Pointers, Classes, Inheritance & Polymorphism

Rahul Deodhar

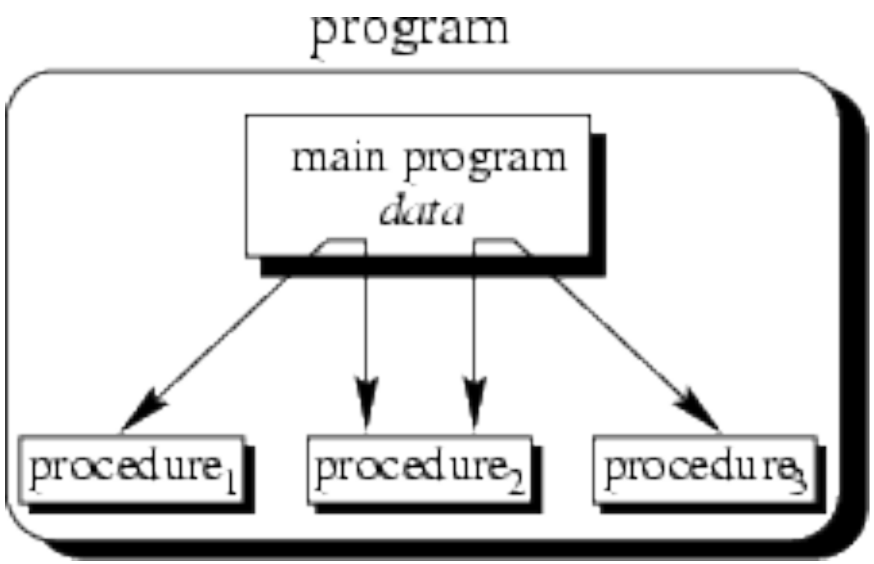
www.rahuldeodhar.com

rahuldeodhar@gmail.com

@rahuldeodhar

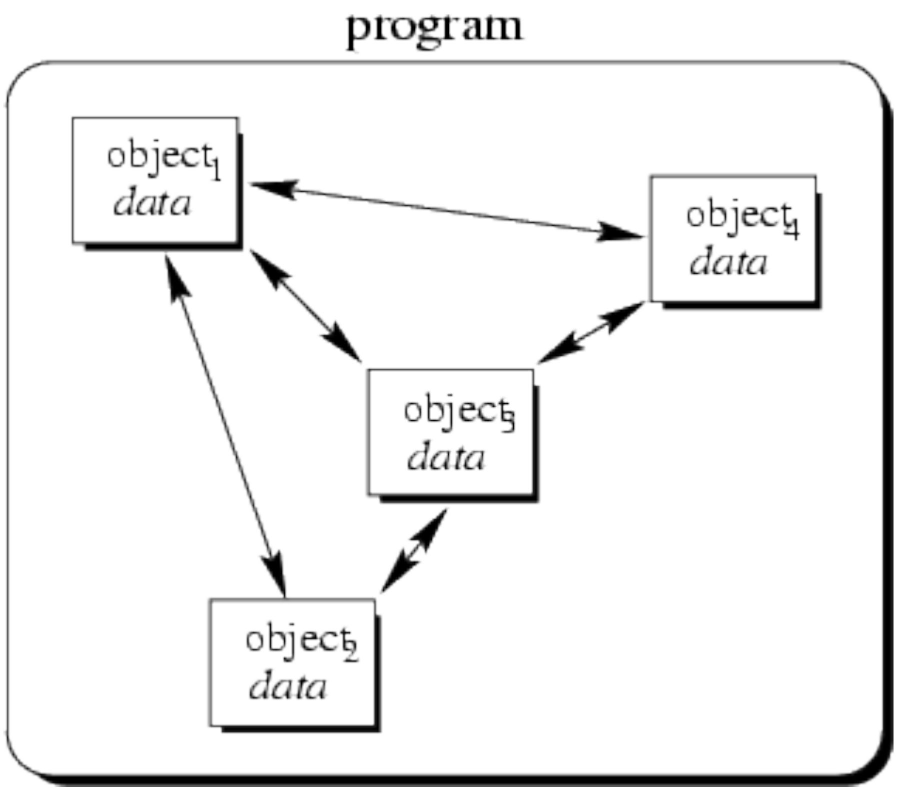
+91 9820213813

Procedural Concept



- The main program coordinates calls to procedures and hands over appropriate data as parameters.

Object-Oriented Concept



- Objects of the program interact by sending messages to each other

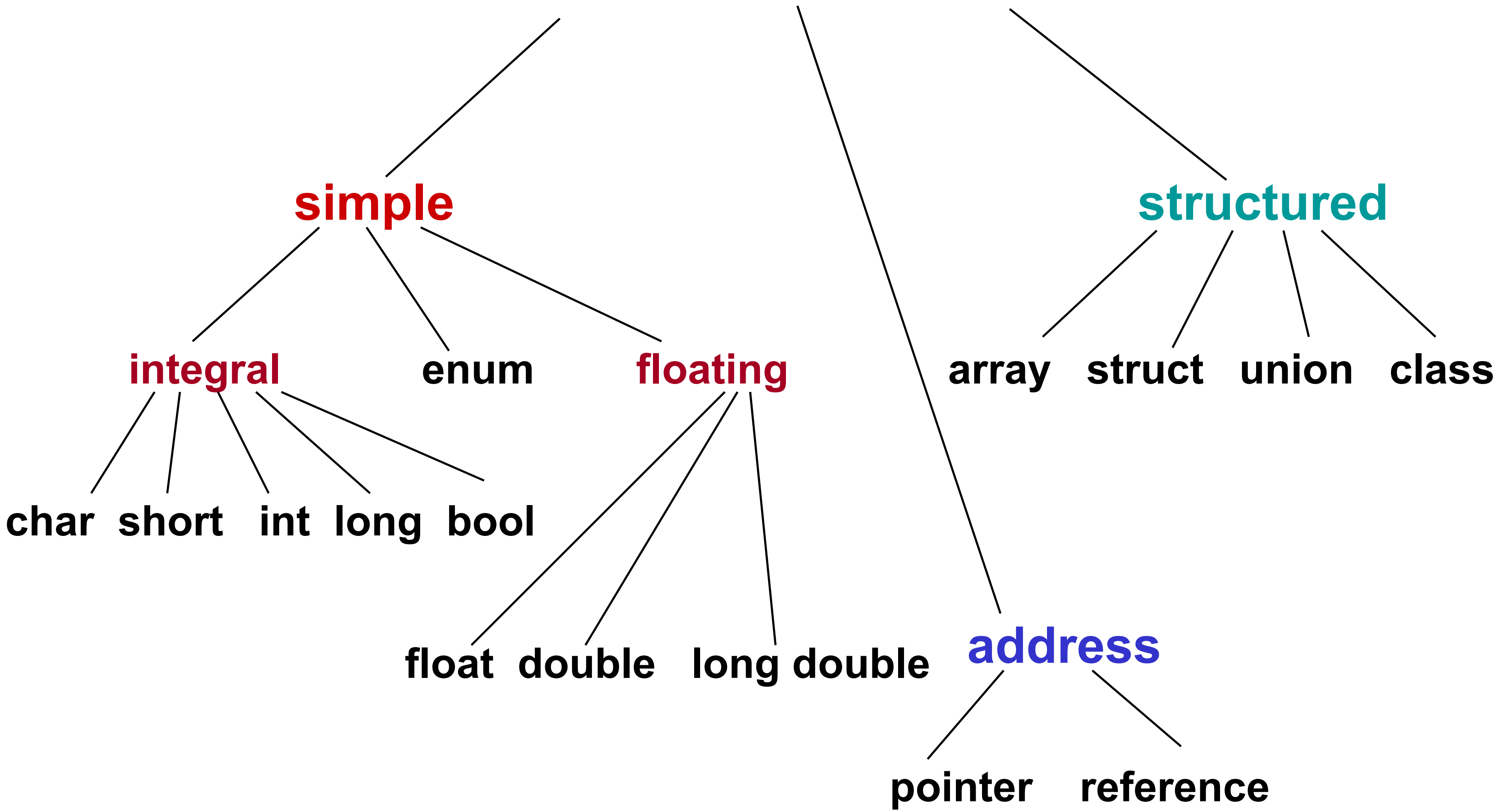
C++

- Supports Data Abstraction
- Supports OOP
  - Encapsulation
  - Inheritance
  - Polymorphism
- Supports Generic Programming
  - Containers
    - Stack of char, int, double etc
  - Generic Algorithms
    - sort(), copy(), search() any container Stack/Vector/List

Pointers, Dynamic Data, and Reference Types

- Review on Pointers
- Reference Variables
- Dynamic Memory Allocation
  - The new operator
  - The delete operator
  - Dynamic Memory Allocation for Arrays

C++ Data Types


simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

Recall that . . .

char str [8];

- **str** is the base address of the array.
- We say **str** is a pointer because its value is an address.
- It is a pointer constant because the value of **str** itself cannot be changed by assignment. It “points” to the memory location of a `char`.

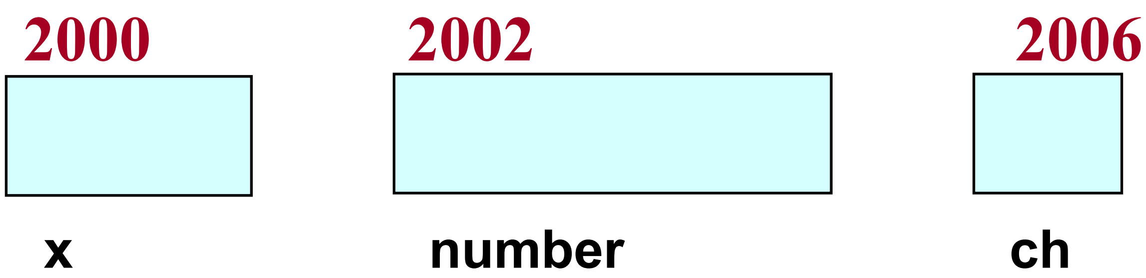
6000

‘H’	**‘e’**	**‘l’**	**‘l’**	**‘o’**	**‘\0’**		
str [0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**

Addresses in Memory

- When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable

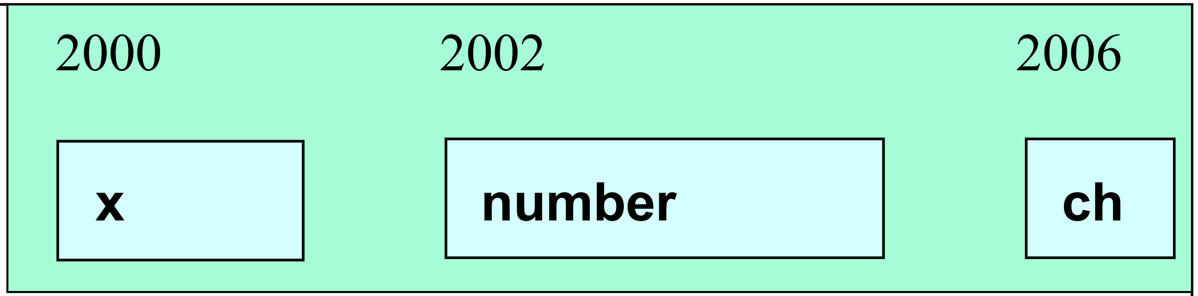
```
int      x;
float    number;
char     ch;
```



Obtaining Memory Addresses

- The address of a *non-array variable* can be obtained by using the address-of operator &

```
int     x;
float   number;
char    ch;
```




```
cout << “Address of x is “ << &x << endl;

cout << “Address of number is “ << &number << endl;

cout << “Address of ch is “ << &ch << endl;
```

What is a pointer variable?

