Topics

Topics

- Click here to topic

Introduction to Object Orientation

Characteristics of Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

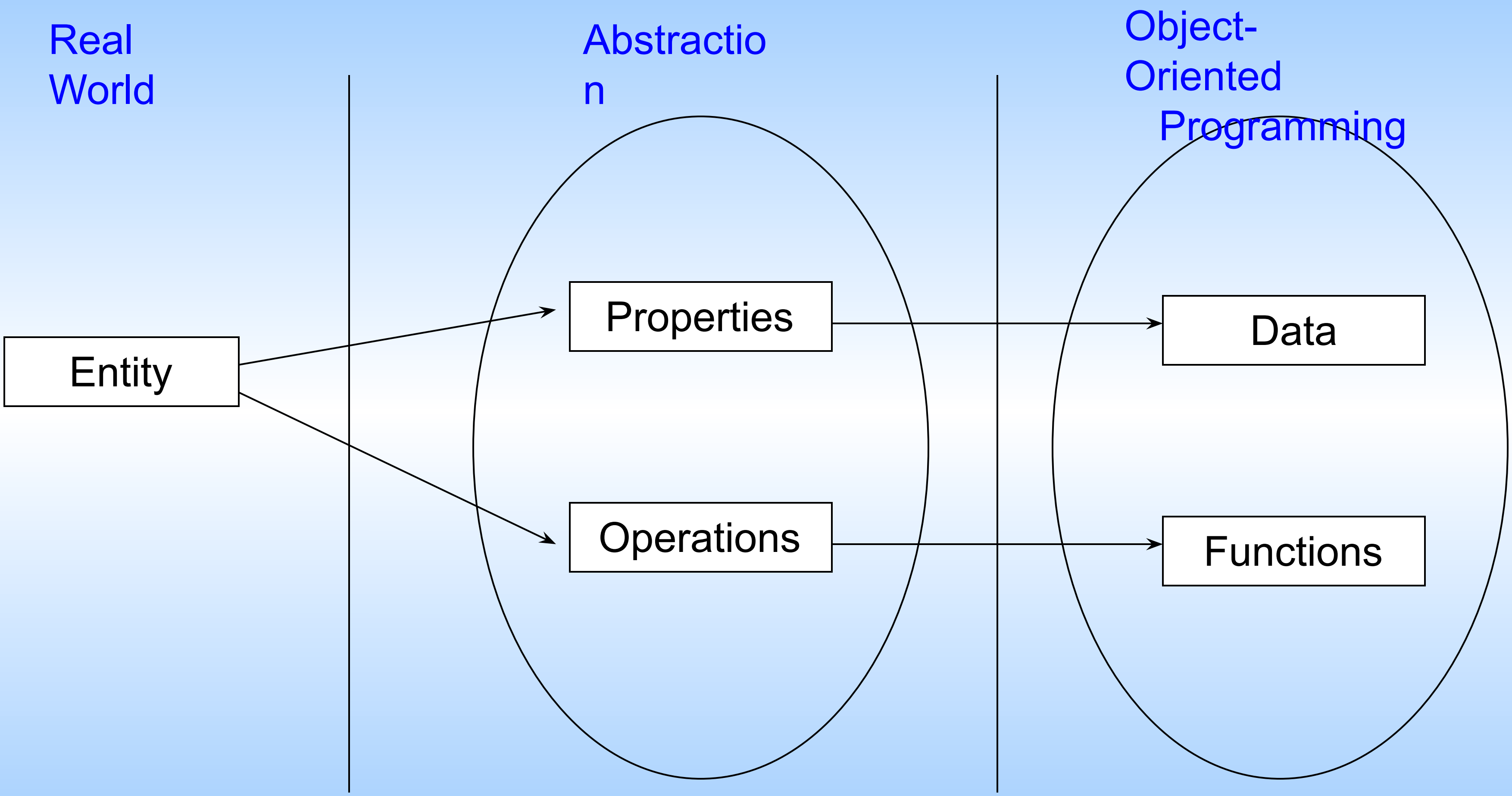
- I treat patients
- I have a stethoscope
- I wear white coat

Doctor

This is Abstraction

Showing only the Essential Part

Abstraction



Mapping real world entity to OOP

Abstraction

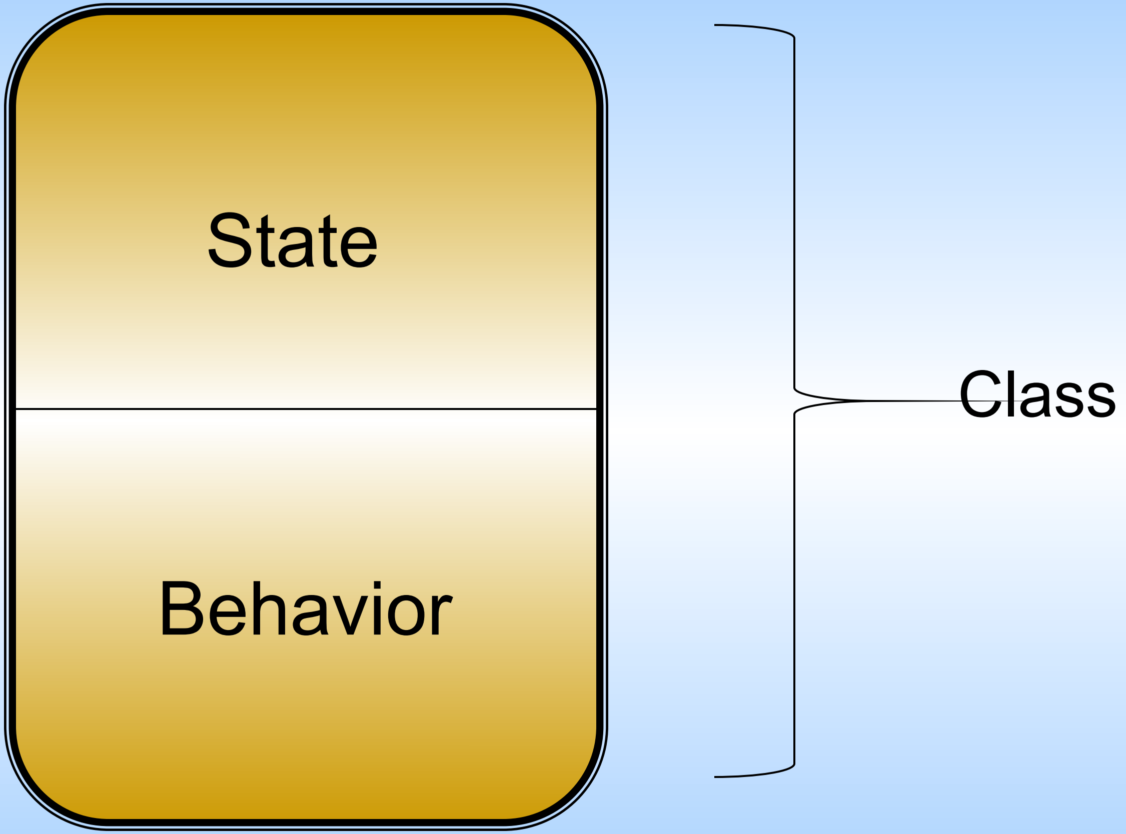
"A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information." - [IEEE, 1983]

- Abstraction specifies necessary and sufficient descriptions rather than implementation details.
- Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).
- Building up data types from the predefined data types is data abstraction.

Characteristics of Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Encapsulation



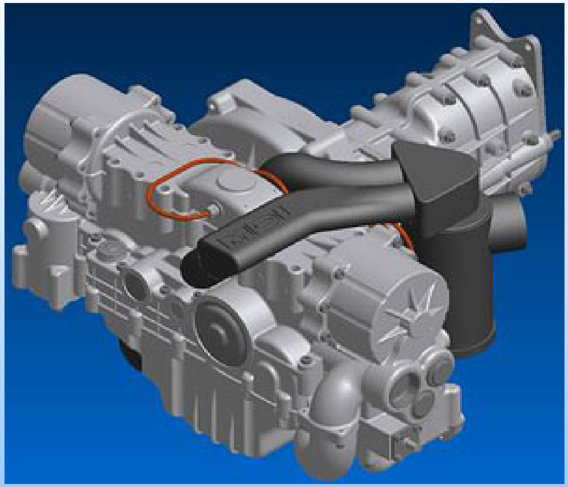
Bundling the data and functions into a unit

Encapsulation

- **Hide implementation details**
  - Only expose the interface
  - Change the implementation without affecting interface

Interfac e

Implementation



Encapsulation

- Bundling of data and functions together into a single unit (class).

- The advantages of encapsulation are
  - Data hiding
  - Information hiding

- The data is not accessible outside, only the function that are wrapped in the class can access it. This insulation of data is called as data hiding.

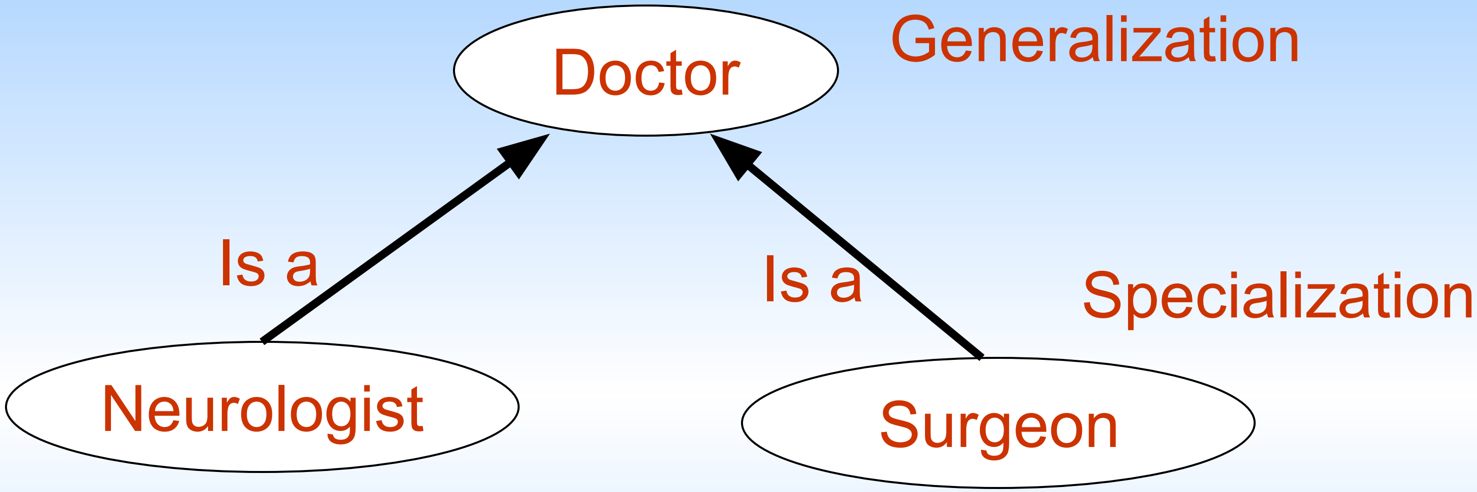
- Hiding the implementation details of the wrapped functions from the user is called as information hiding (Separating what is to be done from how it is to be done).

Example: Driver may know how to use steering(how), but not the steering mechanism(what).

Characteristics of Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Inheritance



states *"is a"*
Relationship

Characteristics of Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Polymorphism

- ***Polymorphic: from the Greek for "many forms"***

- Polymorphism means ability to take multiple or many forms.

- In programming languages, polymorphism means that same code / operation / object behaves differently in different context.

- It provides a single interface to entities of different types.

Classes & Objects

Object

State

Behavior

Identity

(Booch)

State

- The state of an object encompasses all of the (static) properties of the object plus the current (dynamic) values of each of these properties
- A property is an inherent or distinctive characteristic, trait, quality, or feature that contribute to making an object uniquely that object
- We will use the word attribute, or data member, to refer to the state of an object

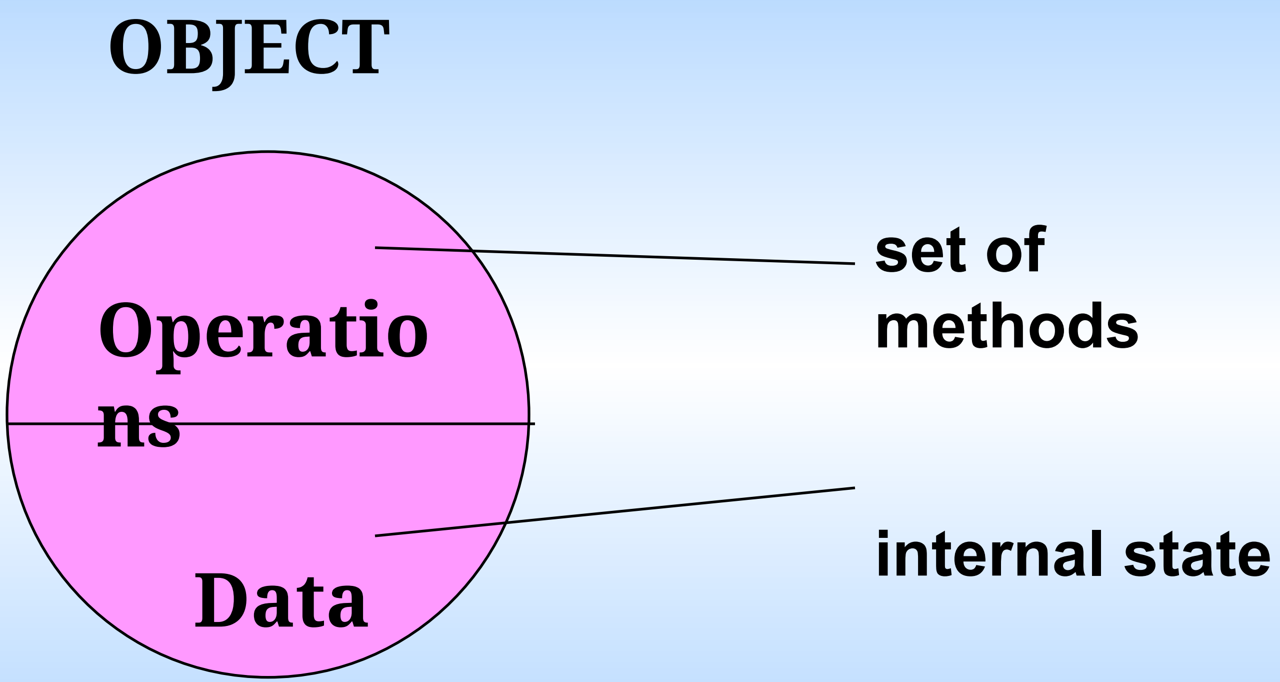
Behavior

- Behavior is how an object acts and reacts, in terms of state changes and interactions with other objects.
- An operation is some action that one object performs upon another in order to elicit a reaction.

Identity

- Identity is the property of an object that distinguishes it from all other objects.

What is an object?



Class & Object

Class:

- Putting behaviours(functions) and attributes(data) together.
- A class does not exist in the memory of computer during program execution.

Object:

- An instance of a class. There can be more than one instance for a class.

Example : **Writing material** is a class. **Pen** and **Pencil** are **objects** of the class.

Class & Object

Account
acct_no
name
balance
withdraw()
deposit( )

Account: acct1
431245
Patrick
50567.78

Account: acct2
431246
Meyer
35567.78

Class & Object

Class	Object
Class is a data type	Object is an instance of class data type
It is the prototype or model.	It is a container
It does not occupy memory.	It occupies memory
It is compile-time entity	It is run-time entity

C++

C++ : The History

- *Bjarne Stroustrup* developed C++ (originally named "*C with Classes*") during the 1980s.
- The name C++ was coined by Rick Mascitti in 1983.
- Standard was ratified in 1998 as *ISO/IEC 14882:1998*
- New version of the standard (known informally as C++0x) is being developed.

