CS-1004 Object Oriented programming
Week 8

Instructor:
ATIYA
LECTURER
(COMPUTER SCIENCE DEPARTMENT)
NATIONAL UNIVERSITY- FAST (KHI CAMPUS)
EMAIL: ATIYA.JOKHIO@NU.EDU.PK
ROOM: (LECTURER ROOM#19, CS DEPT BASEMENT-2)

Exercise

- Create two classes named Mammals and MarineAnimals. Create another class named BlueWhale which inherits both the above classes. Now, create a function in each of these classes which prints "I am mammal", "I am a marine animal" and "I belong to both the categories: Mammals as well as Marine Animals" respectively. Now, create an object for each of the above class and try calling
  1 - function of Mammals by the object of Mammal
  2 - function of MarineAnimal by the object of MarineAnimal
  3 - function of BlueWhale by the object of BlueWhale
  4 - function of each of its parent by the object of BlueWhale
- Types of Inheritance = ??
- Write a Basic Sekeletone

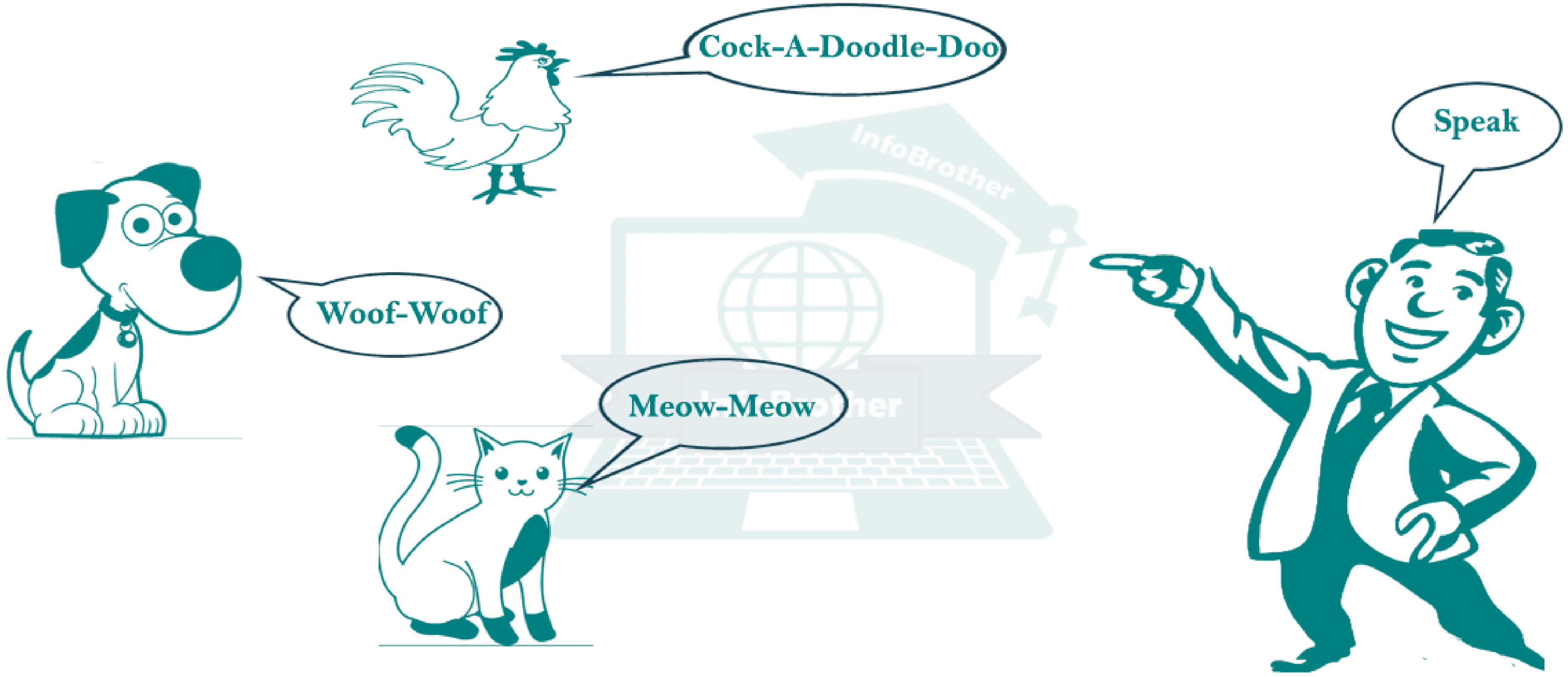
Polymorphism

- Polymorphism is derived from 2 Greek words: **poly** and morphs. The word "poly" means many and **morphs** means forms. So, polymorphism means many forms.
- Polymorphism is a feature of **OOPs** that allows the object to behave differently in different conditions.
- we can define polymorphism as the ability of a message to be displayed in more than one form.
- First concept given by Hindley and Milner.

Polymorphism-contd.

