➢ **Data Member** declared as constant.

- Data member of a class can also be declared as constant…

```
// Using a member initializer to initialize a constant of a built-in
// data type.
#include <iostream.h>
class Increment {
 public:
  Increment( int c = 0, int i = 1 );
  void print() const;
 private:
  int count;
  const int incVal;     // const data member
};
// Constructor for class Increment with initializer for const member
Increment::Increment( int c, int i) {
 count = c;
 incVal = i;[JAN1]
}
// Print the data
void Increment::print() const {
 cout << "count = " << count
      << ", incVal = " << incVal << endl;
}
```

Syntax Error

```
void main() {
 Increment value( 10, 5 );
 value.print();
}
```


- Constant data members of an object, must be given a value at **construction time**…
- A **member initializer** is used to provide the constructor with a value of that *const data member* of the *declared object*…

```
// Using a member initializer to initialize a constant of a built-in
// data type.
#include <iostream.h>
class Increment {
 public:
  Increment( int c = 0, int i = 1 );
  void addIncrement() { count += incVal; }
  void print() const;
 private:
  int count;
  const int incVal;     // const data member
};
// Constructor for class Increment with initializer for const member
Increment::Increment( int c, int i) : incVal( i)[JAN2]
{
 count = c;
}
// Print the data
void Increment::print() const {
 cout << "count = " << count
      << ", incVal = " << incVal << endl;
}
```



```
// Using a member initializer to initialize a constant of a built-in
// data type.
#include <iostream.h>
class Increment {
 public:
  Increment( int c = 0, int i = 1 );
  void addIncrement() { count += incVal; }
  void print() const;
 private:
  int count;
  const int incVal;     // const data member
};
// Constructor for class Increment with initializer for const member
Increment::Increment( int c, int i) : incVal( i)[JAN3]
{
 count = c;
}
// Print the data
void Increment::print() const {
 cout << "count = " << count
      << ", incVal = " << incVal << endl;
}
```

```
void main() {
 Increment value( 10, 5 );
 cout << "Before incrementing: ";
 value.print();
 for ( int j = 0; j < 3; j++ ) {
  value.addIncrement();
  cout << "After increment " << j + 1 << ": ";
  value.print();
 }
}
```


- Multiple member initializers …

```
#include <iostream.h>
class More {
 public:
  More ( int c = 0, int i = 1, int j = 2 );
  void Print() const {
   cout << count << " " << Val_1 << " "
        << Val_2 << " ";
  }
 private:
  int count;
  const int Val_1;
  const int Val_2;
};
// Constructor for class Increment with multiple initializer.
More::More( int c, int i, int j) : Val_1(i),
Val_2(j)[JAN4] {
 count = c;
}
```

```
void main() {
 More One( 10, 5, 3 );
 One.Print();
}
```

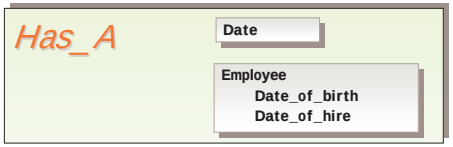

- Any data members can be initialized using member initializer syntax

```
#include <iostream.h>
class All {
 public:
  All ( int c, int i, int j, int k);
  void Print() const {
   cout << count << " " << Ndx  << " "
    << Val_1  << " "  << Val_2;
  }
 private:
  int count;
  int Ndx;
  const int Val_1;
  const int Val_2;

};

All::All( int c, int i, int j, int k)
          : Ndx (k), Val_1( i ), Val_2( j )[JAN5]
{
 count = c;
}
```

```
void main() {
 All One( 10, 5, 3 ,1);
 One.Print();
}
```


Composition: Objects as Members of Classes

- Sometime, a class data member is an object of another class… This is **Composition**.



