Objects

Objects

- An object is :
  - representation of an entity in the system.
  - not necessarily something that you can see or touch.
  - something that has state and behavior.
  - an instance of a class.
  - something that performs an action.

Objects

- Example:
  - Circle c; // c is an object of type Circle
  - Square s; // s is an object of type Square

  This means that Circle and Square are classes.

  In C, you would say something like

  - struct Circle c; or struct Square s;

  Complex c; // c is an object of type Complex.

- Every class usually defines an interface and attributes. So, an object has an interface for communication with other objects.

State

- The state of an object at any given instant in time is the properties (attributes) of the object and the value of those attributes.

- Example:

  - Stack object may have some of the following attributes:
    - number of elements currently in the stack
    - size of the stack (max. number of elements it can hold)

  - Circle object's attributes:
    - coordinates of the center
    - radius
    - color

Behavior

- Behavior of an object refers to how the object reacts when it gets a message.
- An object may or may not change its state when a message is sent to it.
- The state of an object at any given instant is in a sense the cumulative result of its behavior from instantiation to that instant.
- Example:

```
Circle c(0, 0, 10);      // center is at origin, radius is 10 units
c.getRadius();           // will return 10
c.resize(15);            // ask it to resize radius to 15 units
c.getRadius();           // now returns 15
```

Behavior

- Different classes can define the same behavior.
- Example :
  - Circle and Square can derive from an abstract class (say Shape class) and would have to implement the interface of the Shape class.
  - This can be methods (messages) like
    - Draw()
    - GetColor()
    - GetArea()
    - GetPerimeter()

Class

- A class is the definition of what its objects will have in terms of attributes and behavior.

- So, a class describes objects that will share a common structure and behavior. (Similar in that sense to a struct / record that describes variables that will share a common structure).

- Looking at a class declaration, one can tell
  - what attributes objects of that class will have
  - what messages they will respond to
  - which class(es) they derive from (i.e., who is/are the parent(s))
  - But you cannot tell which classes will derive from it

Class

- Class is similar to struct in C.
- Except that in C++, a struct can have methods in addition to data, and can specify access specifiers.

- Example:
  - struct Student {

```
        char  name[ MAX_NAME_SIZE];
        char  address[ MAX_ADDRESS_SIZE ];
        // etc.

        char *          getName() ;
        char *          getAddress() ;
        // etc.
    };
```

Class Interface

- This is also called the API or public interface.
- Interface can be public or protected. (private methods can only be used by the class itself).
- Protected interface is how communication is done with derived classes.
- Public interface is how communication is done with everyone else. (Derived classes can also use public interface of parent, since anyone can use the public interface).

- So, the interface is essentially "what the class does", i.e., its behavior or its external view.

- *Example* : Stack has "push", "pop", "isEmpty" on its interface
  Circle has "Draw", "GetArea", etc. on its interface

Implementation

- Implementation is an internal view.
- This is essentially the "how it does what it does".
- This is hidden from the "rest of the world".

- Separation of the class interface from its implementation is a great feature in object oriented programming. It reduces the coupling between various classes.

Interface, Implementation

- Example:
  - A Stack class may be implemented using an *array*, and later on the implementation is changed to use *lists*. As long as the interface remains the same, the client code need not change.
  - Stack s;
  - s.push(4);
  - s.push (5);
  - intValue = s.pop(); // returns 5
  - intValue = s.pop() // returns 4

  Change in implementation of Stack class does not affect above code.

  Assumption is that interface signature as well as semantics do not change (since it is a change in implementation only).

Hello world

```
#include <ostream.h>
int main()
{
        cout << "Hello, world !!!" << endl;    // endl is new line

        return 0;
}
```

- This is a simple C++ program.

- The only object used here is cout.
- cout is the C++ equivalent of stdout in C.
- stdout is available in C++ also.

Class members

- A class has two types of members :
  - data members and
  - method members

- Data members : attributes/properties, this is what helps maintain an object's state.

- A data member can be:
  - public (anyone can access this data member)
  - protected (only the owning class and its derived classes can access this data member)
  - private (only the owning class can access this data member)

Method Members

- Method members:
  - These are the messages that can be sent to an object, in other words, functions that can be called on an object.

