Access Specifiers/Modifiers

- Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be directly accessed by the outside functions.
  There are 3 types of access modifiers available in C++:
- **Public**
- **Private**
- **Protected**

Note: If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be **Private**.

Public Access Specifier

- Public : All the class members declared under the public specifier will be available to everyone.
- The data members and member functions declared as public can be accessed by other classes and functions too.
- The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Example

```
#include<iostream>
using namespace std;
class Circle
{
   public:
      double radius;
      double  compute_area()
      {
         return 3.14*radius*radius;
      }
    };
int main()
{
   Circle obj;
    obj.radius = 5.5;

   cout << "Radius is: " << obj.radius << "\n";
   cout << "Area is: " << obj.compute_area();
   return 0;
}
```

Private Access Specifier

- **Private**: The class members declared as *private* can be accessed only by the member functions inside the class.

- They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class..

Example

```
#include<iostream>
using namespace std;
class Circle
{
   private:
      double radius;
   public:
      double  compute_area(double r)
      {            radius=r;
                    return 3.14*radius*radius;
                    cout << "Radius is: " << radius << endl;
                    cout << "Area is: " << area;
   }
    };
int main()
{
   Circle obj;
   // trying to access private data member
   // directly outside the class
   obj.radius = 1.5;
   obj.compute_area(1.5);
   return 0;
}
```

Protected Access Specifier

- **Protected**: The protected access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class unless with the help of a friend class.
- The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

  Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the mode of Inheritance.

Example

```
class Parent
{
   protected:
   int id_protected;
   };

 // sub class or derived class from public base class
class Child : public Parent
{
   public:
   void setId(int id)
   {
      id_protected = id;
   }
   void displayId()
   {
      cout << "id_protected is: " << id_protected << endl;
   }
};
int main() {
  Child obj1;
   obj1.setId(81);
   obj1.displayId();
   return 0;
}
```

Manipulators in C++

- Manipulators are the operators used to format the data that is to be displayed on screen.
- The most commonly used manipulators are endl and setw.
- Endl:- It is used in output statement and inserts a line feed. It is similar to new line character (“\n”) ex: …….. cout<<<<endl;
- setw:- this manipulator allows a specified width for a field that is to be printed on screen and by default the value printed is right justified.This function is available in header file iomanip.h

```
#include <iostram>
#include <iomanip>
using namespace std;
int main()
 {
int s=123;
cout<<"s="<<setw(10)<<s;
Return 0;
}
```

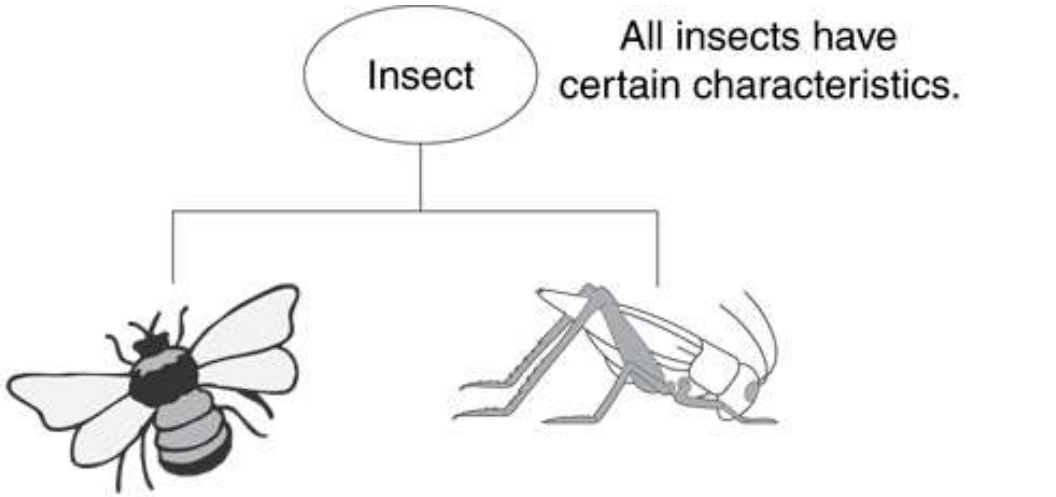
- Insertion operators(<<)
- Extraction operator(>>)

Inheritance, Polymorphism, and Virtual Functions

What Is Inheritance?

- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class

Example: Insect Taxonomy



The "is a" Relationship

- Inheritance establishes an "is a" relationship between classes.
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete

Inheritance – Terminology and Notation in C++

- Base class (or parent) – inherited from
- Derived class (or child) – inherits from the base class
- Notation:

```
class Student           // base class
{
  . . .
};
class UnderGrad : public student
{                       // derived class
  . . .
};
```

Back to the ‘is a’ Relationship

- An object of a derived class 'is a(n)' object of the base class
- Example:
  - an `UnderGrad` is a `Student`
  - a `Mammal` is an `Animal`
- A derived object has **all** of the characteristics of the base class

What Does a Child Have?

An object of the derived class has:

- all members defined in child class
- all members declared in parent class

An object of the derived class can use:

- all `public` members defined in child class
- all `public` members defined in parent class

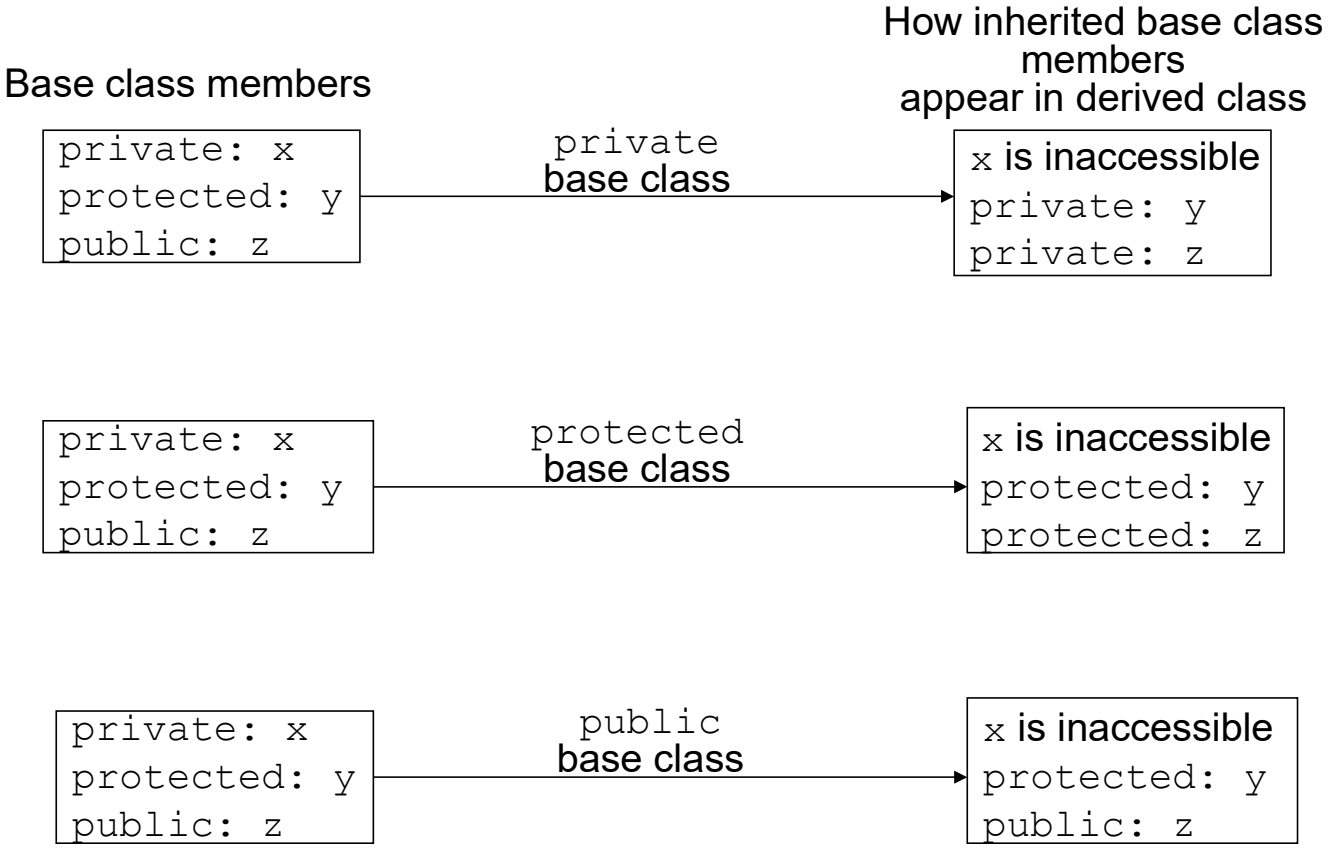
Protected Members and Class Access

- `protected` member access specification: like `private`, but accessible by objects of derived class
- Class access specification: determines how `private`, `protected`, and `public` members of base class are inherited by the derived class

Class Access Specifiers

1) `public` – object of derived class can be treated as object of base class (not vice-versa)
2) `protected` – more restrictive than `public`, but allows derived classes to know details of parents
3) `private` – prevents objects of derived class from being treated as objects of base class.

Inheritance vs. Access



Inheritance vs. Access

class Grade

private members:
```
  char letter;
  float score;
  void calcGrade();
```
public members:
```
  void setScore(float);
  float getScore();
  char getLetter();
```

class Test : public Grade

private members:
```
  int numQuestions;
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `public` class access, it looks like this: →

private members:
```
  int numQuestions:
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
  void setScore(float);
  float getScore();
  char getLetter();
```

Inheritance vs. Access

class Grade

private members:
```
  char letter;
  float score;
  void calcGrade();
```
public members:
```
  void setScore(float);
  float getScore();
  char getLetter();
```

class Test : protected Grade

private members:
```
  int numQuestions;
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `protected` class access, it looks like this: →

private members:
```
  int numQuestions:
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```
protected members:
```
  void setScore(float);
  float getScore();
  float getLetter();
```

Inheritance vs. Access

class Grade

private members:
```
  char letter;
  float score;
  void calcGrade();
```
public members:
```
  void setScore(float);
  float getScore();
  char getLetter();
```

class Test : private Grade

private members:
```
  int numQuestions;
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `private` class access, it looks like this: →

private members:
```
  int numQuestions:
  float pointsEach;
  int numMissed;
  void setScore(float);
  float getScore();
  float getLetter();
```
public members:
```
  Test(int, int);
```

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Constructors and Destructors in Base and Derived Classes

Program 15-4

```
1  // This program demonstrates the order in which base and
2  // derived class constructors and destructors are called.
3  #include <iostream>
4  using namespace std;
5
6  //*********************************
7  // BaseClass declaration         *
8  //*********************************
9
```

Constructors and Destructors in Base and Derived Classes

Program 15-4 *(continued)*

```
10  class BaseClass
11  {
12  public:
13     BaseClass()  // Constructor
14        { cout << "This is the BaseClass constructor.\n"; }
15
16     ~BaseClass() // Destructor
17        { cout << "This is the BaseClass destructor.\n"; }
18  };
19
20  //*********************************
21  // DerivedClass declaration      *
22  //*********************************
23
24  class DerivedClass : public BaseClass
25  {
26  public:
27     DerivedClass()  // Constructor
28        { cout << "This is the DerivedClass constructor.\n"; }
29
30     ~DerivedClass()  // Destructor
31        { cout << "This is the DerivedClass destructor.\n"; }
32  };
33
```

Constructors and Destructors in Base and Derived Classes

```
34  //********************************
35  // main function                 *
36  //********************************
37
38  int main()
39  {
40     cout << "We will now define a DerivedClass object.\n";
41
42     DerivedClass object;
43
44     cout << "The program is now going to end.\n";
45     return 0;
46  }
```

Program Output

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

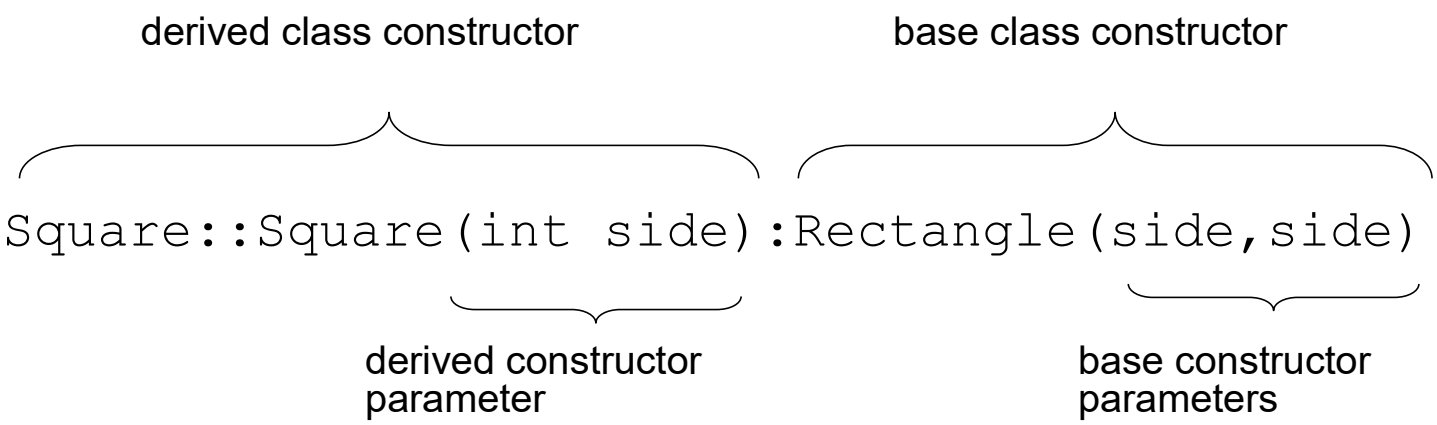
Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:

```
Square::Square(int side) :
  Rectangle(side, side)
```

- Can also be done with inline constructors
- Must be done if base class has no default constructor

Passing Arguments to Base Class Constructor



Redefining Base Class Functions

- Redefining function: function in a derived class that has the *same name and parameter list* as a function in the base class

- Typically used to replace a function in base class with different actions in derived class

Redefining Base Class Functions

- Not the same as overloading – with overloading, parameter lists must be different
- Objects of base class use base class version of function; objects of derived class use derived class version of function

Base Class

```
class GradedActivity
{
protected:
   char letter;            // To hold the letter grade
   double score;           // To hold the numeric score
   void determineGrade();  // Determines the letter grade
public:
   // Default constructor
   GradedActivity()
      { letter = ' '; score = 0.0; }

   // Mutator function
   void setScore(double s)
      { score = s;
        determineGrade();}

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const
      { return letter; }
};
```

Derived Class

```
1  #ifndef CURVEDACTIVITY_H
2  #define CURVEDACTIVITY_H
3  #include "GradedActivity.h"
4
5  class CurvedActivity : public GradedActivity
6  {
7  protected:
8     double rawScore;      // Unadjusted score
9     double percentage;    // Curve percentage
10 public:
11    // Default constructor
12    CurvedActivity() : GradedActivity()
13       { rawScore = 0.0; percentage = 0.0; }
14
15    // Mutator functions
16    void setScore(double s)
17       { rawScore = s;
18         GradedActivity::setScore(rawScore * percentage); }
19
20    void setPercentage(double c)
21       { percentage = c; }
22
23    // Accessor functions
24    double getPercentage() const
25       { return percentage; }
26
27    double getRawScore() const
28       { return rawScore; }
29 };
30 #endif
```

Redefined `setScore` function

Driver Program

```
13      // Define a CurvedActivity object.
14      CurvedActivity exam;
15
16      // Get the unadjusted score.
17      cout << "Enter the student's raw numeric score: ";
18      cin >> numericScore;
19
20      // Get the curve percentage.
21      cout << "Enter the curve percentage for this student: ";
22      cin >> percentage;
23
24      // Send the values to the exam object.
25      exam.setPercentage(percentage);
26      exam.setScore(numericScore);
27
28      // Display the grade data.
29      cout << fixed << setprecision(2);
30      cout << "The raw score is "
31           << exam.getRawScore() << endl;
32      cout << "The curved score is "
33           << exam.getScore() << endl;
34      cout << "The curved grade is "
35           << exam.getLetterGrade() << endl;
```

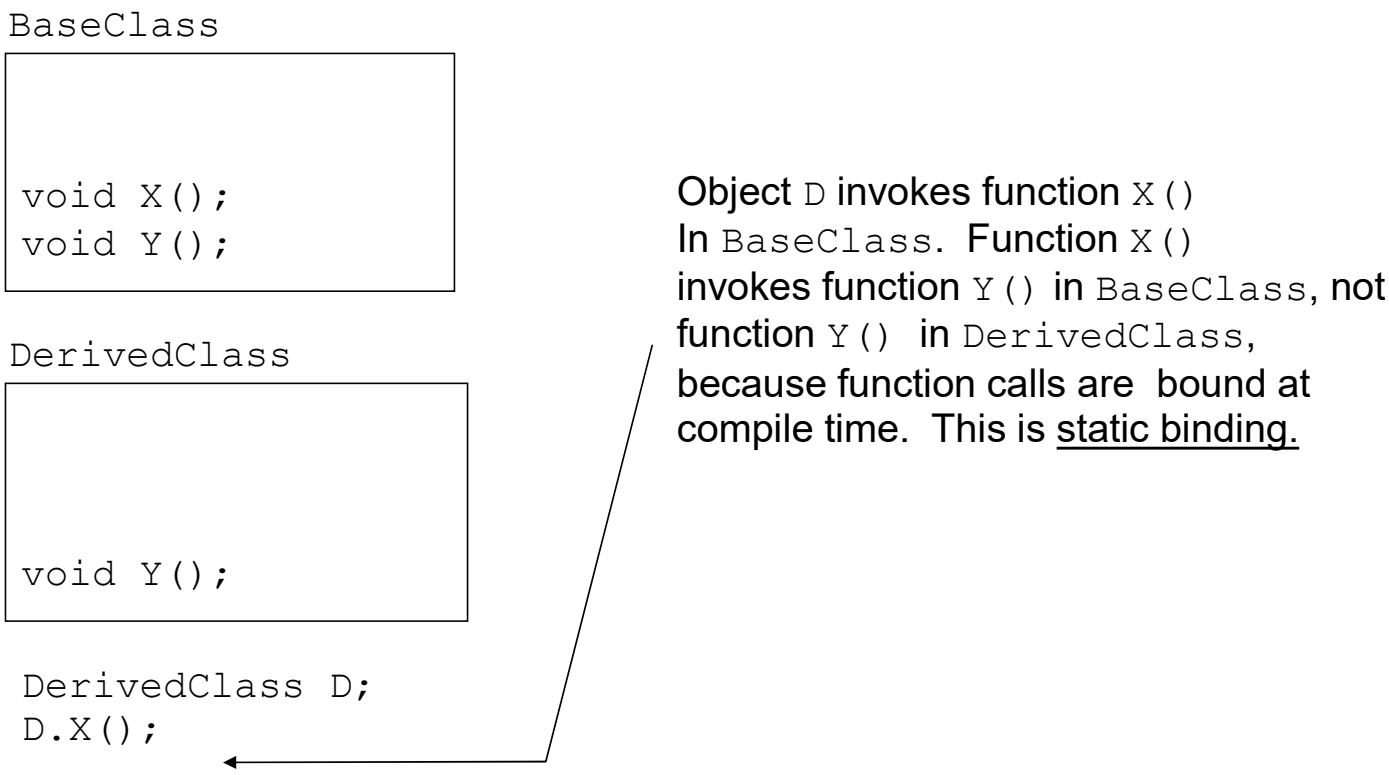
Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87 [Enter]
Enter the curve percentage for this student: 1.06 [Enter]
The raw score is 87.00
The curved score is 92.22
The curved grade is A
```

Problem with Redefining

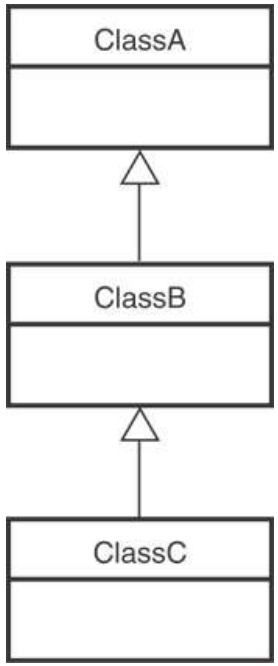
- Consider this situation:
  - Class `BaseClass` defines functions `x()` and `y()`. `x()` calls `y()`.
  - Class `DerivedClass` inherits from `BaseClass` and redefines function `y()`.
  - An object `D` of class `DerivedClass` is created and function `x()` is called.
  - When `x()` is called, which `y()` is used, the one defined in `BaseClass` or the the redefined one in `DerivedClass`?

Problem with Redefining



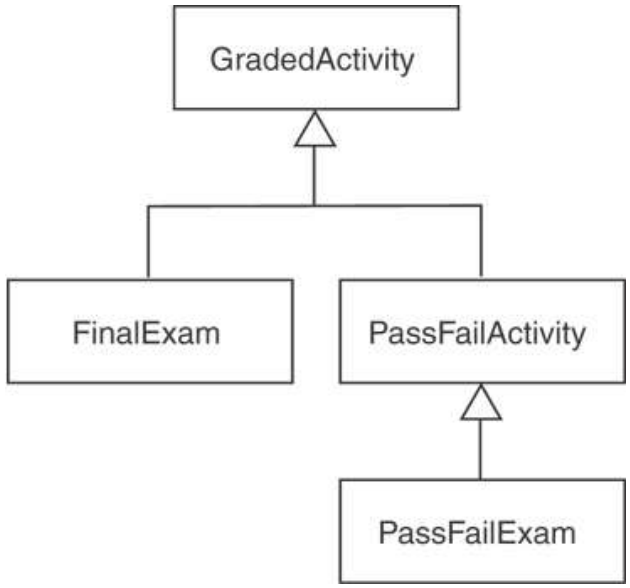
Class Hierarchies

- A base class can be derived from another base class.



Class Hierarchies

- Consider the GradedActivity, FinalExam, PassFailActivity, PassFailExam hierarchy in Chapter 15.



Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class
- Function defined with key word `virtual`:

```
virtual void Y() {...}
```

- Supports dynamic binding: functions bound at run time to function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Polymorphism and Virtual Member Functions

```
29  void displayGrade(const GradedActivity &activity)
30  {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33          << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35          << activity.getLetterGrade() << endl;
36  }
```

Because the parameter in the `displayGrade` function is a GradedActivity reference variable, it can reference any object that is derived from GradedActivity. That means we can pass a GradedActivity object, a FinalExam object, a PassFailExam object, or any other object that is derived from GradedActivity.

A problem occurs in Program 15-10 however...

Program 15-10

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailActivity.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity &);
8
9  int main()
10 {
11    // Create a PassFailActivity object. Minimum passing
12    // score is 70.
13    PassFailActivity test(70);
14
15    // Set the score to 72.
16    test.setScore(72);
17
18    // Display the object's grade data. The letter grade
19    // should be 'P'. What will be displayed?
20    displayGrade(test);
21    return 0;
22 }
```

```
23
24  //*************************************************************
25  // The displayGrade function displays a GradedActivity object's *
26  // numeric score and letter grade.                              *
27  //*************************************************************
28
29  void displayGrade(const GradedActivity &activity)
30  {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33          << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35          << activity.getLetterGrade() << endl;
36  }
```

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is C
```

As you can see from the example output, the `getLetterGrade` member function returned 'C' instead of 'P'. This is because the GradedActivity class's `getLetterGrade` function was executed instead of the PassFailActivity class's version of the function.

Static Binding

- Program 15-10 displays 'C' instead of 'P' because the call to the `getLetterGrade` function is statically bound (at compile time) with the GradedActivity class's version of the function.
  We can remedy this by making the function *virtual*.

Virtual Functions

- A virtual function is dynamically bound to calls at runtime.
  At runtime, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.

Virtual Functions

- To make a function virtual, place the virtual key word before the return type in the base class's declaration:

```
virtual char getLetterGrade() const;
```

- The compiler will not bind the function to calls. Instead, the program will bind them at runtime.

Updated Version of GradedActivity

```
6  class GradedActivity
7  {
8  protected:
9     double score;   // To hold the numeric score
10 public:
11    // Default constructor
12    GradedActivity()
13       { score = 0.0; }
14
15    // Constructor
16    GradedActivity(double s)
17       { score = s; }
18
19    // Mutator function
20    void setScore(double s)
21       { score = s; }
22
23    // Accessor functions
24    double getScore() const
25       { return score; }
26
27    virtual char getLetterGrade() const;
28 };
```

The function is now virtual.

The function also becomes virtual in all derived classes automatically!

Polymorphism

If we recompile our program with the updated versions of the classes, we will get the right output, shown here:
(See Program 15-11 in the book.)

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is P
```

This type of behavior is known as polymorphism. The term *polymorphism* means the ability to take many forms.

Program 15-12 demonstrates polymorphism by passing objects of the GradedActivity and PassFailExam classes to the displayGrade function.

Program 15-12

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailExam.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity &);
8
9  int main()
10 {
11    // Create a GradedActivity object. The score is 88.
12    GradedActivity test1(88.0);
13
14    // Create a PassFailExam object. There are 100 questions,
15    // the student missed 25 of them, and the minimum passing
16    // score is 70.
17    PassFailExam test2(100, 25, 70.0);
18
19    // Display the grade data for both objects.
20    cout << "Test 1:\n";
21    displayGrade(test1);    // GradedActivity object
22    cout << "\nTest 2:\n";
```

```
23      displayGrade(test2);    // PassFailExam object
24      return 0;
25   }
26
27   //***************************************************************
28   // The displayGrade function displays a GradedActivity object's *
29   // numeric score and letter grade.                               *
30   //***************************************************************
31
32   void displayGrade(const GradedActivity &activity)
33   {
34      cout << setprecision(1) << fixed;
35      cout << "The activity's numeric score is "
36           << activity.getScore() << endl;
37      cout << "The activity's letter grade is "
38           << activity.getLetterGrade() << endl;
39   }
```

Program Output

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is P
```

Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer, as demonstrated in the `displayGrade` function.

Base Class Pointers

- Can define a pointer to a *base* class object
- Can assign it the address of a *derived* class object

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);

cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

Base Class Pointers

- Base class pointers and references only know about members of the base class
  - So, you can't use a base class pointer to call a derived class function
- Redefined functions in *derived* class will be ignored unless *base* class declares the function `virtual`

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.
  So, a virtual function is overridden, and a non-virtual function is redefined.

Virtual Destructors

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from.
- See Program 15-14 for an example

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:

```
virtual void Y() = 0;
```

- The `= 0` indicates a pure virtual function
- Must have no function definition in the base class

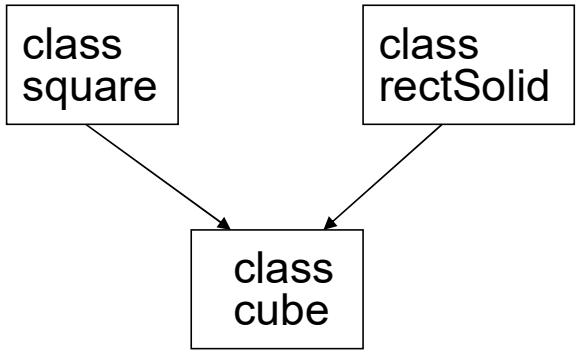
Abstract Base Classes and Pure Virtual Functions

- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function

Multiple Inheritance

- A derived class can have more than one base class
- Each base class can have its own access specification in derived class's definition:

```
class cube : public square,
             public rectSolid;
```



Multiple Inheritance

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
  - Derived class redefines the multiply-defined function
  - Derived class invokes member function in a particular base class using scope resolution operator `::`
- Compiler errors occur if derived class uses base class function without one of these solutions

Friend Function & Class

Friend Functions/Classes

`friends` allow functions/classes access to private data of other classes.

Friend Functions/Classes

Friend functions

A **'friend'** function has access to all 'private' members of the class for which it is a 'friend'.

To declare a 'friend' function, include its prototype within the class, preceding it with the C++ keyword 'friend'.

Sample application that can benefit from friend classes/functions

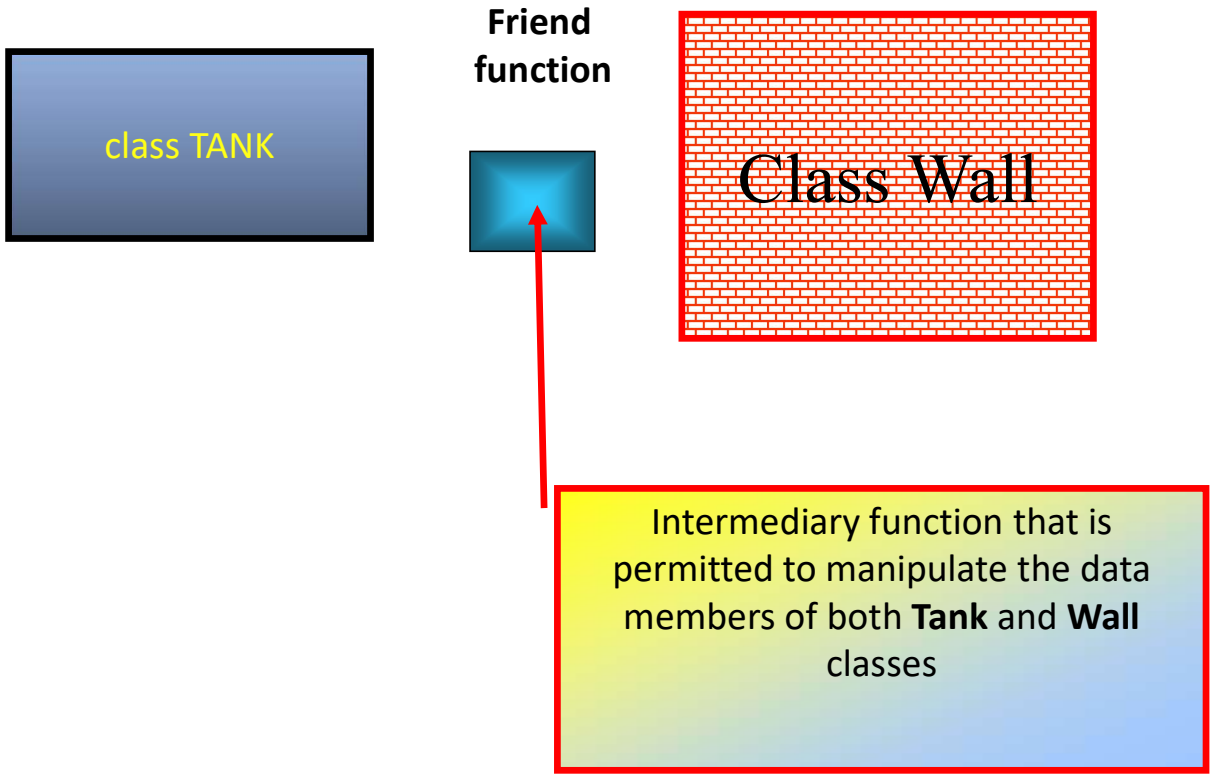
COLLISION PROBLEM

APPROACHES: SIMPLE RECTANGLE , SHRUNKEN RECT, SPRITE IMAGE

- One class for each moving object
- One instance of a class doesn't know the boundaries of other moving objects
- Data members of each object is hidden and protected from other moving objects

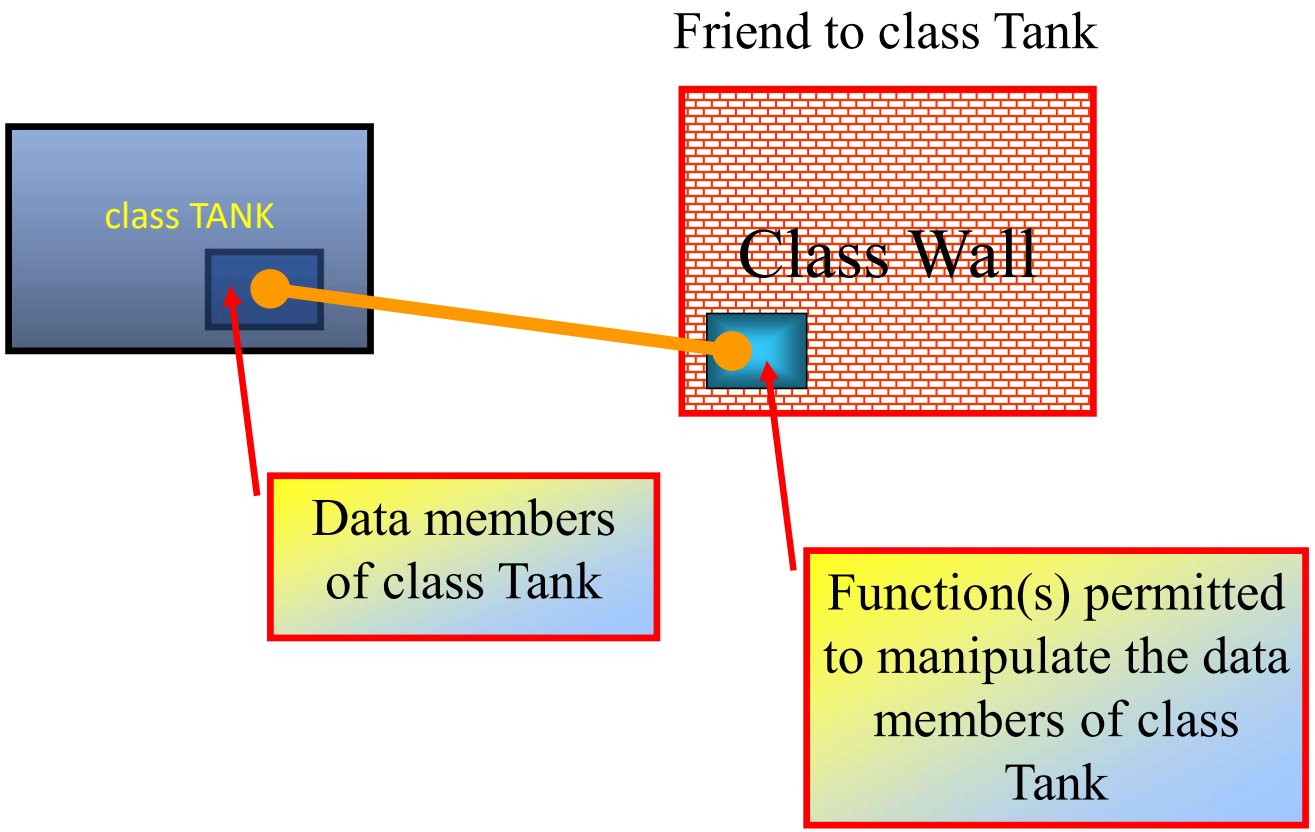
COLLISION PROBLEM

Friend functions/class by-pass object data hiding



COLLISION PROBLEM

Friend functions/class by-pass object data hiding

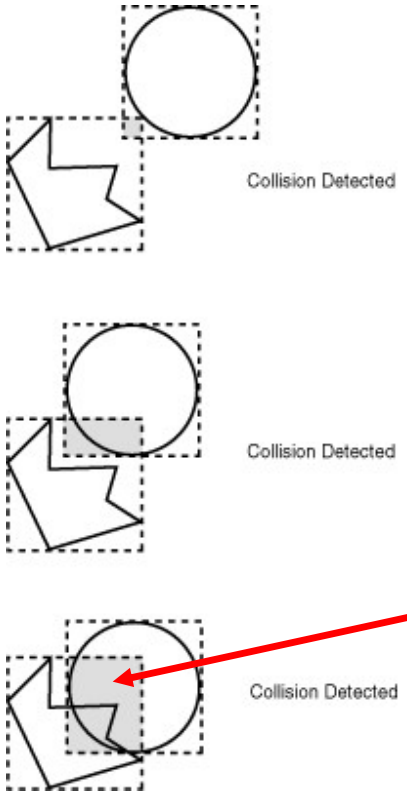


COLLISION PROBLEM

SIMPLE RECTANGLE APPROACH

How to detect collision between objects?

- **not convincing enough!**

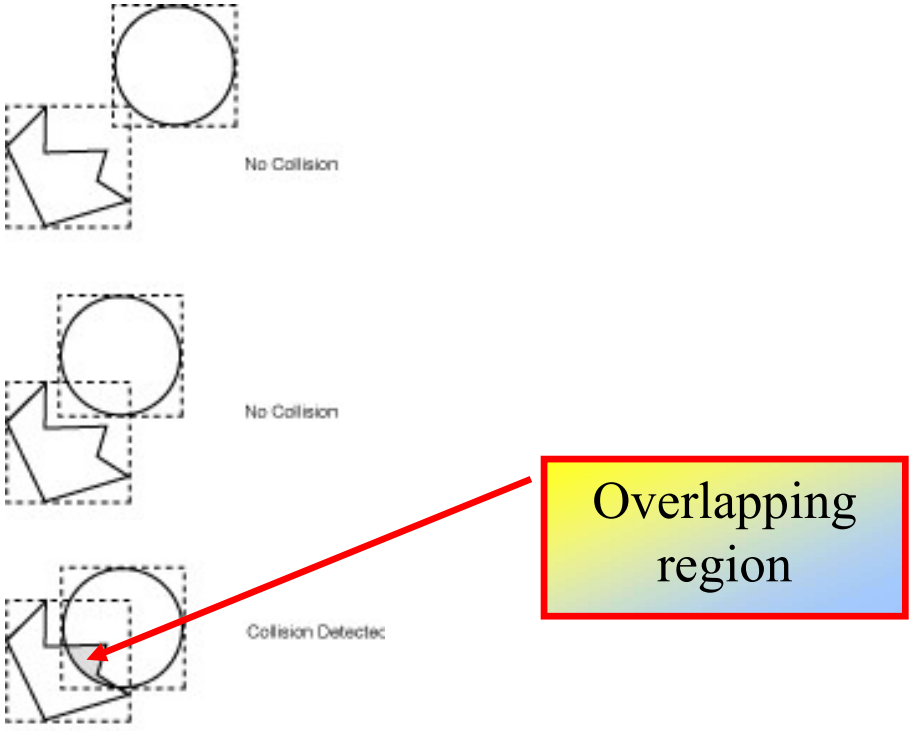


Overlapping region

COLLISION PROBLEM

SPRITE IMAGE APPROACH

How to detect collision between objects?



