n";
}
```

Unit 4

Const Specifier

- A const declaration of a variable forbids changes of the variable after its initialization.
- The const specifier makes the variable of any type read-only within its scope.
- Variable's value cannot be changed during program execution.

```
const int size = 10;
// size = 12; // violates const declaration
```

- A pointer can be declared const as well.
- The value of the pointer cannot change, but the value it refers to can change.
- A pointer can be declared to point to a constant value.

```
                    // the pointer,    the data it refers to
                    // -----------------------------------
      int*       p; // non-const       non-const
      int* const q; // const           non-const
const int*       r; // non-const       const
const int* const s; // const           const
```

Unit 4

```
#include <iostream>    // Example 4-6
using namespace std;
int main()
{
     const int Xdata = 50;
     const int* Xdata_ptr;
     Xdata_ptr = &Xdata;
     cout << "Xdata costant, but pointer non-const:" << Xdata << "\n";

     int Ydata = 100;
     // int* const Ydaya_ptr; // error: uninitialized const pointer
     int* const Ydata_ptr = &Ydata;
     cout << "Ydata non-costant, but pointer const:" << Ydata;
}
```

Unit 4

Constant Array Elements

```
#include <iostream> // Example 4-7
using namespace std;
int main()
{     // illegal to alter any values of the data
      const float data[] = {1.1, 2.2, 3.3, 4.4, 5.5};
      // ERROR!! can not modified value; lvalue specifies const
      data[1] = 7.7;
      for (int index=0; index < sizeof(data)/4; index++)
      cout << data[index] << '\n';
}
```

Unit 5

Constant Objects

- Constant object value can not be changed.
- Constant objects can only access constant member functions.
- Create a constant member function by placing a keyword const after the arguments.
- Constant member function can be called through a non-constant object.
- Constructors and destructors cannot be declared as constant member functions.

```
#include <iostream>    // Example 4-8
using namespace std;
class CApple {
 public:
        CApple(int i) { m_nData = i; }
        void add(int i) { m_nData = i + 10; }
        int GetData() const { return m_nData; }
 private:
        int m_nData;
 };
```

Unit 5

Constant Objects

```
int main()
{
    CApple Washington(60);
    Washington.add(10);

    // non-constant object can call constant member function
    int k = Washington.GetData();
    cout << "The Washington object Value is " << k << "\n";

    // Declare and initilize constant object
    const CApple Macintash = 42;

    // Compiler error [object is constant can not be changed]
    Macintash.add(17);

    // constant object calls constant member function
    int count = Macintash.GetData();
    cout << "The Macintash object Value is " << count << "\n";
}
```

Unit 6

Member initialization List

- Initialize list is used for initialization of data members of a class
- Data members to be initialized are indicated with constructor as a comma separated list followed by a colon
- Reference data members must be initialized using initializer list
- Const data members must be initialized using initializer list
- Initialize all class data members in initializer list instead of assigning values inside constructor body.

```
#include <iostream>    // Example 4-9
using namespace std;
class CMatrix {
 public:
     CMatrix(int s);
     ~CMatrix();
     void FillArray();
     void Display();
 private:
     int **m_pnMatrix;
     const int m_nSize;
};

```

Unit 6

Member initialization List

```
CMatrix::CMatrix(int s) : m_nSize(s), m_pnMatrix(0)
{
     m_pnMatrix = new int*[m_nSize];
     for (int x = 0; x < m_nSize; x++)
         m_pnMatrix[x] = new int [m_nSize];
}

CMatrix::~CMatrix()
{
     for (int x = 0; x < m_nSize; x++) delete m_pnMatrix[x];
     delete [] m_pnMatrix;
}

void CMatrix::FillArray()
{
     for (int x = 0; x < m_nSize; x++)
        for (int y = 0; y < m_nSize; y++)
           *(m_pnMatrix[x] + y) = y;
}
```

Unit 6

```
void CMatrix::Display()
{
     for (int x = 0; x < m_nSize; x++) {
        for (int y = 0; y < m_nSize; y++)
           cout << (*(m_pnMatrix[x] + y)) << " ";
     cout << "\n";
     }
}

int main()
{
     CMatrix m(4);
     m.FillArray();
     m.Display();
}
```

Unit 7

Copy Constructor

- Constructor with reference type argument is called copy constructor.
- The new object is initialized as a copy of some already existing object.
- The default copy constructor is created and initialized for the following cases:
  - When one object is declared and assigned to another object of same type.
  - When object is used as pass by value to a function arguments.
  - When object is returned by value from a function call.

```
#include <iostream>   // Example 4-10
class CHouse {
        public:
            CHouse() { nBedRooms = nBathRooms = 0; }
            CHouse(int nBed, int nBath);
            CHouse(CHouse &Obj);
             void Show();
        private:
             int nBedRooms;
             int nBathRooms;
};
```

Unit 7

```
CHouse::CHouse(int nBed, int nBath)
{
    nBedRooms = nBed;
    nBathRooms = nBath;
}
CHouse::CHouse(CHouse &Obj)
{
    nBedRooms = Obj.nBedRooms;
    nBathRooms = Obj.nBathRooms;
}
void CHouse::Show()
{
    cout << "Total Bed Rooms = " << nBedRooms << "\n";
    cout << "Total Bath Rooms = " << nBathRooms << "\n";
}
int main()
{
    CHouse h1(3,1);
    CHouse h2 = h1;
    h2.Show();
}
```

Unit 7

- **Allocate memory Using Copy constructor**

