


Binding in C++



POWERED BY  NINJA3

By – Mohammad Imran

Concept of Binding

When we compile our program, all variables and functions are mapped to addresses in memory. These addresses are then used for reference in the program. This process of converting variables and functions into addresses is known as **binding**.

Binding is a part of polymorphism that we have discussed earlier. Static polymorphism is known as static binding or early binding while dynamic polymorphism is known as dynamic binding or late binding.

Concept of Binding

Binding helps in **linking a bridge** between a function call and its corresponding function definition.

Before getting to static binding & dynamic binding, we need to understand the concept of binding. When we create a function, we have two crucial things:-

- ✓ A function definition - defines a procedure to execute.
- ✓ A function call - invokes the respective function for implementation.

Now both the function definition and function calls are stored in the memory at **separate addresses**. And our program can have more than one function for its smooth operation. Hence, we need a technique to match the appropriate function call with its definition.

Concept of Binding

Needs

Before looking at static binding & dynamic binding, let us understand the need for binding with a practical example. Suppose we have a class bind with two member functions. The class definition, along with the driver code, is as follows :

Concept of Binding

Example

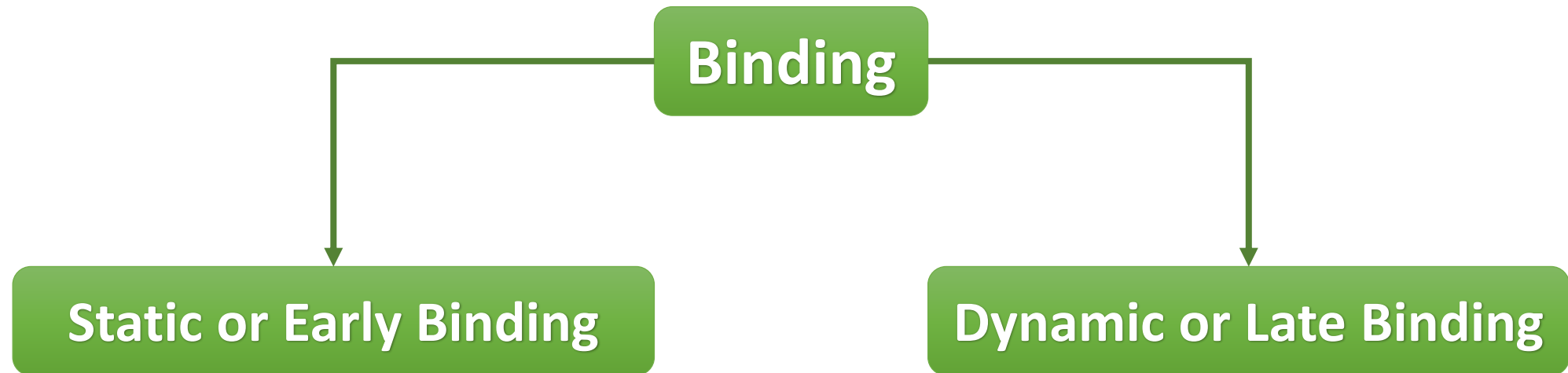
```
#include <iostream> //Static Binding
using namespace std;
class Bind
{
public:
    void fun1()
    {
        cout << "Hi! This is fun1.\n";
    }
    void fun2()
    {
        cout << "Hi! This is fun2.\n";
    }
};
```

```
int main()
{
    bind obj;
    obj.fun1();
    obj.fun2();
    return 0;
}
```

Concept of Binding

Types of Binding

The process of matching a specific function call to its respective function definition is known as **binding**. But why is this of such interest? A lot can be utilized using the two kinds of binding: **static binding** & **dynamic binding**.



Early Binding

Early binding is also called **static binding**. Early binding occurs when we make the explicit or direct function call in our program.

When the compiler encounters a function call in the program, it replaces that function call with a machine language instruction to go to that function.

Therefore, when the point of execution comes in that function call line, it reads that instruction and then jumps to the function given by memory address.