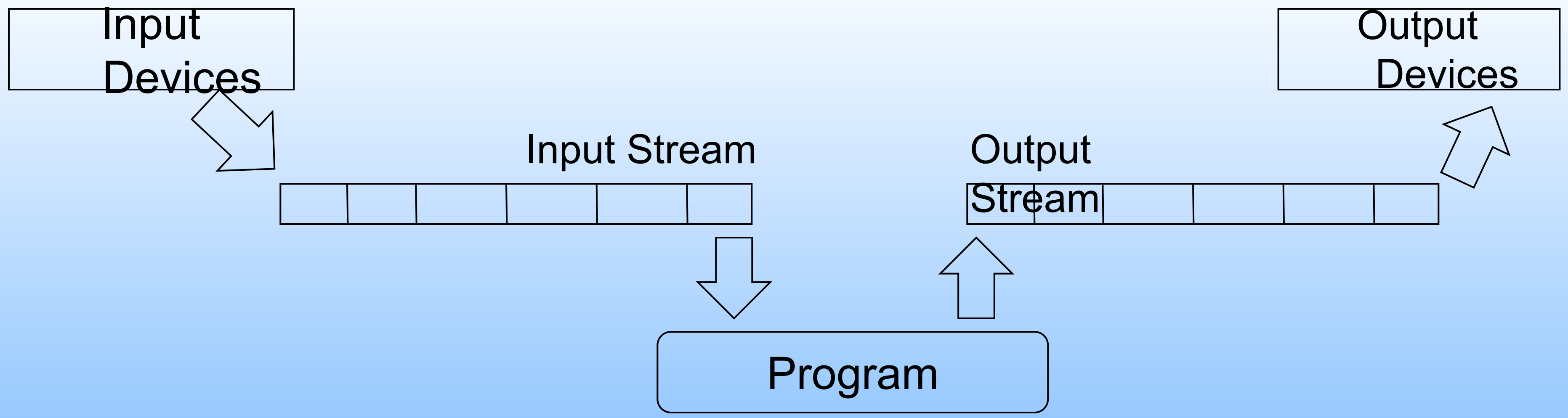
Streams in C++

Streams in C++

Input / Output Statements in C++

- Stream - is an inter-mediator between the I/O device and the user.
  - cin - predefined object for input stream.
  - cout - predefined object for output stream.

- User should include the iostream.h.



Streams for Input and Output

```
#include<stdio.h>
void main()
{
printf("Good
Morning\n");
}
```

Bore

in C

```
#include<iostream.h>
void main()
{
cout<<"Good Morning\n";
}
```

in C++

<<

Good Morning

- *printf() is used in C*
- *Output operator is better than output function*

Why << ?

- Assignment operator = a candidate !
- Input operator different from output operator
- cout=a=b means cout=(a=b)
- How about < and > ?
- << and >> are asymmetric not commonly used

Streams for Input and Output

Precedence of << is low

```
#include<iostream.h>
int main()
{
int a=1,b=2,c=3;
cout<<“a*b+c=“<< a*b+c<<‘\n’;
}
```

Output
:
5

Streams for Input and Output

- Parentheses must be used when operator of lower precedence available

```
#include<iostream.h>
int main()
{
    int a=1,b=2,c=3;
    cout<<"a^b/c ="<<(a^b/c)<<'\n';
}
```

Output
:
1

Streams for Input and Output

- Left shift operator << can be used in an output statement but it must appear within parentheses

```
#include<iostream.h>
void main()
{
int a=1,b=2;
cout<<“a<<b=“<<(a<<b)<<‘\n’;
}
```

Output
:
4

Streams for Input and Output

C++ associates set of manipulators with the output stream

```
#include<iostream.h>
void main()
{
int amount=123;
cout<<dec<<amount<<‘\n ‘
     <<oct<<amount<<‘\n ’
     <<hex<<amount;
}
```

Output:

123

173

7b

Streams for Input and Output

The stream cin is used for input

>> extraction operator (get from)

```
#include<iostream.h>
void main()
{
	int amount;
	cout<<"Enter the amount…\n";
	cin>>amount;
	cout<<"The amount you entered was ";		cout<<amount;
}
```

Enter the amount…
10
The amount you entered was
10

C++ sends the value that you enter to the variable amount

Comments

C++ comments is token // sequence. Wherever this sequence appears (unless it is inside a string), everything to the end of the current line is a comment

```
void main()
// I am a single line comment --- I am very useful…
/* *******we are multi line comments
 we too are very useful********** */
```

Declarations and Definitions

Declarations

- Before any identifier can be used in a C++ program it must be *declared*
- i.e. Its type must be specified to inform the compiler what kind of entity the name refers to

Declarations -Examples

```
char ch;
int count=1;
char* name="NUTS";
struct complex{float re,im;};
complex cvar;
extern complex sqrt(complex); (not defined)
extern int error_number; (not defined)
typedef complex point;
float real(complex* p) { return p->re;};
const double pi=3.141592653589;
struct user;(not defined)
```

Definition

- There must be exactly one definition for each name in a C++ program.

```
int count;
int count;     error
```

- Definition allocates appropriate memory.

- Some definition specify a “value” for the entities they define

```
int sum=0;
```

For types, functions and constants the value is permanent. For non constant data types the initial value may be changed later.

Declarations Vs Definition

- Declarations
  - It can be done more than once
  - Tells the compiler about the entity type and name
- Definition
  - It can be done only ones
  - Allocates memory for the variable

For automatic and register variables, there is no difference between definition and declaration.

Scope

- A declaration introduces a scope

- A declaration of a name in a block can hide a declaration in an enclosing block or a global name

```
int x;        → Global x
void f()
{
int x;        → Local x
x=1;          → Assign to Local X
{
 int x;       → Hides local x
x=2;          → Assign to Second local X
}
              → Assign to first local X
x=3;
}
```

```
int *p=&x;    //take address of global X
```

Scope

```
#include<iostream.h>
int amount=456;
void main()
{
int amount=123;
cout<<::amount;
cout<<'\n';
cout<<amount;
}
```

Global scope

Local scope (function)

:: unary scope resolution operator

Output:

456

123

Lifetimes of Data Object

- The *lifetime of a data* object is the time that it remains in existence during the execution of the program

Scope Vs Lifetime

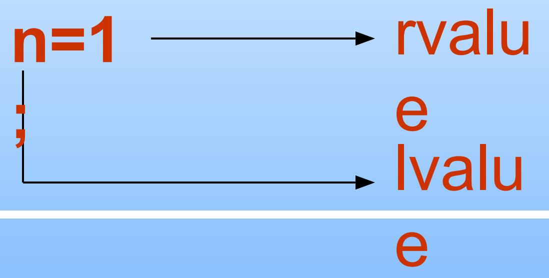
- Identifier has a scope i.e. the part of the program in which it can be referenced (or active).
- Data object has a lifetime i.e. the time it remains in existence (period of time that a variable is assigned memory).
- A data object is a region of memory in which a value can be stored it is characterized by its address, names (if any) type and its value.

lvalue and rvalue

lvalue (location value)

- lvalue is an expression referring to an object or function.
- lvalue is modifiable if it is not a function name, an array name or constant.
- If E is an expression of pointer type the *E is an lvalue referring to the object E points

E.g..



lvalue Vs rvalue

- In assignment operation

  3 = n;

  *Is **wrong** because the left-hand side should be a lvalue*

  Numeric literals, such as **3** and **3.14159**, are *rvalues*.

lvalue Vs rvalue

- An identifier that refers to an object is an lvalue, but an identifier that names an enumeration constant is an rvalue. For example:

```
 enum color { red, green, blue };
color c;
...
c = green;    // ok
blue = green;    // error
```

The second assignment is an error because **blue** is an rvalue.

lvalue Vs rvalue

- Although you can't use an rvalue as an lvalue, you can use an lvalue as an rvalue. For example, given:

  int m, n;

  You can assign the value in **n** to the object designated by **m** using:

  m = n;

  This assignment uses the lvalue expression n as an rvalue.
- Strictly speaking, a compiler performs what the C++ Standard calls an *lvalue-to-rvalue conversion* to obtain the value stored in the object to which **n** refers.

Function Prototyping

Function Prototyping

- ANSI C allows function prototyping

- borrowed from C++

- the return value, function name and number and type of arguments can be specified in the function prototype, right before main()

- While ANSI C allows function prototyping C++ requires it.

```
void swap(int&,int&);
```

→ ***Function prototyping must in C++***

Function Prototyping

```
#include<iostream.h>
void main()
{
void show();
show();
}
void show()
{
cout<<"Welcome to C++\n";
}
```

Function Prototype

Scope of the function is within main()

Actual Definition

Output:
Welcome to C++

Function Prototyping

```
#include<iostream.h>

void main()
{
void f();
void
f();
show();
}
void f()
{
show();
}
void show()
{
cout<<"Welcome to C++\n";
}
```

Scope Problem

Default Function Arguments

Default Function Arguments

A C++ function can have default values for some of the parameters

```
#include<iostream.h>
void function_2(int i,int j=2);
main(void)
{
int i=1;
int j=5;
function_2(i,j);
function_2(j);
}
void function_2(int i,int j)
{ cout<<“i is “<<i<<‘\n’;
cout<<“j is “<<j<<‘\n’;
}
```

→ Function Prototype

Default value for j

→ Actual Definition

output :

i is 1
j is 5
i is 5
j is 2

Default values should be specified in function prototype

Default Function Arguments

What happens if j is initialized at the time that function is called???

```
#include<iostream.h>
void function_2(int i,int j=2);
main(void)
{
int i=1;
int j;
function_2(i,j=8);
}
void function_2(int i,int j)
{ cout<<"i is "<<i<<'\n';
cout<<"j is "<<j<<'\n';
}
```

output
:
i is 1
j is 8

Default Function Arguments

```
#include<iostream.h>
void function_2(int i,int
j=2);
main(void)
{
int i;
function_2(i);
}
void function_2(int i,int j=8)
{ cout<<“i is “<<i<<‘\n’;
cout<<“j is “<<j<<‘\n’;
}
```

?

Error: previously specified default argument cannot be changed in the function

Default Function Arguments

```
#include<iostream.h>
void function_2(int i=1,int j, int k=2);
main(void)
{
int i,k;
int j=2;
function_2(i,j,k);
}
void function_2(int i, int j, int k)
{ cout<<“i is “<<i<<‘\n’;
cout<<“j is “<<j<<‘\n’;
cout<<“k is “<<k<<‘\n’;
}
```

No default value

Rule: Only the last arguments in a parameter list can be initialized.

Error: Default value missing

Default Function Arguments

```
#include<iostream.h>
void function_2(int i,int j=2,k=3);
main(void)
{
int i,k;
int j=2;
function_2(i,j,k);
}
void function_2(int i, int j, int k)
{ cout<<“i is “<<i<<‘\n’;
cout<<“j is “<<j<<‘\n’;
cout<<“k is “<<k<<‘\n’;
}
```

output
:
i is 1
j is 2
k is 3

Default Function Arguments

```
#include<iostream.h>
void show (int  =1,float =2.3, long =4);
void main()
{
show();
show(5);
show(6,7.8);
show(9,10.11,12L);
}
void show(int first, float second, long third)
{
cout<<“first = “<<first<<‘\n’;
cout<<“second = “<<second<<‘\n’;
cout<<“third is “<<third<<‘\n’;
}
```

Rule: You cant omit an argument unless you omit all the arguments to the right

Default Function Arguments

- Default values provides flexibility
- Functions called with same arguments can be given default values

```
#include<iostream.h>
void print(int value, int base=10);
void main()
{
print(31):
print(31,10)
print(31,16)
print(31,2);
}
```

Inline functions

Inline functions

- C++ provides inline keyword
- A new copy of the function to be inserted in each place it is called
- Inline reduces the overhead of a function calls
- Program becomes larger
- Unlike macros they have type checking rules and scope

```
double old_a;
#define DBL(a) ((old_a=a),((a)+(a)))
inline int dbl(int a) {old_a=a; return a+a;}

void f(int* pi, char *pc)
{
double old_a=7;
old_a=dbl(*pi++);
old_a=dbl(pc);//error argument type mismatch
}
```

Macro

Inline function

correct call will expand something like

```
int tmp;
old_a=((tmp=*pi++),(::old_a=tmp), (tmp+tmp)
```

to ensure the argument expression is evaluated only once

```
void f(int* pi, char *pc)
{
double old_a=7;
old_a=DBL(*pi++);
old_a=DBL(pc);//error in expansion
}
```

macro expands to

```
void f(int* pi, char *pc)
{
double old_a=7; // hides the global
    'old_a'
old_a=((old_a=*pi++),((*pi++)+(*pi++)));
old_a= ((old_a=pc),((pc)+(pc)));
}
```

Two errors

1) Adding pointers
2) Assigning char* to double

Inline Vs Macros

Inline	Macros
Part of language easy to debug	Difficult to debug
Not always expanded	Always Expanded
Has type checking rules and scope	Does not have these
Compile time	Preprocessing

References

References

- A reference is an *alternative name* for an object
- It is an '*alias*'
- The unary operator & with typename identifies a reference.
- The notation X& means reference to X

```
int i=1;
int &r=i;// r and i refer to the same int
int x=r; //x=1
r=2;//i=2
```

- A reference must *be initialized*.

Reference

Despite appearance, no operator operates on a reference.

```
int ii=0;
int &rr=ii;
rr++; // ii is incremented
```

The value of a reference can not be changed after initialization.

It always refers to the object it was initialized.

References

```
#include<iostream.h>
void main()
{
int actualint=123;
int
&otherint=actualint;
cout<<'\n'<<actualint;
cout<<'\n'<<otherint;
otherint++;
cout<<'\n'<<actualint;
cout<<'\n'<<otherint;
actualint++;
cout<<'\n'<<actualint;
cout<<'\n'<<otherint;
}
```

Otherint refers to actualint

output
:
123
123
124
124
125
125

Initializing a Reference

Reference should be initialized with explicitly giving it something to refer to.

Some Exceptions

You need not initialize a reference when

- it is declared with extern
- it is a member of a class
- it is declared as a parameter
- in a function declaration or definition
- it is declared as a return type of a function

Reference to an object of different type

```
double dval=3.14159;
const int &Ir=100;
const int &Ia=dval
const double &dr=dval+1.0
```

A const reference can be initialized to an object of a different type (provided there is a conversion from one type to the other) as well as to non-addressable values such as literal const

For non-const reference, the same initialization are not legal.

In the case of non-addressable values such as literal const and objects of a different type, to accomplish this the compilers must generate a temporary object that the reference actually addresses but that the user has no access to it.

References and Pointers

References can be viewed as pointers without usual dereferencing notation.

```
int actualint=123;
int *const intptr=&actualint;
```

intptr is a constant pointer, one cannot make it point to another integer once it has been initialized to actualint.

same is true for references.