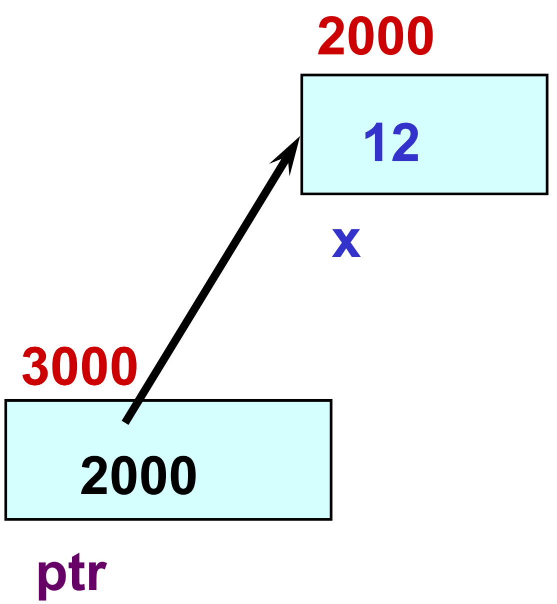
- A pointer variable is a variable whose value is the address of a location in memory.

- To declare a pointer variable, you must specify the type of value that the pointer will point to, for example,

```
int*    ptr; // ptr will hold the address of an int
char*   q;   // q will hold the address of a char
```

Using a Pointer Variable

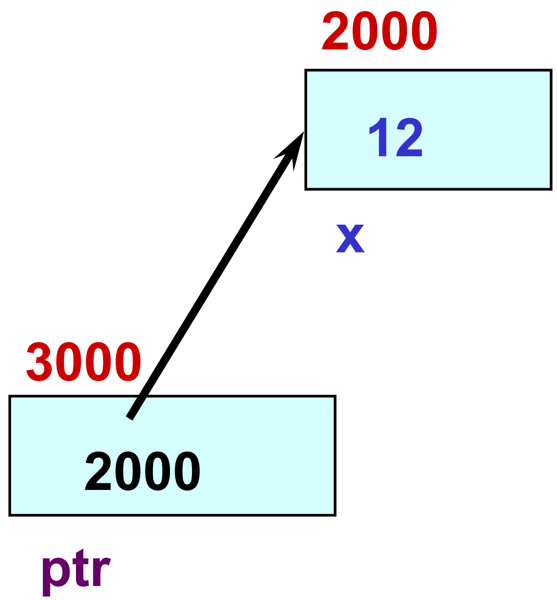
```
int  x;
x = 12;

int*  ptr;
ptr = &x;
```



NOTE: **Because ptr holds the address of x, we say that ptr "points to" x**

*: dereference operator

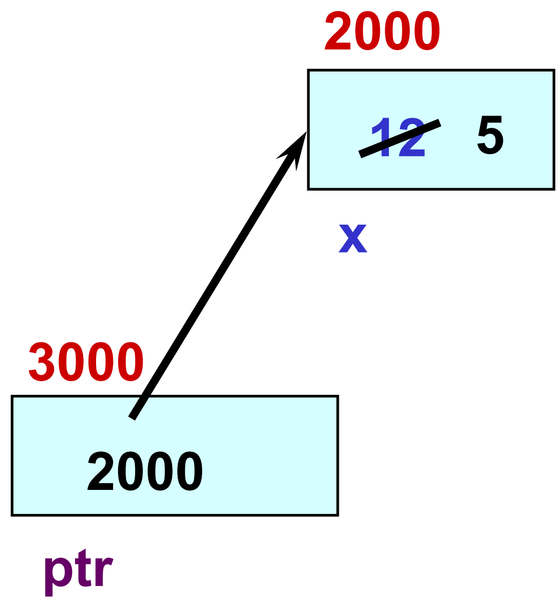
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

cout  <<  *ptr;
```



NOTE: **The value pointed to by ptr is denoted by *ptr**

Using the Dereference Operator

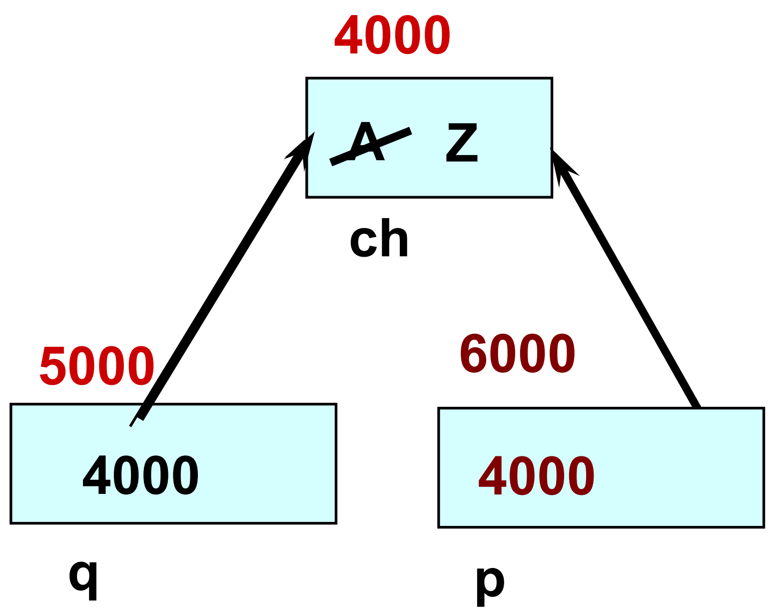
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

*ptr = 5;
```




```
// changes the value at the
   address ptr points to 5
```

Self –Test on Pointers

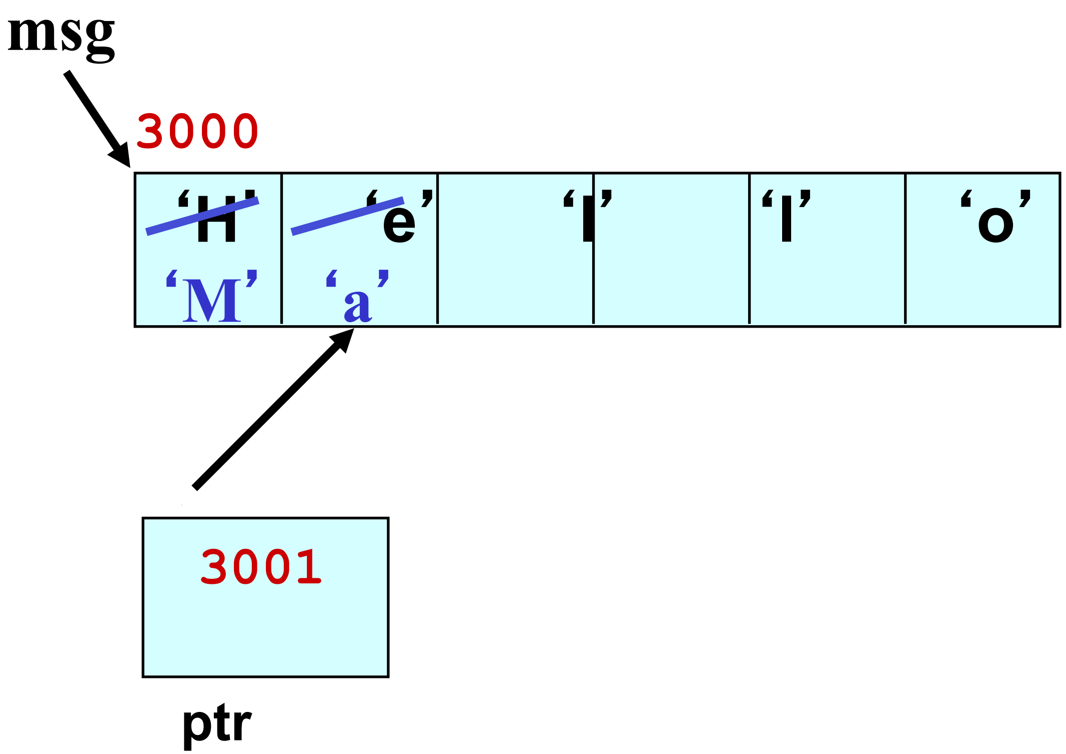
```
char  ch;
ch =  'A';

char*  q;
q  = &ch;

*q = 'Z';
char*  p;
p = q;
```




```
// the rhs has value 4000
// now p and q both point to ch
```

Using a Pointer to Access the Elements of a String

```
char   msg[ ] ="Hello";
char*  ptr;
ptr  =  msg;
*ptr  =  'M'  ;
ptr++;
*ptr =  'a' ;
```



Reference Variables

Reference variable = ***alias for another variable***

- Contains the address of a variable (like a pointer)
- No need to perform any dereferencing (unlike a pointer)
- Must be initialized when it is declared

```
int x = 5;
int &z = x;           // z is another name for x
int &y ;              //Error: reference must be initialized
cout << x << endl;    -> prints 5
cout << z << endl;    -> prints 5

z = 9;                // same as x = 9;

cout << x << endl;    -> prints 9
cout << z << endl;    -> prints 9
```

Why Reference Variables

- Are primarily used as function parameters
- Advantages of using references:
  - you don't have to pass the address of a variable
  - you don't have to dereference the variable inside the called function

Reference Variables Example

```
#include <iostream.h>

// Function prototypes
   (required in C++)

void p_swap(int *, int *);
void r_swap(int&, int&);

int main (void){
 int v = 5, x = 10;
 cout << v << x << endl;
 p_swap(&v,&x);
 cout << v << x << endl;
 r_swap(v,x);
 cout << v << x << endl;
 return 0;
}
```

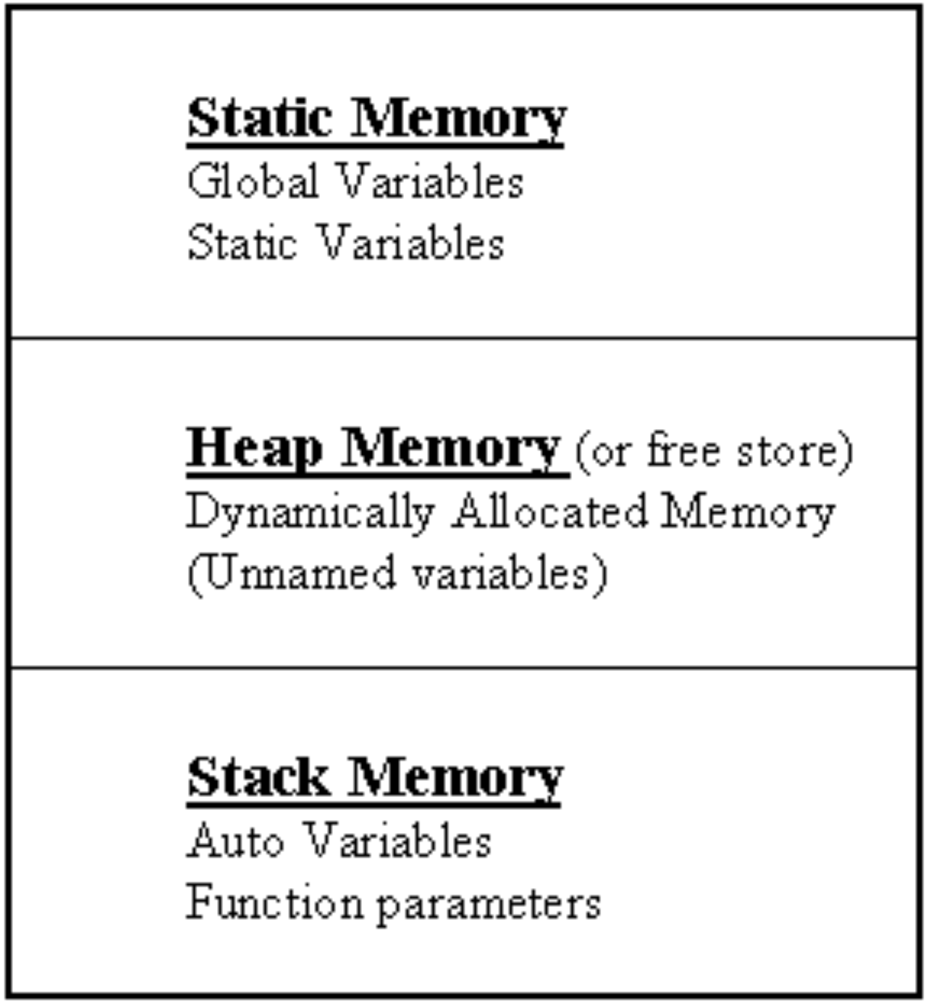
```
void p_swap(int *a, int *b)
{
  int temp;
   temp = *a;      (2)
   *a = *b;              (3)
   *b = temp;
}
```

```
void r_swap(int &a, int &b)
{
 int temp;
 temp = a;              (2)
 a = b;         (3)
 b = temp;
}
```