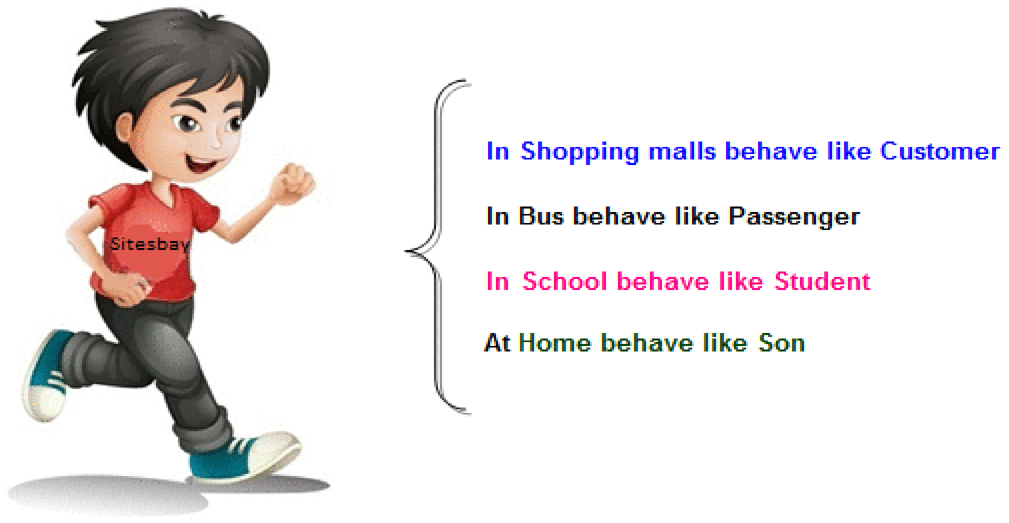
- A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee.
- So the same person posses different behavior in different situations. This is called polymorphism.

Polymorphism-contd.



Real life example of Polymorphism

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.


Sitesbay
In Shopping malls behave like Customer
In Bus behave like Passenger
In School behave like Student
At Home behave like Son

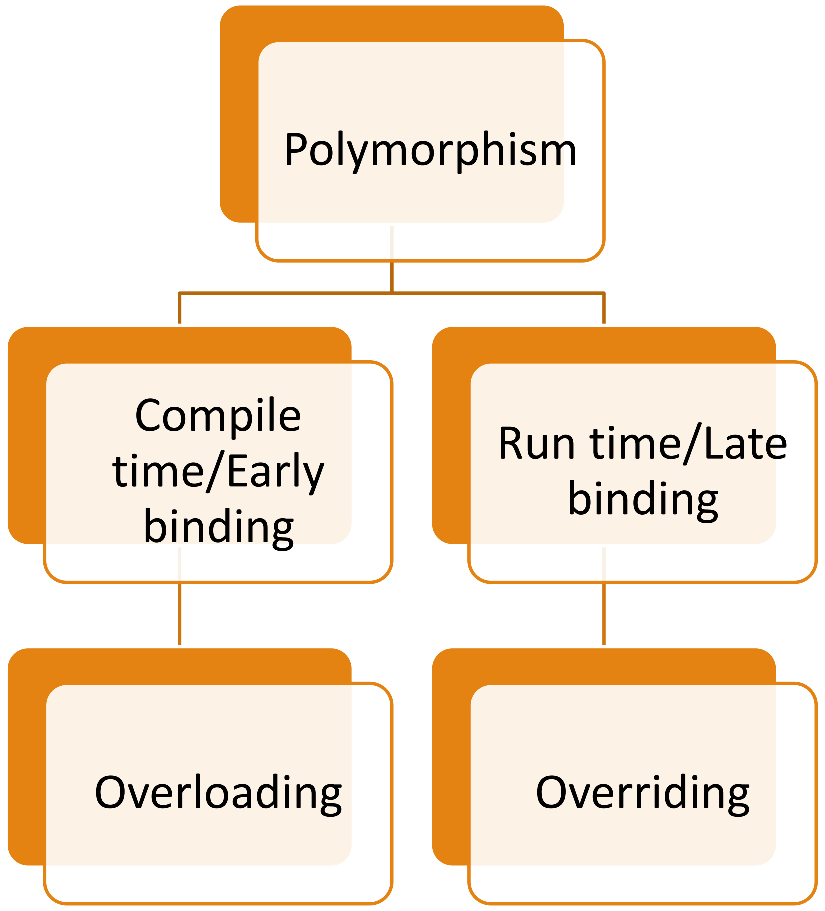
Polymorphism Importance

- Polymorphism saves the programmer a lot of time in re-creating code.
- You don't want to have to write completely different modules for every possible permutation.
- For example, if you had methods for tree growth, it would be hard to have to write a specific growth method for maple, spruce, pine, etc. Instead, you can have a growth function that spans across all tree types.

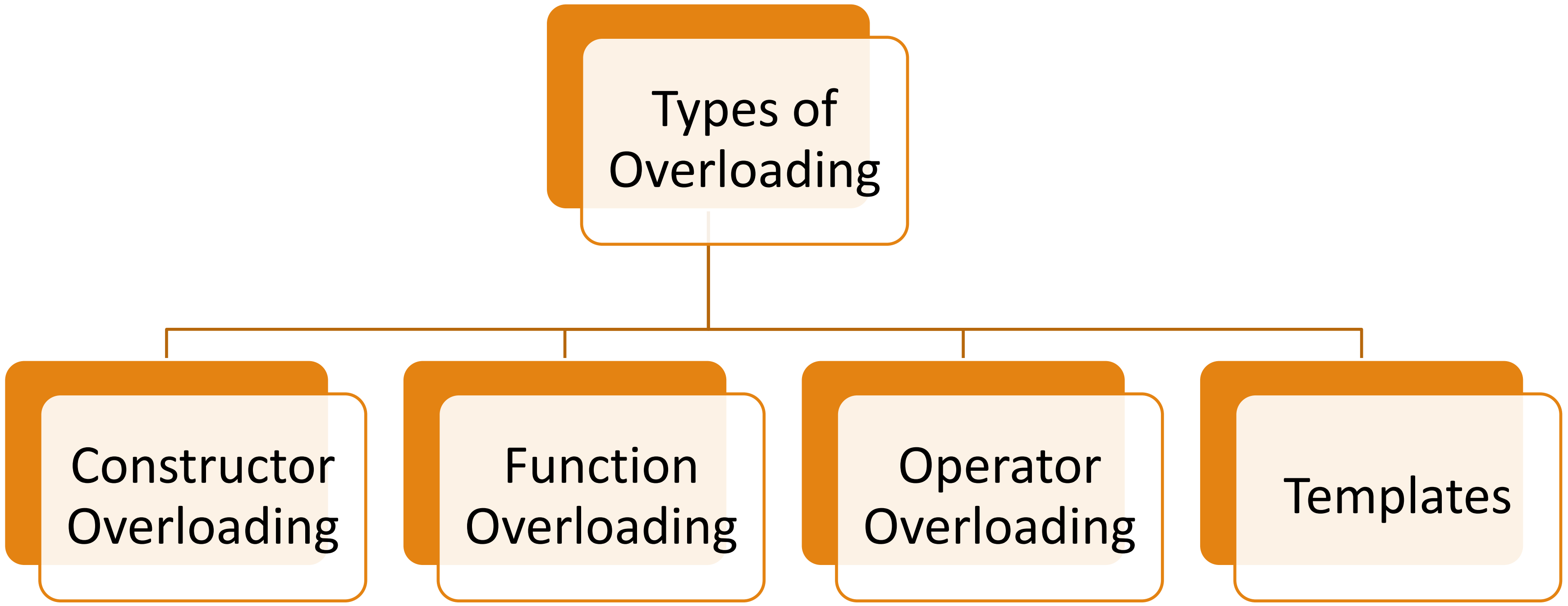
Difference between Inheritance & Polymorphism