References and Pointers

- References can't be manipulated like pointers
- They don't have pointer arithmetic
- They directly act on the object they refer to.
- With pointers, one can use the const keyword to declare constant pointers and pointers to constant

Reference to constant

One can declare a reference to a constant

```
int actualint=123;
const int& otherint=actualint;
```

This makes otherint a readonly alias for actualint. You can't make any modification to otherint, only to actualint

But one *cannot* declare a constant reference

```
int &const otherint=actualint;//error
```

meaningless all references are constant by definition

Parameter passing mechanisms

Parameter passing mechanisms

- Call by value
- Call by address
- Call by reference (in C++)

Parameter passing mechanisms

- Call by Value:

  Value of actual argument is passed to formal argument. Changes made in the formal arguments are local to the block of called function, it does not affect the actual arguments.

```
void swap (int, int);   //prototype
main( )
{    int x=4,y=5;
     cout<<"x="<<x<<" y="<<y;            output:  x=4 y=5
     swap (x, y);                  //x, y actual args
     cout<<"x="<<x<<" y="<<y;            output:  x=4 y=5
}
void swap (int a, int b)                  //a, b formal args
{
     int k;
     k=a;    a=b;    b=k;
}
```

Parameter passing mechanisms

- Call by Address:

  Instead of passing values, address is passed. Hence changes made in the formal arguments are reflected in the actual arguments.

```
void swap (int*, int*);       //prototype
main( )
{    int x=4,y=5;
     cout<<“x=“<<x<<“ y=“<<y;            output: x=4 y=5
     swap (&x, &y);
     cout<<“x=“<<x<<“ y=“<<y;            output: x=5 y=4
}

void swap (int* a, int* b)
{
     int k;
     k=*a;   *a=*b;   *b=k;
}
```

Parameter passing mechanisms

- Call by Reference:

  In C++ it is possible to pass arguments by reference also. When we pass arguments by reference, the ‘formal’ arguments in the function become aliases to the ‘actual’ arguments in the calling function.

```
void swap (int&, int&);          //prototype
main( )
{    int x=4,y=5;
     cout<<"x="<<x<<" y="<<y;          output:  x=4 y=5
     swap (x, y);
     cout<<"x="<<x<<" y="<<y;          output:  x=5 y=4
}
void swap (int &a, int &b)
{
     int k;
     k=a;     a=b;     b=k;
}
```

References as Function Parameter

```
#include<iostream.h>
struct bigone
{int serno;
char text[1000];
}
bo={123,”This is a big
structure”};
void valfunc(bigone v1);
void ptrfunc(const bigone *p1);
void reffunc(const bigone &r1);
void main()
{
valfunc(bo);
ptrfunc(&bo);
reffunc(bo);
}
```

← Pass by value

← Pass by address

← Pass by reference

```
void valfunc(bigone v1)
{
cout<<'\n'<<v1.serno;
cout<<'\n'<<v1.text;
}
void ptrfunc(const bigone *p1)
{
cout<<'\n'<<p1->serno;
cout<<'\n'<<p1->text;
}
void valfunc(const bigone &r1)
{
cout<<'\n'<<r1.serno;
cout<<'\n'<<r1.text;
}
```

The reference function cannot modify the b0 variable. reffunc's parameter is a reference to a constant (readonly)

References as Return values

Reference can also be used to return values from a function.

```
int mynum=0;
int &num();
int &num() {return mynum;}
void main()
{
int i;
i=num();
num()=5;
}
```

return value of the num() is a reference initialized with the global variable mynum. The expression num() acts as an alias for mynum. i.e. a function call appear on the receiving end of an assignment statement.

Reference as return values

```
int& rmax(int &m,int
&n)
{
if(m>=n)
	return m;
else
	return n;
}
```

This function returns a reference to m or n rather than the value of m or n
since rmax(i,j) yields a reference rmax(i,j)=0 is possible.

Initialization of a reference is trivial when the initializer is an lvalue

The initializer of a plain T must be a lvalue or even of type T

int num=10;

double &val=66.6; // not a lvalue

double &post=num; // diff data type

Compile time errors

Returning Reference from function

A function which return reference can also be used as lvalue.

```
#include<iostream.h>
int &f();
int x;
main()
{
      f()=100;
      cout<<x<<'\n';
return 0;
}
```

```
int &f()
{
return x; // not the value of the global variable x but the address
of x in reference form.
}
```

The statement f()=100;
puts the value 100 into x

```
int &f()
{
int x;
return x;
}
```

x is local the object one refers to does not go out of scope

```
int &f()
.
.
.
int *x;
x=f();
```

A reference returned by a function cannot be assigned to a pointer

Reference are similar to pointers, but they are not pointers.

An independent reference can refer to a constant.
const int &ref=100; // valid

Class in C++

Class declaration

Class declaration syntax :

```
class class-name
{
      data-member declarations;
      member-function declarations or definitions;
};
```

- Member function can be defined either inside or outside the class.

Syntax to define the member function outside the class:

```
return-type classname::function-name(argument list)
{  …… };
```

Class Example

```
class account
{
private:
	int acc_no;
	char name[25];
	float balance;
public:
	void read()
	{ cin>>acc_no>>name>>balance; }
	void withdraw(float amt)		// member function defined
				inside a class
	{	balance-=amt;	}
	void deposit();
};
void account::deposit()			// member function defined outside a class
{	float amt;			// using binary scope resolution operator
	cin>>amt;
	balance += amt;
}
```

```
void main()
{	account a1,a2;
	a1.read();
	a2.read();
	a1.withdraw(1000);
	a2.deposit();
}
```

Access modifiers

Members can be declared as

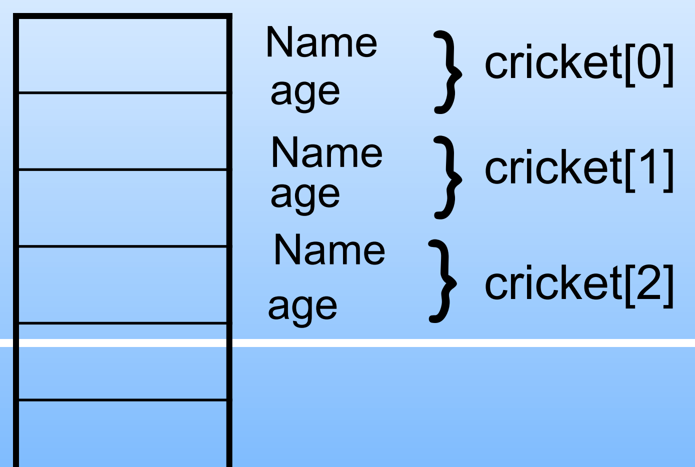
- Private
  visible only inside the class
- Protected
  private + visible to the immediately derived class
- Public
  globally visible

Note: The difference between the structure and class is the default access specifier. In struct it is public, whereas in class it is private.

Array of objects

```
class player
{  char name[20];
   int age;
public:
   void getdata(void);
   void putdata(void);
};
void player::getdata(0
{
  cout<<“\n Enter name: ”;
  cin>>name;
  cout<<“\n Enter age: ”;
  cin>>age;
}
void player::putdata(0
{
  cout<<“\n Player name: ”<<name;
  cout<<“\n Player age: ”;<<age;
}
```

```
main()
{
  player cricket[3];
  cout<<“Enter name and age of 3 players”;
  for (int i=0;i<3;i++)
        cricket[i].getdata();
  for (int i=0;i<3;i++)
        cricket[i].putdata();
}
```



'this' pointer

- 'this' is the keyword can be used inside the class, which represents the current object's address.
- example:

```
class account
{
        int acc_no;
        public:
        void assign(int acc)     // void assign(int acc_no)
        {                        //{
              acc_no=acc;        //    this->acc_no=acc_no;
        }                        //}
};
```

- When a member function is called, it is automatically passed as an implicit argument.

Introducing: `const`

```
void printSquare(const int& i)
{
   i = i*i;
   cout << i << endl;
}

int main()
{
   int i = 5;
   Math::printSquare(i);
}
```

Cannot modify the reference i since it is a constant

Won’t compile.

Can also pass pointers to `const`

```
void printSquare(const int* pi)
{
    *pi = (*pi) * (*pi);
    cout << i << endl;
}
int main()
{
    int i = 5;
    printSquare(&i);
}
```

pointer to a type constant int

Still won’t compile.

Declaring things `const`

```
const River nile;

const River* nilePc;

River* const nileCp;

const River* const nileCpc
```

Read pointer declarations right to left

```
// A const River
const River nile;

// A pointer to a const River
const River* nilePc;

// A const pointer to a River
River* const nileCp;

// A const pointer to a const River
const River* const nileCpc
```

Static Data Members

Static Data Members

A “Static Data Member” is a single shared object to all objects of its class.

```
class SavingsAccount
{
 public:
      SavingsAccount();
      void earnInterest();
      {
       total+=CurrentRate*total;
      }
 private:
      char name[30];
      float total;
      float CurrentRate;
};
```

Not Advisable

- Each object is having its own copy of CurrentRate
- Waste of space
- Has to update changes
- Inefficient
- Lead to Inconsistencies

Why not Global Variable?

?

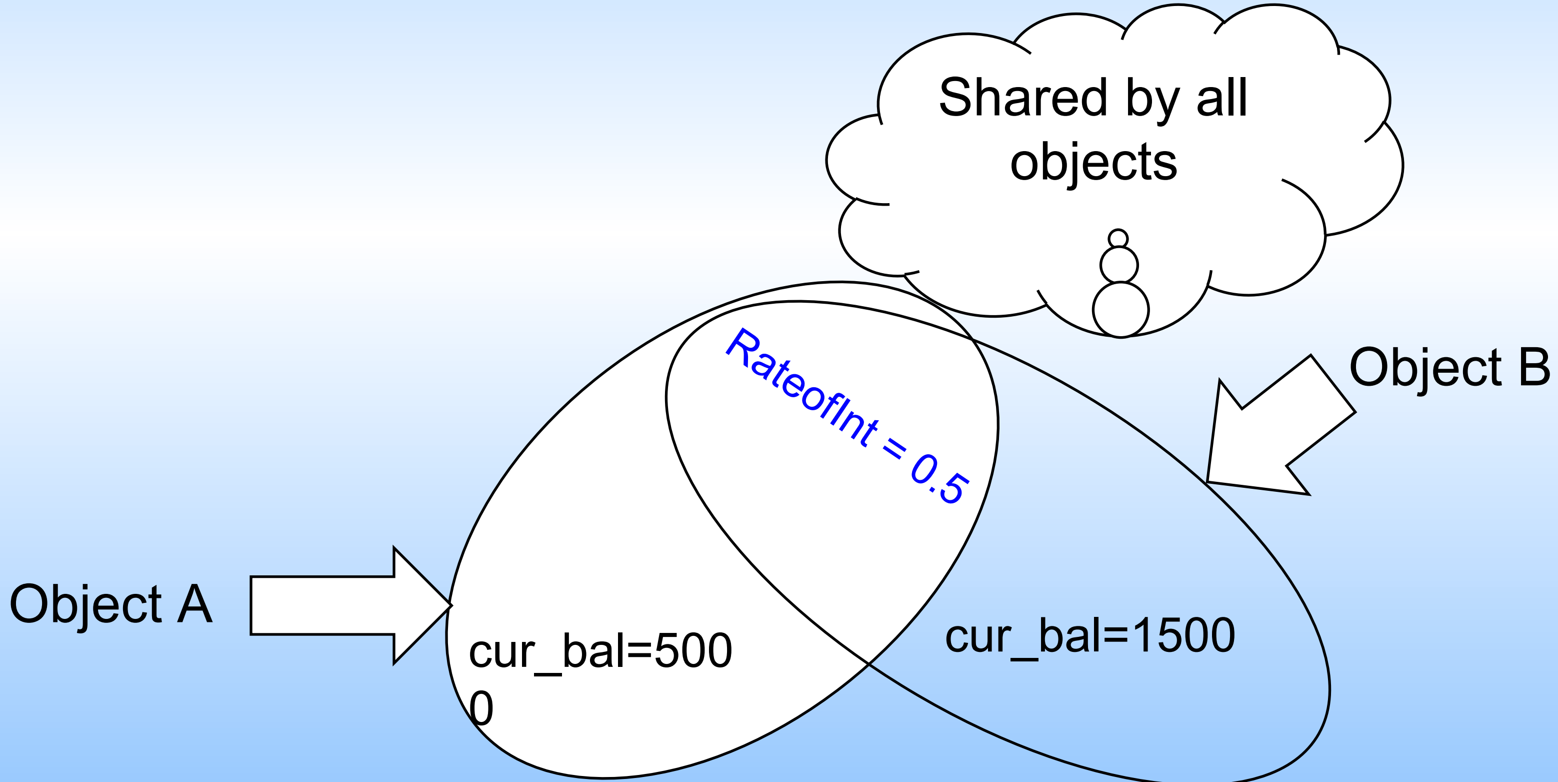
Scop
e

Every function will be able to modify its value

Requirement	Solution
A Kind of Global variable for an individual class All objects of a particular class access the same variable	Static Data Members have only one copy of it allocated No matter how many instances of the class Data Member made static by prefixing with “static” keyword

Static Data Members

- Static Data members may seem like global variables, but have class scope



Static Data Members

```
class SavingsAccount
{
 public:
      SavingsAccount();
      void earnInterest();
      {
       total+=CurrentRate*total;
      }
 private:
      char name[30];
      float total;
      static float CurrentRate;
};
```

Static Data Members

- A static member can be ***public***, making visible to the rest of the program
- If CurrentRate were a public member, one could access it as follows :

```
void main()
{
  SavingsAccount myaccount;
  myaccount.CurrentRate =
0.050;
}
```

- It implies that only CurrentRate of myaccount is being modified

Static Data Members

- A static data member is ***initialized outside the class*** definition in the same manner as a Non member variable.

- The only difference is that the ***class scope operator syntax must*** be used.

```
void main()
{
  float SavingsAccount::CurrentRate =
0.050;
}
```

Static Data Members

- Only one initialization of a static data member can occur within a program.

- Static member initializations should be placed in a file together with the definitions of the inline member functions and not in the class header file.

```
# include “savings.h”
float SavingsAccount::CurrentRate = 0.050;

SavingsAccount::SavingsAccount()
{
CurrentRate = 0.050;          ──→ X
// Cannot initialize a static member from within a constructor
// Because constructor may be called many times
}
```

- A static data member can appear as a default argument to a member function of the class.

- A non static member cannot.

```
extern int a;
class foo
{
 private:
      int a;
      static int b;
 public:
      int mem1(int = a); // ERROR.. There is no associated
                            class object
      int mem2(int = b);   // OK
      int mem3(int = ::a); // OK
};
```

Static Data Members

- A static data member can be an object of the class of which it is a member.

- A non static member is restricted to being declared as a pointer or reference to an object of its class.

```
class Bar
{
 public:
      //…
 private:
      static Bar a;
      Bar *b;
      Bar c;
};
```

Static Member Functions

A "Static Member Function" is a member function that accesses only the
Static data members of a class.

```
class SavingsAccount
{
 public:
      SavingsAccount();
      void earnInterest()
      {  total+=CurrentRate*total;  }
      static void setinterest(float new_value)
      { CurrentRate = new_value; }
 private:
      char name[30];
      float total;
      float CurrentRate;
};
```

Static Member Functions

- A static member function may be invoked through a class object or a pointer to a class object similar to a non static member function.

```
void main()
{
  SavingsAccount myaccount;
  myaccount.setinterest(0.50);
  SavingsAccount::
setinterest(0.50);
}
```

- A static member function does not contain a this pointer, since a static member function doesn't act on any particular instance of the class.
- A static member function
  - can't access any of the class's non static
  - members or call any non static member functions

Example Code

```
// Employ1.h
// Header File
# ifndef EMPLOY1_H
# define EMPLOY1_H
class Employee
{
 public:
      Employee(const char *,const char *) ;
      ~Employee();
      char *getFirstName() const;
      char *getLastName() const;
      static int getcount();
 private:
      char * firstName;
      char * lastName;
      static int count;
};
# endif
```

Contd…

```
# include<iostream.h>
# include<string.h>
# include<assert.h>
# include “Employ1.h”

int Employee :: count = 0;
int Employee :: getcount() { return count; }
Employee :: Employee(const char *first, const char *last)//constructor
{
 firstName = new char [strlen(first)+1];
 assert(firstName != 0);
 strcpy(firstname,first);
 lastName = new char[strlen(last)+1];
 assert(lastName != 0);
 strcpy(lastName,last);
 ++count;
 cout<< “Employee Constructor for”
       << firstName
       << “  ”
       << lastName
       << “called.\n”;
}
```

Contd…

```
Employee :: ~Employee()  //destructor
{
 cout << "~Employee() called for"
        << firstName << "  " << lastName << endl;
 delete firstName;
 delete lastName;
}
char *Employee :: getFirstName() const
{
 char *tempPtr = new char
[strlen(firstName)+1];
 assert(tempPtr != 0);
 strcpy(tempPtr,firstName);
 return tempPtr;
}
char *Employee :: getLastName() const
{
 char *tempPtr = new char [strlen(lastName)+1];
 assert(tempPtr != 0);
 strcpy(tempPtr,lastName);
 return tempPtr;
}
```

```
main()
{
 clrscr();
 cout << “Number of employees before instantiation is ”
       << Employee :: getcount() << endl;
 Employee * e1Ptr = new Employee(“Sujan”,”Baker”);
 Employee * e2Ptr = new Employee(“Roberts”,”Jones”);
 cout << “Number of employees after instantiation is ”
       << e1Ptr -> getcount() << endl;
 cout << “\nEmployee 1 : ”
       << e1Ptr -> getFirstName()
       << “  ” <<e1Ptr -> getLastName()
        << “\nEmployee 2 : ”
       << e2Ptr -> getFirstName()
       << “  ” <<e2Ptr -> getLastName()
       << “\n\n”;
delete e1ptr;
 delete e2Ptr;
 cout << “Number of employees after deletion is ”
      << Employee :: getcount() << endl;
 return 0;
}
```

Output:

```
Number of employees before instantiation is
0
Employee Constructor for Sujan Baker called
Employee Constructor for Roberts Jones called
Number of employees before instantiation is
2
Employee 1 : Sujan Baker

Employee 2 : Roberts Jones
~Employee() called for Sujan Baker
~Employee() called for Roberts Jones
Number of employees after deletion is 0
```

Function Overloading

Function Overloading

- When several different function declarations (signatures) are specified for a single name in the same scope, that name is said to be overloaded
- When that name is used, the correct function is selected by comparing the types of the actual arguments with types of the formal arguments

double abs(double);
int abs(double);
abs(1)//calls abs(int)
abs(1.0) // calls
abs(double)

Function different only in return types cannot be overloaded

Function Overloading

Any type T, a T and a T& accept the same set of initializer values, function with arguments types differing only in this respect may not have the same name.

```
int f(int i)
{
    //..
}
int f(int &r)   // error; function types
          //not sufficiently different
{
    //..
}
```

Function Overloading

The following is possible

```
void f1(int);
void f2(int&);
void f3(const int&);
void g()
{
f1(2.2); //ok
f2(2.2); // error temporary needed
f3(2.2); //ok temporary used
}
```

such is applicable only for const references

Function Overloading

Example:

```
#include <iostream.h>

int square(int x)           { return x * x; }
double square(double y)   { return y * y; }

void main()
{
   cout << "The square of integer 7 is " << square(7);
   cout << "\nThe square of double 7.5 is " << square(7.5)
          << '\n';
}
```

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Function Overloading

- Passing constant values directly can also lead to ambiguity as internal type conversions may take place.