- A class method can be:

  - public (anyone can call this method on the object)
  - protected (only the owning class and derived classes can call this method on the object)
  - private (only the owning class can call this method)

Class Members

- A member can be static, which means that it is associated with the class, and not any specific instance of the class, i.e., not with any specific object.
- A member can be associated with a specific object, i.e., it is not static. These types of members are the ones most commonly used.
- Non static data members are what constitute the state of an object.
- Non static method members constitute the behavior of the object.

Static Data Members

- Static data member means that there is only ONE instance of that member throughout the lifetime of the program.
- When any object accesses it, the same copy/instance of the static data is accessed. (As a result, in a multi-threaded application, you should be careful writing to static data members. Either make them read only, or use critical sections)
- For example, a static data member can be used to keep track of the number of instances of a class, or the number of active instances of a class (or both, if you have two static data members).
- Example :
  - ```
    class Stack
    {
        ...
        static int m_MaxPossibleStackSize;
    }
    ```
    ```

Static Method Members

- Static methods are not bound to any object, i.e., there is no *this* pointer within a static method since it is not called on any object.

- Calls to a static method are qualified with the class name if called from outside the class.

- Example:
  - cout << Stack::GetMaxPossibleStackSize() << endl;
  - Stack s1(100);
  - cout << s1.GetMaxPossibleStackSize() << endl; // this is OK too
  - First form (using class name qualification) should be preferred.

Non-static data members

- Class members that are not static are associated with a specific instance of the class.
- These data members represent the attributes of an object. The state of an object is represented by the values of the data members.
- Example:
  - class Stack
    {
      public:
        int GetSize() const;
        bool IsFull() const;
        void Push(int value);

      private:
        int m_Size;
    }

Non-static method members

- These represent the behavior of an object.
- They may internally use the state of the object to determine the behavior.
- Example:
  - IsFull(), Push() are method members on the Stack class.
  - These methods can be called on an object of the Stack class.
  - They cannot be called using the "Stack::" qualifier like we used for calling static members.
  - All such methods have a *this* variable.

Access specifiers

Following apply to data as well as method members, static as well as non-static members, virtual as well as non virtual members.

- *public* : Public members can be accessed by everybody.
- *protected*: Protected members can be accessed only by sub classes.
- *private:* Private members can only be accessed by the same class and classes or methods declared as *friend* of the class.

- Usually should not have protected data, instead provide protected access and set methods for it and make the data itself private.

- Usually should not have public data.

new operator

- The new operator is used to create an object on the heap. It allocates the memory required for an object, and then calls the constructor for initializing the object.
- Compare this with malloc() in C. New is a lot more powerful than malloc().
- *new* is not a function, it is an operator.
- Return value is a pointer to the dynamically created object, just like in case of malloc().
- Example:
  - MyClass * x = new MyClass();
  - int * intPtr = new int;

delete operator

- The *delete* operator is used to delete an object.
- It calls the destructor of the object before de-allocating the memory.
- Objects created dynamically (i.e., on the heap, using the new operator) must be deleted by calling the delete operator on them. Otherwise there will be a memory leak (no garbage collection in C++).
- Deleting a null pointer is a no-op, i.e., nothing happens.
- An array should be deleted using delete [ ].

delete operator

- Example 1:

```
MyClass  * array = new MyClass[ 5 ];
...
delete [ ] array;              // and not just "delete array"
```

- Example 2:

```
int * intArray = new int [300];
...
delete [ ] intArray;
```

- Example 3:

```
int * iPtr = new int(100);     // Here, we are not allocating
 ...                           // any array, we allocate only
                               // one integer.
delete  iPtr;
```

Constructor

- Constructor is
  - A special method that is called once in the lifetime of an object (and is the very first method called on it automatically).
  - It is typically used to initialize the state of the object, i.e., put the object in its initial state.
  - Has no return value. Cannot even specify void.
  - Has the same name as the class.
- A class can define more than one constructor (or none). Which one gets called depends on the arguments supplied at the time of object creation.

Various types of constructors

- C++ has the following types of constructors that you can define for your class :

  - Default constructor. Does not need user provided arguments, i.e. can be called without any arguments.
  - Conversion constructor. Takes one or more arguments.
  - Copy constructor. Specifies how the class developer wants to create an object of a class A from an existing object of class A, i.e., how to create a copy of an object.

Default constructor

- Default constructor
  - This is the constructor that does not need any user provided arguments. It will usually cause some default initialization.
  - string s1; // default constructor used, create empty string.
  - Stack stack1; // Stack::Stack(int size = 10); size defaults to 10
    - compare above with :
      - Stack stack2(25), i.e. create a stack with size 25

Default constructor

- More examples of default constructors :
  - Complex::Complex(double real = 0.0, double imaginary = 0.0)

    Note the above constructor uses *default* arguments.

    - Can be used as
      - » Complex c1; // real and imaginary are 0.0
      - » Complex c2(25); // real = 25, imaginary = 0.0
  - SomeClassA::SomeClassA() // no arguments
  - Stack::Stack(int size = kDefaultSize) // kDefaultSize is const

Default constructor

- NOTE: Once you have an argument that takes a default value, all arguments after that (if any) have to have a default value.

- Example:
  - We cannot have the following :
    - Complex::Complex(double real = 0.0, double imaginary)
      - Either *imaginary* should have a default value also
      - OR
      - *real* cannot have a default value.
    - void Foo (int arg1, int arg2 = 100, int arg3)
      - arg2 cannot have a default value or arg3 should too.

Conversion constructor

- Conversion constructor
  - This takes one or more arguments.
  - Example:
    - RationalNumber(int numerator, int denominator)
    - string (const char *)
    - Employee(const char * lastName, const char *firstName)
    - Employee(const string & lastName, const string & firstName)

Conversion constructor

- Given two classes SmallInt and Number with constructors :
  - SmallInt::SmallInt(int x);
  - Number::Number (const SmallInt & x);

- See this usage :
  - SmallInt si(100); // create a SmallInt
  - Number n1 (10), n2 (si);

  - Conversion constructor is called for n1 and n2. For n1, the integer 10 is first converted to a SmallInt, creating a temporary SmallInt object, and then the constructor of Number is called.

Conversion constructor usage

- Following forms are generally equivalent :
  - MyClass mc1 ("param1");
  - MyClass mc2 = MyClass ("param1");
  - MyClass mc3 = "param1";
  - Form used in mc1 is recommended.
  - Form used in mc3 can be used only when there is a single argument.

Copy constructor

- Copy constructor is used to create a new object based on the values of an existing object.

- Example :
  - string::string(const string & other)
  - Complex::Complex(const Complex & other)

- If a class does not provide a copy constructor, the compiler generates a copy constructor, which simply does a bit wise copy of each of the data members (not static) of the existing object to the one being created.

  This is a *shallow* copy (this term applies to pointers). See Example →

Example of shallow copy

```
class classA  {
      public:
                  classA ( const char * str );          // conversion
                  classA ( const classA &  other );     // copy
                  ~classA ();                           // destructor
                  ...
      private:
                  char * m_Str;
 }

 classA::classA ( const char * str )
   : m_Str (NULL)
 {
    if ( str)
    {
        m_Str = new char [ strlen (str) + 1 ];          // allocate memory
        strcpy ( m_Str,  str );
    }
 }

 classA::classA ( const classA & other )
 {      m_Str  =  other.m_Str;  }                      // shallow copy, no memory
                                                       // allocation

 classA::~classA()
 {      delete [] m_Str;        }                  // release the memory allocated
```

Copy constructor

- If the compiler generated copy constructor is not sufficient/appropriate, then the class developer must provide a copy constructor.

- A scenario where the above is true :

  - a class does some memory allocation in its constructor, and then deletes this memory in the destructor. If the compiler generated copy constructor is used, then at some point, one of the objects can be pointing to deleted memory.

    - See example of above on next slide

Copy constructor

- Lets say there is a ClassA that :
  - allocates memory in its constructor,
  - deletes this memory in the destructor,
  - does NOT provide a copy constructor.

- Lets see the following segment of code :