Dynamic Memory Allocation

In C and C++, three types of memory are used by programs:

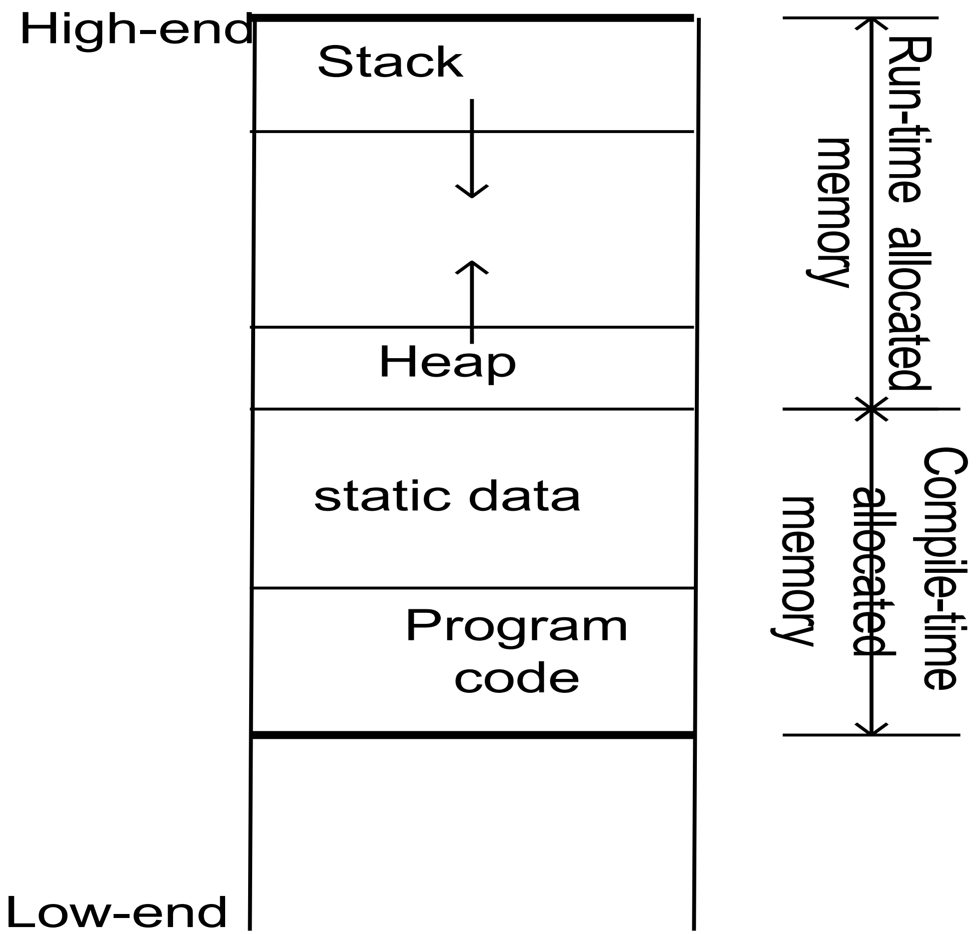
- **Static memory** - where global and static variables live
- **Heap memory** - dynamically allocated at execution time
  - "managed" memory accessed using pointers
- **Stack memory** - used by automatic variables



3 Kinds of Program Data

- STATIC DATA: Allocated at compiler time
- DYNAMIC DATA: explicitly allocated and deallocated during program execution by C++ instructions written by programmer using operators **`new`** and **`delete`**
- AUTOMATIC DATA: automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function

Dynamic Memory Allocation Diagram



Dynamic Memory Allocation

- *In C*, functions such as malloc() are used to dynamically allocate memory from the **Heap**.
- *In C++,* this is accomplished using the **new** and **delete** operators
- **new** is used to allocate memory during execution time
  - returns a pointer to the address where the object is to be stored
  - always returns a pointer to the type that follows the **new**

Operator **`new`** Syntax

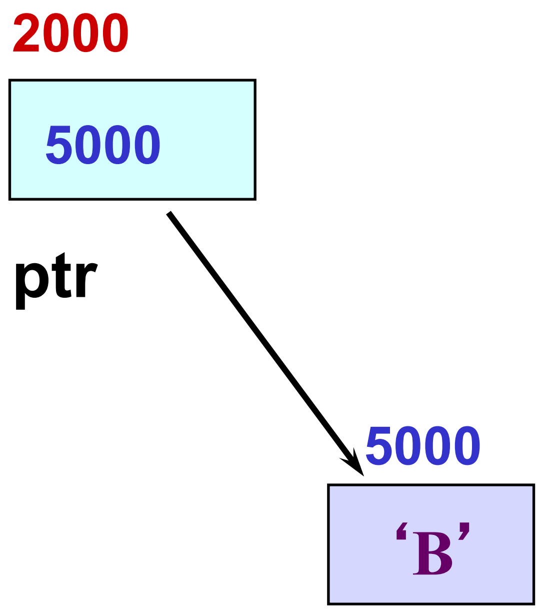
new DataType

new DataType [IntExpression]

- If memory is available, in an area called the heap (or free store) new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.
- Otherwise, program terminates with error message.
- The dynamically allocated object exists until the delete operator destroys it.

Operator new

```
char*  ptr;

ptr = new char;
*ptr = 'B';
cout  <<  *ptr;
```



NOTE: Dynamic data has no variable name

The **NULL** Pointer

- There is a pointer constant called the "null pointer" denoted by NULL
- But NULL is not memory address 0.
- NOTE: It is an error to dereference a pointer whose value is NULL. Such an error may cause your program to crash, or behave erratically. It is the programmer's job to check for this.

```
while (ptr != NULL) {
      . . .          // ok to use *ptr here
}
```

Operator `delete` Syntax

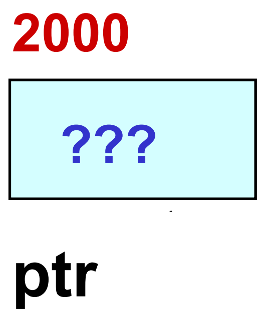
delete Pointer

delete [ ] Pointer

- The object or array currently pointed to by Pointer is deallocated, and the value of Pointer is undefined. The memory is returned to the free store.
- Good idea to set the pointer to the released memory to NULL
- Square brackets are used with delete to deallocate a dynamically allocated array.

Operator **delete**

```
char*  ptr;

ptr = new char;

*ptr = 'B';

cout  <<  *ptr;

delete  ptr;
```

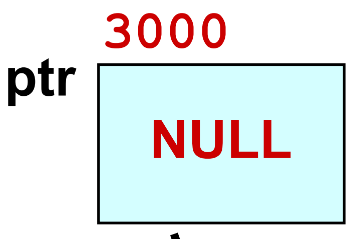


NOTE:

delete deallocates the memory pointed to by ptr

Example

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, "Bye" );

ptr[ 0 ] =  'u' ;


delete [] ptr;

ptr = NULL;
```



// deallocates the array pointed to by ptr
// ptr itself is not deallocated
// the value of ptr becomes undefined

Pointers and Constants

```
char* p;
p = new char[20];

char c[] = “Hello”;
const char* pc = c; //pointer to a constant
pc[2] = ‘a’; // error
pc = p;

char  *const cp = c; //constant pointer
cp[2] = ‘a’;
cp = p; // error

const char *const cpc = c; //constant pointer to a const
cpc[2] = ‘a’; //error
cpc = p; //error
```

Take Home Message

- Be aware of where a pointer points to, and what is the size of that space.
- Have the same information in mind when you use reference variables.
- Always check if a pointer points to NULL before accessing it.

Review: Pointers & Dynamic Data

- A pointer variable is a variable whose value is the address of a location in memory

```
int x;
x = 5;

int* ptr1;
ptr1 = &x;

int* ptr2;
ptr2 = ptr1;
*ptr1 = 6;

cout << ptr1 << endl;
cout << *ptr2 << endl;
```

```
int* ptr3;
ptr3 = new int;
*ptr3 = 5;
delete ptr3;
ptr3 = NULL;

int *ptr4;
ptr4 = new int[5];
ptr4[0] = 100;
ptr4[4] = 123;
delete [] ptr4;
ptr4 = NULL;
```

Review: Reference Types

- Reference Types
  - Alias for another variable
  - Must be initialized when declared
  - Are primarily used as function parameters

```
int main (void){
    int a1 = 5, a2 = 10;
    int *a3 = new int;
    *a3 = 15;
    int &a4 = a3;
    cout << a1 << a2 << a3 << endl;
    increment(a1, a2, a3);
    cout << a1 << a2 << a3 << endl;
    delete a3; a3 = NULL;
    return 0;
}
```

```
void increment(int b1, int &b2, int *b3)
{
    b1 += 2;
    b2 += 2
    *b3 += 2;
}
```

Object-Oriented Programming Introduction to Classes

- Class Definition
- Class Examples
- Objects
- Constructors
- Destructors

Class

- The class is the cornerstone of C++
  - It makes possible encapsulation, data hiding and inheritance
- Type
  - Concrete representation of a concept
    - Eg. **float** with operations like -, *, + (math real numbers)
- Class
  - A user defined type
  - Consists of both data and methods
  - Defines properties and behavior of that type
- Advantages
  - Types matching program concepts
    - Game Program (Explosion type)
  - Concise program
  - Code analysis easy
  - Compiler can detect illegal uses of types
- Data Abstraction
  - Separate the implementation details from its essential properties

Classes & Objects

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
};
```

Objects: Instance of a class

```
Rectangle  r1;
Rectangle  r2;
Rectangle  r3;
   ......
```

```
int  a;
```

Define a Class Type

Header

Body

```
class class_name
{
    permission_label:
        member;
    permission_label:
        member;
    ...
};
```

```
class Rectangle
{
  private:
    int width;
    int length;
  public:
    void set(int w, int l);
    int area();
};
```

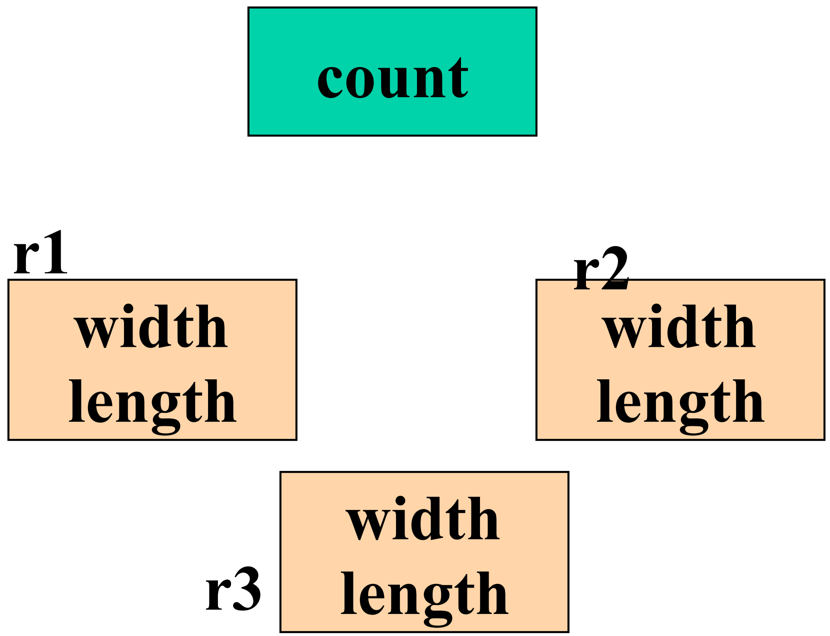
Class Definition
Data Members

- Can be of any type, built-in or user-defined
- *non-static* data member
  - Each class object has its own copy
- *static* data member
  - Acts as a global variable
  - One copy per class type, e.g. counter