S.NO	INHERITANCE	POLYMORPHISM
1.	Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class).	Whereas polymorphism is that which can be defined in multiple forms.
2.	It is basically applied to classes.	Whereas it is basically applied to functions or methods.
3.	Inheritance supports the concept of reusability and reduces code length in object-oriented programming.	Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).
4.	Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance.	Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding).

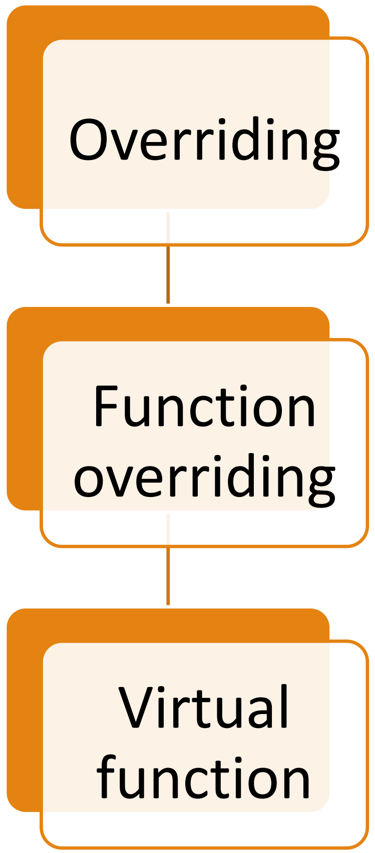
Types of polymorphism:



Compile time / Static / Early Binding polymorphism



Run time / Dynamic / Late Binding polymorphism



static : memory allocated at compile time

In this case, during compile time when the binding happens, compiler links the method definition with the method call. So it gets resolved during compile time

dynamic : memory allocated at run time

Overloading in C++

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as method overloading.

Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different parameter signatures
- The compiler selects the proper function to call by examining the number, types and order of the arguments in the call
- Function overloading allows the program to follow the principle of *"Polymorphism"*
- The *return type* or *name* of parameters don't help the compiler in selecting which overloaded function to call
- Overloading the functions that perform closely related tasks can make programs more readable and understandable

Example

```
void findPerson(string name) { . . . }

void findPerson(int ID) {  . . . }

void findPerson(int ID, string addr)  { . . . }

void findPerson(string addr, int ID) { . . .  }
```

All of above are valid overloaded functions

Example

```
 void showVal(int a)
{  cout << a;  }

 void showVal(char a)
{  cout << a;  }

 void showVal(float a)
{  cout << a;  }
```

```
int main()
{
        showVal('M');
        showVal(20);
        showVal(3.5);
}
```

Can we overload a function on const and non const keyword?

A const member function can be overloaded with a non-const version. The compiler chooses which overloaded member function to use based on the object on which the function is invoked. If the object is const,the compiler   uses the const version. If the object is not const, the compiler uses the non-const version.

Note:Invoking a non-const member function on a const object is a compilation error

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

```
class Test
{
protected:
        int x;
public:
        Test (int i):x(i) { }
        void fun() const
        {
                cout << "fun() const called " << endl;
        }
        void fun()
        {
                cout << "fun() called " << endl;
        }};

int main(){
        Test t1 (10);
        const Test t2 (20);
        t1.fun();
        t2.fun();
        return 0;
}
```

```
fun() called
fun() const called
```

The two methods ‘void fun() const’ and ‘void fun()’ have same signature except that one is const and other is not. Also, if we take a closer look at the output, we observe that, ‘const void fun()’ is called on const object and ‘void fun()’ is called on non-const object.