Early Binding

Binding at compile time is known as **static binding**.

There is always a default setting for any tool or method. For binding, static binding in C++ is the default route. Static binding ensures linking the function call and its function definition at compile-time only. It is also why it is **synonymous** with **compile-time binding** or **early binding**.

But how does the linkage take place? Before getting into a practical example, there are two ways by which static binding can be achieved: **function overloading** & **operator overloading**.

Early Binding

Advantages

- ✓ It performs better than a **dynamic binding** in C++. The compiler knows before runtime about all the methods of the class & is aware of which ones can or can't be overridden. Hence, it is easier for the compiler to associate the objects with their respective class(es). It concludes with not needing an extra overhead.
- ✓ Better performance also results in being more efficient and faster.

NOTE: The downside of static binding is that it reduces flexibility. All information about the values of parameters and function calls is predefined and cannot be changed at runtime.

Early Binding

Function Overloading

Example - 1

```
#include <iostream>
using namespace std;
class func
{
public:
    void stat(int a, int b)
    {
        cout << "No of Parameter: " << a+b;
    }
    void stat(int a, int b, int c)
    {
        cout << "No of Parameter: " << a+b+c;
    }
};
```

Early Binding

Function Overloading

Example - 1

```
int main()
{
    func obj;
    int a, b, c;
    a = 1;
    b = 1;
    c = 1;
    obj.stat(a, b, c);
    cout << endl;
    obj.stat(a, b); return 0;
}
```

OUTPUT

No of Parameter 3
No of Parameter 2

Early Binding

Operator Overloading

Example - 2

```
#include <iostream>
using namespace std;
class Over_num
{
    int x, y;
public:
    void input();
    void input2();
    Over_num operator + (Over_num & ob);
    void print();
};
void Over_num::input()
{
    cout << " Enter first number: ";
    cin >> x;
}
```

```
void Over_num::input2()
{
    cout << " Enter second number: ";
    cin >> x;
}
void Over_num::print()
{
    cout << "Sum of two numbers = " << x;
}
Over_num Over_num::operator + (Over_num &ob)
{
    Over_num A;
    A.x = x + ob.x;
    return (A);
}
```

Early Binding

Operator Overloading

Example - 2

```
int main()
{
    Over_num x1, y1, res;
    x1.input();
    y1.input2();
    res = x1 + y1;
    res.print();
    cout << endl;
    return 0;
}
```

OUTPUT

```
Enter first number:22
Enter second number: 11
Sum of two number = 33
```

Early Binding

Using Pointer Object

By default early binding happens in C++.

Here is CPP Program to illustrate early binding. Any normal function call (without virtual) is binded early. Here we have taken base and derived class example so that readers can easily compare and see difference in outputs.

The function call decided at compile time (compiler sees type of pointer and calls base class function).

Early Binding

Using Pointer Object

Example - 3

```
#include<iostream>
using namespace std;
class Base
{
    public:
    void show()
    {
        cout << " In Base \n";
    }
};
```

OUTPUT

In Base

```
class Derived: public Base
{
    public:
    void show()
    {
        cout << "In Derived \n";
    }
};
int main(void)
{
    Base *bp = new Derived;
    bp -> show(); //Static Binding
    return 0;
}
```


Late Binding

This is run time polymorphism. In this type of binding the compiler adds code that identifies the object type at runtime then matches the call with the right function definition. This is achieved by using virtual function.

Binding at **runtime** is known as **dynamic binding**. **Late binding** is also called **dynamic binding**. Late binding occurs when we make implicit or indirect function calls in our program. An example of this is using **function pointers** or **virtual functions** when using classes. Here, the function call's memory reference is not determined at compile-time, but rather at run-time.

Virtual Function

- ✓ A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- ✓ It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- ✓ There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.