Static Data Member

```
class Rectangle
{
   private:
      int width;
      int length;
      static int count;
   public:
      void set(int w, int l);
      int area();
   }
```

```
Rectangle  r1;
Rectangle  r2;
Rectangle  r3;
```



Class Definition
Member Functions

- Used to
  - access the values of the data members (**accessor**)
  - perform operations on the data members (**implementor**)
- Are declared inside the class body
- Their definition can be placed inside the class body, or outside the class body
- Can access both public and private members of the class
- Can be referred to using dot or arrow member access operator

Define a Member Function

```
class Rectangle
{
    private:
        int width, length;
    public:
        void set (int w, int l);
        int area() {return width*length; }
};
```

inline

r1.set(5,8);

rp->set(8,10);

class name

member function nan

```
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

scope operator

Class Definition
Member Functions

- **const** member function
  - declaration
    - *return_type func_name* (*para_list*) const;
  - definition
    - *return_type func_name* (*para_list*) const { … }
    - *return_type class_name* :: *func_name* (*para_list*) const { … }
  - Makes no modification about the data members (safe function)
  - It is illegal for a const member function to modify a class data member

Const Member Function

```
class  Time
{
  private :
     int     hrs, mins, secs ;

  public :

     void     Write ( )  const ;

} ;
```

function declaration

function definition

```
void Time :: Write( ) const
{
    cout <<hrs << ":" << mins << ":" << secs
<< endl;
}
```

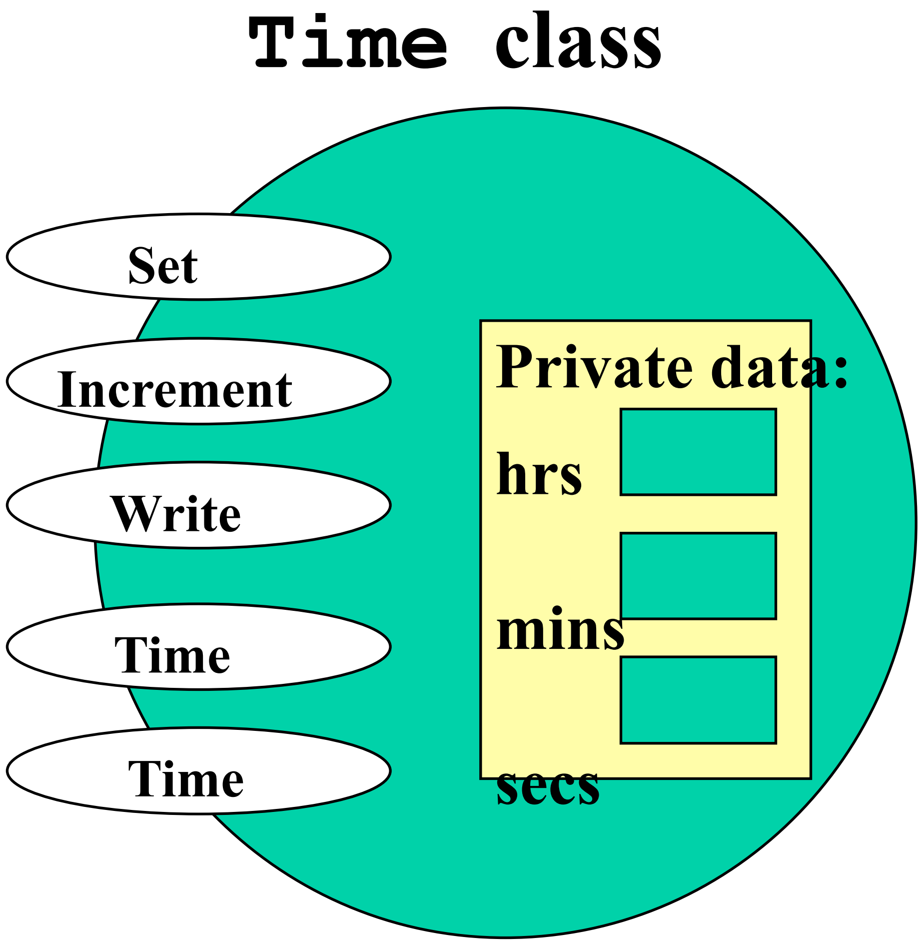
Class Definition - Access Control

- Information hiding
  - To prevent the internal representation from direct access from outside the class
- Access Specifiers
  - public
    - may be accessible from anywhere within a program
  - private
    - may be accessed only by the member functions, and friends of this class
  - protected
    - acts as public for derived classes
    - behaves as private for the rest of the program

class Time Specification

```
class  Time
{

 public :

    void      Set ( int  hours , int  minutes , int  seconds ) ;
    void      Increment ( ) ;
    void      Write ( )  const ;
    Time      ( int  initHrs, int  initMins,  int  initSecs ) ;   // constructor
    Time      ( ) ;                                               // default constructor

 private :

    int         hrs ;
    int         mins ;
    int         secs ;
} ;
```

Class Interface Diagram

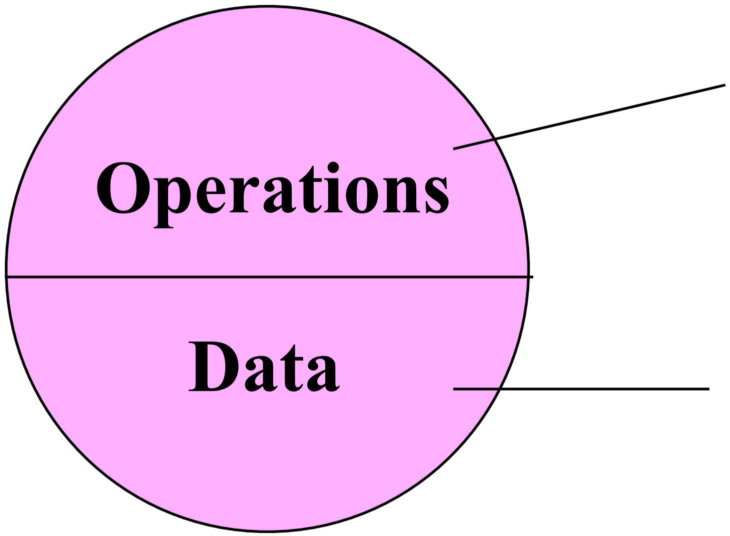


Class Definition
Access Control

- The default access specifier is *private*
- The data members are usually private or protected
- A **private** member function is a helper, may only be accessed by another member function of the same class (exception *friend* function)
- The **public** member functions are part of the *class interface*
- Each access control section is optional, repeatable, and sections may occur in any order

What is an object?

OBJECT



set of methods
(member functions)

internal state
(values of private data members)

Declaration of an Object

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
};
```

```
main()
{
   Rectangle r1;
   Rectangle r2;

   r1.set(5, 8);
   cout<<r1.area()<<endl;

   r2.set(8,10);
   cout<<r2.area()<<endl;
}
```

Another Example

```
#include <iostream.h>

class circle
{
   private:
      double radius;

   public:
      void store(double);
      double area(void);
      void display(void);

};
```

```
// member function definitions

void circle::store(double r)
{
   radius = r;
}

double circle::area(void)
{
   return 3.14*radius*radius;
}

void circle::display(void)
{
   cout << "r = " << radius << endl;
}
```

```
int main(void) {
   circle c;   // an object of circle class
   c.store(5.0);
   cout << "The area of circle c is " << c.area() << endl;
   c.display();
}
```

Declaration of an Object

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
};
```

r1 is statically allocated

```
main()
{
      Rectangle r1;
  ➡ r1.set(5, 8);
}
```

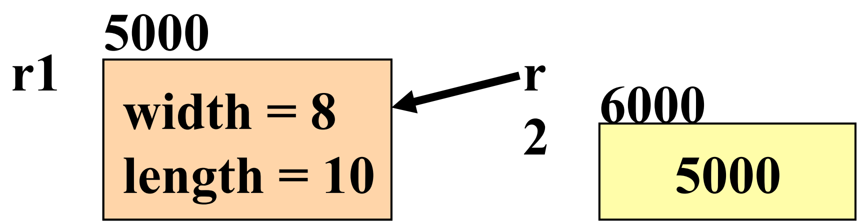
r1	
	width = 5
	length = 8

Declaration of an Object

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
};
```

r2 is a pointer to a Rectangle obje

```
main()
{
    Rectangle r1;
    r1.set(5, 8);          //dot notation

    Rectangle *r2;
    r2 = &r1;
    r2->set(8,10);         //arrow notation
}
```

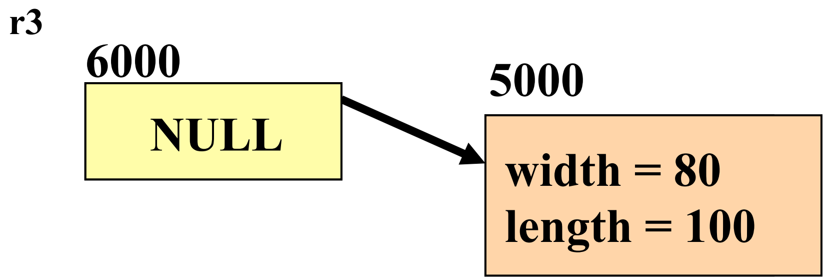


Declaration of an Object

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
};
```

r3 is dynamically allocated

```
main()
{
      Rectangle *r3;
      r3 = new Rectangle();

      r3->set(80,100);      //arrow notatior

      delete r3;
      r3 = NULL;
}
```



Object Initialization

```
#include <iostream.h>

class circle
{
  public:
    double radius;
};
```

1. By Assignment

- Only work for public data members
- No control over the operations on data members

```
int main()
{
  circle c1;                  // Declare an instance of the class circle
  c1.radius = 5;   // Initialize by assignment

}
```

Object Initialization

```
#include <iostream.h>

class circle
{
  private:
      double radius;

   public:
      void set (double r)
              {radius = r;}
      double get_r ()
              {return radius;}

};
```

2. By Public Member Functions

```
int main(void) {
    circle c;                    // an object of circle class
    c.set(5.0);                  // initialize an object with a public member function
    cout << "The radius of circle c is " << c.get_r() << endl;
        // access a private data member with an accessor
}
```

Declaration of an Object

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
}
```

r2 is a pointer to a Rectangle object

```
main()
{
    Rectangle r1;
    r1.set(5, 8);        //dot notation

    Rectangle *r2;
    r2 = &r1;
    r2->set(8,10);   //arrow notation
}
```

r1 and r2 are both initialized by public member function set

Object Initialization

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
      Rectangle();
      Rectangle(const Rectangle &r);
      Rectangle(int w, int l);
      void set(int w, int l);
      int area();
}
```

3. By Constructor

- Default constructor
- Copy constructor
- Constructor with parameters

They are publicly accessible
Have the same name as the class
There is no return type
Are used to initialize class data members
They have different signatures

Object Initialization

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
};
```

When a class is declared with no constructors, the compiler automatically assumes default constructor and copy constructor for it.

- Default constructor

```
Rectangle :: Rectangle() { };
```

- Copy constructor

```
Rectangle :: Rectangle (const Rectangle &
   r)
{
   width = r.width;  length = r.length;
};
```

Object Initialization

```
class Rectangle
{
   private:
      int width;
      int length;
   public:
      void set(int w, int l);
      int area();
}
```

- Initialize with default constructor

```
Rectangle r1;
Rectangle *r3 = new Rectangle();
```

- Initialize with copy constructor

```
Rectangle r4;
r4.set(60,80);

Rectangle r5 = r4;
Rectangle r6(r4);

Rectangle *r7 = new Rectangle(r4);
```

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        Rectangle(int w, int l)
            {width =w; length=l;}
        void set(int w, int l);
        int area();
}
```

If any constructor with any number of parameters is declared, no default constructor will exist, unless you define it.