```
ClassA    * a1 = new ClassA( "Hello");      // memory allocated in constructor
ClassA    * a2 = new ClassA( *a1 );         // member pointer is shallow copied
                                            //  into a2
delete  a1;
```

After the delete statement, a2 will have a data member pointing to deleted memory

Copy constructor

- Some classes do not want a copy constructor to be used. The solution here is to declare it as private, and not to simply omit the declaration.

- In this case, the assignment operator should also be declared as private, as the compiler will generate a default assignment operator if the class does not provide one.

- Note that the argument to a copy constructor should be passed by reference (usually a const reference), but cannot be passed by value. Why?

A note on default constructor

- If you do not provide **any** constructor, the compiler will provide a default constructor that takes no arguments. This one will not do any member initialization.

- If you write any constructor at all, the compiler doesn't provide the default constructor. And if you need a default constructor, then you have to write it.

Initializing within constructor

- The purpose of the constructor is to initialize the object being created to a defined initial state.

- Order of calling of constructors for a derived class is top - down, i.e., the root class constructor is called first, then its child, and so on.
- Example: classC --> classB --> classA classA is root

  If an object of type classC is constructed, the order of constructors called is classA, then classB and classC at the end.

- You should use initialization lists for initializing the data members.

- Initialization can also be done in constructor body (although strictly speaking that is assignment, not initialization).

Initializing

- Initializing in the constructor body instead of initialization list can cause a slight performance hit.

- Example:
  - classA::classA (const string & s1, const string & s2)

```
    : m_StrA(s1)                <-- this is true initialization
    {
        m_StrB = s2;        // m_StrB was first initialized to empty string,
                            // then assigned the value in s2. So this really
                            // is assignment at this point.
    }
```

- The performance hit is not an issue with built in types, i.e., if we were to assign values to int, char or double, etc. within the constructor body, that is ok. However, initialization list should still be preferred.

Initializing (Circle Example)

Consider the conversion constructor in a Circle class :
Let the data members of Circle class be *m_Radius*, *m_XCoord* and *m_YCoord* .

```
Circle::Circle (int   radius,  int   xCoord,  int   yCoord)
{
     // Assign values passed in as parameters to the conversion constructor
     m_Radius = radius;
     m_XCoord = xCoord;
     m_YCoord = yCoord;
}
```

Above is not really initializing the data members *m_Radius*, *m_XCoord* and *m_YCoord*
It is assigning a value.

For true initialization:

```
Circle::Circle (int   radius,  int   xCoord,  int   yCoord)
: m_Radius (radius),    m_XCoord (xCoord),   m_YCoord (yCoord)
{
}
```

Initializing

- Initializing reference data member and constant data members must be done in the initialization list. It cannot be done in the body.

- A class with const data member or references cannot be default constructed (unless you provide default arguments for initializing them).

- ```
  class MyClass
  {
        …
        private:
                int &           m_IntRef;
                const int       m_ConstInt;
  }

  MyClass::MyClass(int & ref, int value1)
  : m_IntRef( ref ), m_ConstInt( value1 )      <-- This is correct.
  {
        m_IntRef = ref;                        <-- This is incorrect, compilation error
        m_ConstInt = value1;                   <-- This is incorrect, compilation error
  }
  ```
  ```

Don't do this!!!

- If you have a reference data member, it needs a reference to be passed in to be initialized. Do NOT initialize it with a parameter passed in by value.
- Example :
  - MyClassA {
    public:
    MyClassA (MyClassB b);
    …
    private:
    MyClassB & m_B;
    }
  - MyClassA::MyClassA(MyClassB b)
    : m_B (b) <-- You made m_B a reference to a temporary
    { }

Right thing to do is MyClassA (MyClassB & b)

Don't do this !!!

- When using initialization lists, the order of initialization of the data members is NOT the order in which you specify the list, it is the order in which the data members are declared in the class specification.

- When initializing in the initialization list, do not make one initialization depend on the value of another data member.

- Example :
- MyClass:: MyClass (int value)

```
 : m_Value1 ( value ),
   m_Value2 ( value )              // don't say   m_Value2 ( m_Value1 )
 { }
```

Saying **m_Value2 (m_Value1)** would mean that we are assuming that m_Value1 is initialized before m_Value2.

Takeaway

- When initializing in the initialization list, if possible, do not make one initialization depend on the value of another data member.

- If you make the initialization of a data member depend on another data member, then even though you may do the initialization in the correct order (i.e., in the order they are declared in the class), its possible that some time later, someone may change the order of declaration because they are making some changes to the class, and now the initialization list could potentially be doing incorrect initialization (unless they fix the order in the initialization list as well, which could be unlikely as that would be easy to miss).

