First Year First Semester Course

M.Tech. (CS) [Batch 2021-23]

Lecture #16

Introduction to Programming

C++: Inheritance (Part-II) and Polymorphism

Course Instructor:

Dr. Monidipa Das

DST-INSPIRE Faculty

Machine Intelligence Unit (MIU), Centre for Artificial Intelligence and Machine Learning (CAIML)

Indian Statistical Institute (ISI) Kolkata, India

Constructor in Derived Classes

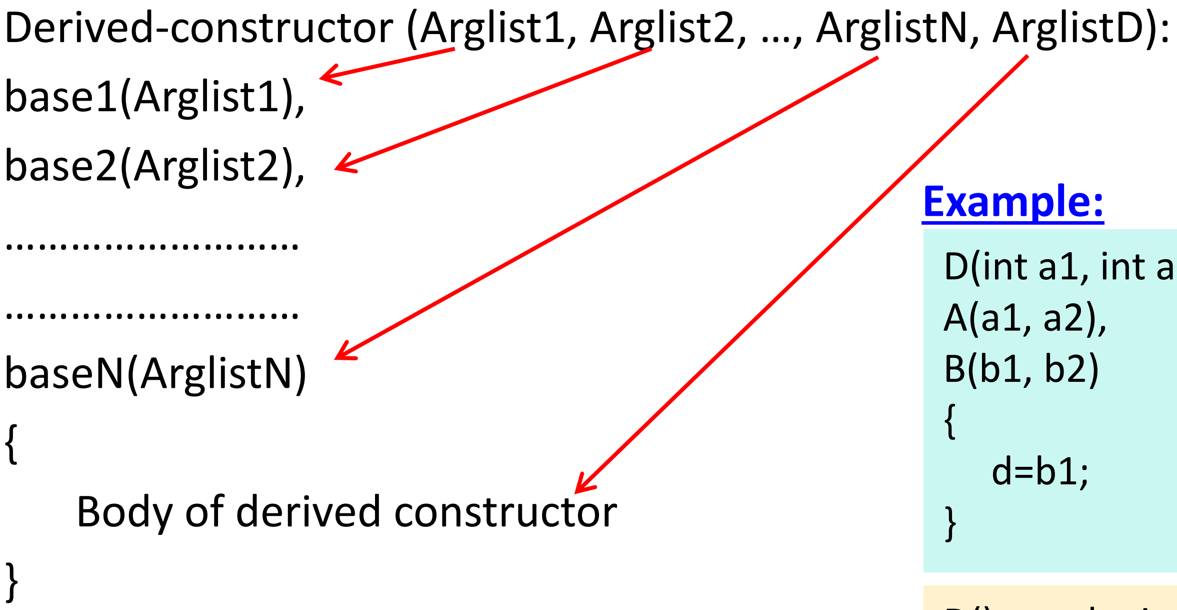
- Mandatory for a derived class if the base class contains a constructor with one or more arguments

- When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed

  – Multilevel Inheritance: Constructors are executed in the order of inheritance

  – Multiple Inheritance: Constructors are executed in the order in which they appear in the declaration of the derived class

Constructor in Derived Classes



```
Derived-constructor (Arglist1, Arglist2, …, ArglistN, ArglistD):
base1(Arglist1),
base2(Arglist2),
………………………
………………………
baseN(ArglistN)
{
    Body of derived constructor
}
```