```
Rectangle r4;        // error
```

- Initialize with constructor

```
Rectangle r5(60,80);
Rectangle *r6 = new Rectangle(60,80);
```

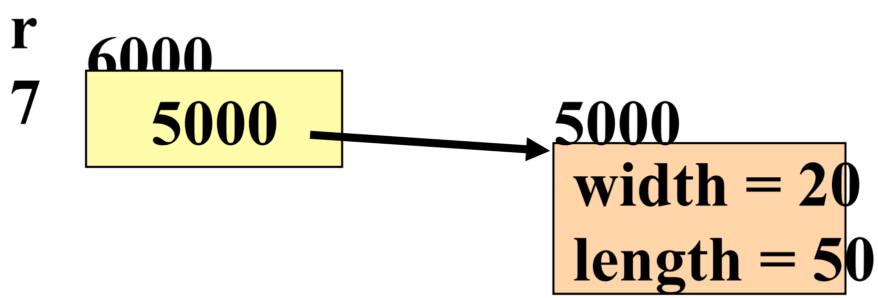
Object Initialization

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
      Rectangle();
      Rectangle(int w, int l);
      void set(int w, int l);
      int area();
}
```

Write your own constructors

```
Rectangle :: Rectangle()
{
    width = 20;
    length = 50;
};
```

```
Rectangle  *r7 = new Rectangle();
```



Object Initialization

```
class Account
{
    private:
      char *name;
      double balance;
      unsigned int id;
    public:
      Account();
      Account(const Account &a);
      Account(const char *person);
}
```

```
Account :: Account()
{
    name = NULL; balance = 0.0;
    id = 0;
};
```

With constructors, we have more control over the data members

```
Account :: Account(const Account &a)
{
    name = new char[strlen(a.name)+1];
    strcpy (name, a.name);
    balance = a.balance;
    id = a.id;
};
```

```
Account :: Account(const char *person)
{
    name = new char[strlen(person)+1];
    strcpy (name, person);
    balance = 0.0;
    id = 0;
};
```

So far, …

- An object can be initialized by a class constructor
  - default constructor
  - copy constructor
  - constructor with parameters
- Resources are allocated when an object is initialized
- Resources should be revoked when an object is about to end its lifetime

Cleanup of An Object

```
class Account
{
    private:
      char *name;
      double balance;
      unsigned int id;  //unique
    public:
      Account();
      Account(const Account &a);
      Account(const char *person);
      ~Account();
}
```

Destructor

```
Account :: ~Account()
{
    delete[] name;
}
```

- Its name is the class name preceded by a ~ (tilde)
- It has no argument
- It is used to release dynamically allocated memory and to perform other "cleanup" activities
- It is executed automatically when the object goes out of scope

Putting Them Together

```
class Str
{
      char *pData;
      int nLength;
  public:
       //constructors
      Str();
      Str(char *s);
      Str(const Str &str);

       //accessors
      char* get_Data();
      int get_Len();

       //destructor
       ~Str();
};
```

```
Str :: Str() {
   pData = new char[1];
   *pData = '\0';
   nLength = 0;
};
```

```
Str :: Str(char *s) {
   pData = new char[strlen(s)+1];
   strcpy(pData, s);
   nLength = strlen(s);
};
```

```
Str :: Str(const Str &str) {
   int n = str.nLength;
   pData = new char[n+1];
   nLength = n;
   strcpy(pData,str.pData);
};
```

Putting Them Together

```
class Str
{
    char *pData;
    int nLength;
  public:
     //constructors
    Str();
    Str(char *s);
    Str(const Str &str);

     //accessors
    char* get_Data();
    int get_Len();

     //destructor
     ~Str();
};
```

```
char* Str :: get_Data()
{
    return pData;
};
```

```
int Str :: get_Len()
{
    return nLength;
};
```

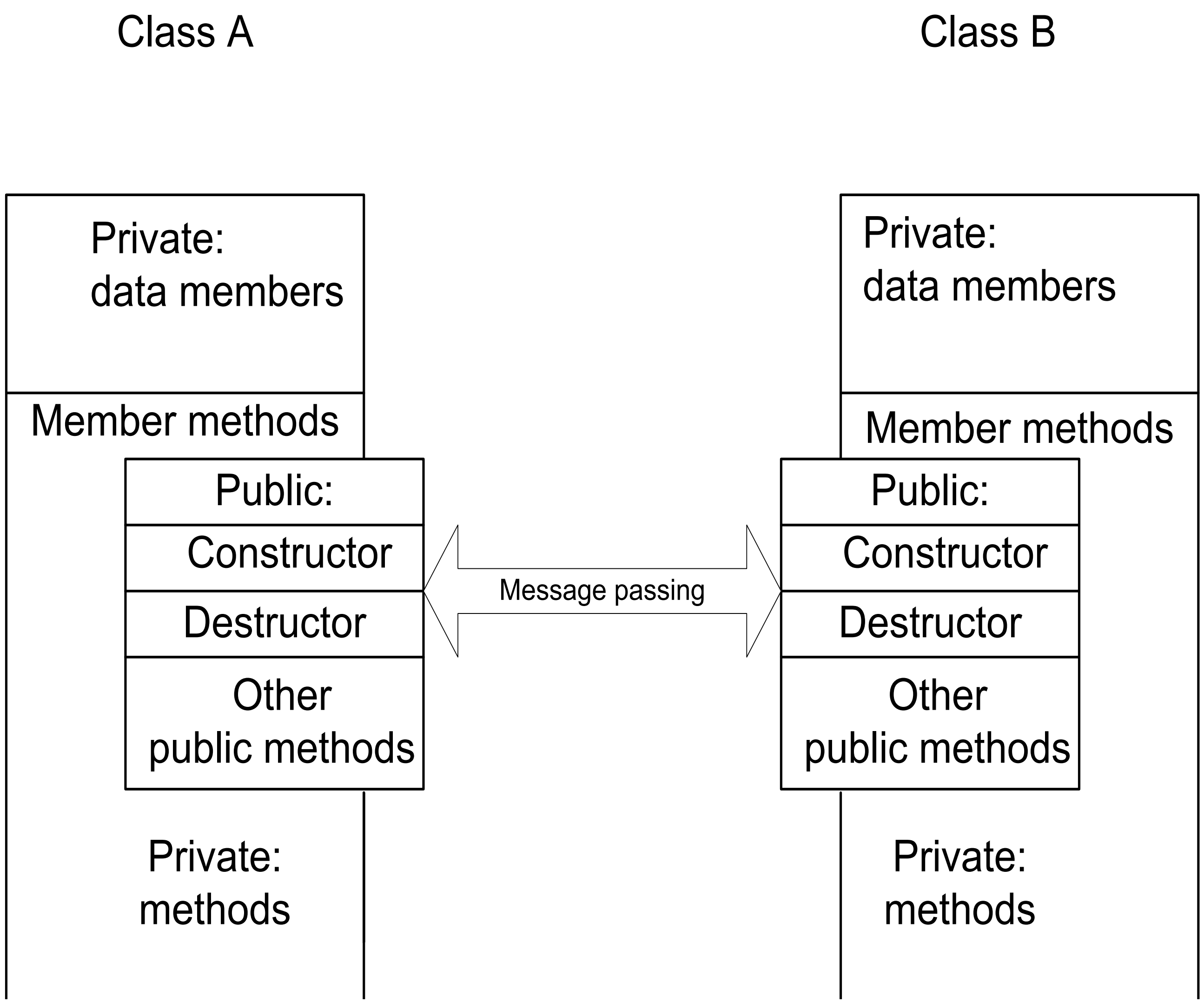
```
Str :: ~Str()
{
    delete[] pData;
};
```

Putting Them Together

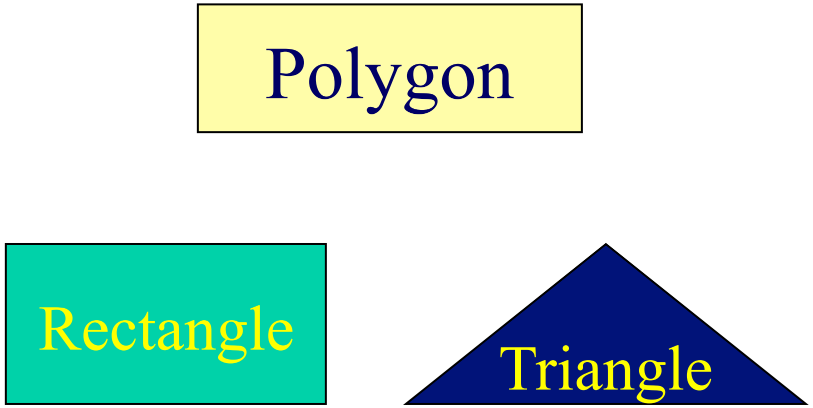
```
class Str
{
    char *pData;
    int nLength;
  public:
     //constructors
    Str();
    Str(char *s);
    Str(const Str &str);

     //accessors
    char* get_Data();
    int get_Len();

     //destructor
     ~Str();
};
```

```
int main()
{
    int x=3;
    Str *pStr1 = new Str("Joe");
    Str *pStr2 = new Str();
}
```

Interacting Objects



Inheritance Concept

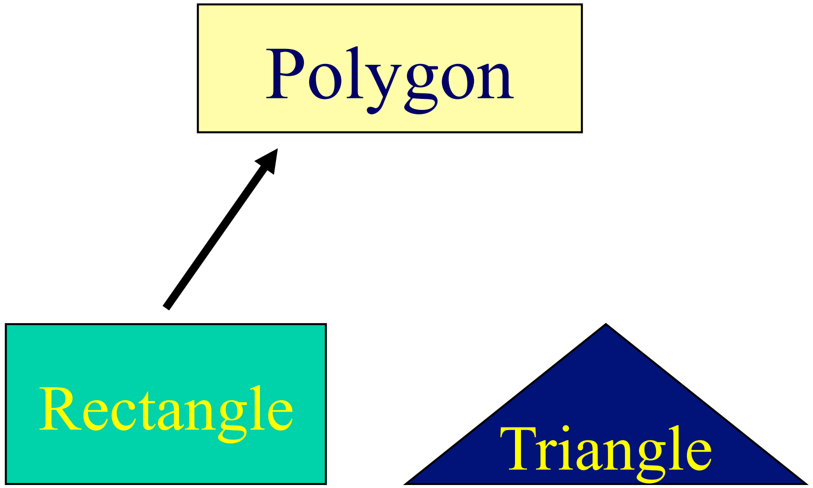



```
class Rectangle{
    private:
      int numVertices;
      float *xCoord, *yCoord;
    public:
      void set(float *x, float *y, int nV);
      float area();
};
```

```
class Polygon{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
};
```

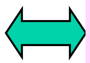
```
class Triangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

Inheritance Concept



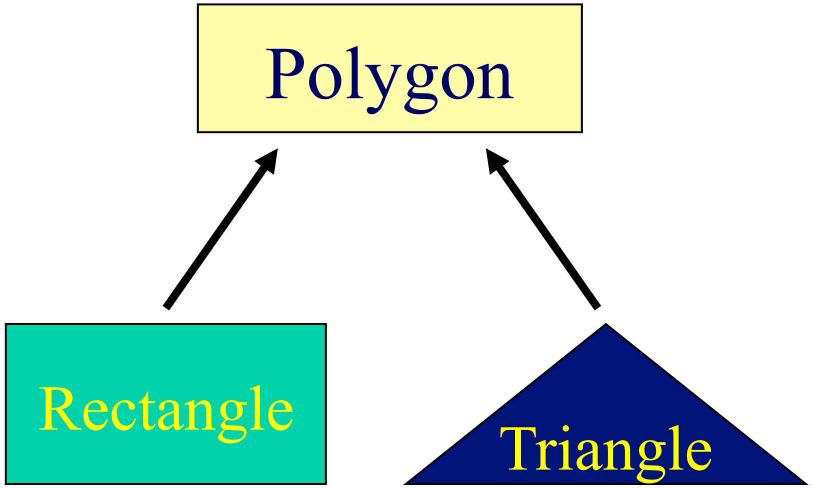