Virtual Function

- ✓ A '**virtual**' is a keyword preceding the normal declaration of a function.
- ✓ When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Virtual Function

Rules

- ✓ Virtual functions must be members of some class.
- ✓ Virtual functions cannot be static members.
- ✓ They are accessed through object pointers.
- ✓ They can be a friend of another class.
- ✓ A virtual function must be defined in the base class, even though it is not used.

Virtual Function

Rules

- ✓ The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- ✓ We cannot have a virtual constructor, but we can have a virtual destructor
- ✓ Consider the situation when we don't use the virtual keyword.

Late Binding

Example - 3

```
#include<iostream>
using namespace std;
class Base
{
    public:
    virtual void show()
    {
        cout << " In Base \n";
    }
};
```

OUTPUT

In Derived

```
class Derived: public Base
{
    public:
    void show()
    {
        cout << "In Derived \n";
    }
};
int main(void)
{
    Base *bp = new Derived;
    bp -> show(); //Dynamic Binding
    return 0;
}
```

Late Binding

Example - 4

```
#include <iostream>
using namespace std;
class A
{
public:
    virtual void display()
    {
        int x = 5;
        cout << "X = " << x << endl;
    }
};
```

OUTPUT

Y = 10

```
class B : public A
{
public:
    void display()
    {
        int y = 10;
        cout << "Y = " << y << endl;
    }
};
int main()
{
    A *a = new A;
    B b;
    a = &b;
    a -> display();
    return 0;
}
```

Late Binding

Using Function Pointer

Example - 5

```
#include<iostream>
using namespace std;
void display(int x)
{
    cout << x;
}

int main()
{
    void (*ptrFunc)(int) = display;
    ptrFunc(5);
    return 0;
}
```

OUTPUT

5

Pure Virtual Function

A pure virtual function in C++ is a virtual function for which we do not have an implementation. We do not write any functionality in it. Instead, we only declare this function. A pure virtual function does not carry any definition related to its base class. A pure virtual function is declared by assigning a **zero (0)** in its declaration. Any class containing one or more pure virtual functions can not be used to define any object. For this reason, these classes are known as abstract classes. Classes derived from abstract classes need to implement the pure virtual functions of these classes.

Pure Virtual Function

- ✓ A virtual function is not used for performing any task. It only serves as a placeholder.
- ✓ When the function has no definition, such function is known as "**do-nothing**" function.
- ✓ The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- ✓ A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.

Pure Virtual Function

- ✓ The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

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

Pure Virtual Function

Example - 5

```
#include <iostream>
using namespace std;
class A
{
    public:
        virtual void show() = 0;
};
class B : public A
{
    public:
        void show()
        {
            cout << "B derived from A" << endl;
        }
};
```

OUTPUT

B derived from A

```
int main()
{
    A *aptr = new A;
    B b1;
    aptr = &b1;
    aptr->show();
    return 0;
}
```

By – Mohammad Imran

Concept of Binding

Constructor is a class member function with the same name as the class name. The main job of the constructor is to allocate memory for class objects. Constructor is automatically called when the object is created. It is very important to understand how constructors are called in inheritance.

We know when we create an object, then automatically the constructor of the class is called and that constructor takes the responsibility to create and initialize the class members. Once we create the object, then we can access the data members and member functions of the class using the object.

Coding Questions

By – Mohammad Imran

Question - 1

In a C++ program designed to perform basic arithmetic operations, you have two classes: **Addition** and **Multiplication**. The **Addition** class takes two integers by argument constructor and computes their sum, while the **Multiplication** class inherits from **Addition** and computes the product of the same two integers using argument constructor while we created only derived class object.

[Click here to see code](#)
By – Mohammad Imran

QUIZ

By – Mohammad Imran

Quiz - 1

```
class A
{
    public:
        void foo()
        {
            cout << "A::foo()" << endl;
        }
};

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

```
class C : public B
{
    public:
        void bar()
        {
            foo();
        }
};

int main()
{
    C obj;
    obj.bar();
    return 0;
}
```

OUTPUT

B::foo()