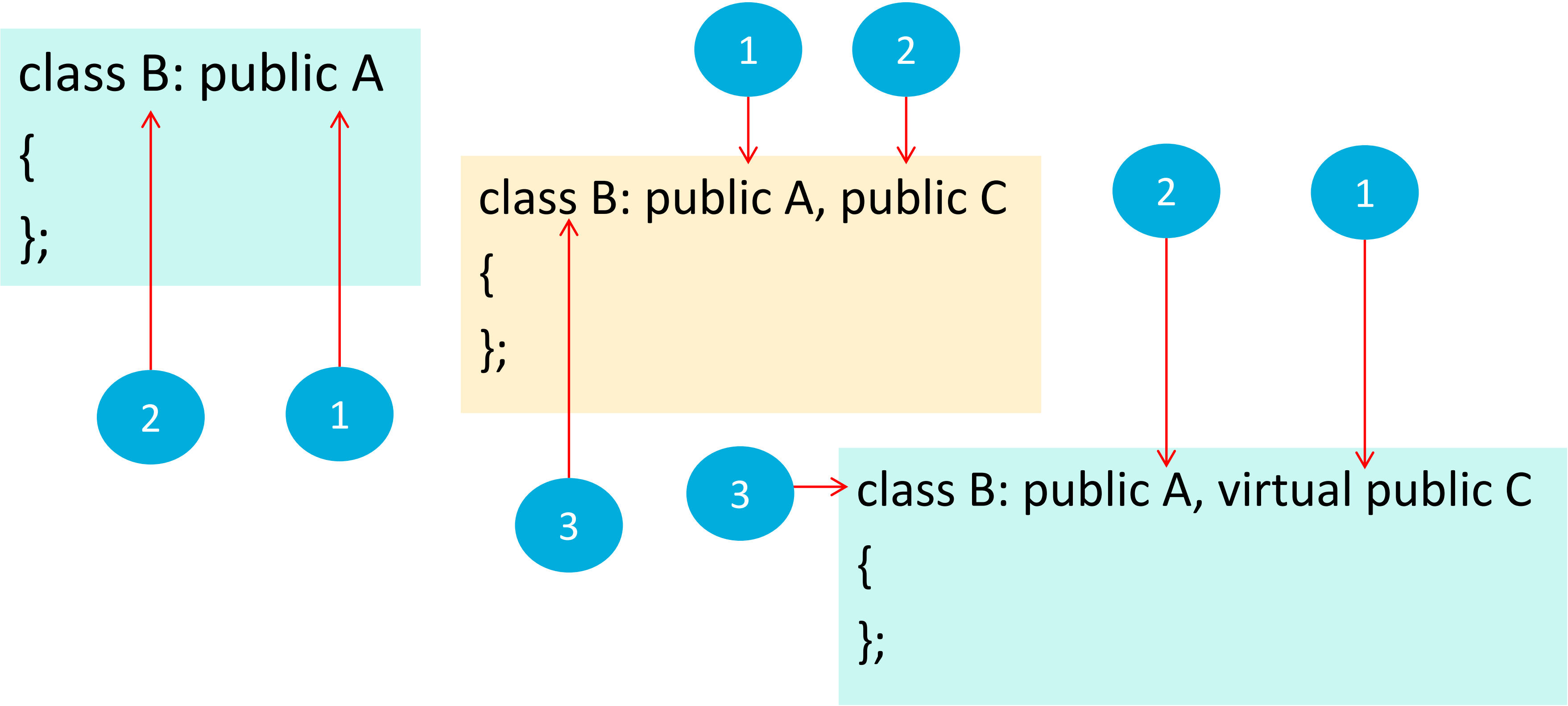
Example:

```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),
B(b1, b2)
{
    d=b1;
}
```

D() may be invoked as follows:

D objD(5, 12, 2.5, 7.54, 30)

Order of Execution of Base Class Constructors



```
class B: public A
{
};
```

```
class B: public A, public C
{
};
```

```
class B: public A, virtual public C
{
};
```

Polymorphism

Polymorphism

- One name multiple forms
- Compile time polymorphism
  - Early binding/ Static binding/ Static Linking
  - Operator overloading
  - Function overloading
- Run-time polymorphism
  - **Virtual functions**

Pointer in C++

- A derived data type that refers to another variable by storing the variable's memory address rather than data
- Provides alternative means to access the other data objects

- Declaring and Initialization

data-type *pointer-variable;

```
int *ptr, a;    //declaration
 ptr=&a;         //initialization
```

- Manipulations:

***pointer-variable**

```
int *ptr, a=10;    ptr=&a;  *ptr=(*ptr)/5;  cout<<"Value of a is: "<<a;
```

- Pointer Expressions and Pointer Arithmetic:

```
int a[10]; int *ptr; ptr=&a[0]; ptr++
```

Pointer in C++

- Pointers with Arrays and Strings

```
int *ptr, number[]={12,23,34,45,56,67,78,89,90};
ptr=number;    //or ptr=&number[0];
char institute[]=“ISI”;
char *iptr=“ISI”;
```

- Pointers to Functions
  - Allows C++ program to select a function dynamically at the run time.
  - Function can be passed as an argument to another function through the function pointer
  - Cannot be de-referenced
  - Function pointer comparison is allowed in C++