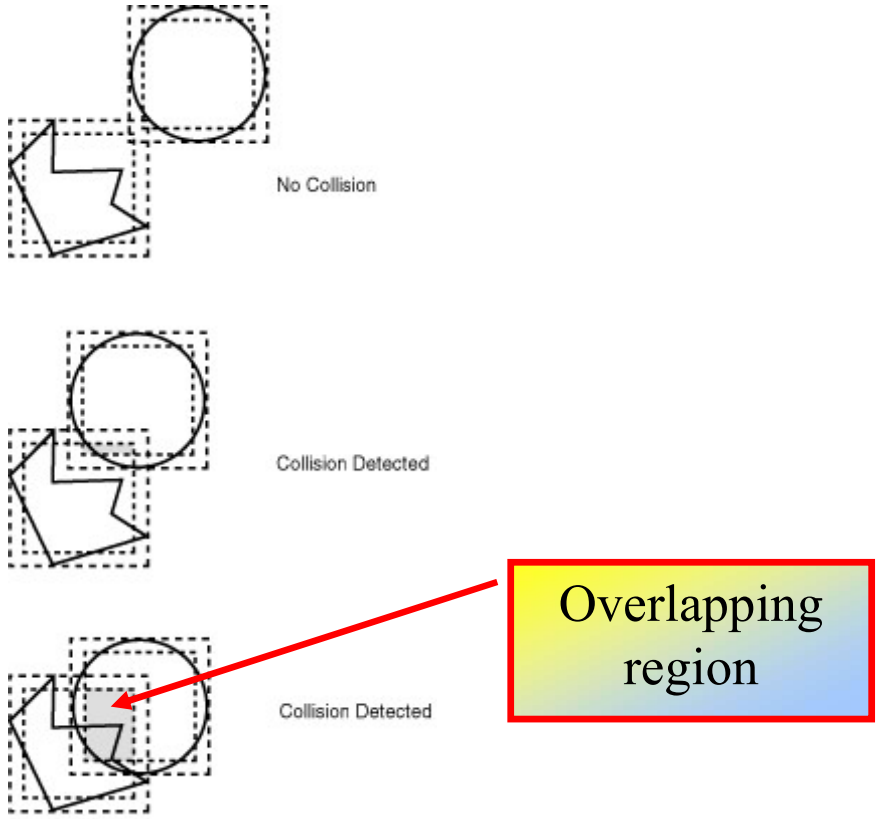
- **can be a major bottleneck in performance to get a decent animation**

COLLISION PROBLEM

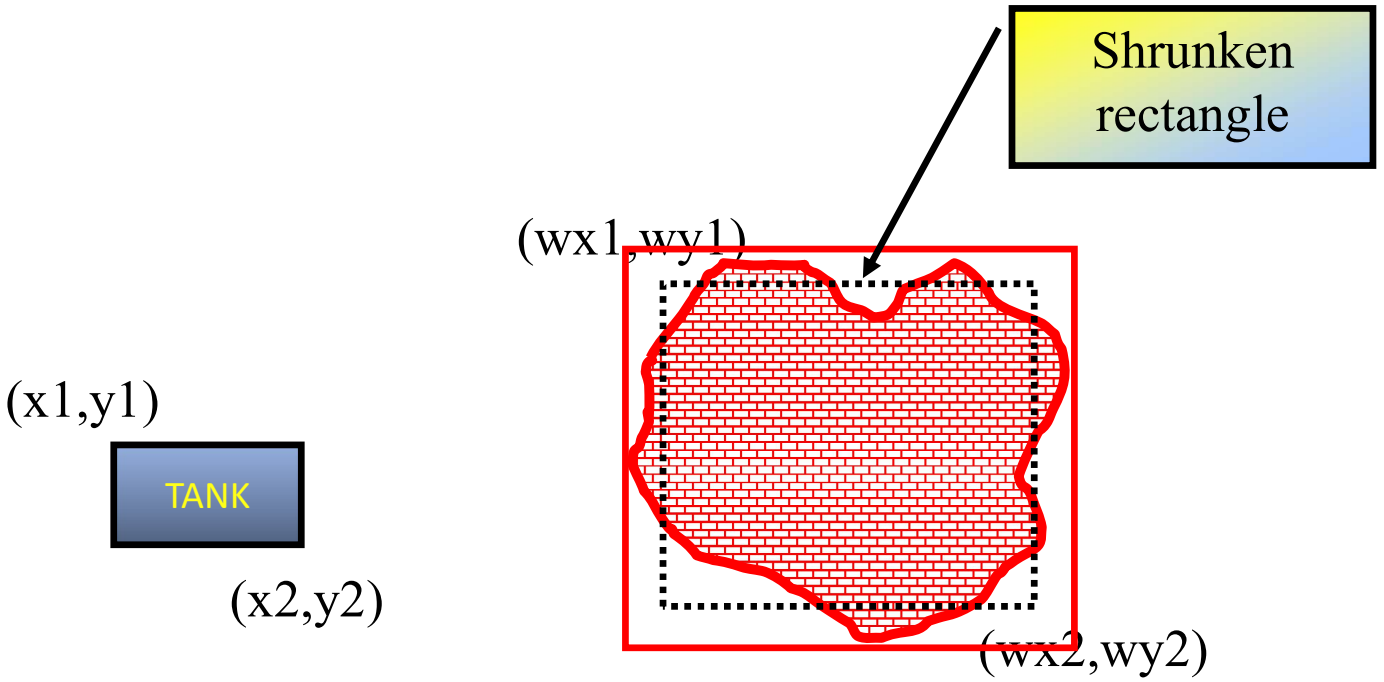
SHRUNKEN RECTANGLE APPROACH

How to detect collision between objects?

- **Fast**
- **Simple to implement**
- **perfect for a game with many moving objects**

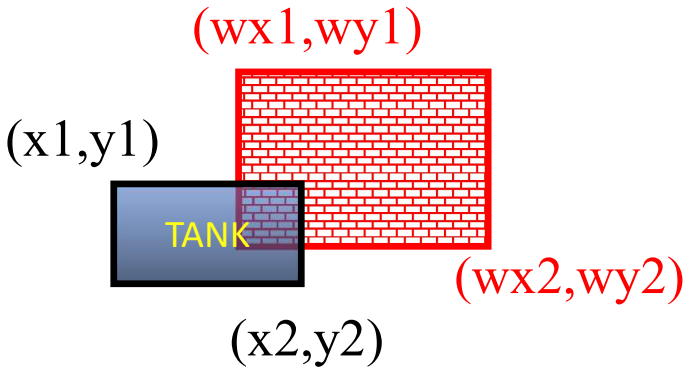


COLLISION PROBLEM

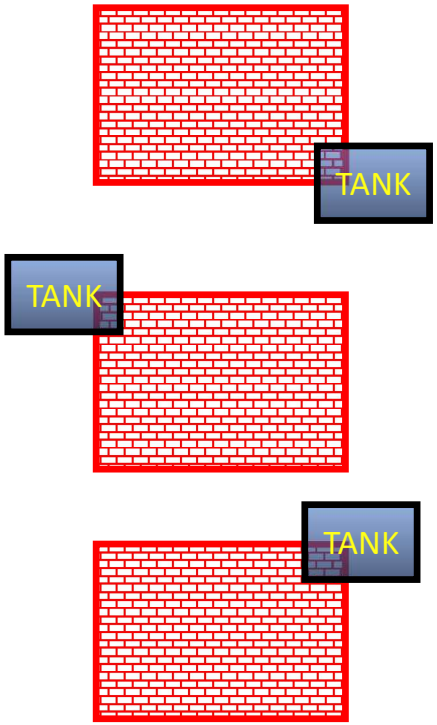


Non-rectangular object

COLLISION PROBLEM



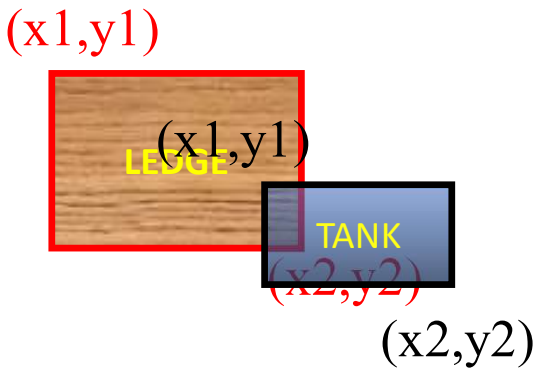
Sample Collision Cases:



COLLISION PROBLEM

Sample Collision Cases:

(t.x1 >= ledge.x1) && (t.x1 <= ledge.x2)



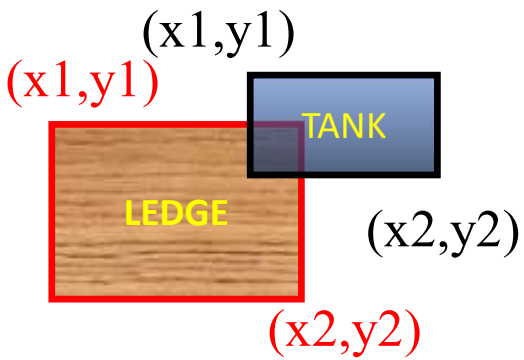
Note: This example is only using the device system of coordinates and is therefore not scalable.

```
if(  ((t.x1 >= ledge.x1) && (t.x1 <= ledge.x2) && (t.y1 >= ledge.y1) && (t.y1 <= ledge.y2))    ||
     ((t.x1 >= ledge.x1) && (t.x1 <= ledge.x2) && (t.y2 >= ledge.y1) && (t.y2 <= ledge.y2)) ||
     ((t.x2 >= ledge.x1) && (t.x2 <= ledge.x2) && (t.y1 >= ledge.y1) && (t.y1 <= ledge.y2)) ||
     ((t.x2 >= ledge.x1) && (t.x2 <= ledge.x2) && (t.y2 >= ledge.y1) && (t.y2 <= ledge.y2))     )
```

COLLISION PROBLEM

Sample Collision Cases:

(t.x1 >= ledge.x1) && (t.x1 <= ledge.x2)



Note: This example is only using the device system of coordinates and is therefore not scalable.

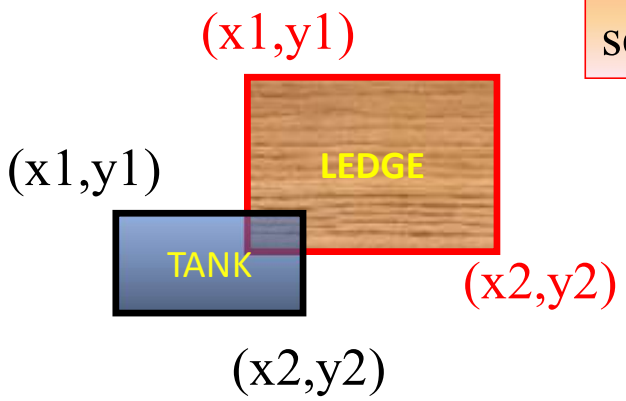
```
if(  ((t.x1 >= ledge.x1) && (t.x1 <= ledge.x2) && (t.y1 >= ledge.y1) && (t.y1 <= ledge.y2))    ||
     ((t.x1 >= ledge.x1) && (t.x1 <= ledge.x2) && (t.y2 >= ledge.y1) && (t.y2 <= ledge.y2)) ||
     ((t.x2 >= ledge.x1) && (t.x2 <= ledge.x2) && (t.y1 >= ledge.y1) && (t.y1 <= ledge.y2)) ||
     ((t.x2 >= ledge.x1) && (t.x2 <= ledge.x2) && (t.y2 >= ledge.y1) && (t.y2 <= ledge.y2))     )
```

COLLISION PROBLEM

Sample Collision Cases:

(t.x2 >= ledge.x1) && (t.x2 <= ledge.x2)

Note: This example is only using the device system of coordinates and is therefore not scalable.

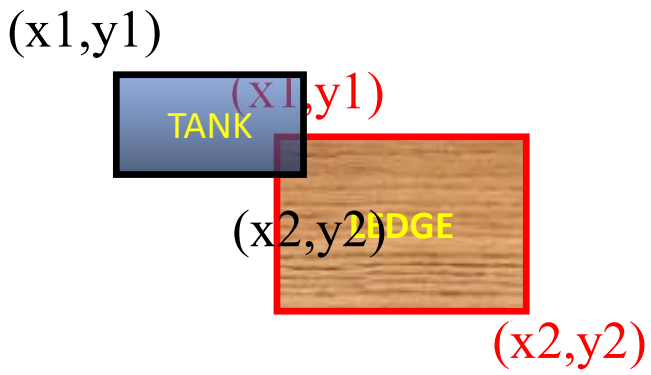



```
if(  ((t.x1 >= ledge.x1) && (t.x1 <= ledge.x2) && (t.y1 >= ledge.y1) && (t.y1 <= ledge.y2))    ||
     ((t.x1 >= ledge.x1) && (t.x1 <= ledge.x2) && (t.y2 >= ledge.y1) && (t.y2 <= ledge.y2)) ||
     ((t.x2 >= ledge.x1) && (t.x2 <= ledge.x2) && (t.y1 >= ledge.y1) && (t.y1 <= ledge.y2)) ||
     ((t.x2 >= ledge.x1) && (t.x2 <= ledge.x2) && (t.y2 >= ledge.y1) && (t.y2 <= ledge.y2))     )
```

COLLISION PROBLEM

Sample Collision Cases:

(t.x2 >= ledge.x1) && (t.x2 <= ledge.x2)



Note: This example is only using the device system of coordinates and is therefore not scalable.

```
if(  ((t.x1 >= ledge.x1) && (t.x1 <= ledge.x2) && (t.y1 >= ledge.y1) && (t.y1 <= ledge.y2))    ||
     ((t.x1 >= ledge.x1) && (t.x1 <= ledge.x2) && (t.y2 >= ledge.y1) && (t.y2 <= ledge.y2)) ||
     ((t.x2 >= ledge.x1) && (t.x2 <= ledge.x2) && (t.y1 >= ledge.y1) && (t.y1 <= ledge.y2)) ||
     ((t.x2 >= ledge.x1) && (t.x2 <= ledge.x2) && (t.y2 >= ledge.y1) && (t.y2 <= ledge.y2))     )
```

Friend Functions/Classes

```
class T {
 public:
   friend void a();
   int m();
 private:  // ...
};

void a() {// can access
          // private data in T...}


class S {
 public:
   friend int T::m();
   //...
};

class X {
public:
   friend class T;
   //...
};
```

- Global function `a()` can access private data in `T`
- `m()` can access private data in `S`
- all functions of `T` can access private data in `X`

`friends` should be used with caution: they by-pass C++'s data hiding principle.

It is the responsibility of the code for which access is to be given to say who it's friends are - i.e. who does it trust!

Friend Functions/Classes

```
class Demo {
   friend void Change( Demo obj );
 public:
    Demo(double x0=0.0, int y0=0.0):x(x0),y(y0){}
    void print();
 private:
    double x; int y;
};

void Demo::print(){
   cout << endl << "This is x " << x << endl;
   cout << "This is y " << y << endl;
}

void Change( Demo obj ) {
   obj.x += 100;
   obj.y += 200;
   cout << "This is obj.x" << obj.x << endl;
   cout << "This is obj.y" << obj.y << endl;
}
```

Friend Functions/Classes

```
#include <iostream>
using namespace std;

const int DaysInMonth[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
enum Months { unused, January, February, March, April, May, June,
              July, August, September, October, November, December };
const char *MonthNames[]={"Unused", "January", "February", "March", "April", "May", "June",
       "July", "August", "September", "October", "November", "December" };

class Date{
 friend bool before( Date  & d1, Date & d2 );

public:
 Date( int d=1, int m=1, int y=1970 ){  // do not use initialiser list in constructor
  setMonth( m);                    // as here we want to reuse the checks
  setDay(d);                      // in the set methods
  setYear(y);  // remember  month is  used to verify day validity
 }
 ~Date(){} // no special needs for destructor

 Date( const Date & date ){  // supply an explicit copy constructor
  day = date.day;
  month = date.month;
  year = date.year;
 }

 int getDay(){ return day; }
 void setDay( int d ){
  if( d > 0 && d <= DaysInMonth[month] ){
    day = d;
  }
  else{
    cerr << "Invalid  Day value " << d << endl;
    exit(1);
  }
 }
 int getMonth(){ return month; }
 void setMonth( int m ){
  if( m >= 1 && m <= 12 ){
    month = m;
  }
  else{
    cerr << "Invalid  Month  value " << m << endl;
    exit(1);
  }
 }

 int getYear(){return year;}
 void setYear( int y ){
  // no restrictions placed on year
  year = y;
 }

 void print(){
  cout << day << " of " << MonthNames[month] << ", " << year << endl;
 }

private:
 int day;
 int month;
 int year;
};
```

- **Continued on next slide…**

Friend Functions/Classes

```
bool before( Date  & d1, Date & d2 );  // prototype

int main(){  // test the Calendar collection
 Date today;
 Date lecture11( 6, 8, 2008 );
 today.setMonth( August );
 today.setDay( 7 );
 today.setYear( 2008 );
 today.print();
 lecture11.print();
 if( before( lecture11 , today ) )
    cout << "lecture11 was before today" << endl;
 Date tomorrow( 8, 8, 2008 );
 if( before( tomorrow , today ) )
    cout << "tomorrow was before today" << endl;
 else
    cout << "tomorrow was not before today" << endl;
 return 0;
}

// return  true if   Date1 is before Date2
bool before( Date  & d1, Date & d2 ){
 if( d1.year < d2.year )
     return true;
 else if( d1.year == d2.year ){
     if( d1.month < d2.month )
        return true;
     else if( d1.month == d2.month ){
       if( d1.day < d2.day )
           return true;
     }
 }
 return false;
}
```

- We wanted the global before() function to have access to the internals of the Date class
- Date declares it as a friend function

- Example output:

4 of August, 2022

3 of August, 2022

lecture11 was before today

tomorrow was not before today

Friend Functions/Classes

Thank You.

Friend Functions/Classes

Questions

1. **Is there any difference between `List x;` and `List x();`?**
2. **Is the default constructor for `Fred` always `Fred::Fred()`?**
3. **Should constructors use "initializer lists" or "assignment"?**
4. **How do you know when to declare a function as a member function or a `friend` function in your class?**

Friend Functions/Classes

The following questions pertain to a class called Circle.

a) The only element in a circle considered unique is the radius. Write one line of code that declares the necessary data member.

b) In which section of the class does this data member get put?

c) Write the prototype for the constructor which takes a radius as its parameter.

d) In which section of the class does this constructor get put?

e) Write the function **definition** for the member function Area which computes the area of a Circle object. *(Just use 3.14159 for Pi. And remember that area of a circle of radius R is given by:* πR^2*.)*

Inheritance

Definition

- Inheritance is one of four pillars of Object-Oriented Programming (OOPs).
- It is a feature that enables a class to acquire properties and characteristics of another class.
- Inheritance allows you to reuse your code since the derived class or the child class can reuse the members of the base class by inheriting them.
- Consider a real-life example to clearly understand the concept of inheritance.

  A child inherits some properties from his/her parents, such as the ability to speak, walk, eat, and so on. But these properties are not especially inherited in his parents only. His parents inherit these properties from another class called mammals. This mammal class again derives these characteristics from the animal class. Inheritance works in the same manner.
- During inheritance, the data members of the base class get copied in the derived class and can be accessed depending upon the visibility mode used. The order of the accessibility is always in a decreasing order i.e., from public to protected.

Why and When to Use Inheritance?

Inheritance makes the programming more efficient and is used because of the benefits it provides -

- **Code reusability**: One of the main reasons to use inheritance is that you can reuse the code. For example, consider a group of animals as separate classes - Tiger, Lion, and Panther. For these classes, you can create member functions like the predator() as they all are predators, canine() as they all have canine teeth to hunt, and claws() as all the three animals have big and sharp claws. Now, since all the three functions are the same for these classes, making separate functions for all of them will cause data redundancy and can increase the chances of error. So instead of this, you can use inheritance here. You can create a base class named carnivores and add these functions to it and inherit these functions to the tiger, lion, and panther classes.
- **Transitive nature**: Inheritance is also used because of its transitive nature. For example, you have a derived class mammal that inherits its properties from the base class animal. Now, because of the transitive nature of the inheritance, all the child classes of 'mammal' will inherit the properties of the class 'animal' as well. This helps in debugging to a great extent. You can remove the bugs from your base class and all the inherited classes will automatically get debugged.

What Are Child and Parent classes?

To clearly understand the concept of Inheritance, you must learn about two terms on which the whole concept of inheritance is based - Child class and Parent class.

- Child class: The class that inherits the characteristics of another class is known as the child class or derived class. The number of child classes that can be inherited from a single parent class is based upon the type of inheritance. A child class will access the data members of the parent class according to the visibility mode specified during the declaration of the child class.
- Parent class: The class from which the child class inherits its properties is called the parent class or base class. A single parent class can derive multiple child classes (Hierarchical Inheritance) or multiple parent classes can inherit a single base class (Multiple Inheritance). This depends on the different types of inheritance in C++.