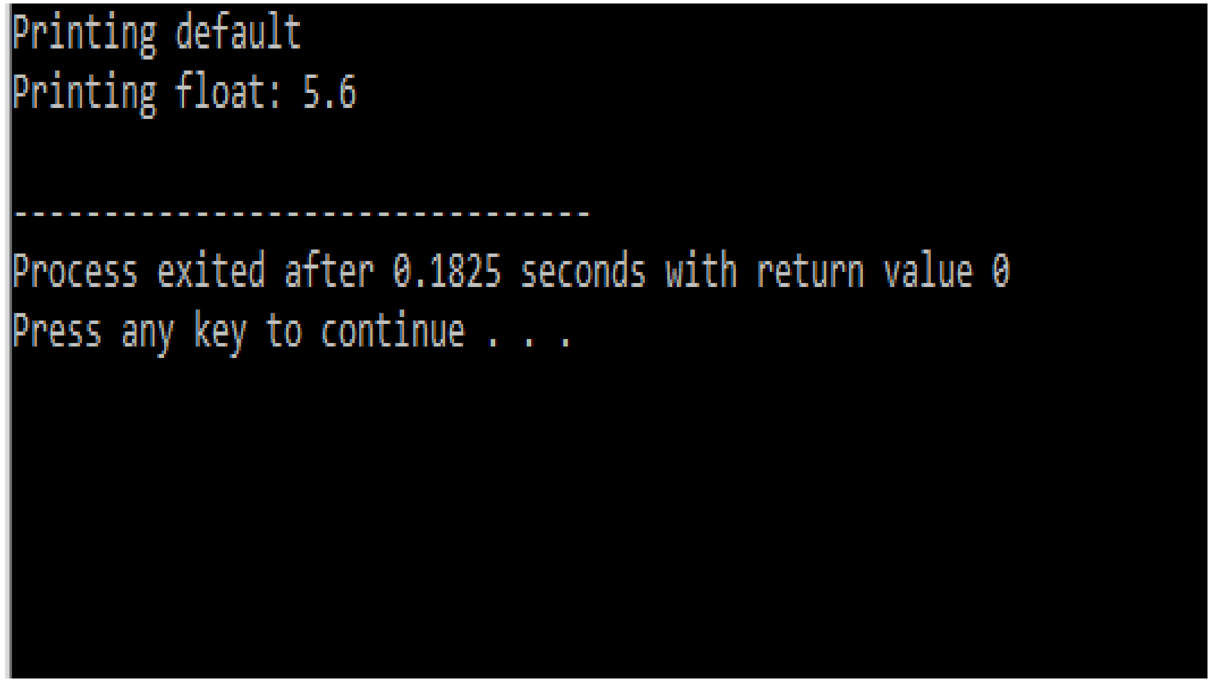
Pitfall

Avoid using default arguments with overloaded functions as it can be dangerous i.e. can lead to error conditions that are hard to trace

For example:

In a program, a function that takes no argument and another function of the same name that takes all default arguments

```
#include <iostream>
using namespace std;
class printData {
   public:
      void print() {
         cout << "Printing default "  << endl;}
      void print(float f) {
         cout << "Printing float: " << f << endl; }};
int main(void) {
   printData pd;
   pd.print(); // Call print to print default
   pd.print(5.6); // Call print to print float
   return 0;}
```



```
Printing default
Printing float: 5.6


--------------------------------
Process exited after 0.1825 seconds with return value 0
Press any key to continue . . .
```

```
#include <iostream>
using namespace std;
class printData {
   public:
      void print() {
         cout << "Printing default "  << endl;   }
      void print(float f=0) {
         cout << "Printing float: " << f << endl;   }};
int main(void) {
   printData pd;
   pd.print();  //[Error] call of overloaded 'print()' is ambiguous
   pd.print(5.6);
   return 0;}
```

Is Function Overloading supported in C?

Any object code produced by compilers is usually linked with other pieces of object code by a type of program called a linker. The linker needs a great deal of information on each program entity. For example, to correctly link a function it needs its name, the number of arguments and their types, and so on. These requirements enable use of function overloading in C++. C, only distinguished the pieces of code by their name that’s why function overloading is not supported by C

Name Mangling

How does C++ compiler distinguishes between different functions when it generates object code – it changes names by adding information about arguments. This technique of adding additional information to function names is called <u>Name Mangling</u>.

Name Mangling

- ✓The compiler encodes each function identifier with the number and types of its parameters
- ✓This is called **name mangling** or **name decoration**

Name Mangling

C++ standard doesn’t specify any particular technique for name mangling, so different compilers may append different information to function names.

g++ name_decoration / Name Mangling

- Each mangled name (other than main) begins with two underscores (__) followed by the letter Z, a number and the function name

- The number that follows Z specifies how many characters are in the function’s name

- The function name is then followed by an encoding of its parameter list

Name Mangling

```
void func(int i, char c) { . . . }

int func2(string x, float y) { . . . }

void func3(int a, float b, int& c) { . . . }
```

Mangled Names:

```
__4Zfuncic

__5Zfunc2sf

__5Zfunc3ifRi
```

Question

You cannot overload function declarations that differ only by return type.

WHY?

Answer

Return type of functions is not a part of the mangled name which is generated by the compiler for uniquely identifying each function in the case of function overloading.

- No of arguments
- Type of arguments &
- Sequence of arguments

are the parameters which are used to generate the unique mangled name for each function. It is on the basis of these unique mangled names that compiler can understand which function to call even if the names are same.

Can we overload functions on const and non const parameters?

```
#include <iostream>
using namespace std;
class base{
        public:
        void func(int a){
                        cout << "inside base"<< endl;           }
          void func(const int a){
                        cout << "inside base"<< endl;           }
};
        int main(){
        base b;
        b.func(3) ;  // error  can’t be overloaded
}
```

C++ Overload Static Function with Non-Static Function ????

Void print(){}

Static void print (){}

- No, it is directly prohibited by the standard:
- ISO 14882:2003 C++ Standard 13.1/2 – Overloadable declarations
- Certain function declarations cannot be overloaded:
- Function declarations that differ only in the return type cannot be overloaded.
- Member function declarations with the same name and the same parameter types
- cannot be overloaded if any of them is a static member function declaration

Question

Can we overload main() function in C++?

- in c++ global main cannot be overloaded it will generate compile time error, in is because you cannot have multiple entry point for the same program since when you run the program the program will not know where to begin
- But if you write main function inside a class then it will compile fine however it will not be treated as program entry point

Exercise

Create a **Message** class with a constructor that takes a single **string** with a default value. Create a private member **string**, and in the constructor simply assign the argument **string** to your internal **string**. Create two overloaded member functions called **print( )**: one that takes no arguments and simply prints the message stored in the object, and one that takes a **string** argument, which it prints in addition to the internal message. Also write name mangling for these overloaded methods

Method Overriding in C++

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

Requirements for Overriding a Function

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.