```
/* This program creates a "safe" array class. Since space
* for the array is dynamically allocated, a copy constructor
* is provided to allocate memory when one array object is
* used to initialize another. */
#include <iostream>    // Example 4-11
#include <cstdlib>
using namespace std;

class CArray {
   public:
CArray(int sz);       // Normal constructor
~CArray() {delete [] m_ptr; }
CArray(CArray& a);
int get (int i) { return m_ptr[i]; }

   private:
int *m_ptr;
const int m_size;
};
```

Unit 7

```
int main()
{
        CArray redLED(10);
        for (int i = 9; i >= 0; i--) cout << redLED.get(i);
        cout << "\n";

        CArray greenLED = redLED;
        for (int i = 0; i < 10; i++) cout << greenLED.get(i);
        cout << "\n";
}

void CArray::put(int i, int j)
{
        if (i >= 0 && i < m_size) m_ptr[i] = j;
}
```

Unit 7

```
CArray::CArray(int sz) : m_size(sz), m_ptr(0)
{
      m_ptr = new int[sz];
      if (!m_ptr) exit(1);
      cout << "Using 'normal' constructor\n";
      for (int i = 0; i < sz; i++) put(i, i);
}

CArray::CArray(CArray& a) : m_size(a.m_size), m_ptr(0)
{
      m_ptr = new int[m_size]; // allocate memory for copy
      if (!m_ptr) exit(1);
      for (int i = 0; i < m_size; i++) m_ptr[i] = a.m_ptr[i];
      cout << "Using copy constructor\n";
}
```

OUTPUT:
Using 'normal' constructor
9876543210
Using copy constructor
0123456789

Unit 8

Static Data Members

- Normally each object receives a copy of all data members
- Objects can be created in a way that all data members are shared
- Use static keyword before data member declaration.
- Only one copy of data member exists.

```
#include <iostream>    // Example 4-12
using namespace std;

 class CComputer {
     public:
           void Env(int b) { m_nBuffers = b; }
           int Display() { return(m_nBuffers); }
     private:
           static int m_nBuffers;
};
```

Unit 8

```
// m_nBuffers is still private to a Ccomputer class
int CComputer::m_nBuffers;

int main()
{
    CComputer Config, Windows;

    Config.Env(10);

    cout << "Config object can see m_nBuffers value as: ";
    cout << Config.Display() << "\n"; // display 10
    cout << "Windows object can see m_nBuffers value as: ";
    cout << Windows.Display() << "\n"; // also display
}
```

OUTPUT:

Config object can see m_nBuffers value as: 10

Windows object can see m_nBuffers value as: 10

Unit 8

Access Static Data Member Independently

- Use scope resolution operator to access static data member independently of any objects.
- All static data member are initialized to zero by default.
- Reason for static data members is to prevent the use of the global variables.
- Classes that rely upon global variables always violate data hiding rules.
- Static data member exist before any object is created

```
// Reference a static independent of any object.
#include <iostream>    // Example 4-13
using namespace std;

class CComputer {
      public:
            static int m_nBuffers;
            int Display() { return m_nBuffers; }
};
```

Unit 8

```
int CComputer::m_nBuffers;
int main()
{
      CComputer Config, Windows;

      // set buffers directly
      CComputer::m_nBuffers = 100; // no object is referenced.

      cout << "CComputer::m_nBuffers ";
      cout << CComputer::m_nBuffers << "\n";
      cout << "Config object can see m_nBuffers value as: ";
      cout << Config.Display() << "\n";
      cout << "Windows object can see m_nBuffers value as: ";
      cout << Windows.Display() << "\n";
 }
```

OUTPUT:

CComputer::m_nBuffers 100
Config object can see m_nBuffers value as: 100
Windows object can see m_nBuffers value as: : 100

Unit 9

Static Member Functions

- Static member function can be called without referencing an object.
- Static member function does not have the "**this**" pointer.
- It can not be used to call non-static member functions.
- Only static data member can be manipulated by static member functions.
- Static function belongs to the class rather than to an object of the class.

```
#include <iostream>    // Example 4-14
using namespace std;
class CStat {
public:
     static void Increment() { m_Number++; }
     void Show() { cout << m_Number << "\n"; }
private:
     static int m_Number;
};
```

Unit 9

```
int CStat::m_Number = 57;
int main()
{
    CStat alpha;

     alpha.Show();
     CStat::Increment();
     alpha.Show();
 }
```

OUTPUT:

57

58

Unit 10

Design Pattern Using Static Member Function

- Design patterns provide general solutions that is not tied to a particular problem.
- The Singleton Design pattern is used, where only one instance of an object is needed throughout the lifetime of an application.
- The Singleton object is often used for controlling the access to resources such as database connections or sockets. (license for only one database connection)
- By declaring a class that contains a static member function we can get a single object.

```
#include <iostream>    // Example 4-15
using namespace std;
class Singleton {
  public:
        static Singleton* GetInstance();
        void show() { cout << "Single object\n"; }
  private:
        Singleton();
        static Singleton* pSingleton;  // singleton instance
};
```

Unit 10

Design Pattern Using Static Member Function

```
Singleton* Singleton::pSingleton= 0;  // assignment NULL
Singleton::Singleton()
{
   // do init stuff
}

Singleton* Singleton::GetInstance()
{
   if (pSingleton== NULL) {
      pSingleton = new Singleton();
   }

  return pSingleton;
}

int main()
{
    Singleton::GetInstance()->show();
}
```