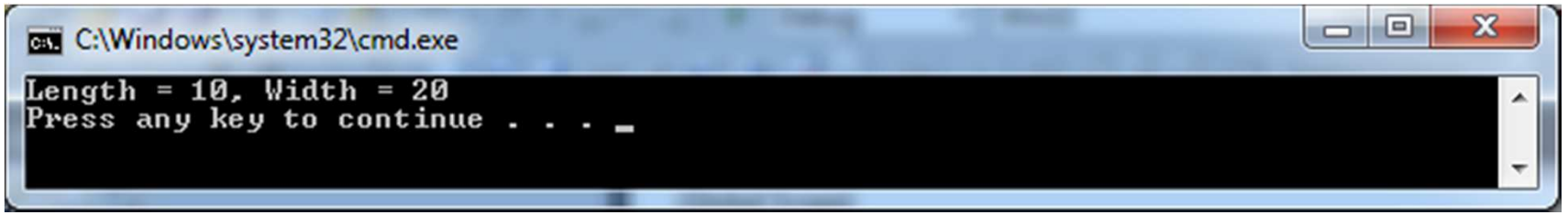
```
#ifndef RECTANGLE2D_H
#define RECTANGLE2D_H

#include <iostream>
using namespace std;

template <class Type>
class Rectangle2D
{
public:
Rectangle2D (Type len = 0, Type wth = 0);
// The class does not have pointers (dynamic memory), therefore, you dont need to write your own
// copy constructor, assignment operator overloading, and destrcutor

void Print () const;
private:
Type length, width;
};

template <class Type>
Rectangle2D<Type>::Rectangle2D (Type len, Type wth)
: length (len), width (wth)
{
}

template <class Type>
void Rectangle2D<Type>::Print () const
{
cout<<"Length = "<<length<<", Width = "<<width;
}
#endif
```

```
#ifndef RECTANGLE3D_H
#define RECTANGLE3D_H

#include "Rectangle2D.h"

template <class Type>
class Rectangle3D : public Rectangle2D<Type>
{
public:
Rectangle3D (Type len = 0, Type wth = 0, Type hg = 0);
// The class does not have pointers (dynamic memory), therefore, you dont need to write your own
// copy constructor, assignment operator overloading, and destrcutor

void Print () const;
private:
Type height;
};

template <class Type>
Rectangle3D<Type>::Rectangle3D (Type len, Type wth, Type hg)
: Rectangle2D (len, wth), height (hg)
{
}

template <class Type>
void Rectangle3D<Type>::Print () const
{
Rectangle2D::Print ();
cout<<", Height = "<<height<<endl;
}
#endif
```

```
#include "Rectangle2D.h"
#include "Rectangle3D.h"

void main ()
{
	Rectangle2D<int>* pRec2D = new Rectangle2D<int> (10, 20);
	pRec2D->Print ();
	cout<<endl<<endl;

	Rectangle3D<int>* pRec3D = new Rectangle3D<int> (50, 60, 70);
	pRec3D->Print ();
}
```

C++ allows the user to pass an object of a derived class to a formal parameter of the base class type.

```
#include "Rectangle2D.h"
#include "Rectangle3D.h"

void main ()
{
        // a pointer of base class points to an object of derived class
        Rectangle2D<int>* pRec3DD = new Rectangle3D<int> (10, 20, 30);
        pRec3DD->Print ();
        cout<<endl;
}
```



Make Print () method virtual

virtual void Print () const;

```
#include "Rectangle2D.h"
#include "Rectangle3D.h"

void main ()
{
        // a pointer of base class points to an object of derived class
        Rectangle2D<int>* pRec3DD = new Rectangle3D<int> (10, 20, 30);
        pRec3DD->Print ();
        cout<<endl;
}
```

```
#ifndef BASE_H
#define BASE_H

class Base
{
public:
        Base (int = 0);

        void Display () const;
        virtual void Print () const;

private:
        int x;
};

#endif
```