2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

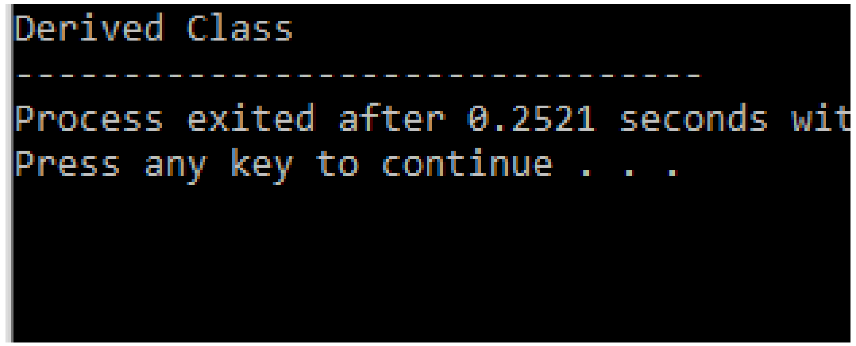
Early binding

```
class A {
   public:
   void display() {
      cout<<"Base class";  } };
class B:public A {
   public:
   int display() {
      cout<<"Derived Class";
               return 1; } };
int main() {
   B obj;
   obj.display();
   return 0;  }
```

Early binding

- In the above example, we are calling the overridden function using derived class object. Compiler gives preference to object type.
- Base class object will call base version of the function and derived class's object will call the derived version of the function. This is called early binding
- In early binding return type and storage classes are ignored

Early binding



```
Derived Class
--------------------------------
Process exited after 0.2521 seconds wit
Press any key to continue . . .
```

Function Overloading VS Function Overriding

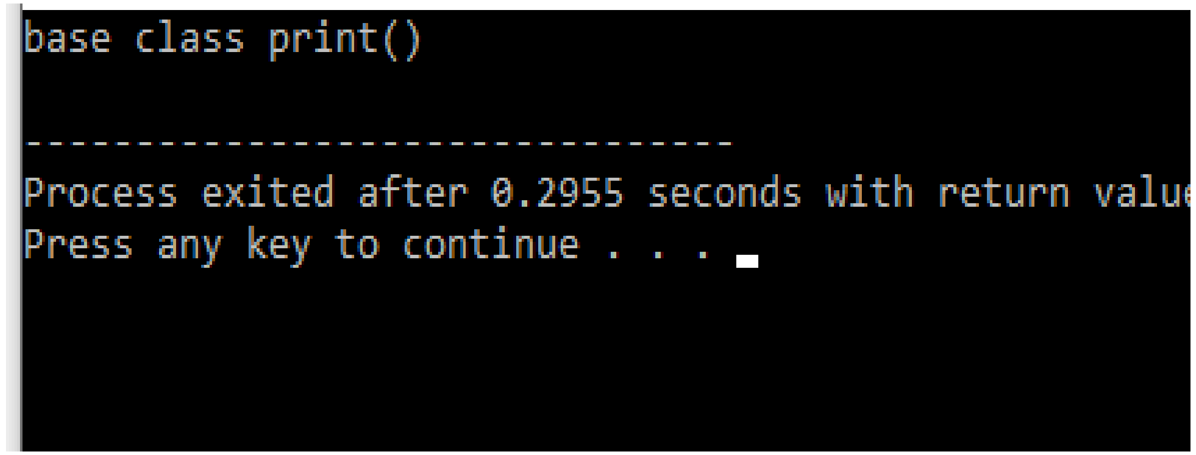
1. **Inheritance:** Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.
2. **Function Signature:** Overloaded functions must differ in function signature ie either number of parameters or type of parameters should differ. In overriding, function signatures must be same.
3. **Scope of functions:** Overridden functions are in different scopes; whereas overloaded functions are in same scope.
4. **Behavior of functions:** Overriding is needed when derived class function has to do some added or different job than the base class function. Overloading is used to have same name functions which behave differently depending upon parameters passed to them.

Runtime Polymorphism

Compile-time(early binding)

```
class Base {
   public:
    void print() {
    cout << "base class print()" << endl; }};
class Derived : public Base {
   public:
    void print() {
        cout << "derived class print()" << endl;}};
int main() {
    Derived derived1;
    Base* base1 = &derived1; //up casting
    base1->print();}
```

Compile-time(early binding)



```
base class print()


--------------------------------
Process exited after 0.2955 seconds with return value
Press any key to continue . . .
```

Problem

we aimed a base-class pointer at a derived-class object, then invoked member function print through that pointer. but the type of the handle(pointer) determines which class’s functionality to invoke. In that case, the Base class pointer invoked the member function print on the derived-class object, even though the pointer was aimed at a derived-class object that has its own customized print function.

Solution

With virtual functions, the type of the object being pointed to, not the type of the handle, determines which version of a virtual function to invoke.

virtual function

A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you 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 expression used to make the function call.