```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
};
```

```
class Rectangle : public Polygon{
    public:
       float area();
};
```



```
class Rectangle{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
       float area();
};
```

Inheritance Concept

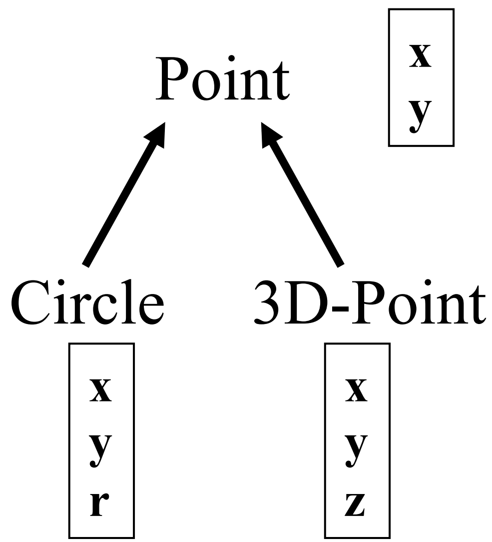



```
class Polygon{
     protected:
       int numVertices;
       float *xCoord, float *yCoord;
     public:
        void set(float *x, float *y, int nV);
};
```

```
class Triangle : public Polygon{
     public:
      float area();
};
```

```
class Triangle{
     protected:
       int numVertices;
       float *xCoord, float *yCoord;
     public:
        void set(float *x, float *y, int nV);
        float area();
};
```

Inheritance Concept



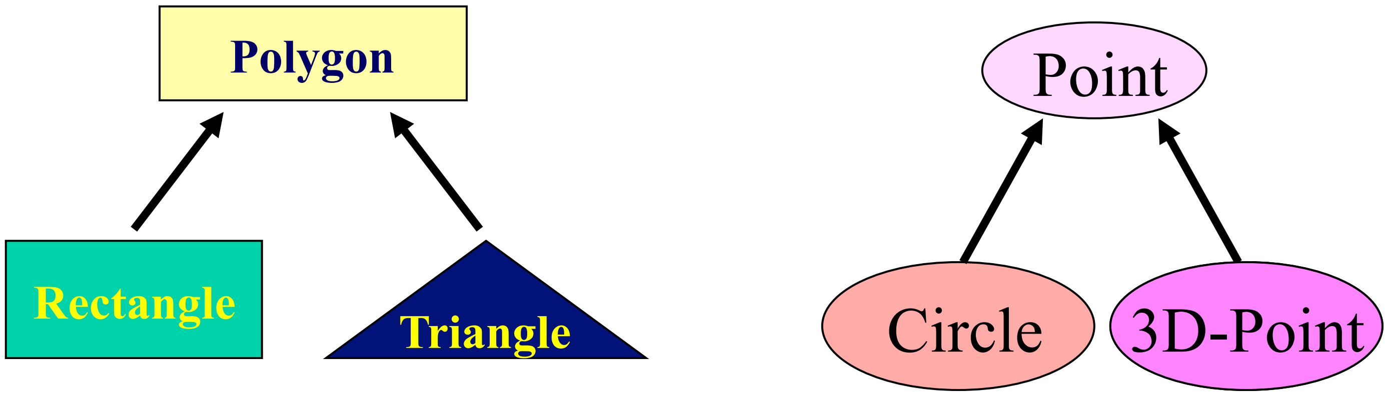

```
class Point{
    protected:
      int x, y;
    public:
      void set (int a, int b);
};
```

```
class Circle : public Point{
    private:
          double r;
};
```

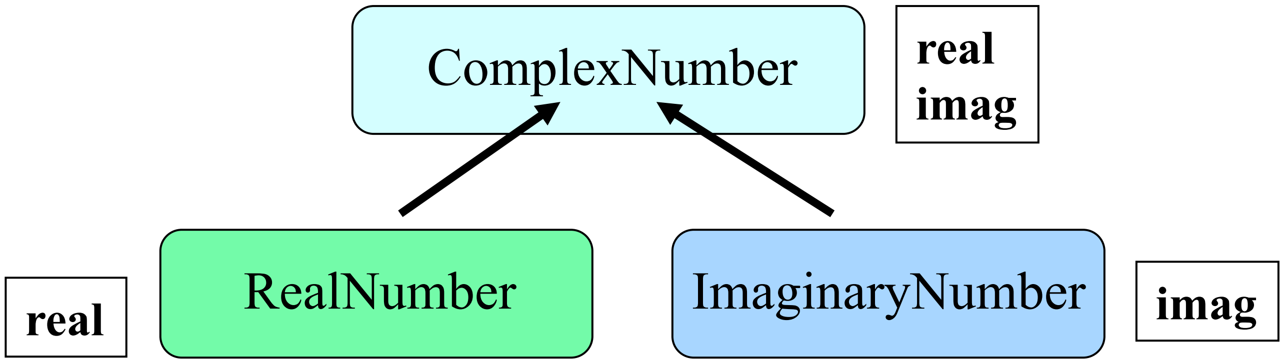
```
class 3D-Point: public Point{
    private:
          int z;
};
```

Inheritance Concept

- Augmenting the original class



- Specializing the original class



Why Inheritance ?

Inheritance is a mechanism for

- building class types from existing class types
- defining new class types to be a
  - specialization
  - augmentation

  of existing types

Define a Class Hierarchy

- Syntax:

class *DerivedClassName* : access-level *BaseClassName*

where
  - access-level specifies the type of derivation
    - private by default, or
    - public
- Any class can serve as a base class
  - Thus a derived class can also be a base class

Class Derivation

Point

↑

3D-Point

↑

Sphere

```
class Point{
    protected:
        int x, y;
    public:
        void set (int a, int b);
};
```

```
class 3D-Point : public Point{
    private:
        double z;
    … …
};
```

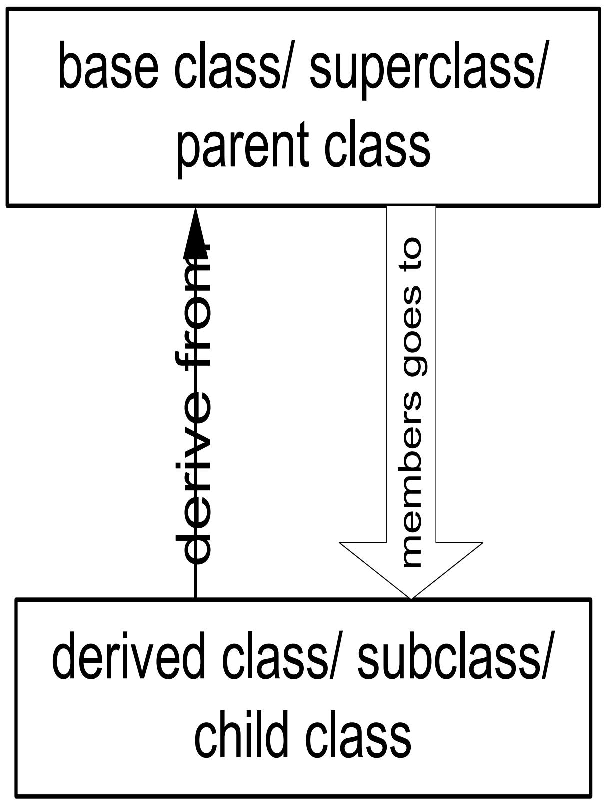
```
class Sphere : public 3D-Point{
    private:
        double r;
    … …
};
```

Point is the base class of 3D-Point, while 3D-Point is the base class of Sphere

What to inherit?

- In principle, every member of a base class is inherited by a derived class
  - just with different access permission

Access Control Over the Members



- Two levels of access control over class members
  - class definition
  - inheritance type

```
class Point{
    protected: int x, y;
    public: void set(int a, int b);
};
```

```
class Circle : public Point{
    … …
};
```

Access Rights of Derived Classes

Type of Inheritance

Access Control for Members

	private	protected	public
private	-	-	-
protected	private	protected	protected
public	private	protected	public

- The type of inheritance defines the access level for the members of derived class that are inherited from the base class

Class Derivation

```
class mother{
    protected: int mProc;
    public: int mPubl;
    private: int  mPriv;
};
```

private/protected/public

```
class daughter : --------- mother{
    private: double dPriv;
    public: void dFoo ( );
};
```

```
void daughter :: dFoo ( ){
    mPriv = 10;   //error
    mProc = 20;
};
```

```
class grandDaughter : public daughter {
    private: double gPriv;
    public: void gFoo ( );
};
```

```
int main() {
    /*….*/
}
```

What to inherit?

- In principle, every member of a base class is inherited by a derived class
  - just with different access permission
- However, there are exceptions for
  - constructor and destructor
  - operator=() member
  - friends

  Since all these functions are class-specific

Constructor Rules for Derived Classes

The default constructor and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

```
class A {
  public:
    A ( )
      {cout<< “A:default”<<endl;}
    A (int a)
      {cout<<“A:parameter”<<endl;}
};
```

```
class B : public A
{
  public:
    B (int a)
        {cout<<“B”<<endl;}
};
```

```
B test(1);
```

output:

```
A:default
B
```

Constructor Rules for Derived Classes

You can also specify an constructor of the base class other than the default constructor

DerivedClassCon (derivedClass args) : BaseClassCon (baseClass args)
{ DerivedClass constructor body }

```
class A {
  public:
    A ( )
      {cout<< "A:default"<<endl;}
    A (int a)
      {cout<<"A:parameter"<<endl;}
};
```

```
class C : public A {
  public:
    C (int a) : A(a)
      {cout<<"C"<<endl;}
};
```

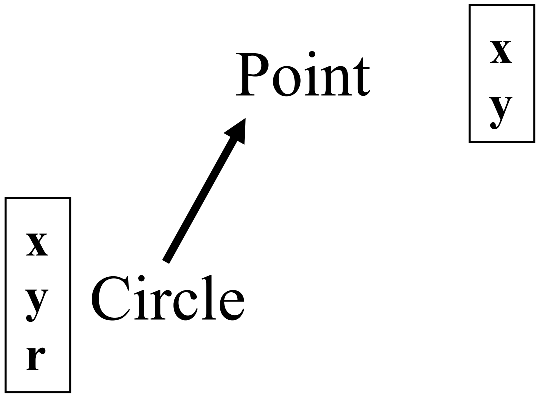
```
C test(1);
```

output:

```
A:parameter
C
```

Define its Own Members

The derived class can also define its own members, in addition to the members inherited from the base class




```
class Circle : public Point{
    private:
        double r;
    public:
        void set_r(double c);
};
```

```
class Point{
    protected:
        int x, y;
    public:
        void set(int a, int b);
};
```

```
class Circle{
    protected:
        int x, y;
    private:
        double r;
    public:
        void set(int a, int b);
        void set_r(double c);
};
```

Even more …

- A derived class can override methods defined in its parent class. With overriding,
  - the method in the subclass has the identical signature to the method in the base class.
  - a subclass implements its own version of a base class method.