Pointer in C++

- Declaring function pointer

  data-type (*function_name)(argument-list);

  int (*funcptr)(int,int);

```
#include <iostream>
using namespace std;
int (*funcptr)(int,int);  // function pointer declaration
int add(int a , int b)
{
    return a+b;
}
int subtract(int a , int b)
{
    return a-b;
}
```

```
int main(){
 funcptr=&add;
 cout << "The result is :" <<funcptr(5,4);
 funcptr=&subtract;
 cout << "\nThe result is :" <<funcptr(5,4);
  return 0;
}
```

Pointers to Objects

- An object of a class behaves identically as any other variable.
- Pointers can be defined for an object type.

```
class employee {
 int code;
 char name [20] ;
public:
 inline void getdata ( )= 0 ;
 inline void display ( )= 0 ;
};
```

```
employee *abc;
```

This declaration creates a pointer variable **abc** that can point to any object of employee type.

Example: Pointers to Objects

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

class item{
   int code;
   float price;
 public:
   void getdata(int a, float b)
   {
          code=a;
          price=b;
   }
   void show(void)
   {
          cout<<"Code: "<<code<<"\n";
          cout<<"Price: "<<price<<"\n";
   }
};

const int size=2;
```

```
int main(){
item *p=new item[size];
item *d=p;
int x, i;
float y;
   for(i=0; i<size; i++){
          cout<<"Input code and price for item-“
          <<i+1<<": ";
          cin>>x>>y;
          p->getdata(x,y);
          p++;
   }
   for(i=0; i<size; i++){
          cout<<"\nItem-"<<i+1
          <<"\n-----------\n";
          d->show();
          d++;
   }
   return 0;
}
```

this Pointer

- 'this' is a pointer that points to the object for which *this* function was called.
- **objA**.add() will set the pointer **this** to the address of the object **objA**
- **this** pointer acts as the implicit argument to all the member functions.
- **Example:**

```
class ABC
{
        int a;
        ---
        ---
};
```

Within any member function

a=123; and **this->a=123;**

are equivalent

- **Application of *this* pointer:** can be used to return the object it points to

```
return *this;  //will return the object that invoked the function
```

Example: **this** Pointer

```
#include <iostream>
#include<string.h>
using namespace std;

class person{
   char name[20];
   float age;
 public:
          person(char *s, float a){
                     strcpy(name,s); age=a;
          }
          person & greater(person &x){
                     if(x.age>age)
                                return x;
                     else
                                return *this;

          }
   void show(void){
                     cout<<"Name: "<<name<<"\n";
                     cout<<"Age: "<<age<<"\n";
   }
};
```

```
int main()
{
          person P1("Nandini", 24.6),
          P2("Ali", 22.5), P3("Jack",20.3);
          person tP=P2.greater(P1);
          cout<<"Elder person is: \n";
          tP.show();

          tP=P2.greater(P3);
          cout<<"\n\nElder person is: \n";
          tP.show();

          return 0;
}
```

Pointer to Derived Classes

- We can use pointer to the objects of derived classes too.
- C++ allows a pointer in a base class to point to either a base class object or to any derived class object.

```
class base {
          //Data Members
          //Member Functions
};
class derived : public base {
          //Data Members
          //Member functions
};
int main( ) {
    base *ptr;   //pointer to class base
    derived obj ;
    ptr =  &obj ;      //indirect  reference obj to the pointer
    ……. .
    ……. .
}
```

```
int main()  {
    base  obja;
    derived *ptr;
    ptr =  &obja; //invalid.... .
    //can be resolved by explicit type casting
}
```

Pointer to Derived Classes [contd.]

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

class Base
{
 public:
            int b;
            void show(){
                        cout<<"b="<<b<<"\n";
            }
};

class Derived : public Base
{
 public:
            int d;
            void show(){
            cout<<"b="<<b<<"\n"<<"d="<<d<<"\n";

            }
};
```

```
int main(){
            Base *bptr, base;
            bptr=&base;

            bptr->b=300;
            cout<<"bptr points to base object \n";
            bptr->show();

            Derived derived;
            bptr=&derived;
            bptr->b=400;
            cout<<"\nbptr points to derived object \n";
            bptr->show();

            Derived *dptr;
            dptr=&derived;
            dptr->d=500;
            cout<<"\ndptr is derived type pointer \n";
            dptr->show();

            cout<<"\nUsing type casting \n";
            ((Derived *)bptr)->d=490;
            ((Derived *)bptr)->show();
            return 0;}
```