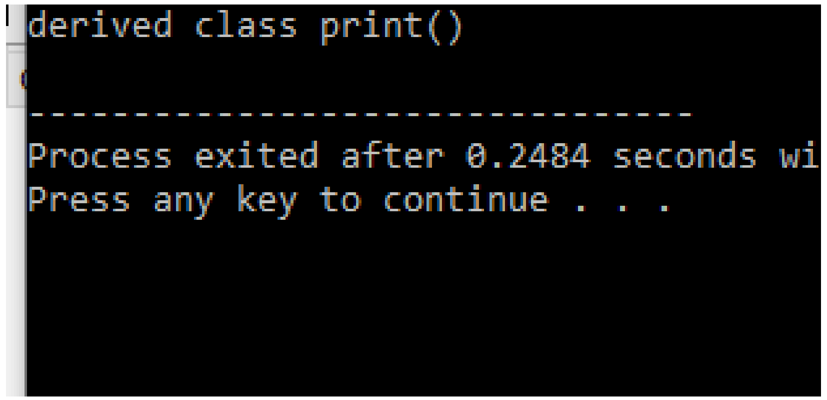
virtual function

- 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 Run-time.

virtual function

```
class Base {
  public:
   virtual void print() {
   cout << "base class print()" << endl; }};
class Derived : public Base {
  public:
   void print() {
       cout << "derived class print()" << endl;}};
int main() {
   Derived derived1;
   Base* base1 = &derived1;
   base1->print();
}
```

virtual function



```
derived class print()

--------------------------------
Process exited after 0.2484 seconds wi
Press any key to continue . . .
```

Runtime polymorphism is achieved only through a pointer (or reference) of base class type

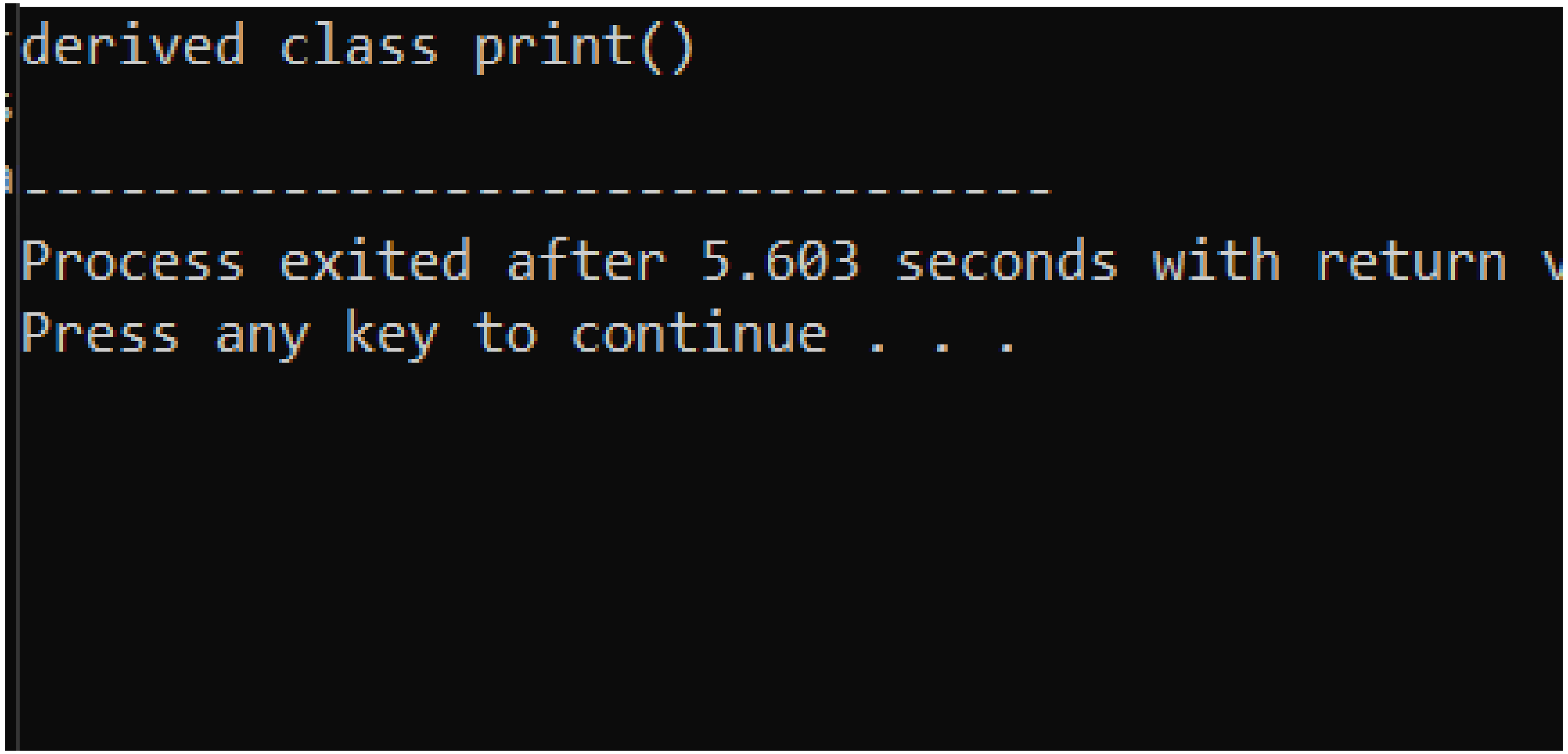
Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer

When calling a function using pointers or references, the following rules apply:

1. Functions in derived classes override virtual functions in base classes only if their type is the same. A function in a derived class cannot differ from a virtual function in a base class in its return type only; the argument list must differ as well.
2. A call to a virtual function is resolved according to the underlying type of object for which it is called.
3. A call to a nonvirtual function is resolved according to the type of the pointer or reference.
4. The virtual keyword can be used when declaring overriding functions in a derived class, but it is unnecessary; overrides of virtual functions are always virtual.