Example: **sum(int,int) and sum(float,float)**

- The compiler will not be able to distinguish between the two calls made below

sum(2,3);
sum(1.1,
2.2);

Constructor

Constructors

- Constructors are *special member functions* (should be declared in public section) with the same name as the class. They *do not return values.*

- The constructor is invoked whenever an object of its associated class is created.

- A constructor is called whenever an object is defined or dynamically allocated using the “new” operator.

- They are normally used to allocate memory for the data members and initialize them.

- If no constructor is explicitly written then a default is created by the compiler (not in the case of const and reference data members).

Example

```
class Stack {
public:
Stack(int sz);
void Push(int value);
bool Full();
private:
int size;
int top;
int * stack;
};
Stack::Stack(int sz) {
size = sz;
top = 0;
stack = new int[size];
}
```

Constructor

Example

- To specify how an object is initialized we write a constructor for it as a member function.
- This member function has the same name as the class name i.e, Stack().
- Stack(int) is the constructor in this example

Types of Constructors

- Default Constructor
- Parameterized Constructor
- Overloaded Constructor
- Constructor with Default Arguments
- Dynamic Constructor
- Copy Constructor

Default Constructor

- The constructor without arguments.

```
class sum
{
  public:
    int x, y;
    sum();
};
sum::sum()          ← Default Constructor
{
    x=0;y=0;
}
```

Parameterized Constructors

- Parameters can be passed to the constructors

```
class sum
{
  public:
      int x, y;
      sum(int i,int j);
};
sum::sum(int i,int j)
{
      x=i;y=j;
}
void main( )
{
      sum s1(10,20);
      sum s2 = sum(30,40);
      cout<<"x= " <<
s1.x<<"y= "<<s1.y;
      cout<<"x= " <<
s2.x<<"y= "<<s2.y;
}
```

Parameterized Constructor

Constructor called implicitly

Explicit

Object s1: x= 10 y= 20

Object s2: x= 30 y= 40

Overloaded Constructors

- Overloaded Constructors - Multiple constructors declared in a class
- All constructors have different number of arguments
- Depending on the number of arguments, compiler executes corresponding constructor

```
sum( ) {x=10;y=20;};              No arguments

sum (int, int) {x=i; y=j;};      Two arguments
```

Constructors with default arguments

- Constructors can be defined with default arguments

```
class sum
{
        int x, y;
 public:
        void print()
        {       cout<<"x = "<<x<<"y ="<<y;  }
        sum(int i, int j=10);
};

sum::sum(int i,int j)
{       x=i;y=j;  }
void main()
{       sum s1(1),s2(8,9);
        s1.print();
        s2.print();
}
```

Default value for j=10 (Used by the object s1)

Object s1	x= 1 y= 10
Object s2	x= 8 y= 9

Dynamic Constructor

- Allocates the right amount of memory during execution for each object when the object's data member size is not the same using new operator in the constructor.
- The allocated memory should be released when the object is no longer needed by delete operator in the destructor.

```
#include<string.h>
class String
{   char* data;
    public:
    String(const char* s = "")
    {   data = new char[20];
        strcpy(data,s);     }
    ~String( )
    {       delete [ ] data;    }
    void display()
    { cout << data;         }
};
```

```
void main()
{
    String s = "hello";

    cout <<"s=";
    s.display( );
    String s1("hello world");
    cout<<"s1= ";
    s1.display();
}
```

```
s = hello
s1=hello world
```

Copy Constructors

- Constructor that takes a reference to an object of the same class as argument is Copy constructor. C++ calls a copy constructor (deep copy or member-wise copy) to make a copy of an object.
- Copy constructor is invoked in any of the following three ways.
  - ✔ When one object explicitly initializes another.
    - ❖ **sum s3=s1;**
  - ✔ When a copy of an object is made to be passed to a function.
    - ❖ **show(s2);**
  - ✔ When a temporary object is generated.
    - ❖ **s3=fun();**
- If there is no copy constructor defined for the class, C++ uses the default copy constructor which copies each data member, ie, makes a shallow copy or bit-wise copy.

Copy Constructors

Syntax:

class_name(class_name &object_name) {…}   //by reference is a must

```
class sum
{
  public:
     int x;
     sum( ){ }
     sum(int i) {x=i;}
     sum(sum &j) {x=j.x;}
};
```

```
void main( )
{      sum s1(10);
       sum s2(s1);
       sum s3=s1;
       cout<<"\nx in s1= " << s1.x;
       cout<<"\nx in s2= " << s2.x;
       cout<<"\nx in s3= " << s3.x;
}
```

Objects   s1   s2   s3

x=10   x=10   x=10

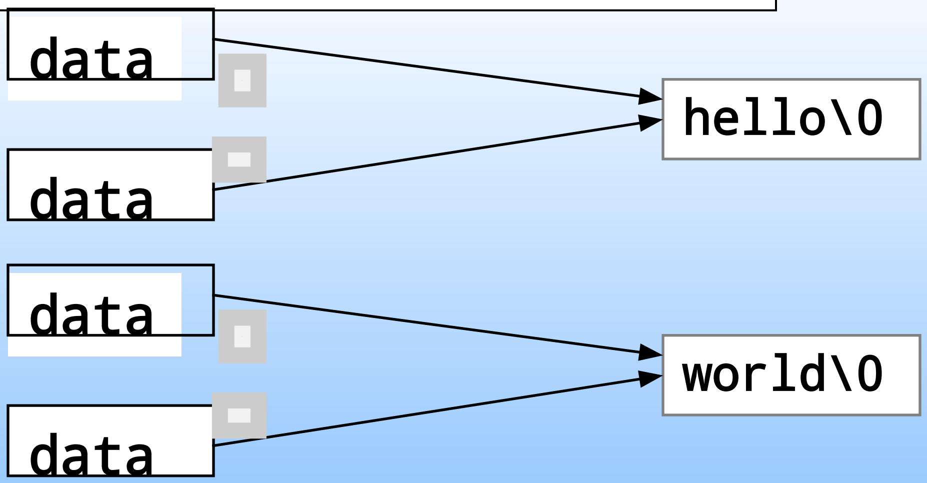
Copy Constructors

Shallow Copy copies the address of s.data to t .data

```
#include<string.h>
class String
{   char* data;
    public:
    String(const char* s = "")
    {   data = new char[20];
        strcpy(data,s);
    }
    ~String( )
    {     delete [ ] data;   }

   void assign(char *str)
    {     strcpy(data,str);        }

    void display()
    {    cout << data;             }
};
```

```
void main()
{  String s = "hello";
   String t = s; // same as String t(s);
   s.display( );
   t.display( );
   t.assign("world");
   s.display( );
   t.display( );
}
```

hello
hello
world
world



Copy Constructors

```
String(const String& s)
{
   data = new char[strlen(s.data)+1];
   strcpy(data, s.data);
}
```

<u>Deep</u> <u>Copy</u> copies the value of the object s to t

Constructor with initialization list

- Initialization of data members can also be done using initialization list.
- Syntax:

   class_name(arg_list) : InitializationSection

   { }

   Example:

```
class A
{
      int a;
      int b;
public:
A(int x):a(x),b(2*x){ }
void print( ){
cout<<a<<b; }
};
```

```
void main( )
{
      A object1(10);
      object1.print()
}
```

```
10 20
```

- For references and const data members constructor with initialization list is a must (no default constructor is provided).

Destructor

Destructors

- When an object goes out of scope then it is automatically destroyed.
- It performs clean up of the object (in the case of allocating memory inside the object using new)
- Destructors have the same name as class preceded with tilde (~)
- They have no arguments and thus cannot be overloaded

Syntax:

```
~ classname() { }
```

Example:

```
~sum( ) { }
```

Allocating memory using new

Point *p = new Point(5, 5);

- **new** can be thought of a function with slightly strange

  syntax
- **new** allocates space to hold the object.
- **new** calls the object's constructor.
- **new** returns a pointer to that object.

Deallocating memory using delete

```
// allocate memory
Point *p = new Point(5, 5);

...
// free the memory
delete p;
```

For every call to **new**, there must be exactly one call to **delete**.

Using new with arrays

```
int x = 10;
int* nums1 = new int[10];  // ok
int* nums2 = new int[x];   // ok
```

- Initializes an array of 10 integers on the heap.
- C++ equivalent of

```
int* nums = (int*)malloc(x * sizeof(int));
```

Using new with multidimensional arrays

```
int x = 3, y = 4;
int* nums3 = new int[x][4][5];// ok
int* nums4 = new int[x][y][5];// BAD!
```

- Initializes a multidimensional array
- Only the first dimension can be a variable. The rest must be constants.
- Use single dimension arrays to fake multidimensional ones

Using delete on arrays

```
// allocate memory
int* nums1 = new int[10];
int* nums3 = new int[x][4][5];

. . .
// free the memory
delete[ ] nums1;
delete[ ] nums3;
```

- Have to use **delete[].**

Friends

Friends

- A friend of a class is a function that is not a member of the class but is permitted to use the private and protected member names from the class.
- The name of the friend is not in the scope of the class
- It is not called with the member access operator
- It uses the friend keyword

Difference between friends and member function

```
class X {
int a;
friend void friend_set(X*,int);
public:
void member_set(int);
};
void friend_set(X* p,int i) {p->a=i;}
void X::member_set(int i) {a=i;}
```

```
void f()
{
X obj;
friend_set(&obj,10);
obj.member_set(10);
}
```

Friend Class

- Entire class can be declared as friend for another class.

- When a class is declared as friend, it means the members of the friend class have access to all the public / private / protected members of the class in which the declaration was made.

Friend class example

```
#include <iostream.h>
class two;
class one
{       int a1,b1;
        public:
        assign(void)
        {   a1=5,   b1=10;     }
        friend class two;
//friend void two::assign(one);
};
class two
{
        int a,b;
        public:
        void let(int x, int y)
        {     a=x; b=y;  }
        void print(void)
        { cout<<"a="<<a<<"b="<<b;  }
        void assign(one x)
        {       a=x.a1;
                b=x.b1;
        }
};

void main(void)
{       one o1;
        two t1;
        o1.assign();
        t1.let(4,2);
        t1.print();
        t1.assign(o1);
        t1.print();
}
```

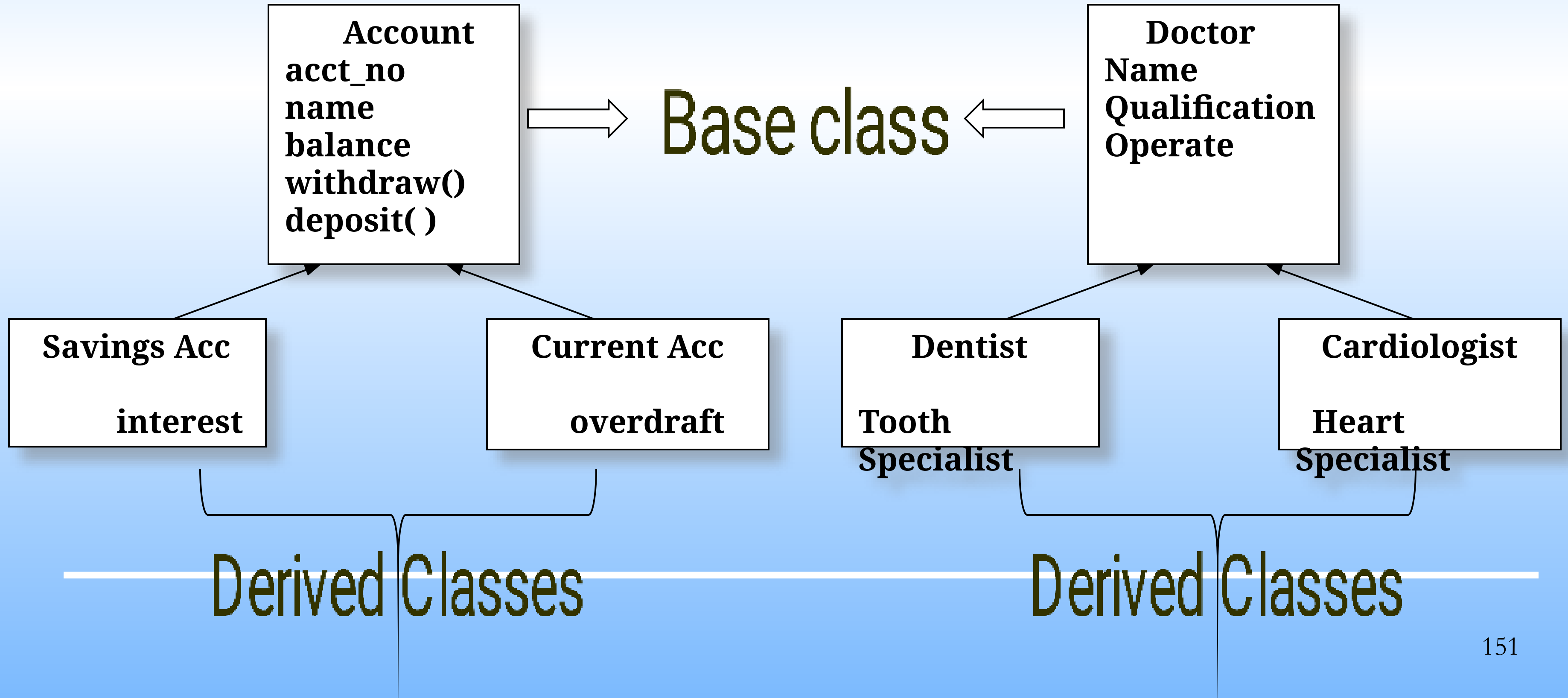
a = 4 b= 2
a= 5   b=10

Inheritance

Inheritance – is a relationship

- Inheritance is a process of one class acquiring the features of another class.
- It is a way of creating new class(derived class) from the existing class(base class) providing the concept of reusability.