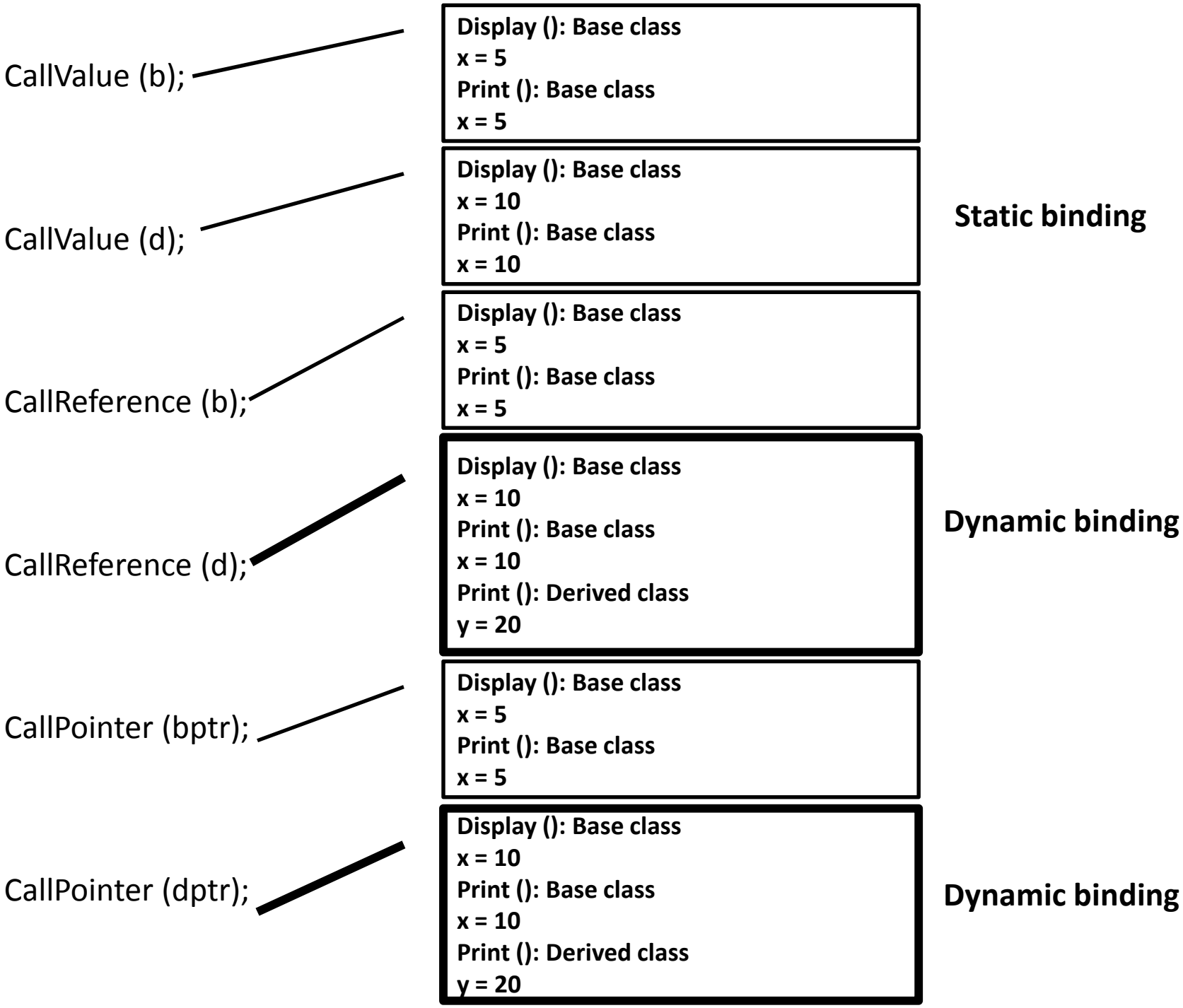
```
#include <iostream>
using namespace std;

#include "Base.h"

Base::Base (int a)
: x (a)
{
}

void Base::Display () const
{
        cout<<"Dispaly (): Base class\n x =
"<<x<<endl;
}

void Base::Print () const
{
        cout<<"Print (): Base class\n x =
"<<x<<endl;
}
```

```
#ifndef DERIVED_H
#define DERIVED_H

#include "Base.h"

class Derived : public Base
{
public:
        Derived (int = 0, int = 0);

        void Display () const;
        virtual void Print () const;

private:
        int y;
};

#endif
```