Destructor

- Destructor of an object is called automatically when it is deleted.
- For an object on the stack, this happens when it goes out of scope.
- For an object on the heap, this happens when operator **delete** is called on it.
- Destructor is where you would release any memory or other resources that are held by the object.
- Its name is the same as that of the class, prefixed with a tilde "~".
- Nothing is returned by the destructor.

Destructor

- Example:

```
class MyClass
{
    public:
                ...
                ~MyClass();
    private:
                char *     m_Ptr;     // points to memory allocated in constructor
}
MyClass::~MyClass()
{
      delete   m_Ptr1;                // m_Ptr was pointing to char *
}
```

Not deleting the memory in the destructor would be a memory leak (unless there was some other means of deleting that memory)

Calling the destructor

- You don't have to call the destructor yourself (but there is an exception. See backup slide).

- When operator **delete** is called on a pointer (in case of object on the heap), the destructor gets called.

  OR

- When an object goes out of scope, its destructor gets called.

- However, if the object is created by using the placement **new** operator, then there may be a need to call the destructor, and that too if any memory or resource needs to be released.

The *this* variable

- In C++, there is a special variable called "this". It always refers to the object to which the message has been sent.
- It is accessible within a non-static member function.
- *this* is a keyword.

- Example:
  - Stack s1, s2;
    s1.push (10); ← First call to Push on object s1
    s2.push (20); ← Second call to Push on object s2

  - ```
    void Stack::push( int value )
    {
        // When first Push is called, this is a pointer to the object "s1".

        // When the second push is called, this is a pointer to "s2".
    }
    ```
    ```

The *this* variable

- You don't have to use *this* to access an object's members.
- Example:
  - ```
    bool Stack::IsFull() const
    {
        return   m_NumElements  ==  m_Size ;

        // don't have to say  this->m_NumElements
        // or
        // this->m_Size
    }

    void Stack::push( int value )
    {
        if ( IsFull() )              // don't have to say  this->IsFull()
        …
    }
    ```
    ```

Overloading methods

- C++ supports function overloading.
- This means that that the same name can be given to more than one functions (within certain guidelines).
- Compiler can determine which function to call based on the arguments supplied.
- Example:
  - void Stack::Push(int value)
  - void Stack::Push(int value1, int value2) // not recommending this interface :=)
  - In C, or a language not providing this feature, this would have to be called PushInt and PushTwoInts (or something like that)
- Note: C++ compilers do name mangling to support this.

Overloading methods

- C++ does not distinguish the function to be called based on the return type.

- Example:
  - **Cannot** have :
    - void Stack::Push (int value) and
    - bool Stack::Push(int value) // not legal

Overloading methods

- The argument(s) to overloaded functions should be different enough so that the compiler can distinguish between the two.

- Example:
  - **Cannot** have any two or more of the following together:
    - void Stack::Push (int value) and
    - void Stack::Push (int & value)
    - void Stack::Push (const int value)

Parameters

- Parameter list of a function has to be specified.
- Function with no parameters can be represented as :
  - int myFunction(); or
  - int myFunction(void);
- Parameters in the list are separated by comma:
  - int myFunction (int value, char * str, double d)
- As already seen, two functions can have the same name, but different parameters.

Parameter Passing

- Parameters are passed to functions :
  - by value or
    - void MyFunction1 (int value, MyClass mc);
  - by reference
    - void MyFunction2 (int & value, const MyClass & mc);
  - by address
    - void MyFunction3 (int * value, const MyClass * mc1, MyClass * mc2)

Parameter passing by value

- When a parameter is passed by value, its copy is passed to the function.
- As a result, any changes made to the parameter are only made to the copy, and not the original. Hence these changes are not visible outside the function.

- Example:

```
void  MyFunction1 ( int  value,    MyClass mc );
{
        value  = value * 2;
        mc.IncrementCount();          //  increments a value inside object mc
}
main()
{
    int  x = 10;
    MyClass  mc( 25 );
    MyFunction1 ( x,  mc );

    // value of x  after function call is still 10,  and  mc  is also unchanged.
}
```

When not to pass by value

- Do not pass a parameter by value if any one of the following is true:
  - You are expecting the function to change the value of the parameter.
  - A large object needs to be passed (copy constructor gets called, resulting in an unnecessary performance hit)
  - In the first case, pass the parameter by reference.
  - In the second case also, pass the parameter by reference, however, make it a constant reference to prevent modification by mistake.