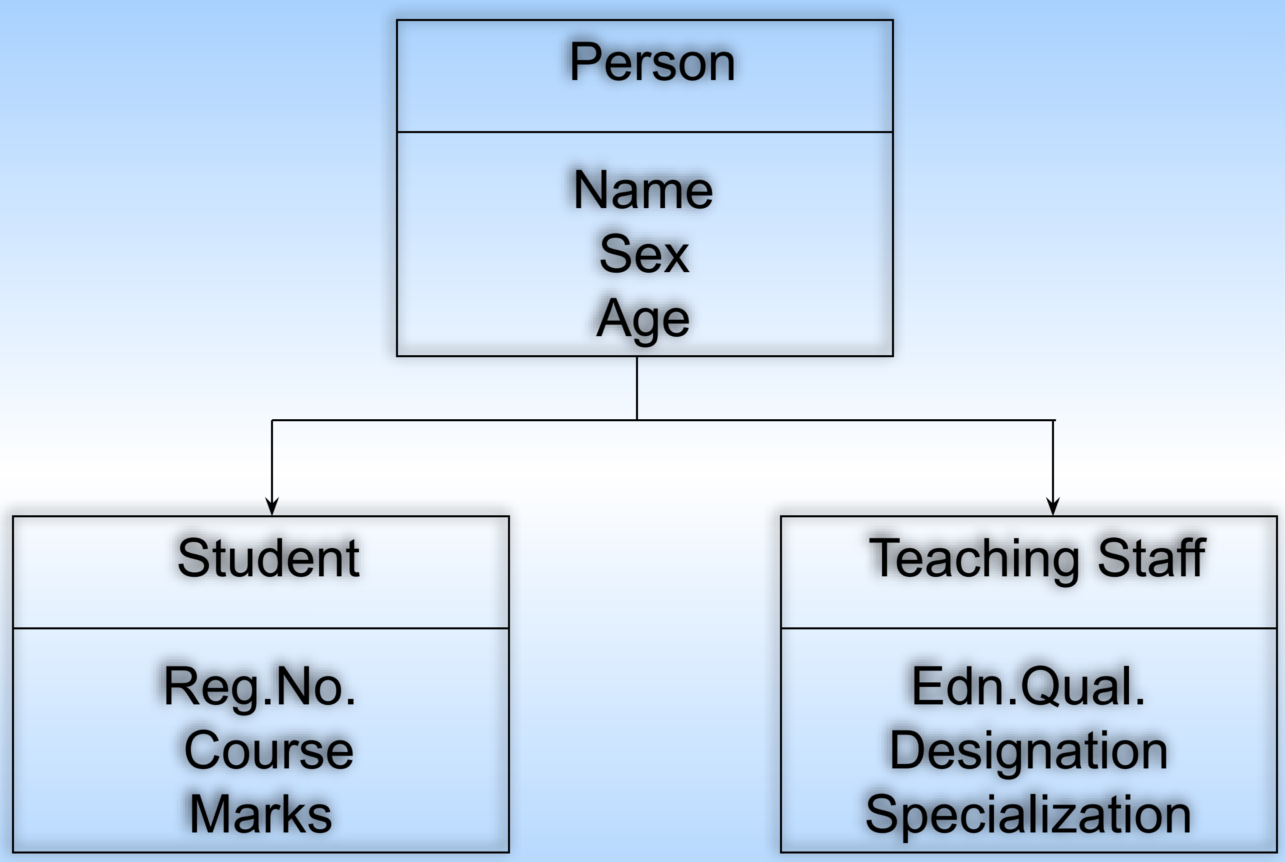
A Surgeon "is a" Doctor. A Doctor need not be a surgeon



Inheritance

- By using the concepts of inheritance, it is possible to create a new class from an existing one and add new features to it.
- The class being refined is called the superclass or base class and each refined version is called a subclass or derived class.
- Semantically, inheritance denotes an "*is-a*" relationship between a class and one or more refined version of it.
- Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass providing the mechanism for class level Reusability .

Inheritance



"Person" is a generalization of "Student".
"Student" is a specialization of "Person".

Defining Derived Class

- The general form of deriving a subclass from a base class is as follows

```
Class derived-class-name : visibility-mode base-class-name
{
            ………………….  //
        ………………….// members of the derived class
};
```

- The visibility-mode is optional.
- It may be either private or public or protected, by default it is private.
- This visibility mode specifies how the features of base class are visible to the derived class.

Access control

Access Specifier and their scope

Base Class Access Mode	Derived Class Access Modes		
	Private derivation	Public derivation	Protected derivation
Public	Private	Public	Protected
Private	Not inherited	Not inherited	Not inherited
Protected	private	Protected	Protected

Access control to class members

Function Type	Access Directly to		
	Private	Public	Protected
Class Member	Yes	Yes	Yes
Derived Class Member	No	Yes	Yes
Friend	Yes	Yes	Yes
Friend Class Member	Yes	Yes	Yes

Access Specifiers with Inheritance

```
class X
{     int priv;
      protected:
       int prot;
      public:
       int publ;
       void m( );
};
void  X::m( )
{     priv =1;    //Ok
      prot =1;    //Ok
      publ =1;    //Ok
}
```

```
class Y : public X
{      void  mderived( );
};
Y::mderived( )
{     priv =1;    //Error priv is private and
                  //cannot be inherited
      prot =2;    // Ok
      publ=3;     // Ok
}
void global_fun(Y  *p)
{
p->priv = 1;  //Error : priv is private of X
p->prot = 2;  //Error : prot is protected and
              //the function global_fun( )
              //is not a friend or a member of X or Y
p->publ =3;      // Ok
}
```

Public, Protected and Private derivation

Class A	Class B	Class B: Public A
private : int a1;	private : int b1;	private : int b1;
protected : int a2;	protected : int b2;	protected: int a2; int b2
public : int a3;	public : int b3;	public: int b3;int a3;

Public Derivation

Public derivation - example

```
class A
{
        private    :  int a;
      protected: int b;
     public     : void get_a( )  {  cin>>a;}
     void get_b( )  { cin>>b;}
             void print_a( ) { cout<<a;}
      void print_b( ) {cout<<b;}
};
class B : public A
{
      private    :  int c;
   protected:  int d;
   public     :  void get_c( ) { cin>>c;}
   void get_d( ) {cin >>d;}
   void get_all( ) { get_a( ); cin>>b>>c>>d;}
   void print_cd( ){ cout<<c<<d;}
   void print_all( ) { print_a( ); cout<<b<<c<<d; }
};
```

```
void  main( )
 {
        B  b1;
        b1.get_a( );
        b1.get_b( );
        b1.get_c( );
        b1.get_d( );
        b1.print_all( );
 }
```

Protected derivation - example

- The inherited public and protected members of a base class become protected members of the derived class

Class A	Class B	Class B : Protected A
private : int a1;	private : int b1;	private : int b1;
protected : int a2;	protected : int b2;	protected: int a2; int b2,a3;
public : int a3;	public : int b3;	public: int b3;

Protected Derivation

Protected derivation - example

```
class A
{
        private:  int a;
        protected: int b;
        public    : void get_a( )  {  cin>>a;}
        void get_b( )  { cin>>b;}
        void print_a( ) { cout<<a;}
        void print_b( ) {cout<<b;}
};


class B : protected A
{       private    :  int c;
        protected:  int d;
        public     :  void get_c( ) { cin>>c;}
        void get_d( ) {cin >>d;}
        void get_ab( ) { get_a( ); get_b( );}
        void print_cd( ){ cout<<c<<d;}
        void print_all( ) { print_a( ); cout<<b<<c<<d;};
 }
```

```
void  main( )
{
          B  b1;
          b1.get_a( );  //ERROR
          b1.get_b( ); //ERROR
          b1.get_ab( );
         b1.get_c( );
         b1.get_d( );
         b1.print_all( );
}
```

Private derivation - example

The inherited public and protected members of a private derivation become private members of the derived class.

Class A	Class B	Class B : private A
private : int a1;	private : int b1;	private : int b1; int a2,a3;
protected : int a2;	protected : int b2;	protected: int b2;
public : int a3;	public : int b3;	public: int b3;

Private Derivation

Private derivation - example

```
class A
{
        private:  int a;
        protected: int b;
        public    : void get_a( )  {  cin>>a;}
        void get_b( )  { cin>>b;}
        void print_a( ) { cout<<a;}
        void print_b( ) {cout<<b;}
};


class B : private A
{
        private    :  int c;
        protected:  int d;
        public     :  void get_c( ) { cin>>c;}
        void get_d( ) {cin >>d;}
        void get_ab( ) { get_a( ); get_b( );}
        void print_cd( ){ cout<<c<<d;}
        void print_abcd( ) { print_a( ); cout<<b<<c<<d; }
};
```

```
Class C : public B
{       public :
        void get_all( )
        {       get_a( );   //ERROR
                get_b( );  //ERROR

                get_ab( );  //Ok
                get_c( );   //Ok
                get_d( );   //Ok        }
   void print_all( )
   {            print_a( );   //ERROR
                print_b( );  //ERROR
                print_cd( );     //Ok
                 print_abcd( ); //Ok  } };
void  main( )
{         C  c1;
          c1.get_a( );  //ERROR
          c1.get_b( );  //ERROR
          c1.get_c( );  // Ok
          c1.get_d( );  //Ok
          c1.getall( );  //Ok
          c1.print_all( ); //Ok
}
```

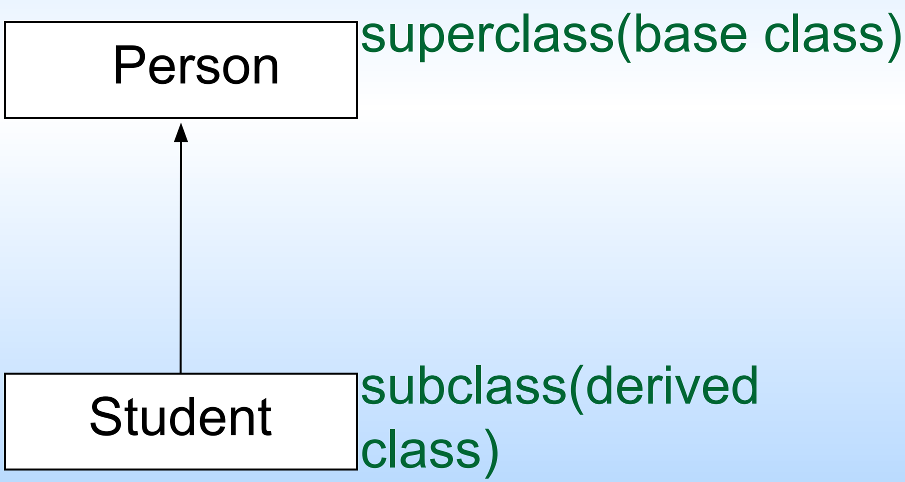
Types of Inheritance

Inheritance are of the following types

- Simple or Single Inheritance
- Multi level or Varied Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance
- Virtual Inheritance

Simple or Single Inheritance

- This is a process in which a sub class is derived from only one superclass.
- a Class Student is derived from a Class Person

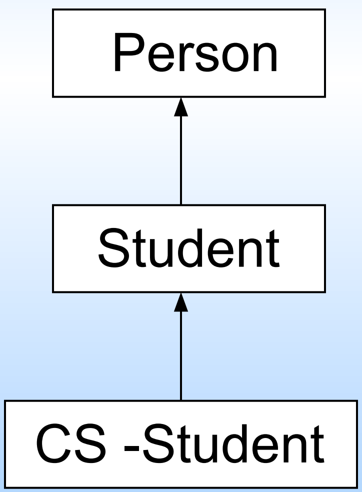



```
class  Person

      {  .....     };

      class Student :  public  Person

      {
                  ...........
      };
```

visibility mode

Multilevel or Varied Inheritance

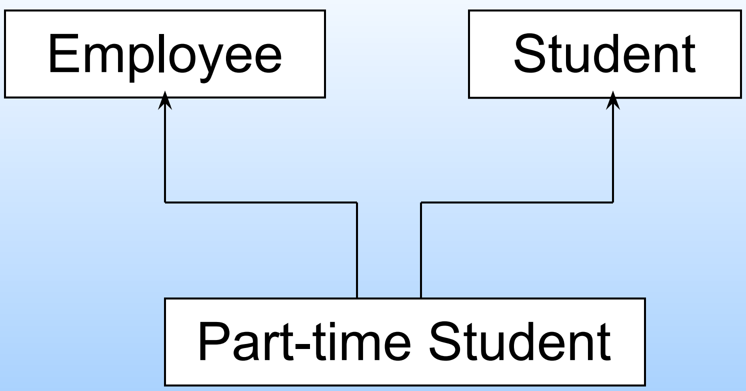
- The method of deriving a class from another derived class is known as Multiple or Varied Inheritance.
- A derived class CS-Student is derived from another derived class Student.




```
Class Person
      { ……};
Class Student : public Person
      { ……};
Class CS -Student : public Student
      {  …….};
```

Multiple Inheritance

- A class is inheriting features from more than one super class.
- Class Part-time Student is derived from two base classes, Employee and Student .

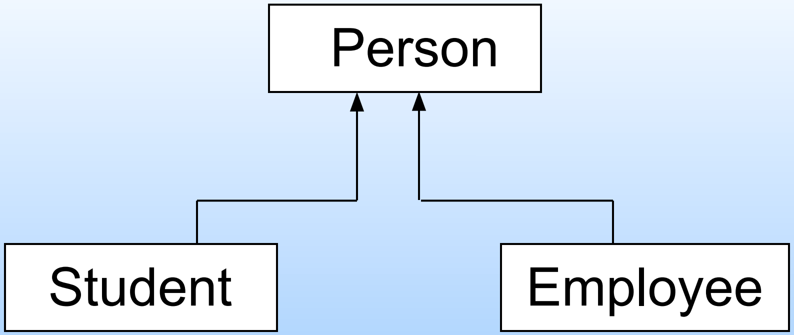



```
Class Employee
        {……..};
Class Student
        {……..};
Class Part-time Student : public Employee,
                           public Student
        {…….};
```

Hierarchical Inheritance

- Many sub classes are derived from a single base class

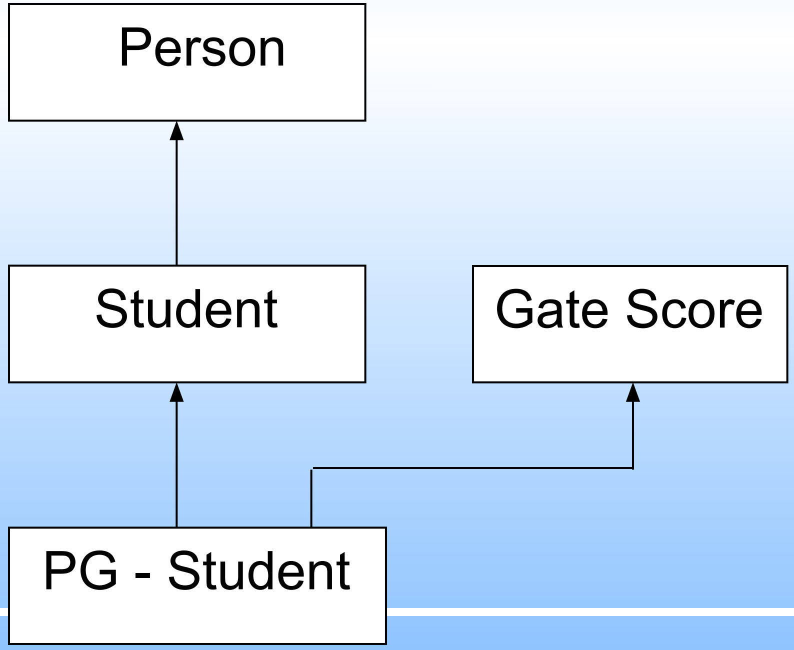
- The two derived classes namely Student and Employee are derived from a base class Person.




```
Class Person
      {……};
Class Student : public  Person
      {……};
Class Employee : public  Person
      {……};
```

Hybrid Inheritance

- In this type, more than one type of inheritance are used to derive a new sub class.
- Multiple and multilevel type of inheritances are used to derive a class PG-Student.