Rule 4

```
class Base {
   public:
    virtual void print() {
    cout << "base class print()" << endl; }};
  class Derived : public Base {
   public:
   virtual void print() { // virtual keyword with derived version can be used
        cout << "derived class print()" << endl;}};
  int main() {
    Derived derived1;
    Base* base1 = &derived1;
    base1->print();}
```



```
derived class print()

--------------------------------
Process exited after 5.603 seconds with return v
Press any key to continue . . .
```

Derived-Class Member-Function Calls via Base-Class Pointers

Off a base-class pointer, the compiler allows us to invoke only base-class member functions. Also called **object slicing**

Thus, if a base-class pointer is aimed at a derived-class object, and an attempt is

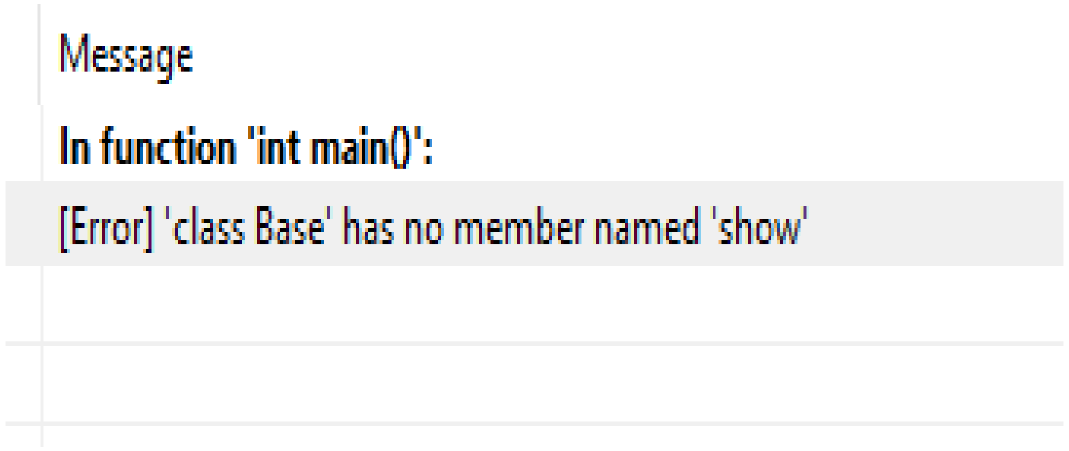
made to access a ***derived-class-only member function***, a compilation error will occur.

The compiler generates errors on these calls, because they’re not made to member functions of base-class

Derived-Class Member-Function Calls via Base-Class Pointers

```
class Base {
   public:
    virtual void print() {
    cout << "base class print()" << endl; }};
class Derived : public Base {
   public:
    void print() {
        cout << "derived class print()" << endl;}
    void show() {
        cout << "only derived class member function" << endl;}};
int main() {
    Derived derived1;
    Base* base1 = &derived1;
    base1->print();
    base1->show();}
```

Derived-Class Member-Function Calls via Base-Class Pointers



```
Message
In function 'int main()':
[Error] 'class Base' has no member named 'show'
```

Can I override static function C++?

Can I override static function C++?

Static methods cannot be overridden because method overriding only occurs in the context of dynamic (i.e. runtime) lookup of methods. Static methods (by their name) are looked up statically (i.e. at compile-time).

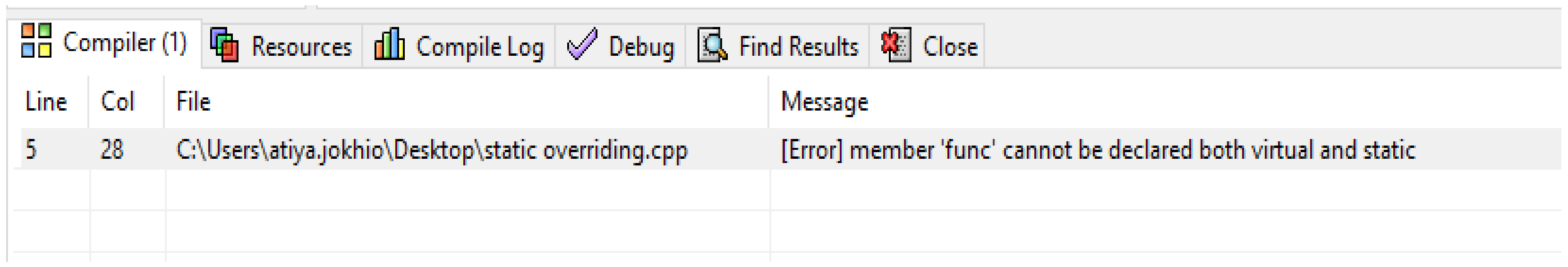
Early binding

```
#include <iostream>
using namespace std;
class base{
		public:
		 static void func(){
				cout << "inside base"<< endl;
}};
class child: public base{
		public:
		void func(){
				cout << "inside child"<< endl;
}};

int main(){
		base* b ;
		child c;
		b = &c;
		b -> func();
}
```

Static in run time polymorphism

```
#include <iostream>
using namespace std;
class base{
                public:
                 virtual static void func(){
                                cout << "inside base"<< endl;
}};
class child:public base{
                public:
                 void func(){
                                cout << "inside child"<< endl;
}};
int main(){
                base* b ;
                child c;
                b = &c;
                b -> func();
}
```



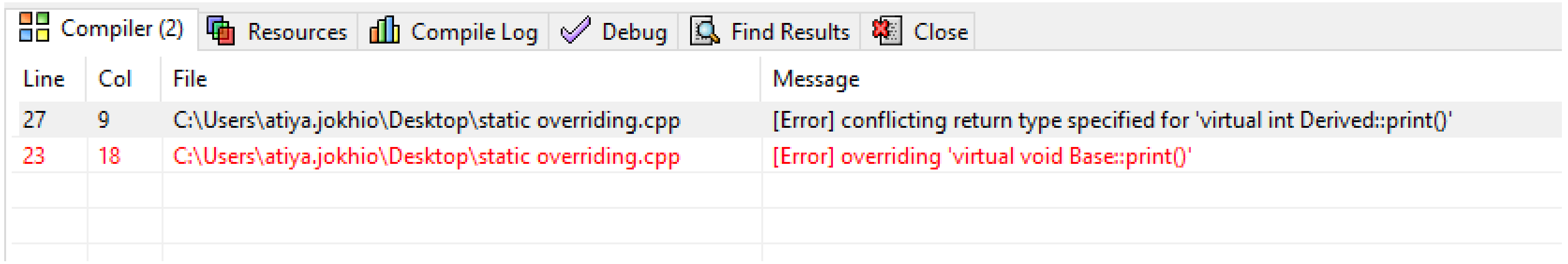
```
Line  Col  File                                               Message
5     28   C:\Users\atiya.jokhio\Desktop\static overriding.cpp  [Error] member 'func' cannot be declared both virtual and static
```