Syntax

The syntax for defining the child class and parent class in all types of Inheritance in C++ is given below:

```
class parent_class
{
   //class definition of the parent class
};
class child_class : visibility_mode parent_class
{
  //class definition of the child class
};
```

Visibility Mode

The visibility mode specifies how the features of the base class will be inherited by the derived class. There are three types of visibility modes for all types of Inheritance in C++:

Public Visibility Mode:

In the public visibility mode, it retains the accessibility of all the members of the base class. The members specified as public, protected, and private in the base class remain public, protected, and private respectively in the derived class as well. So, the public members are accessible by the derived class and all other classes. The protected members are accessible only inside the derived class and its members. However, the private members are not accessible to the derived class.

The following code snippet illustrates how to apply the public visibility mode in a derived class:

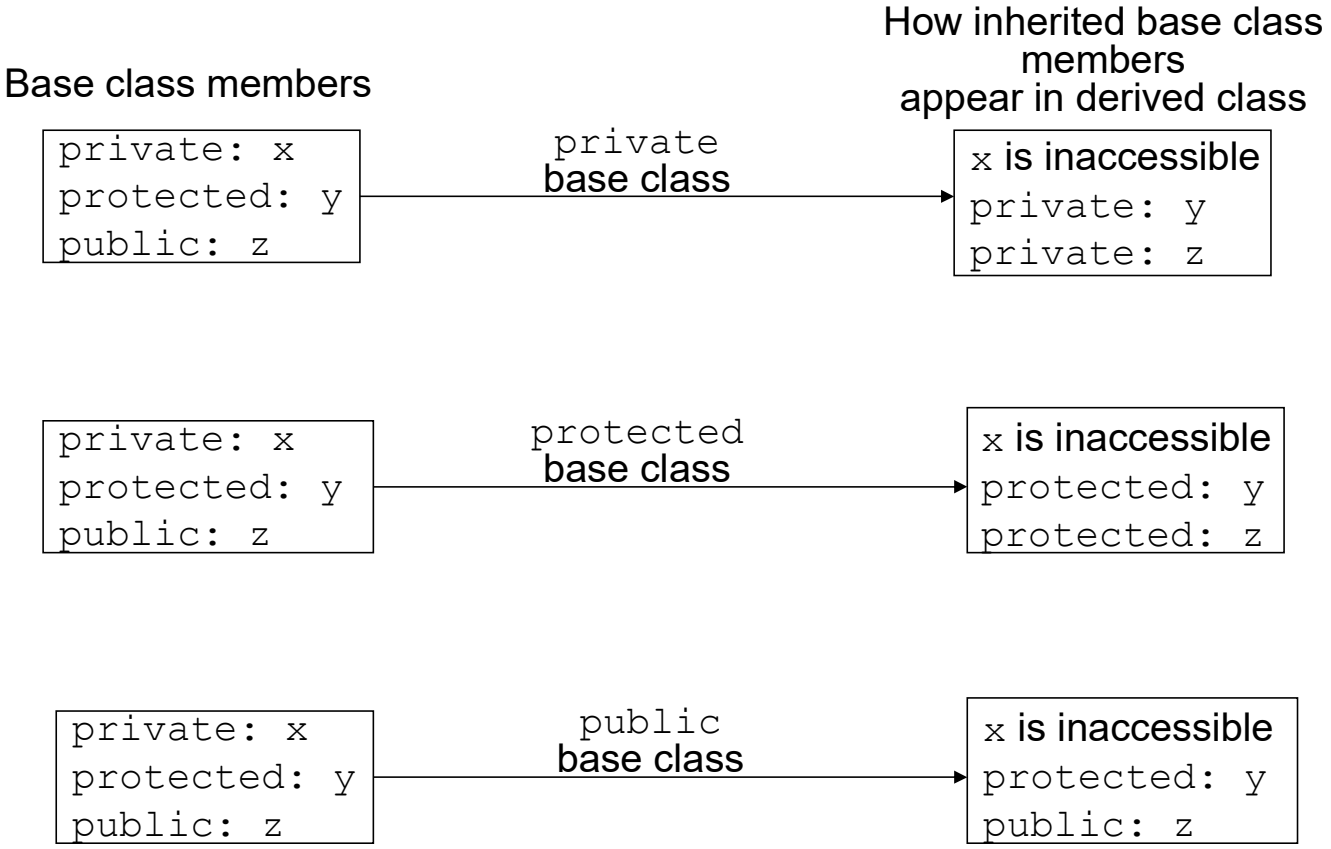
```
class base_class_1
{
   // class definition
};
class derived_class: public base_class_1
{
   // class definition
};
```

Example

The following code displays the working of public visibility mode with all three access specifiers of the base class:

```
class base_class
{
private:
   //class member
   int base_private;
protected:
   //class member
   int base_protected;
public:
   //class member
   int base_public;
};
class derived_class : public base_class
{
private:
   int derived_private;
   // int base_private;
protected:
   int derived_protected;
   // int base_protected;
public:
   int derived_public;
   // int base_public;
};
int main()
{
   // Accessing members of base_class using object of the //derived_class:
   derived_class obj;
   obj.base_private;   // Not accessible
   obj.base_protected; // Not accessible
   obj.base_public;    // Accessible}
```

Inheritance vs. Access



Inheritance vs. Access

class Grade

private members:
```
  char letter;
  float score;
  void calcGrade();
```
public members:
```
  void setScore(float);
  float getScore();
  char getLetter();
```

class Test : public Grade

private members:
```
  int numQuestions;
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `public` class access, it looks like this: →

private members:
```
  int numQuestions:
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
  void setScore(float);
  float getScore();
  char getLetter();
```

Inheritance vs. Access

class Grade

private members:
```
  char letter;
  float score;
  void calcGrade();
```
public members:
```
  void setScore(float);
  float getScore();
  char getLetter();
```

class Test : protected Grade

private members:
```
  int numQuestions;
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `protected` class access, it looks like this: →

private members:
```
  int numQuestions:
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```
protected members:
```
  void setScore(float);
  float getScore();
  float getLetter();
```

Inheritance vs. Access

class Grade

private members:
```
  char letter;
  float score;
  void calcGrade();
```
public members:
```
  void setScore(float);
  float getScore();
  char getLetter();
```

class Test : private Grade

private members:
```
  int numQuestions;
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `private` class access, it looks like this: →

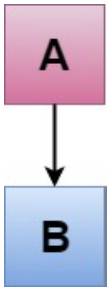
private members:
```
  int numQuestions:
  float pointsEach;
  int numMissed;
  void setScore(float);
  float getScore();
  float getLetter();
```
public members:
```
  Test(int, int);
```

Types of Inheritance

- There are mainly five types of Inheritance in C++
- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Level Inheritance

- **Single inheritance** is defined as the inheritance in which a derived class inherits only one base class.



Where 'A' is the base class, and 'B' is the derived class.

Example 1

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
     Programmer p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

Example2

```
class A  {
   int a = 4;
   int b = 5;
   public:
   int mul() {
      int c = a*b;
      return c;  }
};
 class B : private A  {
   public:
   void display()  {
      int result = mul();
      std::cout <<"Multiplication of a and b is : "<<result<< std::endl;  }
};
int main()  {
  B b;
  b.display();
   return 0;
}
```

Multilevel Inheritance

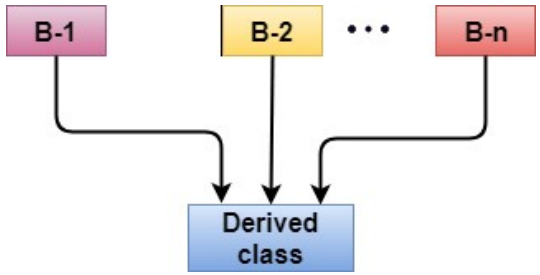
When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Multi level Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Example

```
 class Animal {
  public:
 void eat() {
   cout<<"Eating..."<<endl;   }
  };
  class Dog: public Animal
  {
     public:
    void bark(){
   cout<<"Barking..."<<endl;    }
  };
  class BabyDog: public Dog
  {
     public:
    void weep() {
   cout<<"Weeping...";    }
  };
int main(void) {
   BabyDog d1;
   d1.eat();
   d1.bark();
    d1.weep();   return 0;  }
```

Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more base classes.

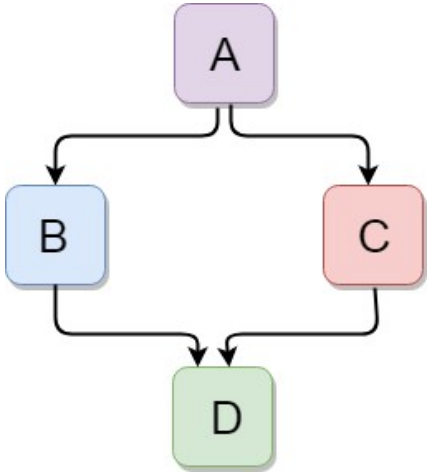


Example

```
#include <iostream>
 class A  {
   protected:
    int a;
   public:
   void get_a(int n)  {
      a = n;  }
};
 class B  {
   protected:
   int b;
   public:
   void get_b(int n)  {
      b = n;  }
};
class C : public A, public B  {
  public:
   void display()  {
        std::cout << "The value of a is : " <<a<< std::endl;
       std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;  }
};
int main()  {
  C c;
  c.get_a(10);
  c.get_b(20);
  c.display();
 return 0;
}
```

Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Example

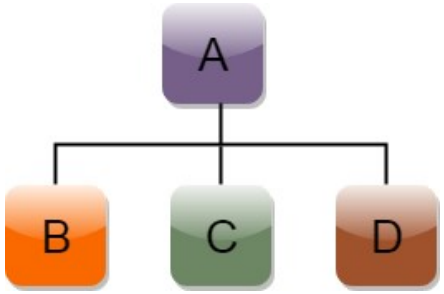
```
class A  {
   protected:
   int a;
   public:
   void get_a()  {
      std::cout << "Enter the value of 'a' : " << std::endl;
      cin>>a;  }
};
  class B : public A   {
   protected:
   int b;
   public:
   void get_b()  {
       std::cout << "Enter the value of 'b' : " << std::endl;
      cin>>b;   }
};
class C
{
   protected:
   int c;
   public:
void get_c()  {
       std::cout << "Enter the value of c is : " << std::endl;
       cin>>c;   }
};
```

Continue..

```
class D : public B, public C
{
   protected:
   int d;
   public:
   void mul()
   {
       get_a();
       get_b();
       get_c();
       std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
   }
};
int main()
{
   D d;
   d.mul();
   return 0;
}
```

Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Example

```
class Shape  {
   public:
   int a;
   int b;
   void get_data(int n, int m)  {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape    {
   public:
   int rect_area()  {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape
{
   public:
   int triangle_area()
   {
      float result = 0.5*a*b;
      return result;   }
};
```

Continue

```
int main()
{
   Rectangle r;
   Triangle t;
   int length, breadth, base, height;
   cout << "Enter the length and breadth of a rectangle: " << endl;
   cin>>length>>breadth;
   r.get_data(length, breadth);
   int m = r.rect_area();
   cout << "Area of the rectangle is : " <<m<< endl;
   cout << "Enter the base and height of the triangle: " << endl;
   cin>>base>>height;
   t.get_data(base, height);
   float n = t.triangle_area();
   cout <<"Area of the triangle is : "  << n<<endl;
   return 0;
}
```

Pointers

Introduction

- A pointer is a variable that represents the location (rather than the value) of a data item.
- They have a number of useful applications.
  - Enables us to access a variable that is defined outside the function.
  - Can be used to pass information back and forth between a function and its reference point.
  - More efficient in handling data tables.
  - Reduces the length and complexity of a program.
  - Sometimes also increases the execution speed.

Basic Concept

- Within the computer memory, every stored data item occupies one or more contiguous memory cells.
  - The number of memory cells required to store a data item depends on its type (char, int, double, etc.).
- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.
  - Since every byte in memory has a unique address, this location will also have its own (unique) address.

Contd.

- Consider the statement

  int xyz = 50;

  - This statement instructs the compiler to allocate a location for the integer variable xyz, and put the value 50 in that location.
  - Suppose that the address location chosen is 1380.

xyz	→	variable
50	→	value
1380	→	address

Contd.

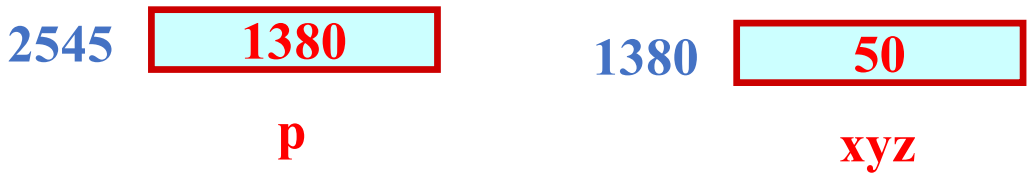
- During execution of the program, the system always associates the name xyz with the address 1380.
  - The value 50 can be accessed by using either the name xyz or the address 1380.
- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.
  - Such variables that hold memory addresses are called pointers.
  - Since a pointer is a variable, its value is also stored in some memory location.

Contd.

- Suppose we assign the address of xyz to a variable p.
  - p is said to point to the variable xyz.

Variable	Value	Address
xyz	50	1380
p	1380	2545

p = &xyz;



Accessing the Address of a Variable

- The address of a variable can be determined using the '&' operator.
  - The operator '&' immediately preceding a variable returns the address of the variable.
- Example:

  p = &xyz;
  - The address of xyz (1380) is assigned to p.
- The '&' operator can be used only with a simple variable or an array element.

  &distance

  &x[0]

  &x[i-2]

Contd.

- Following usages are illegal:

  &235

  - Pointing at constant.

  ```
  int   arr[20];
    :
  &arr;
  ```

  - Pointing at array name.

  &(a+b)

  - Pointing at expression.

Example

```
#include  <stdio.h>
main()
{
   int   a;
   float  b, c;
   double  d;
   char  ch;

   a = 10;   b = 2.5;  c = 12.36;  d = 12345.66;  ch = 'A';
   printf  ("%d is stored in location %u \n",  a,  &a) ;
   printf  ("%f is stored in location %u \n",  b,  &b) ;
   printf  ("%f is stored in location %u \n",  c,  &c) ;
   printf  ("%ld is stored in location %u \n",  d,  &d) ;
   printf  ("%c is stored in location %u \n",  ch,  &ch) ;
}
```

Output:

10 is stored in location 3221224908 a
2.500000 is stored in location 3221224904 b
12.360000 is stored in location 3221224900 c
12345.660000 is stored in location 3221224892 d
A is stored in location 3221224891 ch

Incidentally variables a,b,c,d and ch are allocated to contiguous memory locations.

Pointer Declarations

- Pointer variables must be declared before we use them.
- General form:

```
data_type   *pointer_name;
```

Three things are specified in the above declaration:

1. The asterisk (*) tells that the variable pointer_name is a pointer variable.
2. pointer_name needs a memory location.
3. pointer_name points to a variable of type data_type.

Contd.

- Example:

```
int    *count;
float  *speed;
```

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like:

```
int     *p,  xyz;
:
p = &xyz;
```

  - This is called pointer initialization.

Things to Remember

- Pointer variables must always point to a data item of the *same type*.

```
float   x;
int    *p;
:                    ➔ will result in erroneous output
p = &x;
```

- Assigning an absolute address to a pointer variable is prohibited.

```
int   *count;
:
count = 1268;
```

Accessing a Variable Through its Pointer

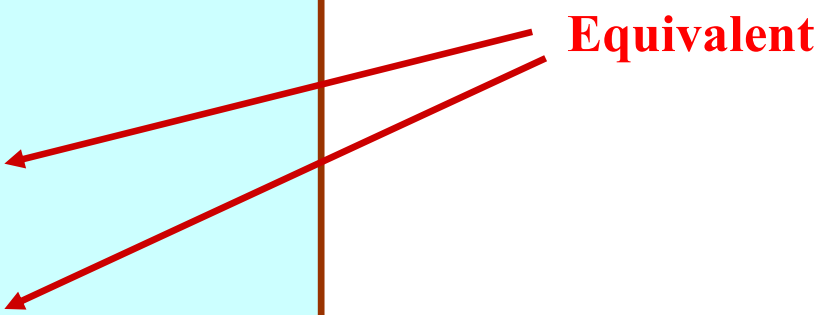
- Once a pointer has been assigned the address of a variable, the value of the variable can be accessed using the indirection operator (*).

```
int   a, b;
int   *p;
:
p = &a;
b = *p;
```

b = a

Example 1

```
#include <stdio.h>
main()
{
   int  a, b;
   int  c = 5;
   int  *p;

   a = 4 * (c + 5) ;

   p = &c;
   b = 4 * (*p + 5) ;
   printf ("a=%d  b=%d \n",  a, b) ;
}
```



Example 2

```
#include  <stdio.h>
main()
{
   int  x, y;
   int  *ptr;

   x = 10 ;
   ptr = &x ;
   y = *ptr ;
   printf  ("%d is stored in location %u \n",  x,  &x) ;
   printf  ("%d is stored in location %u \n",  *&x,  &x) ;
   printf  ("%d is stored in location %u \n",  *ptr,  ptr) ;
   printf  ("%d is stored in location %u \n",  y,  &*ptr) ;
   printf  ("%u is stored in location %u \n",  ptr, &ptr) ;
   printf  ("%d is stored in location %u \n",  y,  &y) ;

  *ptr = 25;
   printf  ("\nNow x = %d \n", x);
}
```

*&x⇔x

ptr=&x;
&x⇔&*ptr

Output:

```
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
3221224908 is stored in location 3221224900
10 is stored in location 3221224904

Now x = 25
```

```
Address of x:   3221224908
Address of y:   3221224904
Address of ptr: 3221224900
```

Pointer Expressions

- Like other variables, pointer variables can be used in expressions.
- If p1 and p2 are two pointers, the following statements are valid:

```
sum  = *p1 + *p2 ;
prod = *p1 * *p2 ;
prod =  (*p1) * (*p2) ;
*p1 = *p1 + 2;
x = *p1 / *p2 + 5 ;
```

Contd.

- What are allowed in C?
  - Add an integer to a pointer.
  - Subtract an integer from a pointer.
  - Subtract one pointer from another (related).
    - If p1 and p2 are both pointers to the same array, them p2–p1 gives the number of elements between p1 and p2.
- What are not allowed?
  - Add two pointers.

```
p1 = p1 + p2 ;
```

  - Multiply / divide a pointer in an expression.

```
p1 = p2 / 5 ;
p1 = p1 – p2 * 10 ;
```

Scale Factor

- We have seen that an integer value can be added to or subtracted from a pointer variable.

```
int   *p1, *p2 ;
int   i, j;
:
p1 = p1 + 1 ;
p2 = p1 + j ;
p2++ ;
p2 = p2 – (i + j) ;
```

- In reality, it is not the integer value which is added/subtracted, but rather the scale factor times the value.

Contd.

Data Type	Scale Factor
char	1
int	4
float	4
double	8

- If p1 is an integer pointer, then

  p1++

  will increment the value of p1 by 4.

E

Returns no. of bytes required for data type representation

```
#include  <stdio.h>
main()
{
   printf ("Number of bytes occupied by int is %d \n", sizeof(int));
   printf ("Number of bytes occupied by float is %d \n", sizeof(float));
   printf ("Number of bytes occupied by double is %d \n", sizeof(double));
   printf ("Number of bytes occupied by char is %d \n", sizeof(char));
}
```

Output:

```
Number of bytes occupied by int is  4
Number of bytes occupied by float is  4
Number of bytes occupied by double is  8
Number of bytes occupied by char is  1
```

Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
  - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
  - Called call-by-reference (or by address or by location).
- Normally, arguments are passed to a function by value.
  - The data items are copied to the function.
  - Changes are not reflected in the calling program.