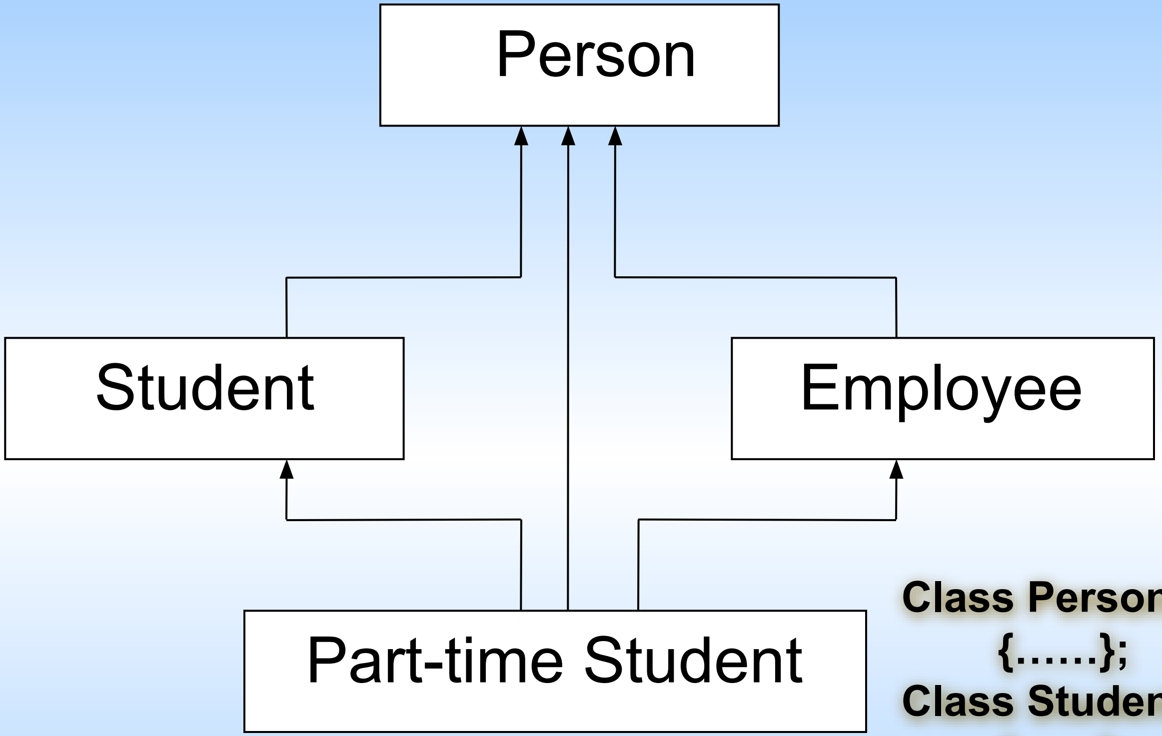



```
class Person
        { ……};
class Student : public Person
        { ……};
class Gate Score
        {…….};
class PG - Student : public Student,
                     public Gate Score
        {………};
```

Virtual Inheritance

- A sub class is derived from two super classes which in-turn have been derived from another class.
- The class Part-Time Student is derived from two super classes namely, Student and Employee.
- These classes in-turn derived from a common super class Person.
- The class Part-time Student inherits, the features of Person Class via two separate paths.

Virtual Inheritance



Class Person
{……};
Class Student : public Person
{……};
Class Employee : public Person
{……};
Class Part-time Student : public Student,
public Employee
{…….};

Derived Class Constructors

- A base class constructor is invoked(if any) , when a derived class object is created.
- If base class constructor contains default constructor, then the derived class constructor need not send arguments to base class constructors explicitly.
- If a derived class has constructor, but base class has no constructor, then the appropriate derived class constructor executed automatically whenever a derived class object is created.

Derived Class Constructors

```
class B
    {
        int x;
        public :
        B( ) { cout<<"B::B( )   Ctor…"<<endl;}
    };
class D : public B
    {
        int y;
        public :
        D( ) { cout<<"D::D()    Ctor …"<<endl;}
    };
    void main( )
    {
        D d;
    }
```

```
B::B( ) Ctor …
D::D( ) Ctor …
```

Derived Class Constructors

```
class B
{
		int a;
		public :
		B( ) { a = 0; cout<<"B::B( ) Ctor…"<<endl;}
		B(int x) { a =x; cout<<"B::B(int) Ctor…"<<endl;}
};
class D : public B
{		int  b;
		public :
		D( ) { b =0; cout<<"D::D( ) Ctor…"<<endl;}
		D(int x) { b =x; cout<<"D::D(int) Ctor…"<<endl;}
		D(int x, int y) : B(y)
		{ b =x; cout<<"D::D(int, int) Ctor…"<<endl;}
};
void main( )
{
		D d;
		D d(10);
		D d(10, 20);
}
```

```
B::B( ) Ctor…
D::D( ) Ctor…
B::B( ) Ctor…
D::D(int) Ctor…
B::B(int) Ctor…
D::D(int, int) Ctor…
```

Derived Class Destructors

Derived class destructors are called before base class destructors.

```
class B
{
        int x;
        public :
          B( ) { cout<<"B Constructor Invoked…"<<endl;
}
        ~B( ) { cout<<"B Destructor Invoked …"<<endl;}
};
class D : public B
{
        int y;
        public :
        D( ) { cout<<"D Constructor Invoked …"<<endl;
}
                ~ D( )  { cout<<"D Destructor
Invoked…"<<endl;}
};
void main( )
{       D d; }
```

```
B Constructor Invoked…
D Constructor Invoked…
D Destructor Invoked…
B Destructor Invoked …
```

Overriding Member Functions

- When the same function exists in both the base class and the derived class, the function in the derived class is executed

```
class A
 {
protected    :  int a;
public       :
void getdata( )  {  cin>>a;}
void  putdata( ) { cout << a;}
};
class B : public A
{
protected:  int b;
public     :  void getdata( )
{ cin>>a>>b;}
void putdata( ) { cout<<a<<b;}
};
```

```
void  main( )
{

B  b1;
b1.getdata( );     // B::getdata( )
                //is invoked
b1.putdata( );     // B::putdata( )
                //is invoked
 b1.A::getdata( );  // A::getdata( )
                // is invoked
 b1.A::putdata( );  // A::putdata( )
                //is invoked             }
```

Composition

- Classes having objects of other classes as their data members - composite classes

```
class x
{       int i;
  public:
  x()
        {i=0;}
  void set(int j)
        { i=j;}
  int read() const
        { return i;}
};
```

```
class  y
{
        int i;
  public:
        X x;
        Y()
        { i=0;}
  void f(intj)
        { i=j;}
  int g() const
        { return i;}
 };
int main()
{       Y y;
        y.f(5);
        y.x.set(10);
}
```

Inheritance : Summary

- A subclass may be derived from a class and inherit its methods and members.
- Semantically, inheritance denotes an "*is-a*" relationship between a class and one or more refined version of it.

- Different types are
  - Single inheritance
  - Multiple inheritance
  - Multilevel inheritance

- Base class constructors are also executed whenever derived class objects created.

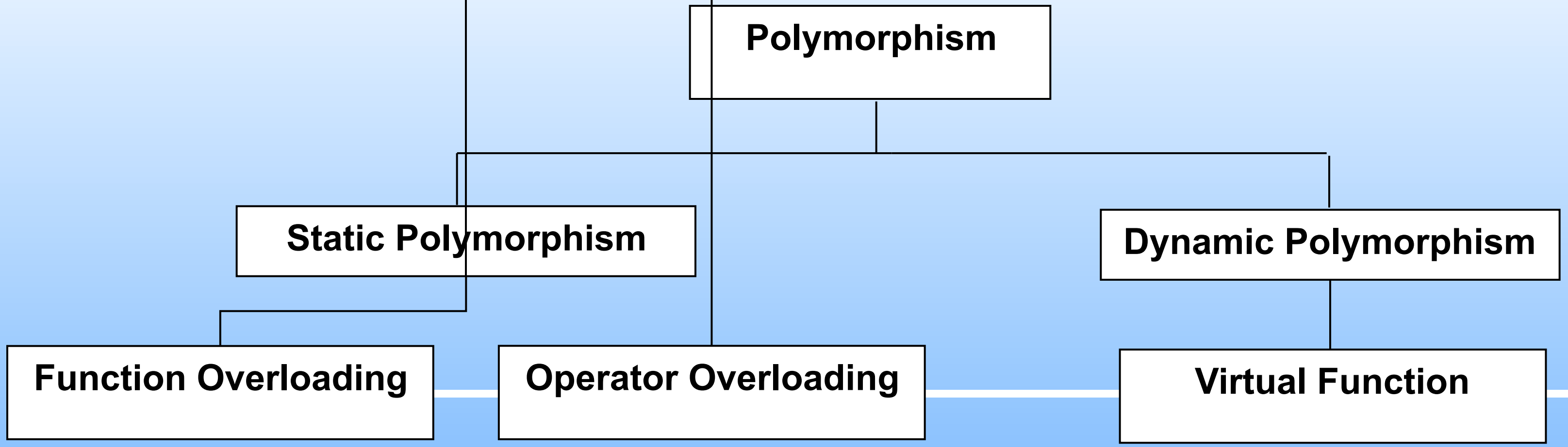
- Derived class can override a member function of a base class.

Virtual Functions and Polymorphism

Polymorphism

- Greek word meaning - many or multiple forms.
- In programming languages, *polymorphism* means that some code or operations or objects behave differently in different contexts
- It provides a single interface to entities of different types.

Types of Polymorphism



Polymorphism

Binding

- Determining a location in memory by the compiler
- Connecting a function call to a function body
- The location may represent the following
  - Variable names bound to their storage memory address (offset)
  - Function names bound to their starting memory address (offset)

Two kinds of binding

- Compile-Time Binding
- Run-Time Binding

Static Polymorphism

- Compile-time binding or early binding is called as static polymorphism.

- Compile-Time Binding means

  - Compiler has enough information to determine an address (offset)

  - Named variables / function calls have their addresses fixed during compilation

Virtual Functions

- Virtual function is a member function that is declared in the base class (using keyword virtual) and is redefined in the derived class.
- Dynamic binding means that the actual function invoked at run time depends on the address stored in address.
- Virtual functions are overriding functions to achieve dynamic binding.

```
class One
{       public:
        virtual void whoami()
                {cout<<"One"<<endl;  }
};
class Two : public One
{       public:
        void whoami()
                {cout<<"Two"<<endl;  }
};
```

```
void main()
{
        One one, *ptr ;
        Two two;
        ptr=&one;
        ptr->whoami( );
        ptr=&two;
        ptr->whoami( );
}
```

One
Two

Virtual Function Implementation

VTABLE (Virtual Table):

- The compiler creates a VTABLE for every class and its derived class having virtual functions, which contains addresses of virtual functions.

- If no function is redefined in the derived class that is defined as virtual in the base class, the compiler takes the address of base class function.

VPTR (Virtual Pointer):

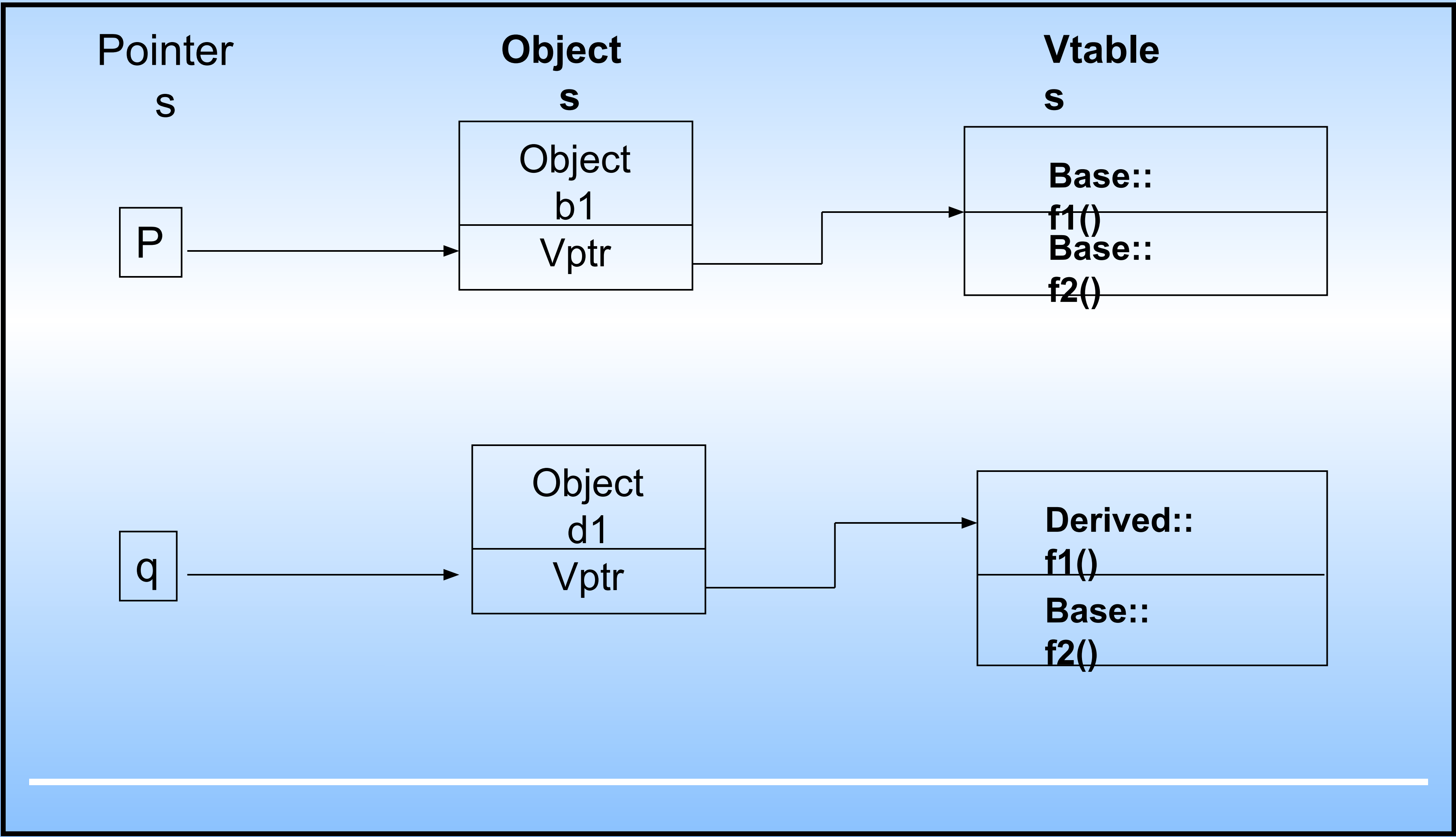
- When objects of base or derived classes are created, a void pointer is inserted in the VTable called VPTR.

Virtual Function Implementation

```
class Base
{
    public :
    Base() { }
    virtual void f1() { cout<<"base::f1( )" << endl; }
    virtual void f2( ) { cout << Base:: f2()"<<endl; }
    void f3() { cout<<"Base :: f3()"<<endl; }
};
class Derived :public Base
{
    public:
    Derived() { }
    void f1() {  cout<<"Derived ::f1()"<<endl;  }
    };
```

```
void main()
{
    Base b1;
    Derived d1;
    Base *p=&b1;
    p->f1(); // base::f1
    p->f2(); //base::f2
    p->f3(); //base::f3
    Base *q =&d1;
    q->f1(); //Derived::f1
    q->f2(); //Base::f2
    q->f3(); //Base::f3
}
```

Virtual Function Implementation contd…



Abstract Class & Pure Virtual Functions

Abstract Class

- A class that serves only as a base class from which other classes can be derived
- If a base class declares contains pure virtual functions , no instance of the class can be created.