Parameter passing by reference

- When a parameter is passed by reference, the function has access to argument that is passed in (and not its copy).
- As a result, any changes made to the parameter inside the function are seen outside of the function call.
- Passing by reference (or const reference) is faster than passing by value. (For built in types, this statement is not true).
- Another reason to pass by reference is to maintain polymorphic behavior. (See example 2 in slides that follow)

Example 1

- void MyFunction2 (int & ii, const MyClass & mc);

- MyFunction2 may modify value of ii, so ii is passed by reference.
- MyFunction2 does NOT modify object mc, so mc is passed by const reference.

- ```
  void MyFunction2 ( int & ii,  const MyClass & mc );
  {
        ii = ii * 2;
        mc->IncrementCount();      <-- Compile ERROR, since mc is a const reference
  }

  main()
  {
      int x = 10;
      MyClass mc( 25 );
      MyFunction2 ( x, mc );

      // value of x after function call is 20, and object mc is unchanged.
  }
  ```
  ```

Example 2

- Example to show maintaining polymorphic behavior :

- void FooRef (const Employee & e)
- void FooVal (Employee e)

- Lets say we have the following declarations:

- Employee emp; Manager m; Admin a;

- Polymorphic behavior in calls below :

- *FooRef (emp);*
- *FooRef (m);* // m stays as a Manager object inside FooRef
- *FooRef (a);* // a stays as an Admin object inside FooRef

- No polymorphic behavior in calls below :
- *FooVal (emp);*
- *FooVal (m);* // Oops! m becomes an Employee object inside FooVal
- *FooVal (a);* // Oops! a becomes an Employee object inside FooVal

Pass by reference or address ?

- Pass by reference as well as pass by address (like in C) allow a function to modify the value.
- So, how do we decide which one to use ?

- Use pass by address (as in void MyFunction (int * ptr)) if
  - the pointer passed in can be NULL.
  - The pointer passed in may be made to point to different objects.
  - You want to do arithmetic operations on the pointer (increment, decrement, etc.)
- When pass by reference is used, the called function need not check for NULL, since a reference has to always refer to an object.

- void MyFunction2 (int ii, MyClass & mcRef, MyClass * mcPtr)

  MyFunction2 (5, NULL, NULL) <-- Error. Cannot pass NULL as 2nd argument.

  MyFunction2 (6, mc, NULL) <-- OK. Assume mc is declared as: *MyClass mc;*

Passing Arrays

- In C++, arrays are not passed by value.
- When you pass an array, a pointer to its first element is passed in. (like in C).

Array of objects

- An array of objects is defined just like you would an array of built-in type
  - MyClass arrayOfMyClass[100];
  - this defines an array of 100 MyClass objects.
  - Each of these objects is initialized with the default constructor of MyClass.

    NOTE : MyClass has to either provide a default constructor or no constructor at all for the above declaration to compile.

Array of objects

```
class MyClass
{
  public:
        MyClass ( const char * str1 );                    // Constructor 1
        MyClass (const char * str1, const char * str2);   // Constructor 2
  …
};

MyClass array[ ] = { "First",
                     "Second",
                     "Third" };
```

array contains three objects, initialized with constructor 1 above.

This form can be used when the constructor requires user to **specify** one argument.

Array of objects

If constructor 2 needs to be used, we have to do :

```
MyClass array[] = {
                    MyClass ("First"),          // calls constructor 1
                    MyClass ("a", "b"),         // calls constructor 2
                    MyClass()                   // calls default constructor
                  };
```

Third object in the array is initialized using the default constructor (if provided, otherwise it will be a compilation error) .

Array of objects

```
MyClass array[ 10 ] = {
                        MyClass ("First", "First"),
                        MyClass ("Second", "Second")
                      };
```

The first two elements are initialized using the conversion constructor specified.

All the rest of the elements are initialized using the default constructor.

Creating an array dynamically

- MyClass * array = new MyClass [30];
- This creates an array of 30 objects that are initialized by the default constructor.
- We cannot specify explicit values when creating the array of objects on the heap (like we did when creating the array on stack).
- However, we can allocate memory for the array, then use placement new to create and initialize the objects (elements) in the array

  (See Example)