Example: passing arguments by value

```
#include <stdio.h>
main()
{
    int a, b;
    a = 5 ;   b = 20 ;
    swap (a, b) ;
    printf ("\n a = %d,  b = %d", a, b);
}

void  swap (int x, int y)
{
    int t ;
    t = x ;
    x = y ;
    y = t ;
}
```

a and b do not swap

x and y swap

Output

a = 5, b = 20

Example: passing arguments by reference

```
#include <stdio.h>
main()
{
    int a, b;
    a = 5 ;   b = 20 ;
    swap (&a, &b) ;
    printf ("\n a = %d,  b = %d", a, b);
}

void  swap  (int  *x, int  *y)
{
    int  t ;
    t = *x ;
    *x = *y ;
    *y = t ;
}
```

*(&a) and *(&b) swap

*x and *y swap

Output

a = 20, b = 5

scanf Revisited

```
int   x,  y ;
printf  (“%d %d %d”,  x, y, x+y) ;
```

- What about scanf ?

```
scanf   (“%d %d %d”, x, y, x+y) ;
```

NO

```
scanf   (“%d %d”, &x, &y) ;
```

YES

Example: Sort 3 integers

- Three-step algorithm:
  1. Read in three integers x, y and z
  2. Put smallest in x
     - Swap x, y if necessary; then swap x, z if necessary.
  3. Put second smallest in y
     - Swap y, z if necessary.

Contd.

```
#include  <stdio.h>
main()
{
     int  x, y, z ;
     ……..
     scanf  (“%d %d %d”, &x, &y, &z) ;
     if  (x > y)   swap (&x, &y);
     if  (x > z)   swap (&x, &z);
     if  (y > z)   swap (&y, &z) ;
     ……..
}
```

sort3 as a function

```
#include  <stdio.h>
main()
{
     int  x, y, z ;
     ……..
     scanf  (“%d %d %d”, &x, &y, &z) ;
     sort3  (&x, &y, &z) ;
     ……..
}

void   sort3  (int *xp,  int *yp,  int *zp)
{
     if  (*xp > *yp)   swap (xp, yp);
     if  (*xp > *zp)   swap (xp, zp);
     if  (*yp > *zp)   swap (yp, zp);
}
```

xp/yp/zp are pointers

Contd.

- Why no ‘&’ in swap call?
  - Because xp, yp and zp are already pointers that point to the variables that we want to swap.

Pointers and Arrays

- When an array is declared,
  - The compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
  - The base address is the location of the first element (index 0) of the array.
  - The compiler also defines the array name as a constant pointer to the first element.

Example

- Consider the declaration:

  int x[5] = {1, 2, 3, 4, 5} ;

  - Suppose that the base address of x is 2500, and each integer requires 4 bytes.

Element	Value	Address
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

Contd.

x ⇔ &x[0] ⇔ 2500 ;

- p = x; and p = &x[0]; are equivalent.
- We can access successive values of x by using p++ or p- - to move from one element to another.

- Relationship between p and x:

p = &x[0] = 2500
p+1 = &x[1] = 2504
p+2 = &x[2] = 2508
p+3 = &x[3] = 2512
p+4 = &x[4] = 2516

***(p+i) gives the value of x[i]**

Example: function to find average

```
#include <stdio.h>
main()
{
    int x[100], k, n ;

    scanf ("%d", &n) ;

    for (k=0; k<n; k++)
       scanf ("%d", &x[k]) ;

    printf ("\nAverage is %f",
                          avg (x, n));
}
```

int *array

```
float avg (int array[ ],int size)
{
    int *p, i , sum = 0;

    p = array ;

    for (i=0; i<size; i++)
        sum = sum + *(p+i);

    return ((float) sum / size);
}
```

p[i]

Structures Revisited

- Recall that a structure can be declared as:

```
struct  stud  {
            int   roll;
            char  dept_code[25];
            float  cgpa;
          };
struct  stud  a, b, c;
```

- And the individual structure elements can be accessed as:

  a.roll , b.roll , c.cgpa , etc.

Arrays of Structures

- We can define an array of structure records as

  struct stud class[100] ;

- The structure elements of the individual records can be accessed as:

  class[i].roll

  class[20].dept_code

  class[k++].cgpa

Example: Sorting by Roll Numbers

```
#include <stdio.h>
struct   stud
{
    int    roll;
    char  dept_code[25];
    float  cgpa;
 };

main()
{
    struc  stud  class[100], t;
    int  j, k, n;

    scanf  ("%d", &n);
                /* no. of students */
```

```
for  (k=0; k<n; k++)
      scanf ("%d %s %f", &class[k].roll,
           class[k].dept_code, &class[k].cgpa);
   for  (j=0; j<n-1; j++)
      for  (k=j+1; k<n; k++)
        {
          if  (class[j].roll > class[k].roll)
             {
                t = class[j] ;
                class[j] = class[k] ;
                class[k] = t
             }
        }
   <<<< PRINT THE RECORDS >>>>
}
```

Pointers and Structures

- You may recall that the name of an array stands for the address of its zero-th element.
  - Also true for the names of arrays of structure variables.
- Consider the declaration:

```
struct  stud  {
            int   roll;
            char  dept_code[25];
            float  cgpa;
          }  class[100],  *ptr ;
```

- The name class represents the address of the zero-th element of the structure array.
- ptr is a pointer to data objects of the type struct stud.

- The assignment

```
ptr = class ;
```

will assign the address of class[0] to ptr.

- When the pointer ptr is incremented by one (ptr++) :
  - The value of ptr is actually increased by sizeof(stud).
  - It is made to point to the next record.

- Once ptr points to a structure variable, the members can be accessed as:

```
ptr –> roll ;
ptr –> dept_code ;
ptr –> cgpa ;
```

  - The symbol "–>" is called the arrow operator.

Example

```
#include <stdio.h>

typedef struct {
        float real;
        float imag;
       } _COMPLEX;
```

```
swap_ref(_COMPLEX *a, _COMPLEX *b)
{
  _COMPLEX tmp;
   tmp=*a;
   *a=*b;
   *b=tmp;
}
```

```
print(_COMPLEX *a)
{
 printf("(%f,%f)\n",a->real,a->imag);
}
```

```
(10.000000,3.000000)
(-20.000000,4.000000)
(-20.000000,4.000000)
(10.000000,3.000000)
```

```
main()
{
 _COMPLEX x={10.0,3.0}, y={-20.0,4.0};

  print(&x); print(&y);
  swap_ref(&x,&y);
  print(&x); print(&y);
}
```

A Warning

- When using structure pointers, we should take care of operator precedence.
  - Member operator "." has higher precedence than "*".
    - ptr –> roll and (*ptr).roll mean the same thing.
    - *ptr.roll will lead to error.
  - The operator "–>" enjoys the highest priority among operators.
    - ++ptr –> roll will increment roll, not ptr.
    - (++ptr) –> roll will do the intended thing.

Structures and Functions

- A structure can be passed as argument to a function.
- A function can also return a structure.
- The process shall be illustrated with the help of an example.
  - A function to add two complex numbers.

Example: complex number addition

```
#include <stdio.h>
struct complex {
                    float re;
                    float im;
                  };

main()
{
   struct complex a, b, c;
   scanf ("%f %f", &a.re, &a.im);
   scanf ("%f %f", &b.re, &b.im);
   c = add (a, b) ;
   printf ("\n %f %f", c.re, c.im);
}
```

```
struct complex add (x, y)
struct complex x, y;
{
    struct complex t;

    t.re = x.re + y.re ;
    t.im = x.im + y.im ;
    return (t) ;
}
```

Example: Alternative way using pointers

```
#include <stdio.h>
struct complex {
                    float re;
                    float im;
                  };

main()
{
   struct complex a, b, c;
   scanf ("%f %f", &a.re, &a.im);
   scanf ("%f %f", &b.re, &b.im);
   add (&a, &b, &c) ;
   printf ("\n %f %f", c,re, c.im);
}
```

```
void add (x, y, t)
struct complex *x, *y, *t;
{
    t->re = x->re + y->re ;
    t->im = x->im + y->im ;
}
```

Dynamic Memory Allocation

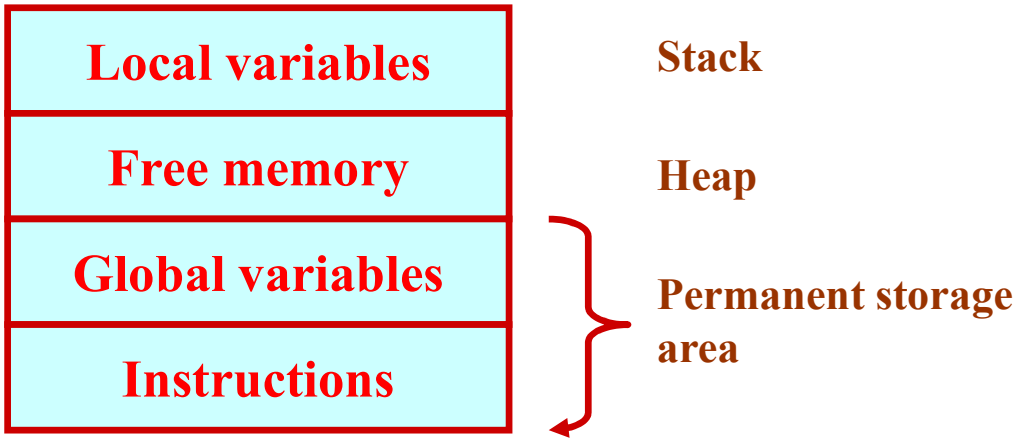
Basic Idea

- Many a time we face situations where data is dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data item keeps changing during program execution.
- Such situations can be handled more easily and effectively using dynamic memory management techniques.

Contd.

- C language requires the number of elements in an array to be specified at compile time.
  - Often leads to wastage or memory space or program failure.
- Dynamic Memory Allocation
  - Memory space required can be specified at the time of execution.
  - C supports allocating and freeing memory dynamically using library routines.

Memory Allocation Process in C



Contd.

- The program instructions and the global variables are stored in a region known as permanent storage area.
- The local variables are stored in another area called stack.
- The memory space between these two areas is available for dynamic allocation during execution of the program.
  - This free region is called the heap.
  - The size of the heap keeps changing

Memory Allocation Functions

- malloc
  - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
- calloc
  - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- free

  Frees previously allocated space.
- realloc
  - Modifies the size of previously allocated space.

Allocating a Block of Memory

- A block of memory can be allocated using the function malloc.
  - Reserves a block of memory of specified size and returns a pointer of type void.
  - The return pointer can be assigned to any pointer type.
- General format:

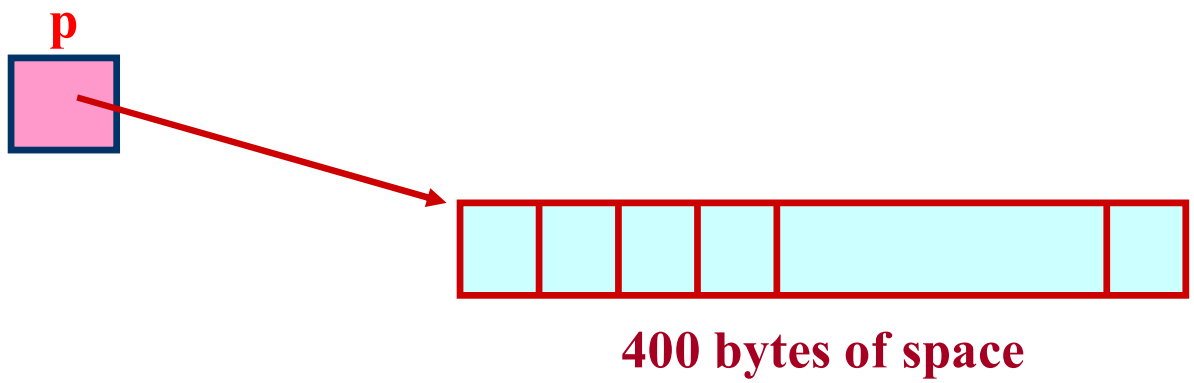
```
ptr  =  (type *)  malloc (byte_size) ;
```

Contd.

- Examples

  p = (int *) malloc (100 * sizeof (int)) ;

  - A memory space equivalent to “100 times the size of an int” bytes is reserved.
  - The address of the first byte of the allocated memory is assigned to the pointer p of type int.



Contd.

```
cptr  =  (char *)  malloc (20) ;
```

- Allocates 10 bytes of space for the pointer cptr of type char.

```
sptr  =  (struct stud *)  malloc (10 *
                                  sizeof (struct stud));
```

Points to Note

- malloc always allocates a block of contiguous bytes.
  - The allocation can fail if sufficient contiguous memory space is not available.
  - If it fails, malloc returns NULL.

Example

```
#include <stdio.h>

main()
{
 int i,N;
 float *height;
 float sum=0,avg

 printf("Input the number of students. \n");
 scanf("%d",&N);

 height=(float *) malloc(N * sizeof(float));
```

```
Input the number of students.
5
Input heights for 5 students
23 24 25 26 27
Average height= 25.000000
```

```
printf("Input heights for %d
udents \n",N);
for(i=0;i<N;i++)
 scanf("%f",&height[i]);

for(i=0;i<N;i++)
 sum+=height[i];

 avg=sum/(float) N;

 printf("Average height= %f \n",
avg);
}
```

Releasing the Used Space

- When we no longer need the data stored in a block of memory, we may release the block for future use.
- How?
  - By using the free function.
- General format:

```
free (ptr) ;
```

where ptr is a pointer to a memory block which has been already created using malloc.

Altering the Size of a Block

- Sometimes we need to alter the size of some previously allocated memory block.
  - More memory needed.
  - Memory allocated is larger than necessary.
- How?
  - By using the realloc function.
- If the original allocation is done by the statement

```
ptr = malloc (size) ;
```

then reallocation of space may be done as

```
ptr = realloc (ptr, newsize) ;
```

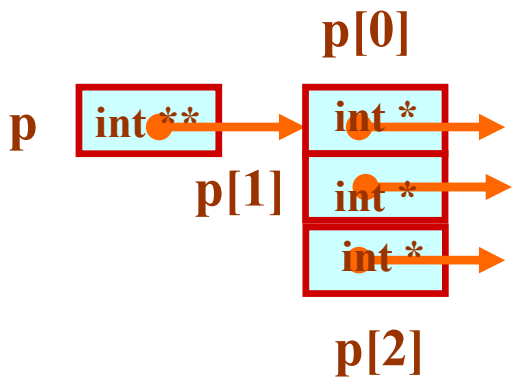
Contd.

- The new memory block may or may not begin at the same place as the old one.
  - If it does not find space, it will create it in an entirely different region and move the contents of the old block into the new block.
- The function guarantees that the old data remains intact.
- If it is unable to allocate, it returns NULL and frees the original block.

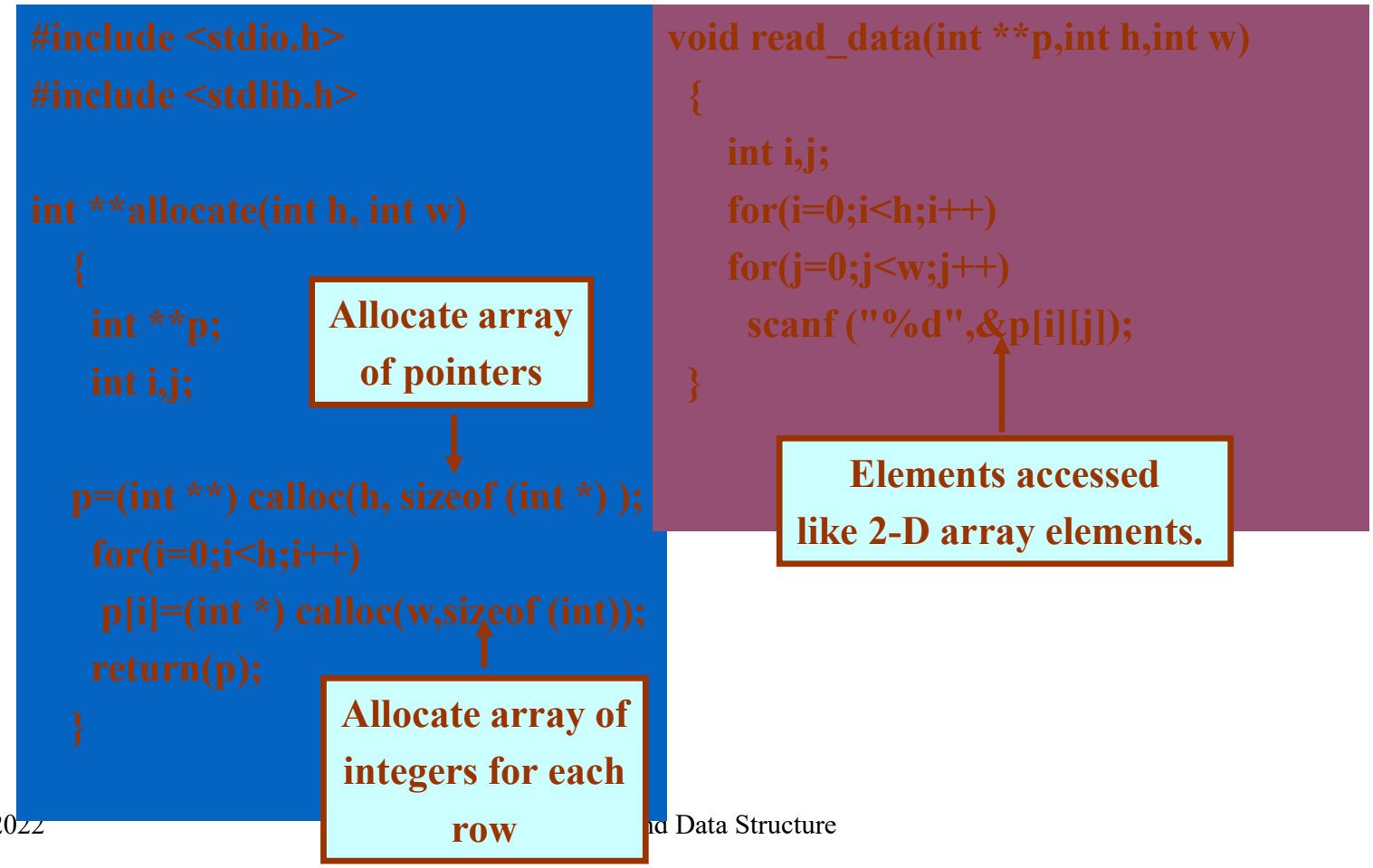
Pointer to Pointer

- Example:

```
int **p;
p=(int **) malloc(3 * sizeof(int *));
```



2-D Array Allocation



2-D Array: Contd.

```
void print_data(int **p,int h,int w)
 {
   int i,j;
    for(i=0;i<h;i++)
    {
    for(j=0;j<w;j++)
     printf("%5d ",p[i][j
    printf("\n");
    }
}
```

```
Give M and N
3 3
1 2 3
4 5 6
7 8 9

 The array read as
   1    2    3
   4    5    6
   7    8    9
```

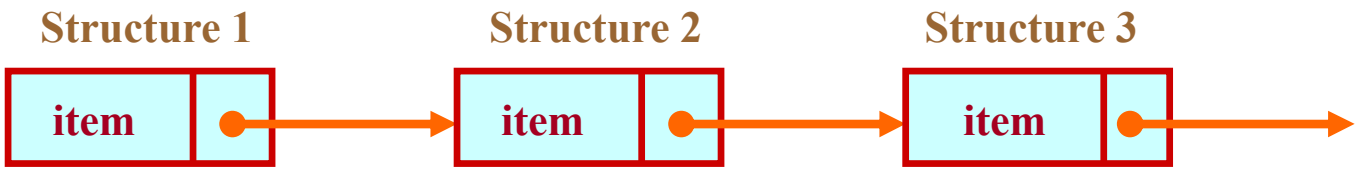
```
main()
{
 int **p;
 int M,N;

printf("Give M and N \n");
scanf("%d%d",&M,&N);
p=allocate(M,N);
read_data(p,M,N);
printf("\n The array read as \n");
print_data(p,M,N);
```

Linked List :: Basic Concepts

- A list refers to a set of items organized sequentially.
  - An array is an example of a list.
    - The array index is used for accessing and manipulation of array elements.
  - Problems with array:
    - The array size has to be specified at the beginning.
    - Deleting an element or inserting an element may require shifting of elements.