- **Example:**

Private data member as an object of another class.

```
class cA {
  public:
    cA(int j=1) {
      setX(j);
    }
    void setX(int i) { x=i;}
    int getX() { return x; }

  private:
    int x;
};
```

```
class cB {
  public:
    cB(int j=2, int k=3) : objA(k)[JAN6] {
      setY(j);
    }
    void setY(int i) { y=i; }
    int getY() { return y;}
    int getObj() { return objA.getX(); }

  private:
    int y;
    cA objA;
};
```

```
void main() {
  cA a1;
  cB b1;
  cout << a1.getX() << endl;
  cout << b1.getObj() << endl;
  cout << b1.getY() << endl;
}
```


- **Example:**

Public data member as an object of another class.

```
class cA {
  public:
    cA(int j=1) {
      setX(j);
    }
    void setX(int i) { x=i;}
    int getX() { return x; }
  private:
    int x;
};
```

```
class cB {
  public:
    cB(int j=2, int k=3) : objA(k) {
      setY(j);
    }
    void setY(int i) { y=i; }
    int getY() { return y;}
    int getObj() { return objA.getX(); }
    cA objA;
  private:
    int y;
};
```

```
void main() {
  cA a1;
  cB b1;
  cout << a1.getX() << endl;
  cout << b1.getY() << endl;
  b1.objA.setX(30);[JAN7]
  cout << b1.objA.getX() << endl;
  cout << b1.getObj() << endl;
}
```


- **Example:**

- Member objects are constructed in order of their declaration in the class …(Not in order of constructor's member initializer list)
- Inner objects are constructed first and destructed last…

```
class cA {
  public:
    cA[JAN8](int j=1) {
      setX(j);
      cout << "Constructor class cA:" << x << endl;
    }
    ~cA[JAN8]() {
      cout << "Destructor class cA:" << x << endl;
    }
    void setX(int i) { x=i;}
  private:
    int x;
};
```

```
class cB {
  public:
    cB[JAN8](int j=2, int k=3, int m=5):objA2(m), objA1(k){
      setY(j);
      cout << "Constructor class cB:" << y << endl;
    }
    ~cB[JAN8]() {
      cout << "Destructor class cB: << y << endl;
    }
    void setY(int i) { y=i;}
  private:
    int y;
    cA objA1;
    cA objA2;
};
```

```
void main() {
  cB b1;
  cout << "Order of construction/destruction\n";
}
```


friend Functions and **friend** Classes

- A **friend function** of a class is defined outside that class's scope, yet has the right to access private and protected members of the class…
- A **friend class** is a class whose member functions have access to the other class's private and protected members…
- Friendship is *granted, not taken*…
- *Not symmetric*… If **B** a friend of **A**, **A** not necessarily a friend of **B**…
- *Not transitive* … if **A** a friend of **B**, **B** a friend of **C**, **A** not necessarily a friend of **C**…

- **friend function**

```
// fig07_05.cpp
// Friends can access private members of a class.
#include <iostream.h>
// Modified Count class
class Count {
  friend [JAN12]void setX( Count &[JAN13], int ); // friend declaration
  public:
    Count() { x = 0; }       // constructor
    void print() const { cout<< x <<endl; }
  private:
    int x;  // data member
};

// Can modify private data of Count because
// setX is declared as a friend function of Count
void setX( Count &c, int val ) {
  c.x = val;                 // legal: setX is a friend of Count
}

int main() {
  Count counter;
  cout << "counter.x after instantiation: ";
  counter.print();
  cout << "counter.x after call to setX friend function: ";
  setX( counter, 8 );  // set x with a friend
  counter.print();
  return 0;
}
```