```
class A {
  protected:
    int x, y;
  public:
    void print ()
          {cout<<"From A"<<endl;}
};
```

```
class B : public A {
  public:
    void print ()
        {cout<<"From B"<<endl;}
};
```

Access a Method

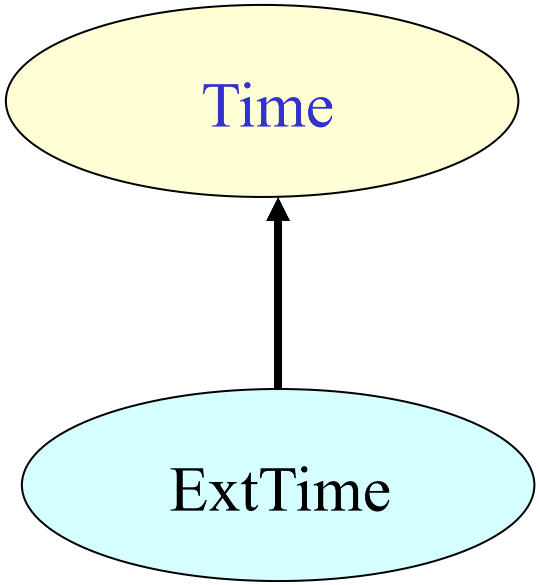
```
class Point{
    protected:
      int x, y;
    public:
      void set(int a, int b)
          {x=a; y=b;}
      void foo ();
      void print();
};
```

```
class Circle : public Point{
  private:  double r;
  public:
    void set (int a, int b, double c) {
        Point :: set(a, b); //same name function call
        r = c;
    }
    void print();  };
```

```
Point A;
A.set(30,50);  // from base class Point
A.print(); // from base class Point
```

```
Circle C;
C.set(10,10,100);   // from class Circle
C.foo ();  // from base class Point
C.print(); // from class Circle
```

Putting Them Together

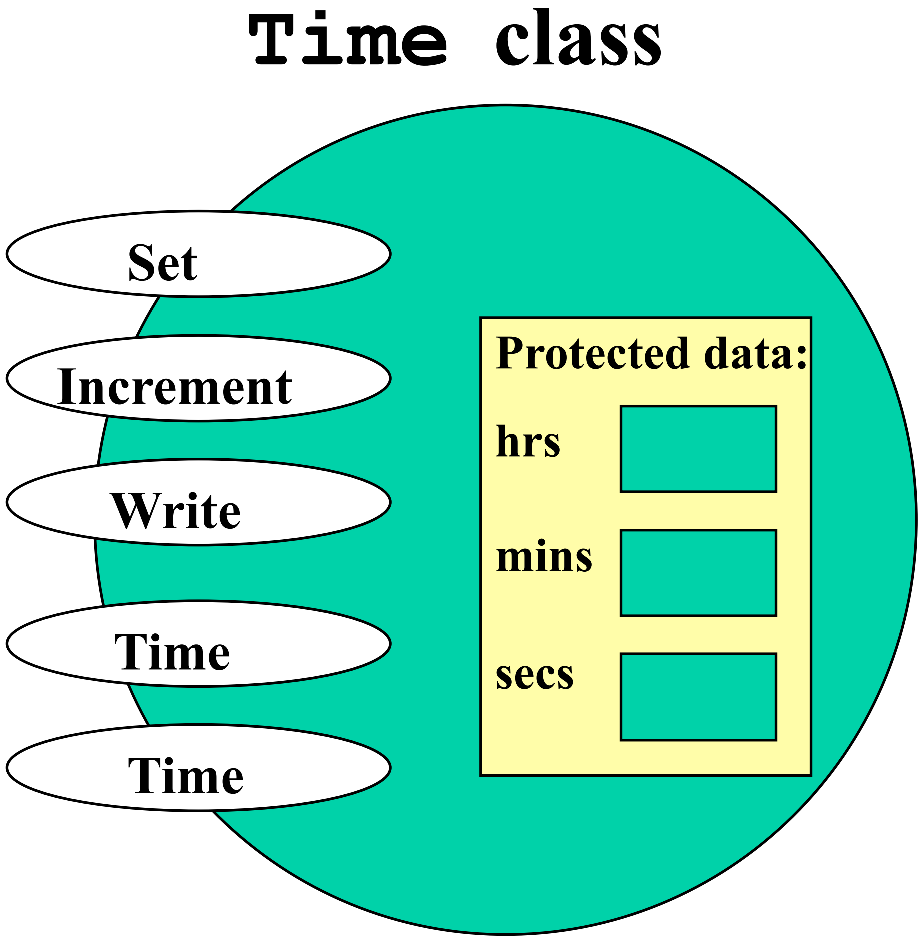


- Time is the base class
- ExtTime is the derived class with public inheritance
- The derived class can
  - inherit all members from the base class, except the constructor
  - access all public and protected members of the base class
  - define its private data member
  - provide its own constructor
  - define its public member functions
  - override functions inherited from the base class

class **Time** Specification

```
// SPECIFICATION   FILE                                    ( time.h)

class  Time{

 public :

    void     Set ( int h, int m, int s ) ;
    void     Increment ( ) ;
    void     Write ( )  const ;
    Time     ( int initH, int initM, int initS ) ;   //  constructor
    Time     ( ) ;                                   //  default constructor

 protected :

    int          hrs ;
    int          mins ;
    int          secs ;
} ;
```

Class Interface Diagram

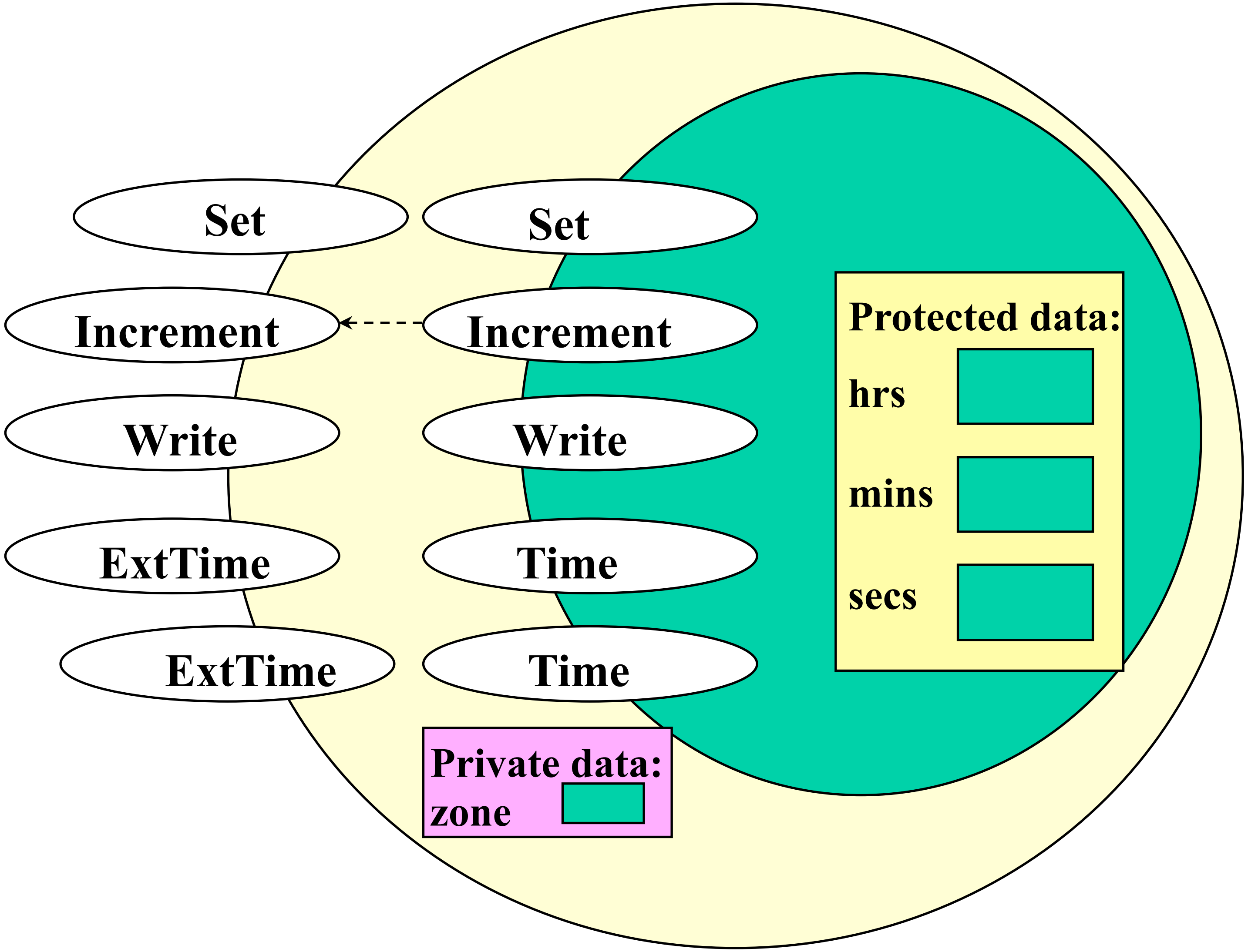


Derived Class **ExtTime**

```
// SPECIFICATION   FILE                                    ( exttime.h)

#include   "time.h"

enum  ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT } ;

class  ExtTime  :  public  Time
             // Time is the base class and use public inheritance
{
  public :
    void          Set ( int h, int m, int s, ZoneType timeZone ) ;
    void          Write ( )  const;     //overridden
    ExtTime     (int initH, int initM, int initS, ZoneType initZone ) ;
    ExtTime     ();    // default constructor

private :
    ZoneType  zone ;         //  added data member

} ;
```

Class Interface Diagram

`ExtTime` class



Implementation of `ExtTime`

Default Constructor

```
ExtTime :: ExtTime ( )
{
    zone = EST ;
}
```

The default constructor of base class, Time(), is automatically called, when an ExtTime object is created.

```
ExtTime et1;
```

et1

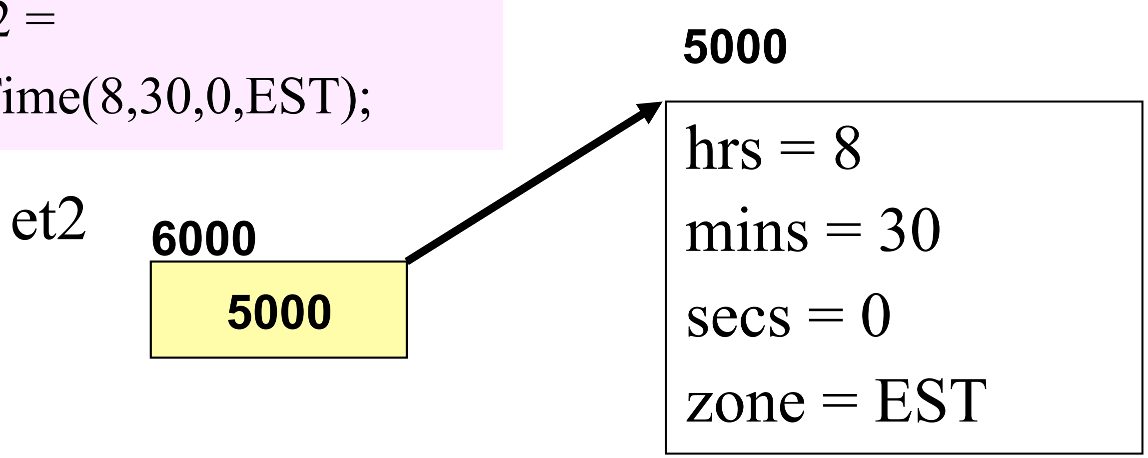
hrs = 0
mins = 0
secs = 0
zone = EST

Implementation of **`ExtTime`**

Another Constructor

```
ExtTime :: ExtTime (int initH, int initM, int initS, ZoneType initZone)
            : Time (initH, initM, initS)
            // constructor initializer
{
        zone  = initZone ;
}
```

```
ExtTime *et2 =
    new ExtTime(8,30,0,EST);
```



Implementation of **`ExtTime`**

```
void  ExtTime :: Set (int h, int m, int s, ZoneType timeZone)
{
      Time :: Set (hours, minutes, seconds);  // same name function call
      zone  = timeZone ;
}
```

```
void  ExtTime :: Write ( )   const  // function overriding
{
   string  zoneString[8] =
            {"EST", "CST", MST", "PST", "EDT", "CDT", "MDT",
     "PDT"} ;

  Time :: Write ( ) ;
  cout  << '   '<<zoneString[zone]<<endl;
}
```

Working with **ExtTime**