Pure Virtual Functions

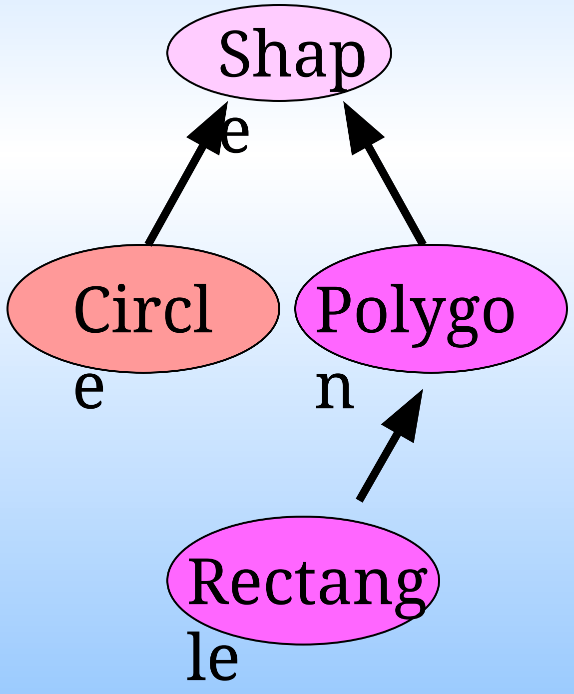
- In practical applications, the member functions of base classes is rarely used for doing any operation (null body).

Syntax:

```
Virtual void display( )=0;
```

Abstract Classes

- An abstract class represents an abstract concept in C++ (such as Shape class)



1. Defines the interfaces that all of the concrete classes (subclasses) share
2. Does not define state and implementation unless it is common to all concrete classes
3. Cannot be instantiated

Pure Virtual Function

```
class Base
{    public:
          virtual  void
    show()=0;
          // Pure Virtual function
};
class Derv1 : public Base
{
     public:
    void show()
    {  cout<<" In Derv 1"; }
};
```

```
class Derv2 : public Base
{  public:
   void show()
   {  cout<<"In Derv2";  }
};
void main()
{
       Base *List[2];
       Derv1 dv1;
       Derv2 dv2;
       List[0]=&dv1;
       List[1]=&dv2;
       List[0]->show();
       List[1]->show();
}
```

Virtual Destructor

- When a derived object pointed to by the base class pointer is deleted, dtor of the derived class as well as the dtors of all its base classes are invoked.

```
#include<string.h>
class Father
{  char* name;
    public:
    Father(const char* s = "")
    {   name = new char[20];
        strcpy(name,s);    }
   virtual  ~Father( )
    {     delete [ ] name;
    }
   virtual void display()
    { cout << name;
    }
};
```

```
class son: public Father
{
    char* name;
    public:
    son(const char* s = "")
    {   name = new char[20];
        strcpy(name,s);
    }
    ~son( )
    {     delete [ ] name;
    }
    void display()
    { cout << name;
    }
};
```

Contd..

Virtual Destructor

```
void  main()
{
   Father *bp,s = “ABC";         // same as Father s(“ABC");
   bp=&s;
   cout <<”s=”<<bp->display( );
   delete bp;
   son s1=“XYZ”;
   bp=&s1;
   cout<<”s1=”<<bp->display();
   delete bp;
}
```

s = ABC
S1=XYZ

Why not virtual constructor??????

Rules for Virtual Functions

- They should not be static.
- They can be friend function of another class.
- Constructors cannot be declared as virtual, but destructors can be declared as virtual.
- The virtual function must be defined in public section of the class.

Polymorphism - Summary

- Function overloading and operator overloading are used to achieve static polymorphism.
- Virtual functions are used to achieve dynamic polymorphism.
- Pointers to objects of base classes are type compatible with pointers to objects of derived classes. Reverse is not possible.
- Virtual functions can be invoked using pointer or reference.
- Abstract base class is the one having pure virtual function.

Overloading Vs Overriding

- Overloading
  - Same name and scope of the class
  - Different signature
  - Doesn't require virtual keyword
- Overriding
  - Same name and Signature
  - Different class scope
  - Require virtual keyword

Operator Overloading

Operator overloading

- Enabling C++'s operators to work with class objects
- Using traditional operators with user-defined objects
- A way of achieving static polymorphism is Operator overloading

Example:

  - Operator << is both the stream-insertion operator and the bitwise left-shift operator
  - + and -, perform arithmetic on multiple types

4 + 5 - integer addition
3.14 + 2.0 - floating point addition
"sita" + "ram" - string concatenation

- Compiler generates the appropriate code based on the manner in which the operator is used

Operator overloading

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded
  - **`operator+`** used to overload the addition operator (**`+`**)
- Two ways of overloading the operators using
  - Member function
  - Friend function

Operator Overloading

Syntax (using member function):

Keyword Operator to be overloaded

```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
    \\ Function body
}
```

- Number of arguments in a member function for
  - Unary operator – 0
  - Binary operator – 1

Operator Overloading

Syntax (using friend function):

Keyword

Operator to be overloaded

```
ReturnType operator OperatorSymbol (argument list)
{
      \\ Function body
}
```

Number of arguments in a friend function for

- Unary operator – 1
- Binary operator – 2

Binary Operator Overloading

- To declare a binary operator function as a member function
  - ret-type operatorop(arg)

```
class time
{
	int hrs, mins;
public:
	void set(int h,int m)
	{	hrs=h;mins=m;}
	void show()
	{	cout<<hrs<<":"<<mins;  }
	void sum(time t)
	{	hrs+=t.hrs;  mins+=t.mins; }
	time operator + (time t)
	{	time temp;
		temp.hrs=hrs+t.hrs;
		temp.mins=mins+t.mins;
		return temp;}
};
```

```
void main(void)
{
	time t1(10,20), t2(11,30);
	t1.show();
	t2.show();
	// t1.sum(t2);
	t1=t1+ t2;
      t1.show();
	t2.show();
}
```

Hrs 10: 20 mins
Hrs 11: 30 mins
Hrs 21: 50 mins
Hrs 11: 30 mins

Binary Operator Overloading

- To declare a binary operator function as a friend function
  - *ret-type* operator*op*(*arg1*, *arg2*)

```
class time
{
      int hrs, mins;
public:
      void set(int h,int m)
      {     hrs=h;mins=m;}
      void show()
      {     cout<<hrs<<":"<<mins;  }
      friend time operator + (time,time);
};
time operator + (time t1,time t2)
{     time temp;
            temp.hrs=t1.hrs+t2.hrs;
            temp.mins=t1mins+t2.mins;
            return temp;   }
```

```
void main(void)
{
      time t1(10,20), t2(11,30);
      t1.show();
      t2.show();
      t1=t1+ t2;
    t1.show();
      t2.show();
}
```

Hrs 10: 20 mins
Hrs 11: 30 mins
Hrs 21: 50 mins
Hrs 11: 30 mins

Binary Operator Overloading

```
class Point {
  public:
    int x,y;
    Point () { };
    Point (int,int);
    Point operator + (Point);
};
Point::Point (int a, int b) {
  x = a;
  y = b;
}
```

```
Point Point::operator+ (Point P)
{
        Point temp;
        temp.x = x + P.x;
        temp.y = y + P.y;
        return (temp);
}
int main ()
{
        Point a (3,1);
        Point b (1,2);
        Point c;
        c = a + b;    // c=a.operator+(b);
        cout << c.x << "," << c.y;
        return 0;
}
```

2,3

Operator Overloading

<table>
<tr><th colspan="8">Operators that can be overloaded</th></tr>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td></tr>
<tr><td>~</td><td>!</td><td>=</td><td><</td><td>></td><td>+=</td><td>-=</td><td>*=</td></tr>
<tr><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td><td><<</td><td>>></td><td>>>=</td></tr>
<tr><td><<=</td><td>==</td><td>!=</td><td><=</td><td>>=</td><td>&&</td><td>||</td><td>++</td></tr>
<tr><td>--</td><td>->*</td><td>,</td><td>-></td><td>[]</td><td>()</td><td>new</td><td>delete</td></tr>
<tr><td>new[]</td><td>delete[]</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><th colspan="5">Operators that cannot be overloaded</th></tr>
<tr><td>.</td><td>.*</td><td>::</td><td>?:</td><td>sizeof</td></tr>
</table>

Operator Overloading

Operators that can be overloaded using member and friend functions:

	Using member function	Using friend function
Operators	= ( ) [ ] ->	<< >>

Unary Operators Overloading

- To declare a unary operator function as a member function
  - return-type operatorop()

```
class Point
{       int x, y;
  public:
        Point() {x = y = 0; }
        Point& operator++()
        {
                x++;
                y++;
                return *this;
        }
        Point& operator--()
        {       x--;   y--;   return *this;}
};
```

```
void main()
{
Point p, p1,p2;
p1= p++;
p2= --p;
}
```

Unary Operators Overloading

- To declare a unary operator function as a friend function
  - ret-type operatorop(arg)

```
class Point
{       int x, y;
  public:
        Point() {x = y = 0; }
        Point& operator++()
        {       x++;
                y++;
                return *this;    }
        friend point operator –(point);
};
Point operator--(point &p)
{       p.x--;
        p.y--;
        return p;
}
```

```
void main()
{
        Point p, p1,p2;
        p1= p++;
        p2= --p;
}
```

Unary Operators Overloading

- There is no distinction between the prefix and postfix overloaded operator functions.
- The new syntax for postfix operator overloaded function is

  ret-type operatorop(int)   // member function
  ret-type operatorop(arg,int)   // friend function

```
class Point
{      int x, y;
  public:
       Point() {x = y = 0; }

       Point& operator++(int)
       {  x++;  y++;  return *this;      }

       Point& operator--()
       {      x--;   y--;   return *this;}
};
```

```
void main()
{
Point p, p1,p2;
p1= p++;
p2= --p;
}
```

Unary Operators Overloading

- The same operators can be defined using the following (friend) function declarators:

```
friend Point& operator++( Point& )       // Prefix increment
friend Point& operator++( Point&, int ) // Postfix increment
friend Point& operator--( Point& )       // Prefix decrement
friend Point& operator--( Point&, int )  // Postfix
decrement
```

Friend operator Functions Add Flexibility

- Overloading an operator by using a friend or a member function makes, no functional difference.

- In exceptional situation in which overloading by using a friend increases the flexibility of an overloaded operator.

- Example:
  - Object + 100
  - 100 + object
    - In this case, it is integer that appears on the left.

Friend operator Functions Add Flexibility

```
class Point
{
        public:
                int x,y;
                Point () {};
                Point (int,int);
   friend Point operator +(int, Point);
   friend Point operator +(Point, int);
};
Point::Point (int a, int b)
{  x = a;  y = b;}
Point operator+ (Point P, int i)
{       Point temp;
        temp.x = P.x + i;
        temp.y = P.y + i;
        return (temp); }
```

```
Point operator+ (int i, Point P)
{       Point temp;
        temp.x = i + P.x;
        temp.y = i + P.y;
        return (temp); }

void main ()
{       Point a (3,1);
        Point b (1,2);
        Point c;
        c = a + 5;
        cout << c.x << "," << c.y;
        c=10+a;
        cout << c.x << "," << c.y;
}
```

Operator Functions as Class Members vs. as friend Functions

- Member vs non-member
  - Operator functions can be member or non-member functions.
  - When overloading **( )**, **[ ]**, **->** or any of the assignment operators, a member function must be used.
- Operator functions as member functions
  - Leftmost operand must be an object (or reference to an object) of the class
    - If left operand of a different type, operator function must be a non-member function
- Operator functions as non-member functions
  - Must be **friend**s if needs to access private or protected members
  - Enable the operator to be commutative

Assignment operator overloading

- Assignment operator (=) is, strictly speaking, a binary operator. Its declaration is identical to any other binary operator.

  Exceptions:

- It must be a non-static member function. No operator = can be declared as a non-member function.
- It is not inherited by derived classes.
- A default operator= function can be generated by the compiler for class types if none exists (*bitwise shallow copy*)
- User defined operator= function performs *member wise deep copy*.

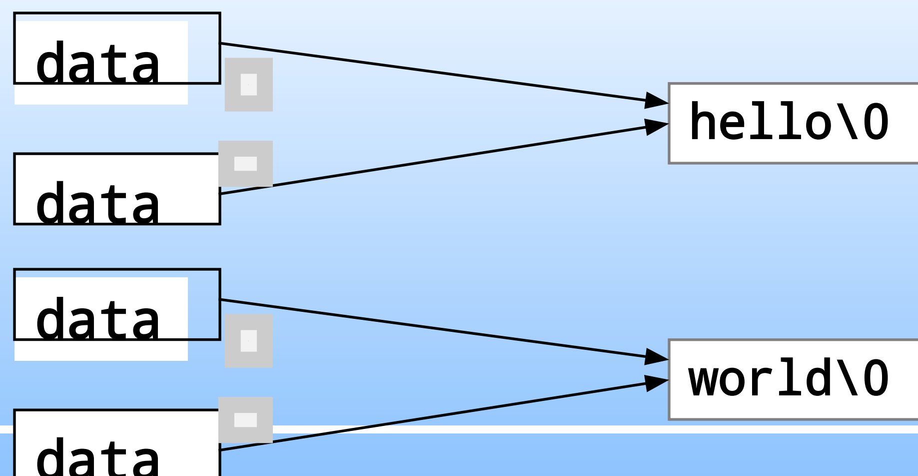
Assignment operator overloading

```
class String
{
    char* data;
    public:
    String(){ data=NULL; }
    String(const char* s = "")
    {   data = new char[20];
        strcpy(data,s);
    }
    ~String( )
    {     delete [ ] data;
    }
    void assign(char *str)
    {  strcpy(data,str);
    }
    void display()
    {  cout << data;
    }
};
```

```
void main()
{
   String s = "hello";
   String t;
   t=s;
   s.display( );
   t.display( );
   t.assign("world");
   s.display( );
   t.display( );
}
```

Default assignment

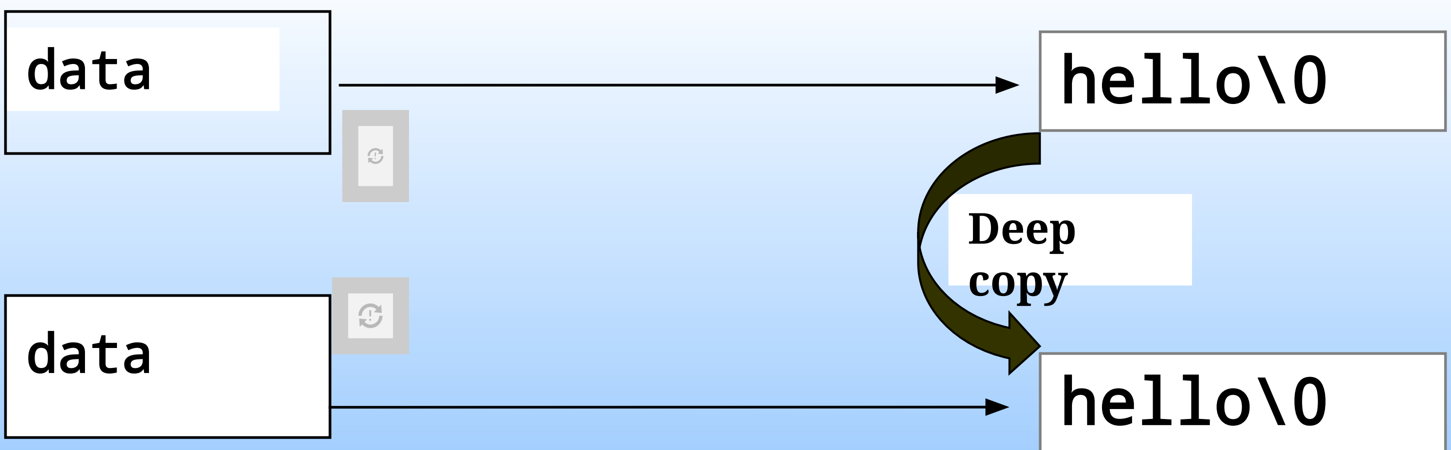
```
hello
hello
world
world
```



Assignment operator overloading

```
void operator=(const String& s)
{
   data = new char[strlen(s.data)+1];
   strcpy(data, s.data);
}
```

Overloaded operator function



Overloading IO Stream operators

Overloaded << and >> operators

- Overloaded to perform input/output for user-defined types
- Left operand of types `ostream` & and `istream` &
- Must be a non-member function because left operand is not an object of the class
- Must be a `friend` function to access private data member

Example: cin>>account;

cout<<account;

Syntax:

Keyword
Output stream
Reference types
Keyword
Ostream operator
Ostream object: cout
User defined object

```
friend ostream & operator << ( ostream &out, arg)
{       //display attributes
        return out
}
```

Overloading IO Stream operators

```
class Point
{
	public:
		int x,y;
		Point () {};
		Point (int,int);
  friend      ostream&      operator<<(ostream&,      const
     point&);
	friend istream& operator>>(istream&, const point&);
};
Point::Point (int a, int b)
{  x = a;  y = b;}
ostream& operator<<(ostream& os, const Point& a)
{	os << a.x;
	os << a.y;
	 return os;
}
istream& operator>>(istream& is, Point& a)
{	is >> a.x;
	is >> a.y;
	return is;
}
```

```
void main()
{
   Point              p1(2,3),
     p2(0,0);
       cin>>p2;
       cout<<p1;
       cout<<p2;
}
```

Type Conversion

- Compiler supports data conversion of only built-in data types.
- In case of user defined data type conversion, the data conversion interface function must be explicitly specified by the user.
- A single argument constructor or an operator function could be used for conversion of objects of different classes.