- **friend class**

```
#include <iostream.h>
class A {
  friend class B;[JAN14]
  public:
    A (int a) { setA(a);}
    void setA (int a) { valA = a; }
    void printA() { cout <<"valA is " <<valA <<endl;}
  private:
    int valA;
};
class B {
  public:
    B (int b) { setB(b);}
    void setB (int b) { valB = b; }
    void incA( A &[JAN15] a ) {a.valA++;}
    void printB() { cout <<"valB is " <<valB <<endl;}
    void printAnB(A &[JAN16] a) {
      cout <<"valA printed in B is " <<a.valA << endl;
      cout << "valB is " << valB << endl;
    }
  private:
    int valB;
};
void main() {
  A a1(10);
  B b1(5);
  a1.printA();
  b1.printB();
  b1.incA(a1);
  b1.printAnB(a1);
}
```

Using the **this** Pointer

- Allows objects to access their own address…
- It is not part of the object itself… Thus it is not reflected in the results of a sizeof operation on the object…
- When a **nonstatic member function** is called for an object, the *address of the object* is passed (*By the compiler*) as a hidden argument to the function… The **object's address** is available from within the member function as the ***this*** pointer…

```
void Date::setMonth( int mn ) {
   month = mn;              // These three statements
   this->month = mn[JAN17];       //     are equivalent
   (*this).month = mn[JAN18];
}
```

- The type of the ***this*** pointer depends upon the type of the object and whether the member function using ***this*** is ***constant*** or ***non-constant***…

object *** const**
const ***object*** *** const**


```
// fig07_07.cpp: Using the this pointer to refer to object members.
#include <iostream.h>
class Test {
 public:
  Test( int = 0 );
  void setX( int);
  void print() const;
 private:
  int x;
};
Test::Test( int a ) { x = a; }

void Test::setX( int a) {
 x = a;
 this[JAN19]->x =a;
 ( *this ).x = a;
}
void Test::print() const {
 cout << "        x = " << x
   << "\n  this[JAN20]->x = " << this->x
   << "\n(*this).x = " << ( *this ).x << endl;
}
void main() {
 Test testObject( 12 );
 testObject.print();
 testObject.setX( 50 );
 testObject.print();

}
```

Test * const

const Test * const

- The ***this*** pointer can be used for ***Cascaded member function calls***…
- When a function returns a reference pointer to the same object

```
return *this;
```

Other functions can operate on that pointer…

```
#include <iostream.h>
class A {
 public:
  A (int a) { setA(a);}
  A &[JAN21] setA (int a) {
   valA = a;
   return *this[JAN22];
  }
  A & incA() {
   valA++;
   return *this;
  }
  void printA() {
   cout <<"valA is " <<valA <<endl;
  }
 private:
  int valA;
};
void main() {
 A a1(10);
 a1.printA();
 a1.setA(20).printA();[JAN23]
 a1.setA(50).incA().printA();[JAN24]
}
```



```
#include <iostream.h>
class A {
 public:
  A (int a) { setA(a);}
  A & setA (int a) {
   valA = a;
   return *this;
  }
  A *[JAN25] incA() {
   valA++;
   return this[JAN26];
  }
  void printA() {
   cout <<"valA is " <<valA <<endl;
  }
 private:
  int valA;
};
void main() {
 A a1(10);
 a1.printA();
 a1.setA(20).printA();
 a1.setA(50).incA()->printA();
}
```


Dynamic Memory Allocation with Operators **new** and **delete**

- The ***new*** and ***delete*** operators provide a superior means to allocate memory than the " ***ANSI C*** " ***malloc*** and ***free*** …
- **ANSI C** :

```
TypeName *typeNamePtr;
...
typeNamePtr = malloc ( sizeof ( TypeName ) );[JAN27]
...
free ( typeNamePtr );
...
```

- **C++** :

```
TypeName *typeNamePtr;
...
typeNamePtr = new TypeName;[JAN28]
...
delete typeNamePtr;
...
```