```
#include "exttime.h"
… …
int main()
{
        ExtTime    thisTime ( 8, 35, 0, PST ) ;
        ExtTime    thatTime ;                      // default constructor called

        thatTime.Write( ) ;                        // outputs 00:00:00 EST

        thatTime.Set (16, 49, 23, CDT) ;
        thatTime.Write( ) ;                        // outputs 16:49:23 CDT

        thisTime.Increment ( ) ;
        thisTime.Increment ( ) ;
        thisTime.Write ( ) ;                       // outputs 08:35:02  PST
}
```

Take Home Message

- Inheritance is a mechanism for defining new class types to be a specialization or an augmentation of existing types.
- In principle, every member of a base class is inherited by a derived class with different access permissions, except for the constructors

Polymorphism

Object-Oriented Concept

- Encapsulation
  - ADT, Object
- Inheritance
  - Derived object
- Polymorphism
  - Each object knows what it is

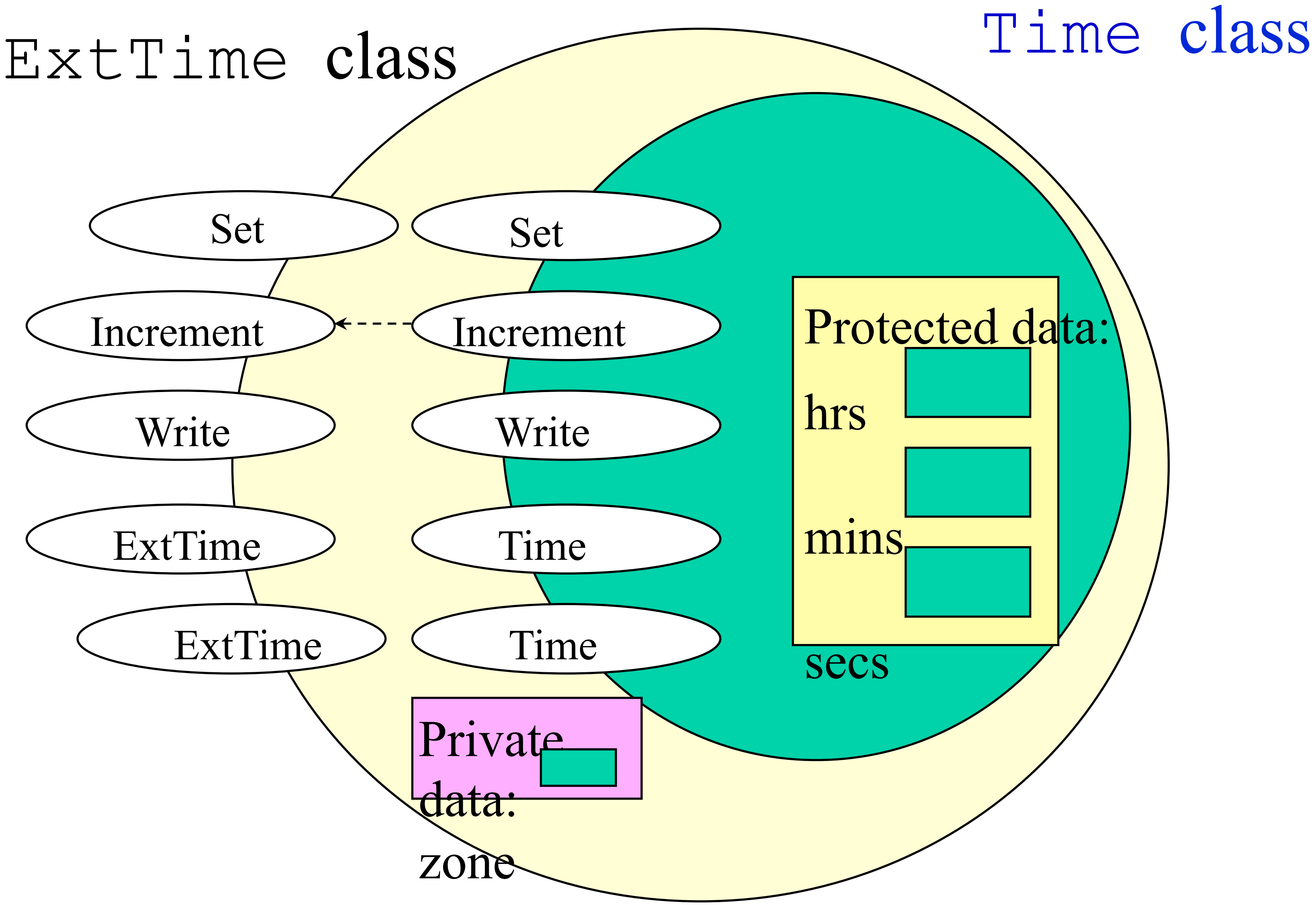
Polymorphism – An Introduction

- *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance

Before we proceed….

- Inheritance – Basic Concepts
  - Class Hierarchy
    - Code Reuse, Easy to maintain
  - Type of inheritance : public, private
  - Function overriding

Class Interface Diagram



Why Polymorphism?--Review: Time and ExtTime Example by Inheritance

```
void  Print  (Time   someTime )  //pass an object by value
{
        cout  <<  "Time is  " ;
        someTime.Write (  ) ;            // Time :: write()
        cout  <<  endl ;
}
```

CLIENT CODE

```
Time        startTime ( 8, 30, 0 ) ;
ExtTime    endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

Time is 08:30:00
Time is 10:45:00

Static Binding

- When the type of a formal parameter is a parent class, the argument used can be:

  the same type as the formal parameter,

  or,

  any derived class type.

- Static binding is the compile-time determination of which function to call for a particular object based on the type of the formal parameter

- When pass-by-value is used, static binding occurs

Can we do better?

```
void  Print  (Time   someTime )  //pass an object by value
{
            cout  <<  "Time is  " ;
            someTime.Write (  ) ;                       // Time :: write()
            cout  <<  endl ;
}
```

CLIENT CODE

```
Time        startTime ( 8, 30, 0 ) ;
ExtTime     endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

Time is 08:30:00
Time is 10:45:00

Polymorphism – An Introduction

- *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance
- Allows run-time interpretation of object type for a given class hierarchy
  - Also Known as "Late Binding"
- Implemented in C++ using virtual functions

Dynamic Binding

- Is the run-time determination of which function to call for a particular object of a derived class based on the type of the argument
- Declaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding
- Dynamic binding requires pass-by-reference

Virtual Member Function

```
//  SPECIFICATION FILE                        ( time.h )

class  Time
{
public :

     . . .

     virtual   void   Write ( )  ;            //  for dynamic binding
     virtual ~Time();                    // destructor

private :

     int          hrs ;
     int          mins ;
     int            secs ;
} ;
```

This is the way we like to see…

```
void  Print  (Time *   someTime )
{
    cout  <<  "Time is  " ;
    someTime->Write (  ) ;
    cout  <<  endl ;
}
```

CLIENT CODE

```
Time          startTime( 8, 30, 0 ) ;
ExtTime    endTime(10, 45, 0, CST) ;

Time *timeptr;
timeptr = &startTime;
Print ( timeptr ) ;                →  Time::write()

timeptr = &endTime;
Print ( timeptr ) ;                →  ExtTime::write()
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00  CST
```

Virtual Functions

- Virtual Functions overcome the problem of run time object determination
- Keyword virtual instructs the compiler to use late binding and delay the object interpretation
- How ?
  - Define a virtual function in the base class. The word virtual appears only in the base class
  - If a base class declares a virtual function, it must implement that function, even if the body is empty
  - Virtual function in base class stays virtual in all the derived classes
  - It can be overridden in the derived classes
  - But, a derived class is not required to re-implement a virtual function. If it does not, the base class version is used

Polymorphism Summary:

- When you use virtual functions, compiler store additional information about the types of object available and created
- Polymorphism is supported at this additional overhead
- Important :
  - virtual functions work only with pointers/references
  - Not with objects even if the function is virtual
  - If a class declares any virtual methods, the destructor of the class should be declared as virtual as well.

Abstract Classes & Pure Virtual Functions

- Some classes exist logically but not physically.
- Example : Shape
  - `Shape s; // Legal but silly..!! : “Shapeless shape”`
  - Shape makes sense only as a base of some classes derived from it. Serves as a “category”
  - Hence instantiation of such a class must be prevented

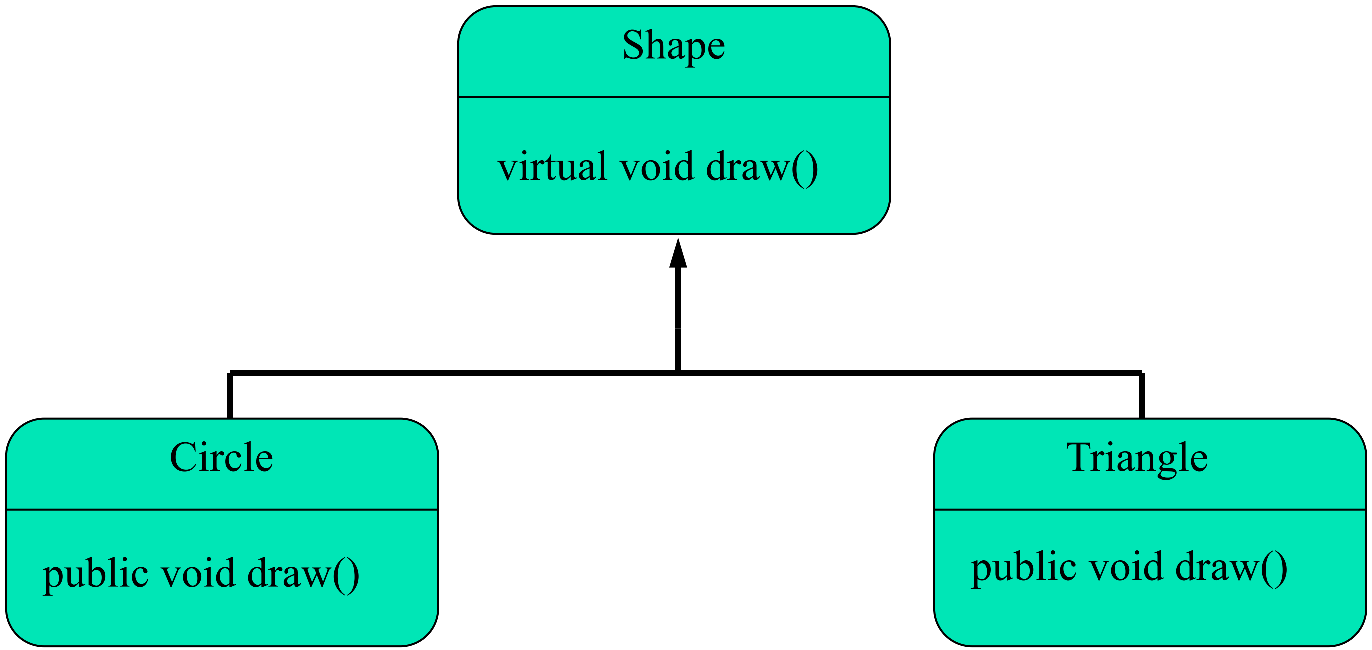
```
class Shape     //Abstract
{
  public :
  //Pure virtual Function
  virtual void draw() = 0;
}
```

A class with one or more pure virtual functions is an Abstract Class

Objects of abstract class can’t be created

```
Shape s; // error : variable of an abstract class
```

Example


Shape
virtual void draw()
Circle
public void draw()
Triangle
public void draw()

- A pure virtual function <u>not defined</u> in the derived class remains a pure virtual function.
- Hence derived class also becomes abstract

```
class Circle : public Shape { //No draw() - Abstract
   public :
   void print(){
     cout << “I am a circle” << endl;
   }}
class Rectangle : public Shape {
   public :
   void draw(){ // Override Shape::draw()
     cout << “Drawing Rectangle” << endl;
   }}
```

```
Rectangle r; // Valid
Circle c; // error : variable of an abstract class
```

Pure virtual functions : Summary

- Pure virtual functions are useful because they make explicit the abstractness of a class
- Tell both the user and the compiler how it was intended to be used
- Note : It is a good idea to keep the common code as close as possible to the root of you hierarchy

Summary ..continued

- It is still possible to provide definition of a pure virtual function in the base class
- The class still remains abstract and functions must be redefined in the derived classes, but a common piece of code can be kept there to facilitate reuse
- In this case, they can not be declared inline

```
class Shape { // Abstract
public :
  virtual void draw() = 0;
};
// OK, not defined
```

```
class Rectangle : public Shape
{
  public :
   void draw(){
     Shape::draw(); // Reuse
```

Take Home Message

- Polymorphism is built upon class inheritance
- It allows different versions of a function to be called in the same manner, with some overhead
- Polymorphism is implemented with virtual functions, and requires pass-by-reference