```
const int kNumElements = 10;
const char * strArray[ ] = { "One", "Two", "Three", "Four"  };

main()
{
      int numStrArrayElements =  sizeof ( strArray )  /  sizeof (char *);
      char * arrayMemory = new char [ sizeof ( MyClass ) * kNumElements ];

      for ( int ii = 0 ; ii  <  kNumElements;  ii ++)
      {
            int offset  =   ii * sizeof( MyClass );                  // compute offset for ii th element
            int elementAddress  =  arrayMemory + offset;

            // Now create an object at the computed address.
            // Note : no memory allocated for the object in the following step, although
            //          the constructor of the object may allocate memory.

            if  (ii  <  numStrArrayElements)
                        new ( elementAddress ) MyClass( ii,  strArray[ ii ] );
            else
                        new ( elementAddress ) MyClass( );
      }
      … use the array of objects
      // now we need to release the memory

      MyClass * mcPtr = (MyClass *) arrayMemory;
      for (ii = 0; ii  <  kNumElements;  ii ++ )
            mcPtr [ ii ].~MyClass();                  // call destructor of each object.

      delete  [ ] arrayMemory;                        // delete the memory block. Note we call delete on
                                                      // arrayMemory and  not on mcPtr.
```

Implicit v.s. explicit constructors

- What does it mean to have an implicit constructor ?
  - It means that a conversion from one type A to another, say, B, can be done automatically by the compiler by calling the conversion constructor of B that converts from A to B.

    B::B (const A & a)

- What does it mean to have an explicit constructor ?
  - It means that the conversion from one type A to another type B will be done ONLY IF explicitly requested by the developer.

- In C++, implicit constructors are the default.

Implicit constructor

- Constructors are implicit by default.
- A constructor can get called if there is an implicit conversion possible.
- Example:

```
class mystring {
        public:
                mystring(const char * str);     // convert str to a string
                mystring(int size);     // to create a string "size" long
};
int main()
{
        mystring s1 (5);
        mystring s2( "param1");

        s2 = 100;       // s2 is assigned a string that can hold 100 characters.
                        // This was possible because the constructor is not
                        // declared as explicit, and is implicit by default
}
```

Implicit constructor

- Implicit constructor can sometimes lead to subtle undesirable effects.

- Example:
  - In the string example, the user may really want to say :

```
s2 = "100";        // assign the string 100 to s2
```

instead of

```
s2 = 100;          // resize s2 to be able to hold 100 characters,
                   // where each character has a default value as
                   // specified in the conversion constructor
```

- To avoid this, you can declare the constructor explicit.

Explicit constructor

- When you declare a constructor explicit, you are telling the compiler that you don't want that constructor to be called unless explicitly called by the program.

- Example

```
class string {
        public:
                string(const char * str);          // convert str to a string
                explicit string(int size);         // to create a string "size" long
};

int main()
{
        string s1 (5);                     // ok, since it is explicitly called
        string s2( "param1");

        s2 = 100;                          // Error. Constructor is declared as explicit
        s2 = string(100);                  // OK, explicit call
}
```

Default constructor

- The following is an incorrect way of declaring an object initialized with the default constructor :

  - MyClass mc();
    if (mc.HasData()) … <-- compile error

  You do not have to put the braces after **mc**.

  Putting the braces means it is declaring **mc** as a function that returns an object of type MyClass.

  As a result, the second line will give a compilation error.

  Correct form :

  MyClass mc;
  if (mc.HasData()) …

Example: Calling destructor

```
char * space = new char [ sizeof( MyClassA ) ];
MyClassA  *  a1 = new (space ) MyClassA ("name1" );  // does NOT allocate any
                                                     // memory
... use a1
a1->~MyClassA();                    // does NOT delete "space "

MyClassA  *  a2 =  new (space ) MyClassA("name2");
... use a2
a2->~MyClassA();

delete [ ] space ;                  // now release the memory that space points to
```

- Note that even though "a2" and "space" point to the same memory, calling delete on "space" does **NOT** call the **destructor** on "a2".