- Creates an object of the proper size, calls its constructor and returns a pointer of the correct type…
- If there is insufficient memory for the allocation request, it returns **NULL** (0) pointer…


```
#include <iostream.h>
#include <string.h>
void main () {
   int *iPtr;
   float *fPtr ;
   char *cArrPtr ;
   fPtr = new float (3.1416);
   iPtr = new int;
   *iPtr = 129;
   cout << iPtr << " - " << *iPtr <<endl;
   delete iPtr;
   iPtr = new int [4];
   iPtr[2]=10;
   cout << iPtr[2] << endl;
   cArrPtr = new char [30] ;
   strcpy (cArrPtr,"This is a test");
   cout << cArrPtr << endl;
   delete [ ] cArrPtr;[J29]
   delete fPtr;
   delete [ ] iPtr;
}
```

- Space created with ***new*** **should not** be released (freed) with ***free*** and space created with ***malloc*** **should not** be released with ***delete***…
- Since C++ programs can contain storage created with ***malloc*** / ***free*** and ***new*** / ***delete***… It is best to use only ***new*** / ***delete*** ...

- Using ***new*** to dynamically create an object of a class will automatically invokes the class's constructor… ***delete*** will automatically invokes the class's destructor…

```
#include <iostream.h>
class A {
   public:
      A (int a=0) {
         setA(a);
         printA();
      }
      void setA (int a) {   valA = a; }
      void printA() {
         cout <<"valA is " <<valA <<endl;
      }
      ~A () {
         cout << "Destructor ";
         printA();
      }
   private:
      int valA;
};
void main () {
   A *aPtr;
   aPtr = new A (10);
   delete aPtr;
   aPtr = 0;[I30]
   aPtr = new A [2];[JAN31]
   delete [ ] aPtr;
   aPtr = 0;
}
```


static Class Members

- Each object of a class has its own copy off all the data members of the class…
- To share a data member by all object of a class, we need to use a ***static*** class member…

```
#include <iostream.h>
class A {
   public:
      A (int a) {
         setA(a);
         incCntA();
      }
      void setA (int a) {   valA = a; }
      void incCntA () {countA++; }
      void decCntA () {countA--; }
      void printA() {
         cout <<"valA is " <<valA <<endl;
         cout <<"countA is " <<countA <<endl;
      }
      ~A () {
         cout << "Destructor ";
         decCntA();
         printA();
      }
   private:
      int valA;
      static int countA;[JAN32]
};
```


- Efficient when a single copy of data is enough … Only the static variable has to be updated
- May seem like global variables, but have class scope… only accessible to objects of same class
- Must be initialized once (only) at file scope… Otherwise, it is a syntax error… (Linker)

```
int A::countA = 0;[JAN33]
void main () {
   A a1( 10 ), a2( 20 );
   a1.printA();
   cout << " Out of main…" << endl;
}
```

- Static data members exist, even if no instances (objects) of the class exist…

- Both variables and functions can be static… They can be ***public***, ***private*** or ***protected***…
- ***public static*** variables are accessible through any object of the class… They can also be accessed thought the class name using the binary scope resolution operator…

```
#include <iostream.h>
class A {
   public:
      static int countA;
      A (int a) { setA(a); }
      void setA (int a) {   valA = a; }
   private:
      int valA;
};
int A::countA=0;
void main () {
   A a1(10);
   A a2(20);
   cout << a1.[I34]countA << endl;
   cout <<a2.countA << endl;
   A::[I35]countA=10;
   cout <<A::countA << endl;
   cout <<a2.[I36]countA << endl;
}
```


- ***private*** and ***protected static*** variables must be accessed through a ***public member function…***
- ***static member functions*** **cannot access** ***non-static data or functions***…
- ***public static member function*** can be called thought by the class name using the binary scope resolution operator…

```
#include <iostream.h>
class A {
   public:
      A (int a) { valA = a; }
      void incCntA () {countA++; }
      static void decCntA () {
         countA--;
         countA = valA + 1;
      }
   private:
      int valA;
      static int countA;
};
int A::countA=0;
void main () {
   A a1(10);
   a1.incCntA();
   A::decCntA();
}
```

Syntax Error