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

#include "Derived.h"

Derived::Derived (int a, int b)
: Base (a), y (b)
{
}

void Derived::Display () const
{
        Base::Display ();
        cout<<"Dispaly (): Derived class\n y =
"<<y<<endl;
}

void Derived::Print () const
{
        Base::Print ();
        cout<<"Print (): Derived class\n y =
"<<y<<endl;
}
```

```
#include "Base.h"
#include "Derived.h"

void CallValue (Base b);
void CallReference (Base& b);
void CallPointer (Base* b);

void main ()
{
        Base b (5);
        Derived d (10, 20);

        CallValue (b);
        CallValue (d);

        CallReference (b);
        CallReference (d);

        Base* bptr = new Base (5);
        Derived* dptr = new Derived  (10, 20);

        CallPointer (bptr);
        CallPointer (dptr);
}
```

```
void CallValue (Base b)
{
        b.Display ();
        b.Print ();
}

void CallReference (Base& b)
{
        b.Display ();
        b.Print ();
}

void CallPointer (Base* b)
{
        b->Display ();
        b->Print ();
}
```


CallValue (b);
Display (): Base class
x = 5
Print (): Base class
x = 5
CallValue (d);
Display (): Base class
x = 10
Print (): Base class
x = 10
Static binding
CallReference (b);
Display (): Base class
x = 5
Print (): Base class
x = 5
CallReference (d);
Display (): Base class
x = 10
Print (): Base class
x = 10
Print (): Derived class
y = 20
Dynamic binding
CallPointer (bptr);
Display (): Base class
x = 5
Print (): Base class
x = 5
CallPointer (dptr);
Display (): Base class
x = 10
Print (): Base class
x = 10
Print (): Derived class
y = 20
Dynamic binding

Static vs Dynamic Binding

- In **compile-time binding**, the necessary code to call a specific function is generated by the compiler. (Compile-time binding is also known as **static binding** or **early binding**.)

- The binding of virtual functions occurs at program execution time, not at compile time. This kind of binding is called **run-time binding** or **late binding.**

Abstract Class

- Abstract class has at least one pure virtual function
- A pure virtual function has initializer **= 0**

  e.g. virtual void Print () const = 0;
- A pure virtual function doesn't have implementation and must be overridden in derived classes
- No objects can be created of an Abstract class
- Abstract classes are merely used as base classes for some other classes.

```
class Number
{
public:
        virtual void print () const = 0;
};
```

```
class IntNumber : public Number
{
public:
        IntNumber (int a)
                : x (a)
        {}
        virtual void print () const
        { cout<<x<<endl;}

private:
        int x;
};
```

```
class FloatNumber : public Number
{
public:
        FloatNumber (float a)
                : x (a)
        {}
        virtual void print () const
        { cout<<x<<endl;}

private:
        float x;
};
```

```
void main ()
{
        Number* numbers[3];

        IntNumber* i1 = new IntNumber (5);
        FloatNumber* f1 = new FloatNumber (10.7f);
        IntNumber* i2 = new IntNumber (8);

        numbers[0] = i1;
        numbers[1] = f1;
        numbers[2] = i2;

        for (int i = 0; i < 3; ++i)
                numbers[i]->print ();
}
```