Important point

A derived class virtual function is only considered overridden if its signature and return type must match exactly

A derived class virtual function is only considered overridden if its signature and return type must match exactly

```
class Base {
  public:
   virtual void print() {
   cout << "base class print()" << endl; }};
class Derived : public Base {
  public:
   int print() {
       cout << "derived class print()" << endl;
                                           return 1;}
                                           };
int main() {
   Derived derived1;
   Base* base1 = &derived1 ; //up casting
   base1->print();
                  }
```



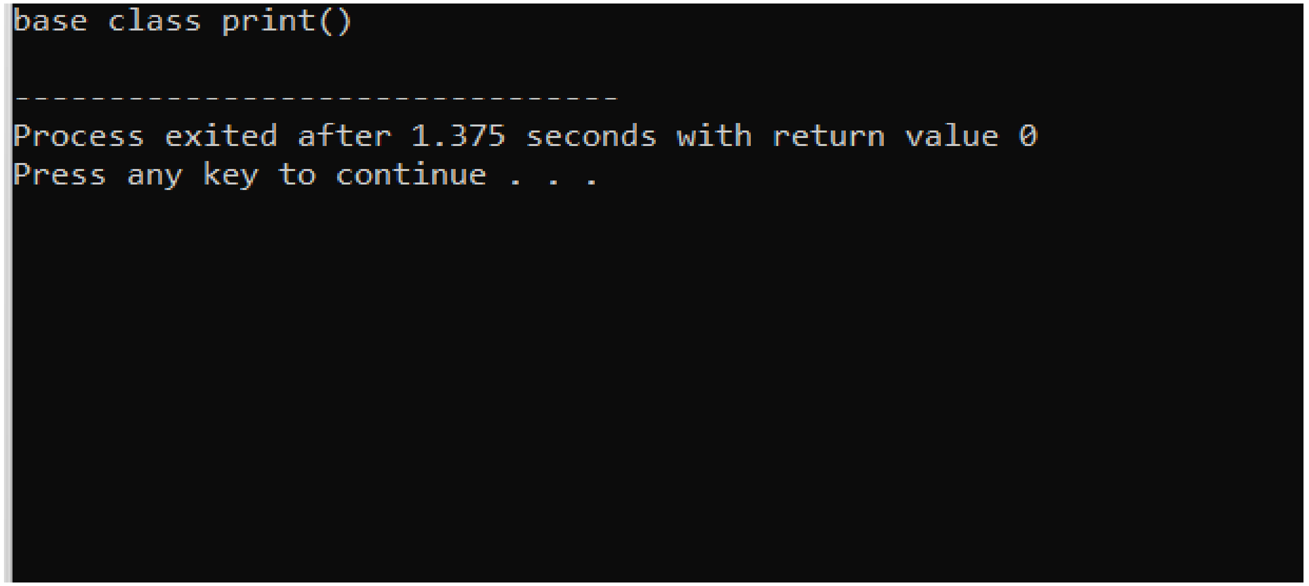
Line	Col	File	Message
27	9	C:\Users\atiya.jokhio\Desktop\static overriding.cpp	[Error] conflicting return type specified for 'virtual int Derived::print()'
23	18	C:\Users\atiya.jokhio\Desktop\static overriding.cpp	[Error] overriding 'virtual void Base::print()'

can we override const method with non const in c++

It is neither overloading nor overriding. Rather, it is *hiding*.

const is part of the signature, and **leaving it off changes the signature, and thus hides the method rather than overrides it**. "

```
class Base {
   public:
     virtual void print() const{
    cout << "base class print()" << endl; }};
class Derived : public Base {
   public:
    void print() {
        cout << "derived class print()" << endl;
                                            }};
int main() {
    Derived derived1;
    Base* base1 = &derived1 ; //up casting
    base1->print();
                    }
```



```
base class print()

--------------------------------
Process exited after 1.375 seconds with return value 0
Press any key to continue . . .
```

Virtual Destructor

- Deleting a derived class object using a pointer to a base class, the base class should be defined with a virtual destructor.

Virtual Destructor

```
#include<iostream>

using namespace std;

class base {
public:
        base()
        { cout<<"Constructing base \n"; }
        ~base()
        { cout<<"Destructing base \n"; }
};
```

```
class derived: public base {
public:
        derived()
        { cout<<"Constructing derived \n"; }
        ~derived()
        { cout<<"Destructing derived \n"; }
};

int main(void)
{
derived *d = new derived();
base *b = d;
delete b;
return 0;
}
```

```
Constructing base
Constructing derived
Destructing base
```

Virtual Destructor

```
#include<iostream>

using namespace std;

class base {
public:
        base()
        { cout<<"Constructing base \n"; }
        virtual ~base()
        { cout<<"Destructing base \n"; }
};
```

```
class derived: public base {
public:
        derived()
        { cout<<"Constructing derived \n"; }
        ~derived()
        { cout<<"Destructing derived \n"; }
};

int main(void)
{
derived *d = new derived();
base *b = d;
delete b;
return 0;
}
```

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```