Conversion type	Source class	Destination class
Basic□ class	**Not applicable**	**Constructor**
Class□ basic	**Casting Operator**	**Not Applicable**
Class□ class	**Casting operator**	**Constructor**

Basic to User defined data type

- To convert basic to user-defined data type, single argument constructor conversion routine should be written in the destination object class.

```
    class Meter
{
        float length;
        public:
        Meter (float len)
        {     length=len;    }
};
main()
{
        float length1=15.56;
        meter1=length1;            // Converts basic data item length1 of float
         type to the object meter1 by      // invoking the one-argument
         constructor.
}
```

This constructor is invoked while creating objects of class Meter using a single argument of type float.

It converts the input argument represented in centimeters to meters and assigns the resultant value to length data member.

User defined data type to Basic Conversion

- To convert user-defined data type to basic, operator function should be written in the source object class.

```
class Meter
{
      float length;
      public:
      Meter (float len)
      {     length=len; }
     operator float()
      {     float len_cms;
            len_cms = length * 100.0; // meter to cm.
            return (len_cms);
      }
};
```

```
main()
{
      float length2;
      Meter meter2(100);
      length2=(float)
     meter2;
//   length2 = float(meter2);
}
```

Class to Class Conversion

Conversion routine in source class: Operator function

- To convert user-defined data type to another user-defined data type, operator function should be written in the source object class.

```
   class Meter
{      float length;
       public:
       Meter (float len)
       {      length=len;      }
    operator CentiMeter()
       {      return CentiMeter(len*100.0);   }
};
class CentiMeter
{      float Clength;
       public:
       CentiMeter (float len)
       {      Clength=len;    }
  };
```

```
main()
{
       Meter m(5);
       CentiMeter cm;
       cm=m;
}
```

Class to Class Conversion

Conversion routine in destination class: constructor function

- To convert user-defined data type to another user-defined data type, constructor function should be written in the destination object class.

```
class Meter
{       float length;
        public:
        Meter (float len)
        {       length=len;     }
        float getlength(){ return length;}
};
class CentiMeter
{       float Clength;
        public:
        CentiMeter (float len)
        {       Clength=len;    }
    CentiMeter(Meter m)
        {       Clength= m.getlength()*100.0);}
};
```

```
main()
{
        Meter m(5);
        CentiMeter cm;
        cm=m;
}
```

Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence and associativity of an operator cannot be changed
  - Arity (number of operands) cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators &, *, + and - each have unary and binary versions
    - Unary and binary versions can be overloaded separately

- No new operators can be created
  - Use only existing operators

- No overloading operators for built-in types (cannot redefine the meaning of operators)
  - Cannot change how two integers are added
  - Produces a syntax error

Implementing Operator Overloading

- Two ways:
  - Implemented as member functions
  - Implemented as non-member or Friend functions
    - the operator function may need to be declared as a friend if it requires access to protected or private data
- Expression *obj1@obj2* translates into a function call
  - *obj1.operator@(obj2),* if this function is defined within class obj1
  - *operator@(obj1,obj2),* if this function is defined outside the class obj1

Implementing Operator Overloading

1. Defined as a member function

```
class Complex {
   ...
 public:
   ...
   Complex operator +(const Complex &op)
   {
     double real   = _real   + op._real,
                 imag = _imag + op._imag;
     return(Complex(real, imag));
   }
   ...
 };
```

c = a+b;

→

c = a.operator+(b);

Implementing Operator Overloading

2. Defined as a non-member function

```
class Complex {
   ...
 public:
   ...
  double real() { return _real; }
    //need access functions
  double imag() { return _imag;
    }
   ...
 };
```

c = a+b;

c = operator+ (a, b);

```
Complex operator +(Complex &op1, Complex
    &op2)
{
   double real   = op1.real()   + op2.real(),
             imag = op1.imag() + op2.imag();
   return(Complex(real, imag));
}
```

Implementing Operator Overloading

3. Defined as a friend function

```
class Complex {
   ...
 public:
   ...
  friend Complex operator +(
     const Complex &,
     const Complex &
   );
   ...
 };
```

c = a+b; → **c = operator+ (a, b);**

```
Complex operator +(Complex &op1, Complex
    &op2)
{
   double real   = op1._real   + op2._real,
            imag = op1._imag + op2._imag;
   return(Complex(real, imag));
}
```

Overloading stream-insertion and stream-extraction operators

- In fact, cout<< or cin>> are operator overloading built in C++ standard lib of iostream.h, using operator "<<" and ">>"
- cout and cin are the objects of ostream and istream classes, respectively
- We can add a <u>friend</u> function which overloads the operator <<

```
friend ostream& operator<< (ostream &ous, const Date &d);
```

```
ostream& operator<<(ostream &os, const Date &d)
{
    os<<d.month<<"/"<<d.day<<"/"<<d.year;
    return os;
}
…
cout<< d1;  //overloaded operator
```

cout ---- object of ostream

Overloading stream-insertion and stream-extraction operators

- We can also add a friend function which overloads the operator >>

```
friend istream& operator>> (istream &in, Date &d);
```

```
istream& operator>> (istream &in, Date &d)
{
   char mmddyy[9];
   in >> mmddyy;
   // check if valid data entered
   if (d.set(mmddyy))   return in;
   cout<< "Invalid date format: "<<d<<endl;
   exit(-1);
 }
```

cin ---- object of istream

```
cin >> d1;
```

Class Template

Class Template

- A C++ language construct that allows the compiler to generate multiple versions of a class by allowing parameterized data types.

Class Template

```
Template < TemplateParamList >
ClassDefinition
```

TemplateParamDeclaration: placeholder

```
class  typeIdentifier
typename variableIdentifier
```

Example of a Class Template

```
template<class ItemType>
class GList
{
 public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( /* in */ ItemType item );
    void Delete( /* in */ ItemType item );
    bool IsPresent( /* in */ ItemType item ) const;
    void SelSort();
    void Print() const;
    GList();                          // Constructor
 private:
    int      length;
    ItemType data[MAX_LENGTH];
};
```

Template parameter

Instantiating a Class Template

- **Class template arguments *must* be explicit.**
- **The compiler generates distinct class types called template classes or generated classes.**
- **When instantiating a template, a compiler substitutes the template argument for the template parameter throughout the class template.**

Instantiating a Class Template

To create lists of different data types

```
// Client code

GList<int> list1;
GList<float> list2;
GList<string> list3;

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

template argument

Compiler generates 3 distinct class types

```
GList_int list1;
GList_float list2;
GList_string list3;
```

Substitution Example

```
class GList_int
{
public:

void Insert( /* in */ ItemType item );

    void Delete( /* in */ ItemType item );

    bool IsPresent( /* in */ ItemType item ) const;

private:
    int         length;
    ItemType data[MAX_LENGTH];
};
```

int

int

int

int

Function Definitions for Members of a Template Class

```
template<class ItemType>
void GList<ItemType>::Insert( /* in */ ItemType item )
{
    data[length] = item;
    length++;
}
```

```
//after substitution of float
void GList<float>::Insert( /* in */ float item )
{
    data[length] = item;
    length++;
}
```

Another Template Example: passing two parameters

```
template <class T, int size>
  class Stack {...
        T buf[size];
  };
Stack<int,128> mystack;
```

non-type parameter

Vectors

Vector

- A sequence that supports random access to elements
  - Elements can be inserted and removed at the beginning, the end and the middle
  - Constant time random access
  - Commonly used operations
    - begin(), end(), size(), [], push_back(…), pop_back(), insert(…), empty()

Example of vectors

```
// Instantiate a vector
vector<int> V;

// Insert elements
V.push_back(2);       // v[0] == 2
V.insert(V.begin(), 3); // V[0] == 3, V[1] == 2

// Random access
V[0] = 5;             // V[0] == 5

// Test the size
int size = V.size(); // size == 2
```

Exception Handling

Exception

- An exception is a unusual, often unpredictable event, detectable by software or hardware, that requires special processing occurring at runtime
- In C++, a variable or class object that represents an exceptional event.

Handling Exception

- If without handling,
  - Program crashes
  - Falls into unknown state
- An **exception handler** is a section of program code that is designed to execute when a particular exception occurs
  - Resolve the exception
  - Lead to known state, such as exiting the program

Standard Exceptions

- Exceptions Thrown by the Language
  - new
- Exceptions Thrown by Standard Library Routines
- Exceptions Thrown by user code, using *throw* statement

The *throw* Statement

Throw: to signal the fact that an exception has occurred; also called *raise*

ThrowStatement

`throw` **Expression**

The `try-catch` Statement

How one part of the program catches and processes the exception that another part of the program throws.

TryCatchStatement

```
try
  Block
catch (FormalParameter)
  Block
catch (FormalParameter)
```

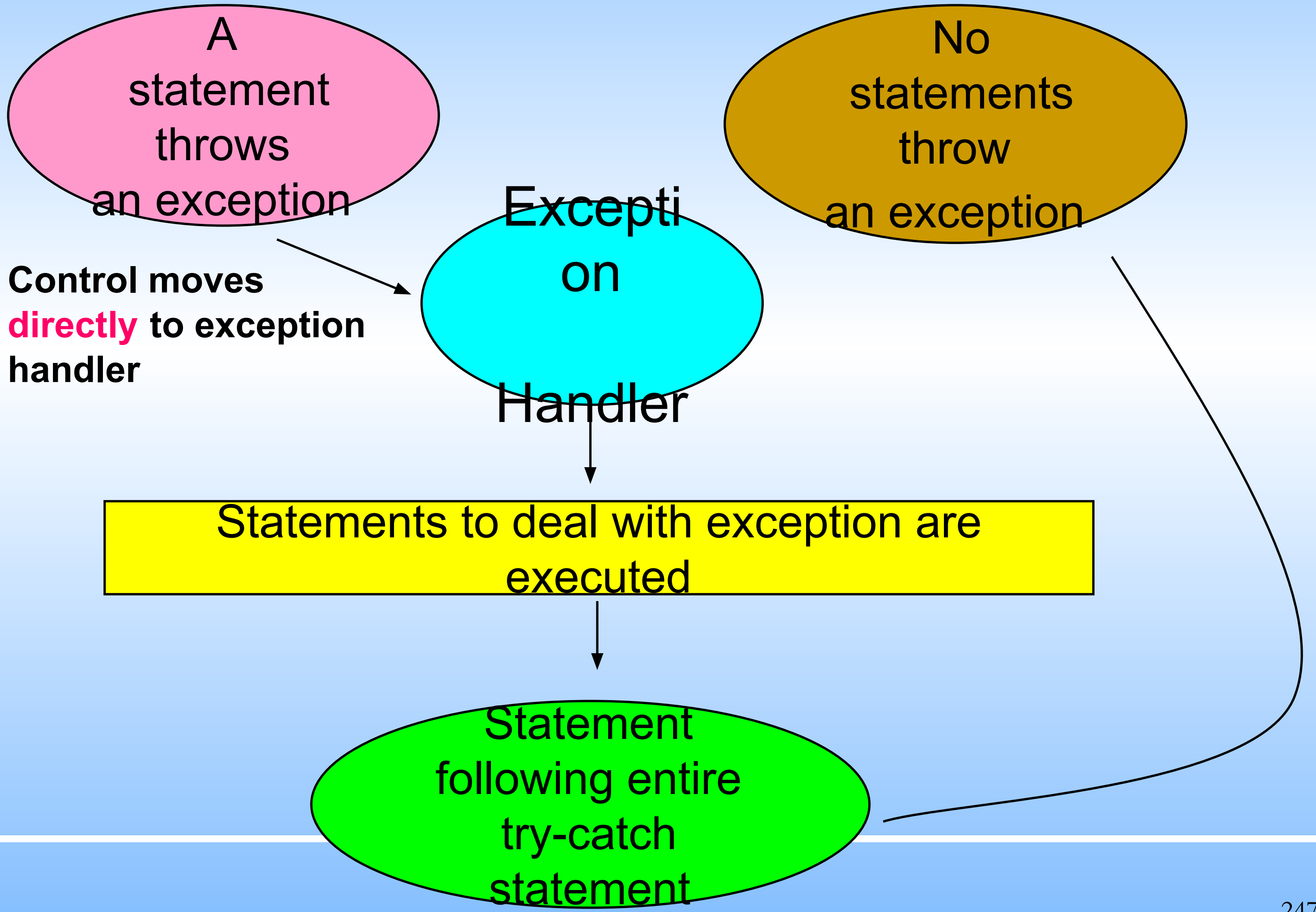
FormalParameter

```
DataType VariableName
...
```

Example of a `try-catch` Statement

```
try
{
        // Statements that process personnel data and may throw
        // exceptions of type int, string, and SalaryError
}
catch ( int )
{
        // Statements to handle an int exception
}
catch ( string s )
{
        cout << s << endl;  // Prints "Invalid customer age"
        // More statements to handle an age error
}
catch ( SalaryError )
{
        // Statements to handle a salary error
}
```

Execution of `try-catch`



Throwing an Exception to be Caught by the Calling Code

```
void Func3()
{

    try
    {

        Func4();

    }
    catch ( ErrType )
    {

    }

}
```

Function call

Normal return

```
void Func4()
{

    if ( error )
        throw ErrType();

}
```

Return from thrown exception

Practice: Dividing by ZERO

Apply what you know:

```
int Quotient(int numer,    // The numerator
       int denom )   // The denominator
{
    if (denom != 0)
        return numer / denom;
    else
        //What to do?? do sth. to avoid program      //
crash
}
```

A Solution

```
int Quotient(int numer,    // The numerator
          int denom )   // The denominator
{
  if (denom == 0)
     throw DivByZero();
   //throw exception of class DivByZero
  return numer / denom;
}
```

A Solution

```
// quotient.cpp -- Quotient program

#include<iostream.h>
#include <string.h>
int Quotient( int, int );
class DivByZero {}; // Exception class
int main()
{
   int numer;  // Numerator
   int denom;  // Denominator

   //read in numerator
     and denominator
while(cin)
{
    try
    {
     cout << "Their quotient: "
                  << Quotient(numer,denom) <<endl;
    }
        catch ( DivByZero )//exception handler
    {
      cout<<"Denominator can't be 0"<< endl;
    }
    // read in numerator and denominator
}
return 0;
}
```

End