Contd.

- A completely different way to represent a list:
  - Make each item in the list part of a structure.
  - The structure also contains a pointer or link to the structure containing the next item.
  - This type of list is called a linked list.



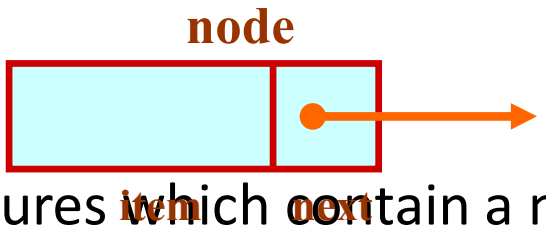
Contd.

- Each structure of the list is called a node, and consists of two fields:
  - One containing the item.
  - The other containing the address of the next item in the list.
- The data items comprising a linked list need not be contiguous in memory.
  - They are ordered by logical links that are stored as part of the data in the structure itself.
  - The link is a pointer to another structure of the same type.

Contd.

- Such a structure can be represented as:

```
struct  node
 {
    int    item;
    struct node  *next;
 };
```



- Such structures which contain a member field pointing to the same structure type are called self-referential structures.

Contd.

- In general, a node may be represented as follows:

```
struct  node_name
 {
    type  member1;
    type member2;
     ……
    struct  node_name  *next;
 };
```

Illustration

- Consider the structure:

```
struct  stud
 {
    int  roll;
    char  name[30];
    int  age;
    struct  stud  *next;
 };
```

- Also assume that the list consists of three nodes n1, n2 and n3.

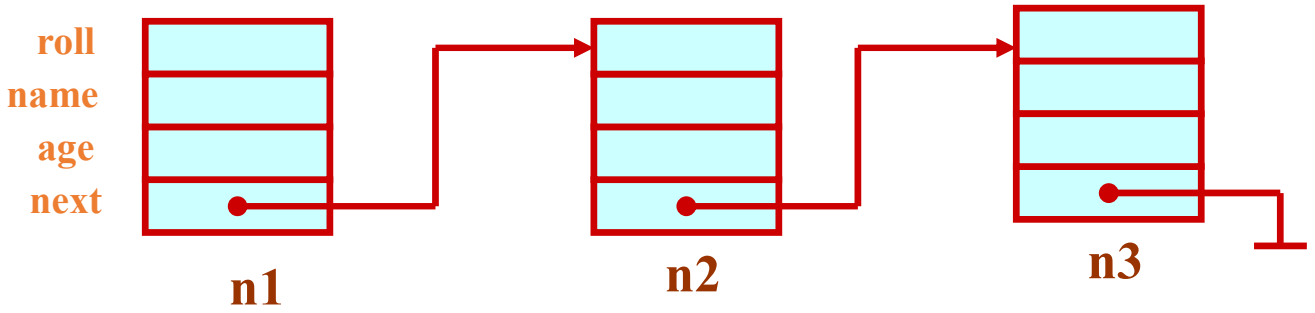
```
struct  stud  n1, n2, n3;
```

Contd.

- To create the links between nodes, we can write:

```
n1.next  =  &n2 ;
n2.next  =  &n3 ;
n3.next  =  NULL ;   /* No more nodes follow */
```

- Now the list looks like:



Example

```
#include <stdio.h>
struct stud
 {
    int roll;
    char name[30];
    int age;
    struct stud *next;
 };

main()
{
   struct stud n1, n2, n3;
   struct stud *p;

   scanf ("%d %s %d", &n1.roll,
                      n1.name, &n1.age);
   scanf ("%d %s %d", &n2.roll,
                      n2.name, &n2.age);
   scanf ("%d %s %d", &n3.roll,
                      n3.name, &n3.age);
```

```
   n1.next = &n2 ;
   n2.next = &n3 ;
   n3.next = NULL ;

/* Now traverse the list and print
   the elements */

   p = n1 ;  /* point to 1st element */
   while (p != NULL)
   {
       printf ("\n %d %s %d",
          p->roll, p->name, p->age);
          p = p->next;
   }
}
```

DIY

1. **Write a program using pointers to compute the sum of all elements stored in an array.**
2. **Write a program using pointers to determine the length of a character string.**

Pointer to Array

Program

```
main()
{
      int *p, sum, i;
      int x[5] = {5,9,6,3,7};
      i  = 0;
      p  = x;      /* initializing with base address of x */
      printf("Element   Value   Address\n\n");
      while(i < 5)
{
      printf(" x[%d] %d %u\n", i, *p, p);
      sum = sum + *p;   /* accessing array element  */
      i++, p++;         /* incrementing pointer     */
}
printf("\n  Sum    =  %d\n", sum);
printf("\n  &x[0]  =  %u\n", &x[0]);
printf("\n  p      =  %u\n", p);
}
```

Output

```
Element      Value      Address
   x[0]        5          166
   x[1]        9          168
   x[2]        6          170
   x[3]        3          172
   x[4]        7          174
   Sum     =  55
   &x[0]   =  166
   p       =  176
```

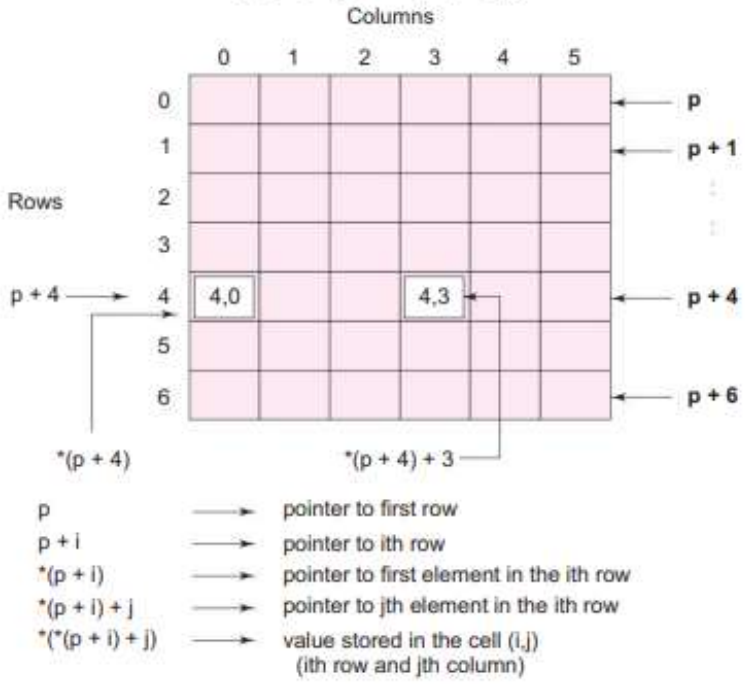
Pointer to 2D-Array

Pointers can be used to manipulate two-dimensional arrays as well. We know that in a one-dimensional array **x**, the expression

```
*(x+i) or *(p+i)
```

represents the element **x[i]**. Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

```
*(*(a+i)+j) or *(*(p+i)j)
```



String Pointer

Program

```
main()
{
    char  *name;
    int   length;
    char  *cptr = name;
    name  = "DELHI";
    printf ("%s\n", name);
    while(*cptr != '\0')
    {
        printf("%c is stored at address %u\n", *cptr, cptr);
        cptr++;
    }
    length = cptr - name;
    printf("\nLength of the string = %d\n", length);
}
```

Array of Pointers

One important use of pointers is in handling of a table of strings. Consider the following array of strings:

```
char name [3][25];
```

This says that the **name** is a table containing three names, each with a maximum length of 25 characters (including null character). The total storage requirements for the **name** table are 75 bytes.

We know that rarely the individual strings will be of equal lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length. For example,

```
char *name[3] = {
                  "New Zealand",
                  Australia",
                  "India"
                };
```

declares **name** to be an *array of three pointers* to characters, each pointer pointing to a particular name as:

Cont..

name [0] ⟶ New Zealand
name [1] ⟶ Australia
name [2] ⟶ India

This declaration allocates only 28 bytes, sufficient to hold all the characters as shown

N	e	w		Z	e	a	l	a	n	d	\0
A	u	s	t	r	a	l	i	a	\0		
I	n	d	i	a	\0						

The following statement would print out all the three names:

```
for(i = 0; i <= 2; i++)
  printf("%s\n", name[i]);
```

To access the jth character in the ith name, we may write as

```
*(name[i]+j)
```

The character arrays with the rows of varying length are called 'ragged arrays' and are better handled by pointers.

Remember the difference between the notations ***p[3]** and **(*p)[3]**. Since * has a lower precedence than [ ], *p[3] declares p as an array of 3 pointers while (***p**)[3] declares **p** as a pointer to an array of three elements.

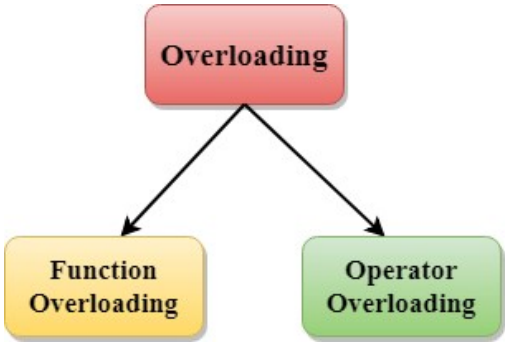
C++ Overloading (Function and Operator)

Definition

- If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:
- methods
- constructors
- indexed properties
- It is because these members have parameters only.

Types of overloading in C++

- Function overloading
- Operator overloading



C++ Function Overloading

- Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++.
- In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.
- The **advantage** of Function overloading is that it increases the readability of the program because we don't need to use different names for the same action.

Example 1

```
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
int main(void) {
   Cal C;                              //   class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Example 2

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   cout << "r1 is : " <<r1<< endl;
   cout <<"r2 is : "  <<r2<< endl;
   return 0;
}
```

Function Overloading and Ambiguity

- When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.
- When the compiler shows the ambiguity error, the compiler does not run the program.
- **Causes of Function Overloading:**
- Type Conversion.
- Function with default arguments.
- Function with pass by reference.

Type Conversion

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   cout << "Value of i is : " <<i<< endl;
}
void fun(float j)
{
   cout << "Value of j is : " <<j<< endl;
}
int main()
{
   fun(10);
   fun(1.2);
   return 0;
}
```

- The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

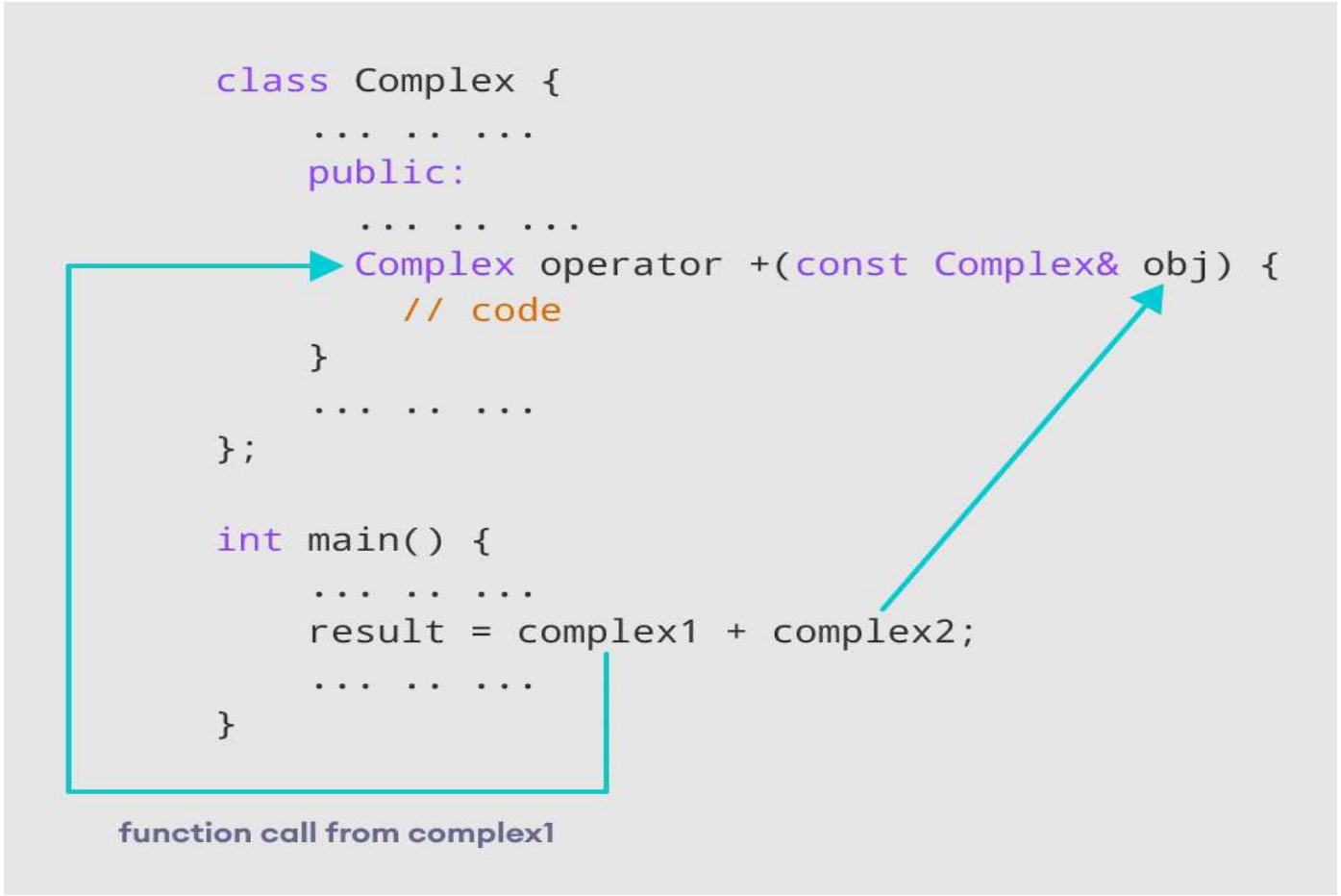
Operator Overloading

- In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.
- Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
  Example:
- ```
  int a;
  float b, sum;
  sum=a+b;
  ```
- Here, variables "a" and "b" are of types "int" and "float", which are built-in data types. Hence the addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.
  ```

Now, consider another example

```
class A
   {

    };

int main()
   {
       A   a1,a2,a3;

        a3= a1 + a2;

        return 0;
   }
```

- In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type i.e. of type “class A” using the “+” operator. This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.

- In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**. For example,
- Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers.
- Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

  result = c1 + c2;

- instead of something like:

  result = c1.addNumbers(c2);

- **Syntax for C++ Operator Overloading**

  To overload an operator, we use a special operator function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

  returnType operator symbol (arguments) { }

Operator Overloading in Unary Operators

- Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

```
class Count {
private:
int value;
public:
// Constructor to initialize count to 5
Count() : value(5) {}
// Overload ++ when used as prefix
void operator ++ () {
++value; }
void display() {
cout << "Count: " << value << endl;
} };
int main() {
Count count1;
 // Call the "void operator ++ ()" function
++count1;
count1.display();
return 0; }
```

Can we overload all operators?

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

- sizeof
- typeid
- Scope resolution (::)
- Class member access operators (.(dot), .* (pointer to member operator))
- Ternary or conditional (?:)

Binary operator Overloading

```
#include <iostream>
using namespace std;
class Complex_num
{
   int x, y;
   public:
      void input()    {
         cout << " Input two complex number: " << endl;
         cin >> x >> y;
      }
      // use binary '+' operator to overload
      Complex_num operator + (Complex_num  &obj)
      {
         // create an object
         Complex_num A;
         // assign values to object
         A.x = x + obj.x;
         A.y = y + obj.y;
         return (A);
      }
      // overload the binary (-) operator
Complex_num operator - (Complex_num  &obj)
      {
         Complex_num A;
         // assign values to object
         A.x = x - obj.x;
         A.y = y - obj.y;
         return (A);
      }
```

```
 // display the result of addition
      void print1()  {
         cout << x << " + " << y << "i" << "\n";  }
// display the result of subtraction
      void print2()  {
         cout << x << " - " << y << "i" << "\n";  }
};
int main ()  {
Complex_num x1, y1, sum, sub;                  // here we created object of class Addition i.e x1 and y1  accepting the values
   x1.input();
   y1.input();
   sum = x1 + y1;                                  // add the objects
   sub = x1 - y1;                                 // subtract the complex number
   // display user entered values
   cout << "\n Entered values are: \n";
   cout << " \t";
   x1.print1();
   cout << " \t";
   y1.print1();
   cout << "\n The addition of two complex (real and imaginary) numbers: ";
   sum.print1(); // call print function to display the result of addition
   cout << "\n The subtraction of two complex (real and imaginary) numbers: ";
   sub.print2(); // call print2 function to display the result of subtraction
   return 0;  }
```

Addition of two Complex Numbers

```
#include <iostream>
using namespace std;
class Complex {
private:
 float real; float imag;
 public: // Constructor to initialize real and imag to 0
Complex() : real(0), imag(0) {}
void input() {
 cout << "Enter real and imaginary parts respectively: ";
 cin >> real;
cin >> imag; }
// Overload the + operator
Complex operator + (Const Complex  &obj) {
Complex temp;
temp.real = real + obj.real;
temp.imag = imag + obj.imag;
 return temp; }
void output() {
if (imag < 0)
 cout << "Output Complex number: " << real << imag << "i";
else
cout << "Output Complex number: " << real << "+" << imag << "i";
 }
};
 int main() {
 Complex complex1, complex2, result;
cout << "Enter first complex number:\n";
complex1.input();
cout << "Enter second complex number:\n";
complex2.input();
 // complex1 calls the operator function // complex2 is passed as an argument to the function
 result = complex1 + complex2;
 result.output();
return 0;
 }
```


class Complex {
... .
public:
... .
Complex operator +(const Complex& obj) {
// code
}
... .
};
int main() {
... .
result = complex1 + complex2;
... .
}
function call from complex1