Virtual Functions

- When the same function name is used in both the base and the derived classes, the function is base class is declared as virtual

- Use keyword virtual proceeding the normal function declaration

- When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer rather than type of the pointer

Example

```
#include<iostream>
using namespace std;
class Base
{
    public:
      void display(){cout<<"\n Display base";}
      virtual void show(){cout<<"\n Show base";}
};
class Derived: public Base
{
    public:
      void display(){cout<<"\n Display derived";}
      virtual void show(){cout<<"\n Show derived";}

};
```

```
int main()
{
   Base B;
   Derived D;
   Base *bptr;

   cout<<"\n bptr points to Base \n";
   bptr=&B;
   bptr->display();
   bptr->show();

   cout<<"\n\nbptr points to Derived \n";
   bptr=&D;
   bptr->display();
   bptr->show();

   return 0;
}
```

Runtime Polymorphism

```
#include<iostream>
#include<string.h>
using namespace std;

class media
{
    protected:
         char title[50];
         float price;
    public:
         media(char * s, float a)
         {
                  strcpy(title,s);
                  price=a;
         }
         virtual void display(){}
};
```

```
class book: public media{
         int pages;
         public:
                  book(char * s, float a,
int p): media(s,a){
                           pages=p;
                  }
                  void display();
};
class tape: public media{
         float time;
         public:
                  tape(char * s, float a,
float t): media(s,a){
                           time=t;
                  }
                  void display();
};
```

Runtime Polymorphism [contd.]

```
void book::display()
{
        cout<<"\n Title: "<<title;
        cout<<"\n Price: "<<price;
        cout<<"\n Pages: "<<pages;

}
void tape::display()
{
        cout<<"\n Title: "<<title;
        cout<<"\n Price: "<<price;
        cout<<"\n Play time:
"<<time<<"mins";
}
```

```
int main()
{
        char *title=new char[30];
        float price, time;
        int pages;

        cout<<"\n Enter Book Details: \n";
        cout<<" Title: "; cin>>title;
        cout<<" Price: "; cin>>price;
        cout<<" Pages: "; cin>>pages;

        book B(title,price,pages);

        cout<<"\n Enter Tape Details: \n";
        cout<<" Title: "; cin>>title;
        cout<<" Price: "; cin>>price;
        cout<<" Play time (mins): "; cin>>time;
```

Runtime Polymorphism [contd.]

```
        tape T(title,price,time);

        media *m[2];
        m[0]=&B;
        m[1]=&T;

        cout<<"\n Media Details: \n";
        cout<<"------- Book -------";
        m[0]->display();

        cout<<"\n\n Media Details: \n";
        cout<<"------- Tape -------";
        m[1]->display();

        return 0;
}
```

Rules for Virtual Functions

- Must be member of some class
- Cannot be static
- Accessed by using object pointers
- Can be friend of another class
- A virtual function in a base class must be defined
- The prototypes of the base class version of a virtual function and all the derived class versions must be identical

- Constructors cannot be virtual

- In a virtual function defined in the base class, it need not be necessarily redefined in the derived class. Function call invoke the base function.

Pure Virtual Function

- A function declared virtual inside a base class and is defined to be empty.
- “do-nothing” function.

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

- Compiler requires each derived class to either define the function or re-declare it as a pure virtual function

- A class containing pure virtual functions cannot be used for declaring an object of its own. Such classes are called ***Abstract base classes***

Example

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

class employee {
    int code;
    char name [20] ;
  public:
    virtual void getdata ( )=0;
    virtual void display ( ) ;
};

class grade: public employee
{
    char grd [90] ;
    float salary ;
  public :
    void getdata ( ) ;
    void display ( );
};
```

```
void employee:: display ( ){
}
void grade :: getdata ( ){
           cout<< "Enter employee's grade: ";
           cin>> grd ;
           cout<< "\nEnter the salary: " ;
           cin>> salary;
}
void grade :: display ( ){
           cout<<"Grade salary \n";
           cout<<grd<<" "<<salary;
}
```

```
int main ( ){
    employee *ptr ;
    grade obj ;
    ptr = &obj ; ptr->getdata();
    ptr->display(); return 0;
}
```

Questions?