Access Functions and Friend Functions

Access functions

- To allow clients to read the value of *private* data, the class can provide a *get* function.
- To allow clients to modify *private* data, the class can provide a *set* function.

```
#include <iostream.h>

class rectangle {
 private:
   float height;
   float width;
   int xpos;
   int ypos;
```

```
// access function
// access function
```

```
float rectangle::get_height()
{
 return (height);
}

void rectangle::set_height(float h)
{
 height = h;
}

void main()
{
 rectangle rc(1.0, 3.0);
 float value;

 value = rc.get_height();
 cout << "height: " << value << endl;

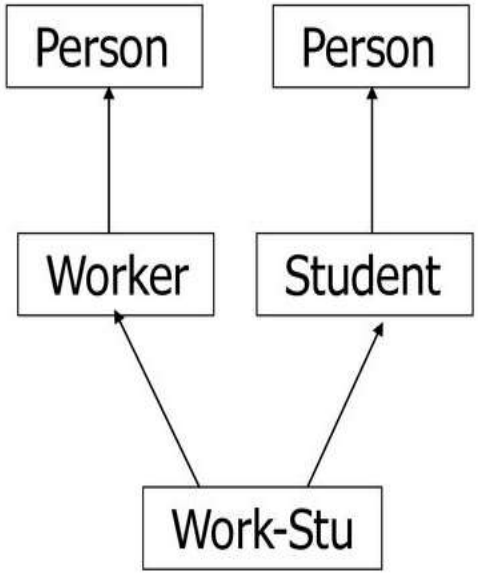
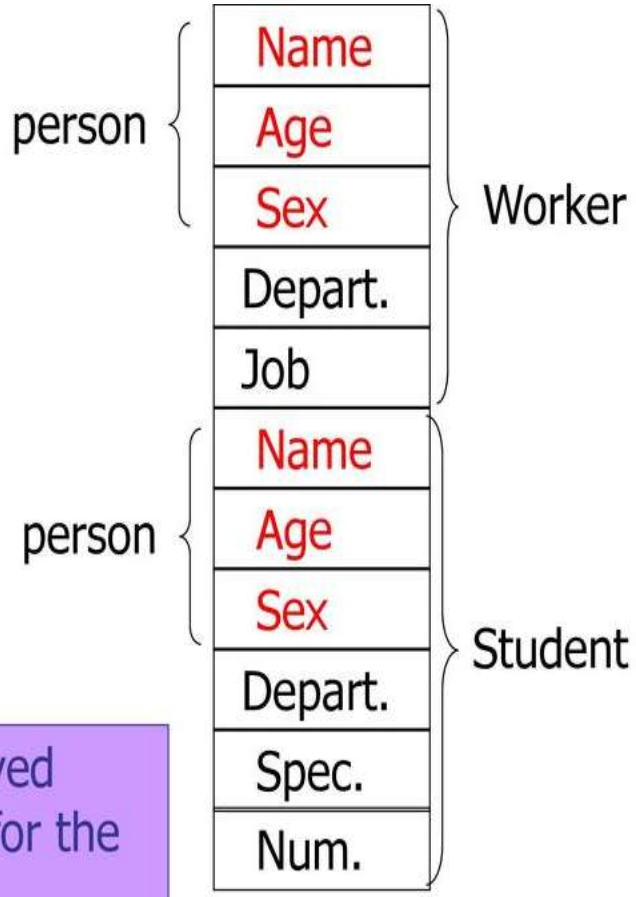
 rc.set_height(10.0);
 value = rc.get_height();
 cout << "height: " << value << endl;

}
```

Friend functions

- Friend functions of an object can "see" inside the object and access private member functions and data.
- To declare a function as a friend of a class, precede the function prototype in the class definition with the keyword *friend*.

```
class rectangle {

 friend void set_value(rectangle&, float);   // friend declaration

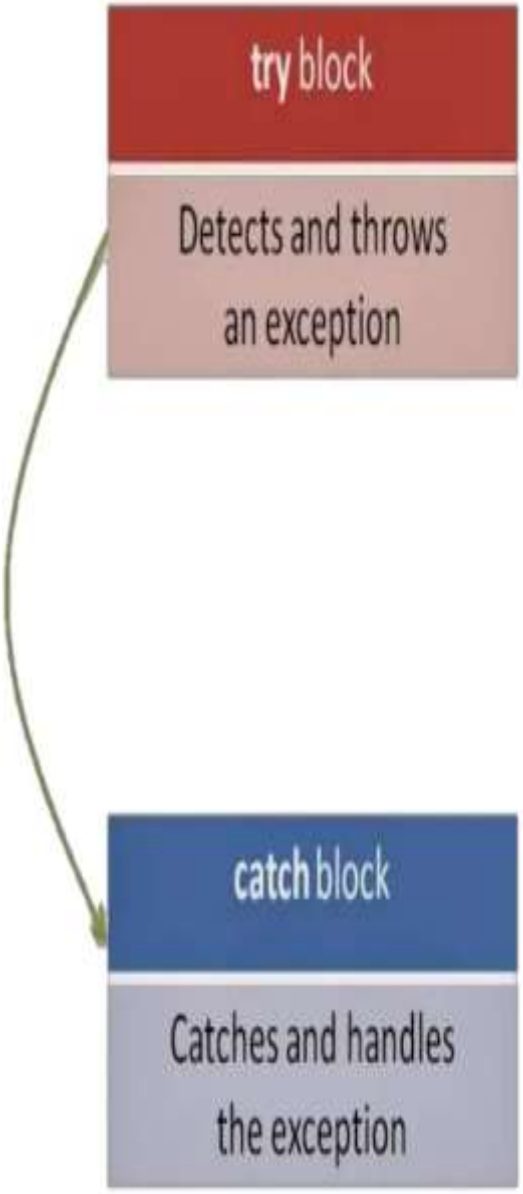
 private:
   float height;
   float width;
   int xpos;
   int ypos;
 public:
   rectangle(float, float);   // constructor
   void draw();               // draw member function
   void posn(int, int);       // position member function
   void move(int, int);       // move member function
   float get_height();        // access function
   void set_height(float);  // access function
};
```

```
void set_value(rectangle &rc, float h)
{
 rc.height = h;
}

void main()
{
 rectangle rc(1.0, 3.0);

 set_value(rc, 10.0);
}
```

Friend function declaration

```
Class ABC {
……
public:
……
friend void xyz(); //declaration
};
```

- The function declaration should be preceded by the keyword friend.
- The function is defined elsewhere in the program.
- The function definition does not use either the keyword friend or the scope resolution operator::

```
class Alpha{
int a;
int b;
public:
void set();
friend void add(Alpha ob2);  //add is declared as friend to class Alpha
};
void Alpha ::set(){
a=10;b=20;
}
void add(Alpha ob2){
int sum = ob2.a + ob2.b;
cout<<“Sum = ”<<sum;
}
int main() {
Alpha ob1;
ob1.set();
add(ob1);
return 0; }
```

Friend Function properties

- It is not in the scope of the class to which it has been declared as friend.
- It can't be called using the object of that class. Thus has to be invoked like a normal C++ function.
- It can be declared either in the public or the private section of a class without affecting it's meaning.
- Usually it takes objects as arguments.
- Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
- Friend function is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
- Friend function is not inherited.
- The concept of friends is not there in Java.

Example 1

```
#include <iostream>
using  namespace std;
class A {
private:
   int a;
 public:
   A() { a = 0; }
   friend class B;        // Friend Class
};
 class B {
private:
   int b;
 public:
   void showA(A& x) {
      // Since B is friend of A, it can access private members of A
         cout << "A::a=" << x.a;
   }};
 int main() {
   A a;
   B b;
   b.showA(a);
   return 0;
}
```

Example 2

```
class X {
int a;
public:
X(int a1) {
a = a1;
}
friend void Y ::add(X);
};
class Y {
int b;
public:
Y(int b1) {
b = b1;
}
void add(X p) {
count<< ( b + p.a)
}};
int main() {
X x1(5);
Y y1(10);
y1.add(x1);
return 0;  }
```

Note : Member function add() of Y is accessing the private data of X So X must declare add() of Y as its friend

Virtual Function & Pure Virtual Function





The structure of the common derived class and the form of the storage for the member data.

```
class A{
public:
    void
    showa(){cout<<"A"<<endl;}
};
class B: public A{
public:
    void
    showb(){cout<<"B"<<endl;}
};
class C: public A{
public:
    void
    showc(){cout<<"C"<<endl;}
};
```

```
class D: public B, public C{
public:
    void
    showd(){cout<<"D"<<endl;}
};

void main()
{
     D d;
    d.showa();
}
```

Virtual Function

- A virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class.
- When we refer to a derived class object using a pointer or a reference to the base class, we can call a virtual function for that object and execute the derived class's version of the function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve runtime polymorphism.
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at runtime.

Rules for Virtual Function

- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
- The prototype of virtual functions should be the same in the base as well as derived class.
- They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

Example 1

```
#include<iostream>
using namespace std;
 class base {
public:
   virtual void print() {
      cout << "print base class\n";
   }
  void show() {
      cout << "show base class\n";
   }
};
class derived : public base {
public:
   void print()  {
      cout << "print derived class\n";
   }
 void show()  {
      cout << "show derived class\n";
   }
};
 int main()   {
   base *bptr;
   derived d;
   bptr = &d;
 bptr->print();
bptr->show();
return 0;
}
```

Example 2

```
#include<iostream>
using namespace std;
 class base {
public:
   void fun_1() { cout << "base-1\n"; }
   virtual void fun_2() { cout << "base-2\n"; }
   virtual void fun_3() { cout << "base-3\n"; }
   virtual void fun_4() { cout << "base-4\n"; }
};
class derived : public base {
public:
   void fun_1() { cout << "derived-1\n"; }
   void fun_2() { cout << "derived-2\n"; }
   void fun_4(int x) { cout << "derived-4\n"; }
};
 int main()  {
   base *p;
   derived obj1;
   p = &obj1;
  p->fun_1();
  // Late binding (RTP)
   p->fun_2();
  p->fun_3();
  p->fun_4();
   return 0;  }
```

Exception Handling in C++

Exception Handling in C++

- The process of converting system error messages into user friendly error message is known as **Exception handling**. This is one of the powerful feature of C++ to handle run time error and maintain normal flow of C++ application.

- **Exception**
- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's Instructions.

Exception Handling in C++

- Exceptions are runtime anomalies that a program encounters during execution. It is a situation where a program has an unusual condition and the section of code containing it can't handle the problem. Exception includes condition such as division by zero, accessing an array outside its bound, running out of memory, etc.
  In order to handle these exceptions, exception handling mechanism is used which identifies and deal with such condition. Exception handling mechanism consists of following parts:
- Find the problem (Hit the exception)
- Inform about its occurrence (Throw the exception)
- Receive error information (Catch the exception)
- Take proper action (Handle the exception)

Exception Handling in C++

Handling the Exception

- Handling the exception is nothing but converting system error message into user friendly error message. Use Three keywords for Handling the Exception in C++ Language, they are;
- try
- catch
- throw

Exception Handling in C++

- **try:** Try block consists of the code that may generate exception. Exception are thrown from inside the try block.
- **try:** represents a block of code that can throw an exception.
- "**try**" block groups one or more program statements with one or more catch clauses.

Exception Handling in C++

- **throw:** Throw keyword is used to throw an exception encountered inside try block. After the exception is thrown, the control is transferred to catch block.

- Raising of an exception is done by "throw" expression.

Exception Handling in C++

- **catch:** Catch block catches the exception thrown by throw statement from try block. Then, exception are handled inside catch block.

- **catch :**The catch block defines the action to be taken, when an exception occur.

Exception Handling in C++

Syntax of Exception Handling

```
try
{
  statements;
  ... ... ...
  throw exception;
}

catch (type argument)
{
  statements;
  ... ... ...
}
```

```
try
{
  statements;
  ... ... ...
  throw exception;
}

catch (type argument)
{
  statements;
  ... ... ...
}
```



Multiple Catch Exception-Syntax

```
try {
   body of try block
}
catch (type1 argument1) {
   statements;
   ... ... ...
}
catch (type2 argument2) {
   statements;
   ... ... ...
}
... ... ...
... ... ...
catch (typeN argumentN) {
   statements;
   ... ... ...
}
```

Example without Exception Handling

```
#include<iostream>
using namespace std;
int main()
{
        int number, ans;
        number=10;
        ans=number/0;
        cout<<"Result: "<<ans;
}
```

Abnormally Terminate Program

Example of Exception Handling

```
#include<iostream>
using namespace std;
int main()
        int number=10, ans=0;
        try
        {
                ans=number/0;
        }
        catch(int i)
        {
                cout<<"Denominator not be zero";
        }
}
```

Denominator not be zero

Example of Exception Handling-1

```
#include <iostream>
using namespace std;
int main()
{
        int n1,n2,result;
        cout<<"Enter 1st number : ";
        cin>>n1;
        cout<<"Enter 2nd number : ";
        cin>>n2;
```

Example of Exception Handling-2

```
try {
    if(n2==0)
        throw n2;    //Statement 1
    else {
        result = n1 / n2;
        cout<<"\nThe result is : "<<result;
    }
}
 catch(int x) {
    cout<<"\nCan't divide by : "<<x;
}

cout<<"\nEnd of program.";

}
```

Output of the Previous Program

```
Enter 1st number : 45
Enter 2nd number : 0

Can't divide by : 0
End of program
```

```
Enter 1st number : 12
Enter 2nd number : 20

The result is : 0
End of program
```

Thank You

Memory Management in C++

Definition

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on **Heap,** and non-static and local variables get memory allocated on **Stack.**
- C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.
- In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.

What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size, which is not possible with compiler allocated memory except for variable-length arrays.
- The most important use is the flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need it and whenever we don't need it anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

There are two operators we use for this purpose –

1. new operator
2. delete operator

The new operator allocates memory to a variable. For example,

```
int* pointVar;
pointVar = new int;
*pointVar = 45;
```

In the above code, the new operator returns the address of the variable's memory location.

But in the case of an array, the new operator returns the address of the first element of the array.

delete Operator

- Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.
- For this, the delete operator is used. It returns the memory to the operating system. This is known as memory deallocation.

The syntax for delete operator is –

```
delete pointerVariable;
```

Example

```
int* pointVar;
pointVar = new int;
*pointVar = 45;
cout << *pointVar;
delete pointVar;
```

Example

```
#include <iostream>
 using namespace std;
 int main() {
 // declare an int pointer
 int* pointInt;
// declare a float pointer
 float* pointFloat;
// dynamically allocate memory
 pointInt = new int;
pointFloat = new float;
 // assigning value to the memory
 *pointInt = 45;
*pointFloat = 45.45f;
cout << *pointInt << endl;
cout << *pointFloat << endl;
 // deallocate the memory
delete pointInt;
delete pointFloat;
return 0;
}
```

Example

```
#include <iostream>
 using namespace std;
 int main() {
 int num;
cout << "Enter total number of students: ";
 cin >> num;
 float* ptr;
 ptr = new float[num];
 cout << "Enter GPA of students." << endl;
 for (int i = 0; i < num; ++i) {
 cout << "Student" << i + 1 << ": ";
 cin >> *(ptr + i);
 }
cout << "\n Displaying GPA of students." << endl;
 for (int i = 0; i < num; ++i) {
 cout << "Student" << i + 1 << ": " << *(ptr + i) << endl;
 }
delete[] ptr;
return 0;
}
```

C++ String Class

- C++ string class internally uses char array to store character but all memory management, allocation, and null termination is handled by string class itself that is why it is easy to use.
- The length of the C++ string can be changed at runtime because of dynamic allocation of memory similar to vectors.
- As string class is a container class, we can iterate over all its characters using an iterator similar to other containers like vector, set and maps, but generally, we use a simple for loop for iterating over the characters and index them using the [] operator.
  C++ string class has a lot of functions to handle string easily. Most useful of them are demonstrated in below code.

Example

```
int main()
{
   // various constructor of string class

   // initialization by raw string
   string str1("first string");

   // initialization by another string
   string str2(str1);

   // initialization by character with number of occurrence
   string str3(5, '#');

   // initialization by part of another string
   string str4(str1, 6, 6); //    from 6th index (second parameter)
                        // 6 characters (third parameter)

   // initialization by part of another string : iterator version
   string str5(str2.begin(), str2.begin() + 5);
   cout << str1 << endl;
   cout << str2 << endl;
   cout << str3 << endl;
   cout << str4 << endl;
   cout << str5 << endl;
   //  assignment operator
   string str6 = str4;
// clear function deletes all character from string
   str4.clear();
   //  both size() and length() return length of string and
```

Continue

```
 //  they work as synonyms
int len = str6.length(); // Same as "len = str6.size();"
cout << "Length of string is : " << len << endl;
// a particular character can be accessed using at /
// [] operator
char ch = str6.at(2); //  Same as "ch = str6[2];"
cout << "third character of string is : " << ch << endl;
//  front return first character and back returns last character
//  of string
char ch_f = str6.front();  // Same as "ch_f = str6[0];"
char ch_b = str6.back();   // Same as below
                    // "ch_b = str6[str6.length() - 1];"

cout << "First char is : " << ch_f << ", Last char is : "
    << ch_b << endl;

// c_str returns null terminated char array version of string
const char* charstr = str6.c_str();
printf("%s\n", charstr);
// append add the argument string at the end
str6.append(" extension");
//  same as str6 += " extension"
// another version of append, which appends part of other
// string
str4.append(str6, 0, 6);  // at 0th position 6 character

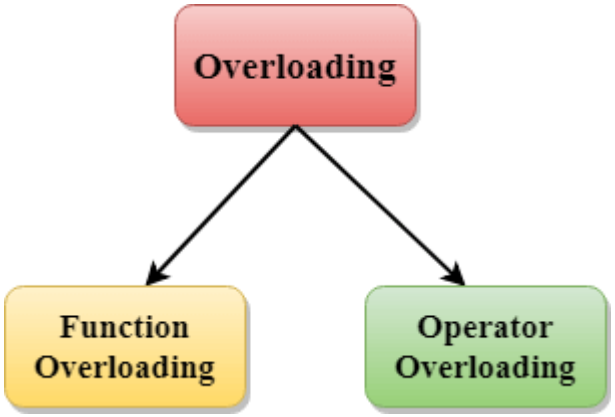
cout << str6 << endl;
cout << str4 << endl;
```

Continue

```
if (str6.find(str4) != string::npos)
        cout << "str4 found in str6 at " << str6.find(str4)
             << " pos" << endl;
    else
        cout << "str4 not found in str6" << endl;
    //  substr(a, b) function returns a substring of b length
    //  starting from index a
    cout << str6.substr(7, 3) << endl;

    //  if second argument is not passed, string till end is
    // taken as substring
    cout << str6.substr(7) << endl;
    //  erase(a, b) deletes b characters at index a
    str6.erase(7, 4);
    cout << str6 << endl;
    //  iterator version of erase
    str6.erase(str6.begin() + 5, str6.end() - 3);
    cout << str6 << endl;

    str6 = "This is a examples";

    //  replace(a, b, str)  replaces b characters from a index by str
    str6.replace(2, 7, "ese are test");

    cout << str6 << endl;

    return 0;
}
```

Output

```
first string
 first string
 #####
string first Length of string is : 6
 third character of string is : r
 First char is : s, Last char is : g
string
string extension
string
str4 found in str6 at 0 pos
Ext
 extension
string nsion strinion These are test examples
```

C++ Overloading (Function and Operator)

Definition

- If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:
- methods
- constructors
- indexed properties
- It is because these members have parameters only.

Types of overloading in C++

- Function overloading
- Operator overloading



C++ Function Overloading

- Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++.
- In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.
- The **advantage** of Function overloading is that it increases the readability of the program because we don't need to use different names for the same action.

Example 1

```
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
int main(void) {
   Cal C;                                  //    class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Example 2

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   cout << "r1 is : " <<r1<< endl;
   cout <<"r2 is : "  <<r2<< endl;
   return 0;
}
```

Function Overloading and Ambiguity

- When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.
- When the compiler shows the ambiguity error, the compiler does not run the program.
- **Causes of Function Overloading:**
- Type Conversion.
- Function with default arguments.
- Function with pass by reference.

Type Conversion

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   cout << "Value of i is : " <<i<< endl;
}
void fun(float j)
{
   cout << "Value of j is : " <<j<< endl;
}
int main()
{
   fun(10);
   fun(1.2);
   return 0;
}
```

- The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Operator Overloading

- In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.
- Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
  Example:
- ```
  int a;
  float b, sum;
  sum=a+b;
  ```
- Here, variables "a" and "b" are of types "int" and "float", which are built-in data types. Hence the addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.
  ```

Now, consider another example

```
class A
    {
    };
int main()
    {
        A   a1,a2,a3;
       a3= a1 + a2;
       return 0;
    }
```

- In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type i.e. of type “class A” using the “+” operator. This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.

- In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**. For example,
- Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers.
- Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

  result = c1 + c2;
- instead of something like:

  result = c1.addNumbers(c2);
- **Syntax for C++ Operator Overloading**

  To overload an operator, we use a special operator function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

  returnType operator symbol (arguments) { }

Operator Overloading in Unary Operators

- Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

```
class Count {
private:
int value;
public:
// Constructor to initialize count to 5
Count() : value(5) {}
// Overload ++ when used as prefix
void operator ++ () {
++value; }
void display() {
cout << "Count: " << value << endl;
} };
int main() {
Count count1;
 // Call the "void operator ++ ()" function
++count1;
count1.display();
return 0; }
```

Can we overload all operators?

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

- sizeof
- typeid
- Scope resolution (::)
- Class member access operators (.(dot), .* (pointer to member operator))
- Ternary or conditional (?:)

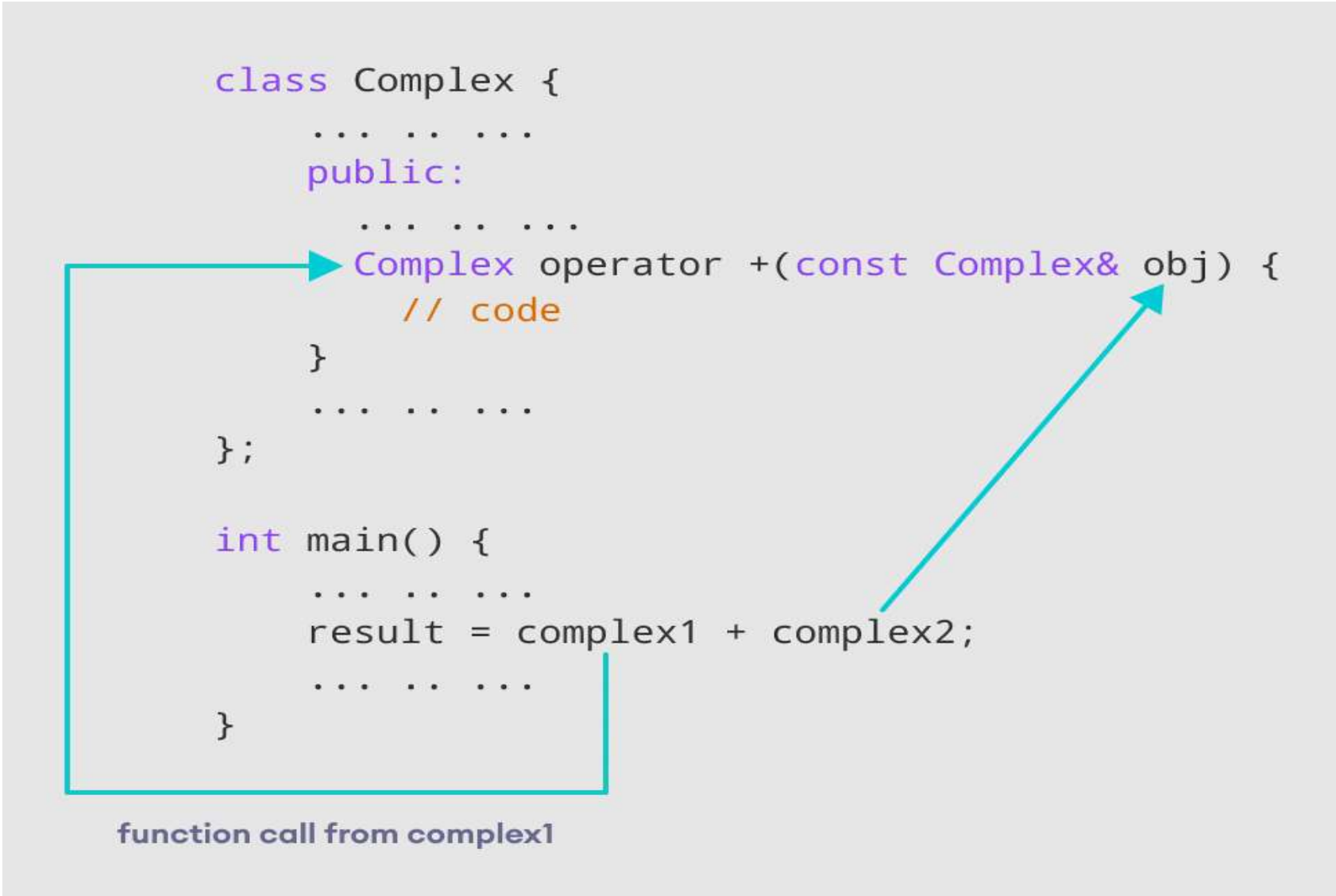
Binary operator Overloading

```
#include <iostream>
using namespace std;
class Complex_num
{
   int x, y;
   public:
      void input()   {
         cout << " Input two complex number: " << endl;
         cin >> x >> y;
      }
      // use binary '+' operator to overload
      Complex_num operator + (Complex_num  &obj)
      {
         // create an object
         Complex_num A;
         // assign values to object
         A.x = x + obj.x;
         A.y = y + obj.y;
         return (A);
      }
      // overload the binary (-) operator
Complex_num operator - (Complex_num  &obj)
      {
         Complex_num A;
         // assign values to object
         A.x = x - obj.x;
         A.y = y - obj.y;
         return (A);
      }
```

```
 // display the result of addition
      void print1()  {
         cout << x << " + " << y << "i" << "\n";  }
// display the result of subtraction
      void print2()  {
         cout << x << " - " << y << "i" << "\n";  }
};
int main ()  {
Complex_num x1, y1, sum, sub;
                  // here we created object of class Addition i.e x1 and y1  accepting the values
   x1.input();
   y1.input();
   sum = x1 + y1;                              // add the objects
   sub = x1 - y1;                             // subtract the complex number
   // display user entered values
   cout << "\n Entered values are: \n";
   cout << " \t";
   x1.print1();
   cout << " \t";
   y1.print1();
   cout << "\n The addition of two complex (real and imaginary) numbers: ";
   sum.print1(); // call print function to display the result of addition
   cout << "\n The subtraction of two complex (real and imaginary) numbers: ";
   sub.print2(); // call print2 function to display the result of subtraction
   return 0;  }
```

Addition of two Complex Numbers

```
#include <iostream>
using namespace std;
class Complex {
private:
 float real; float imag;
 public: // Constructor to initialize real and imag to 0
Complex() : real(0), imag(0) {}
void input() {
 cout << "Enter real and imaginary parts respectively: ";
 cin >> real;
cin >> imag; }
// Overload the + operator
Complex operator + (Const Complex  &obj) {
Complex temp;
temp.real = real + obj.real;
temp.imag = imag + obj.imag;
 return temp; }
void output() {
if (imag < 0)
 cout << "Output Complex number: " << real << imag << "i";
else
cout << "Output Complex number: " << real << "+" << imag << "i";
 }
};
 int main() {
 Complex complex1, complex2, result;
cout << "Enter first complex number:\n";
complex1.input();
cout << "Enter second complex number:\n";
complex2.input();
 // complex1 calls the operator function // complex2 is passed as an argument to the function
 result = complex1 + complex2;
 result.output();
return 0;
 }
```


class Complex {
... .
public:
... .
Complex operator +(const Complex& obj) {
// code
}
... .
};
int main() {
... .
result = complex1 + complex2;
... .
}
function call from complex1

Pure Virtual Function

Definition

Pure virtual functions are used -

- If a function doesn't have any use in the base class but the function must be implemented by all its derived classes.
- For example suppose, we have derived Triangle, Square and Circle classes from the Shape class, and we want to calculate the area of all these shapes.
- In this case, we can create a pure virtual function named calculateArea() in the Shape class.

Example

- A pure virtual function doesn't have the function body and it must end with 0. For example,

```
class shape {
public:
virtual void calculateArea()=0;
}
```

Abstract Class

- A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.
- We cannot create objects of an abstract class. however, we can derive classes from them, and use their data members and member functions (except pure virtual functions).

Example

```
#include <iostream>
 using namespace std;
// Abstract class
class Shape {
protected:
float dimension;
public:
void getDimension() {
cin >> dimension;
}
virtual float calculateArea() = 0;
 };
class Square : public Shape {
public:
float calculateArea() {
 return dimension * dimension;
} };
class Circle : public Shape {
public: float calculateArea() {
 return 3.14 * dimension * dimension;
 } };
```

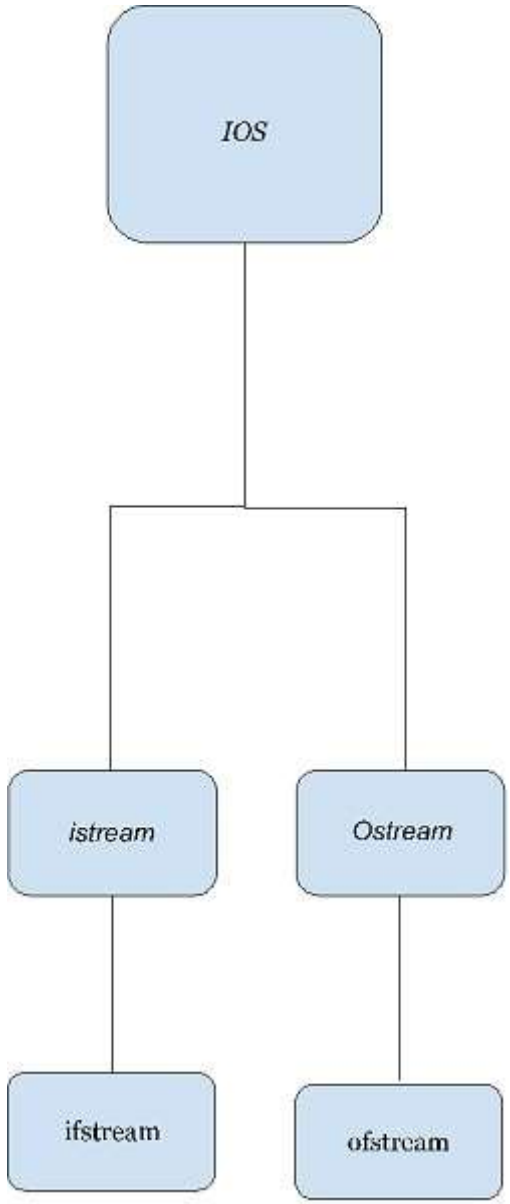
```
int main()
{
Square square;
Circle circle;
cout << "Enter the length of the square: ";
square.getDimension();
cout << "Area of square: " << square.calculateArea() << endl;
cout << "\nEnter radius of the circle: ";
 circle.getDimension();
cout << "Area of circle: " << circle.calculateArea() << endl;
 return 0;
 }
```

C++ Stream Class

Definition

- Stream in C++ means a stream of characters that gets transferred between the program thread and input or output.
- There are a number of C++ stream classes eligible and defined which is related to the files and streams for providing input-output operations.
- All the classes and structures maintaining the file and folders with hierarchies are defined within the file with iostream.h standard library.
- Classes associated with the C++ stream include ios class, istream class, and ostream class.
- Class ios is indirectly inherited from the base class involving iostream class using istream class and ostream class which is declared virtually.

- There is a number of stream classes in the hierarchy which is defining and giving different flows for the varied objects in the class.
- The hierarchy is maintained in a way where it gets started from the top class which is the ios class followed by all the other classes involving istream class, ostream class, iostream class, istream_withassign class, and ostream_withassign class.
- The iosclass in the hierarchy is the parent class which is considered as a class from where both the istream and ostream class gets inherited. Both the istream class and ostream class constitute the ios class which is the highest level of the entire hierarchy of C++ stream classes.
- The other classes which include functions for the operations include assignment operation like _withassign classes.


IOS
istream
Ostream
ifstream
ofstream

Various stream classes in C++ are as follows:

- istream class
- ostream class
- iostream class
- ios class
- ostream_withassign class
- istream_withassign class

C++ FileStream example: writing to a file

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ofstream filestream("testout.txt");
  if (filestream.is_open())
  {
    filestream << "Welcome to CSVTU.\n";
    filestream << "C++ Class.\n";
    filestream.close();
  }
  else
cout <<"File opening is fail.";
  return 0;
}
```

C++ FileStream example: reading from a file

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  string srg;
  ifstream filestream("testout.txt");
  if (filestream.is_open())
  {
    while ( getline (filestream,srg) )
    {
      cout << srg <<endl;
    }
    filestream.close();
  }
  else {
      cout << "File opening is fail."<<endl;
    }
  return 0;
}
```

Thank You

Templates

Definition

- Templates in C++ is an interesting feature that is used for generic programming. Generic Programming is an approach of programming where generic types are used as parameters in algorithms to work for a variety of data types.
- It is defined as a blueprint or formula for creating a generic class or a function. To simply put, you can create a single function or single class to work with different data types using templates.
- It works in such a way that it gets expanded at compiler time, just like macros and allows a function or class to work on different data types without being rewritten.

Types of Template

There are two types of templates in C++

- **Function template**
- **Class template**

Function Template

- Function template in c++ is a single function template that works with multiple data types simultaneously, but a standard function works only with one set of data types.

C++ Function Template Syntax -

template<class type>ret-type func-name(parameter list)

{

//body of the function

}

Here, type is a placeholder name for a data type used by the function. It is used within the function definition.

The class keyword is used to specify a generic type in a template declaration.

Example

```
#include<iostream.h>
using namespace std;
template<classX>        //can replace 'class" keyword by
                        "typename" keyword
X func( Xa,Xb) {
 return a;
}
int main()
{
count<<func(15,8),,endl;                    //func(int,int);
    count,,func('p','q'),,endl;             //func(char,char);
    count<<func(7.5,9.2),,endl;             //func(double,double)
    return();
}
```

Class Template

The class template in c++ is like function templates. They are known as generic templates. They define a family of classes in C++.

Syntax of Class Template

```
template<class Ttype>
class class_name
{
//class body;
}
```

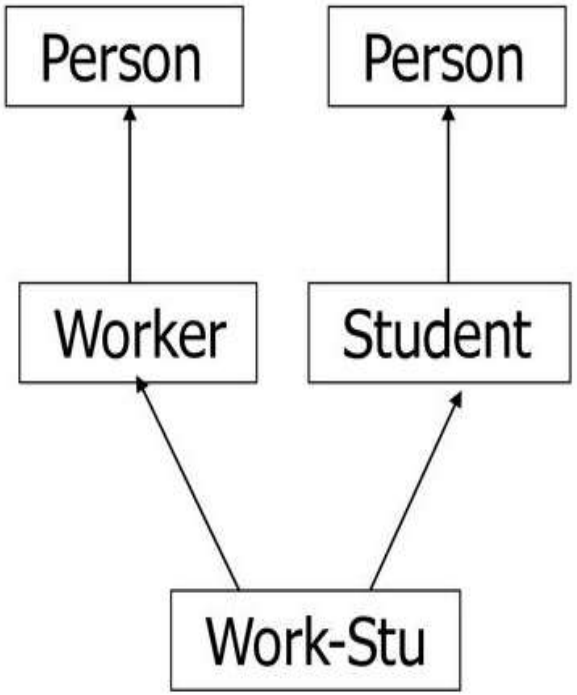
Here Ttype is a placeholder type name, which will be specified when a class instantiated.

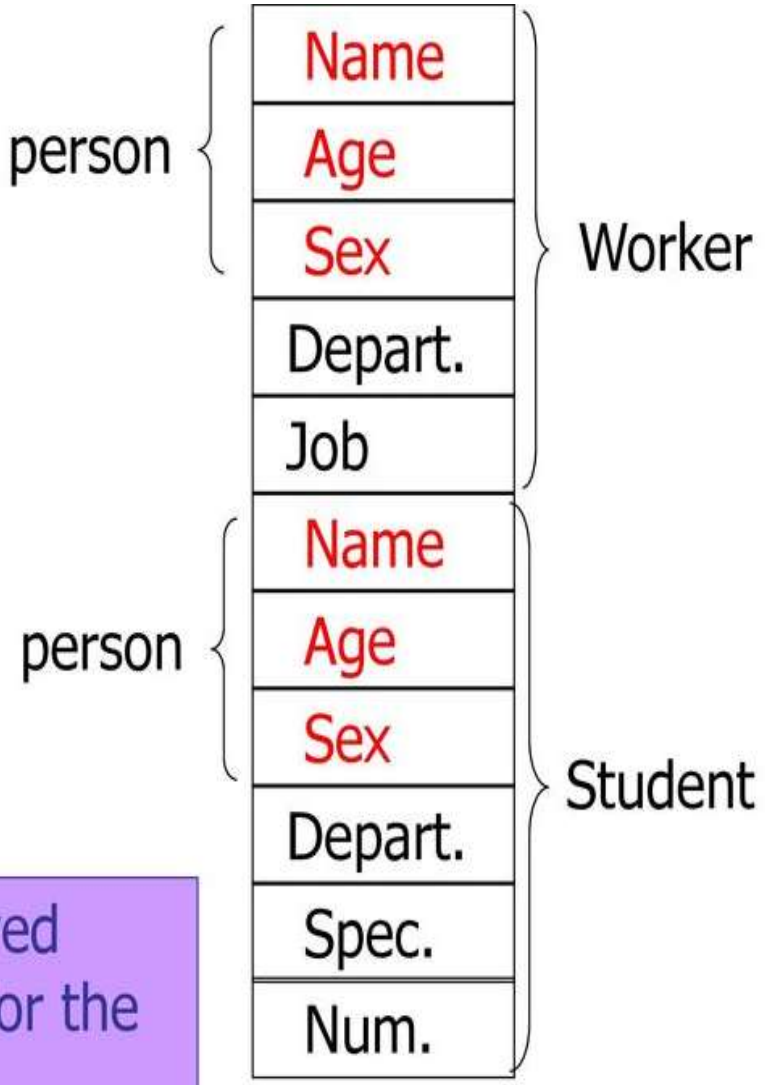
The Ttype can be used inside the body of the class.

Example

```
#include<iostream.h>
using namespace std;
template <class C> class A {
 private:
 C a,b;
public:
A(Cx,Cy){
a=x; b=y;
}
void show() {
count<<"The Addition of"<<a<<"and"<<b<<"is"<<add()<<endl;
 }
 C add() {
C c=a+b;
return c;
 }
};
 int main(){
A addint(8,6);
A addfloat(3.5,2.6);
A aaddouble(2.156,5.234);
A addint.show();
cout<<endl;
adddouble.show();
 count<<endl;
return 0;
 }
```

Virtual Function & Pure Virtual Function





The structure of the common derived class and the form of the storage for the member data.

```
class A{
public:
    void
    showa(){cout<<"A"<<endl;}
};
class B: public A{
public:
    void
    showb(){cout<<"B"<<endl;}
};
class C: public A{
public:
    void
    showc(){cout<<"C"<<endl;}
};
```

```
class D: public B, public C{
public:
    void
    showd(){cout<<"D"<<endl;}
};

void main()
{
     D d;
    d.showa();
}
```

Virtual Function

- A virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class.
- When we refer to a derived class object using a pointer or a reference to the base class, we can call a virtual function for that object and execute the derived class's version of the function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve runtime polymorphism.
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at runtime.

Rules for Virtual Function

- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
- The prototype of virtual functions should be the same in the base as well as derived class.
- They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

Example 1

```
#include<iostream>
using namespace std;
  class base {
public:
    virtual void print() {
       cout << "print base class\n";
    }
   void show() {
       cout << "show base class\n";
    }
};
class derived : public base {
public:
    void print()  {
       cout << "print derived class\n";
    }
  void show()  {
       cout << "show derived class\n";
    }
};
  int main()   {
    base *bptr;
    derived d;
    bptr = &d;
  bptr->print();
 bptr->show();
return 0;
}
```

Example 2

```
#include<iostream>
using namespace std;
 class base {
public:
   void fun_1() { cout << "base-1\n"; }
   virtual void fun_2() { cout << "base-2\n"; }
   virtual void fun_3() { cout << "base-3\n"; }
   virtual void fun_4() { cout << "base-4\n"; }
};
class derived : public base {
public:
   void fun_1() { cout << "derived-1\n"; }
   void fun_2() { cout << "derived-2\n"; }
   void fun_4(int x) { cout << "derived-4\n"; }
};
 int main()  {
   base *p;
   derived obj1;
   p = &obj1;
  p->fun_1();
  // Late binding (RTP)
   p->fun_2();
  p->fun_3();
  p->fun_4();
   return 0;  }
```