Intermediate Programming

Day 33

Outline

- Exercise 11-3
- Dynamic dispatch
- Function hiding and abstract classes
- Virtual destructors
- Review questions

Exercise 11-3 (part 2)

Aclass.h

```
...

class A
{
private:
	int a;

protected:
	double d;
	...
```

Bclass.h

```
...

class B : public A
{
private:
	int b;

public:
	B(int val = 0): b(val) { };
	B(int bval, int aval, double dval ) :
		A(aval, dval), b(bval)
	{
		d = 17;
		a = 27;
		...
```

main1.cpp

```
...

int main( void )
{
	A aobj(1);
	A *aptr;
	B bobj(2);
	B *bptr;
	...
	 aobj.d = 17.5;
	...
	aptr->setb(15);
	...
	A a5(5);
	bobj = a5;
	...

}
```

Exercise 11-3 (part 2)

Aclass.h

```
...

class A
{
private:
	int a;

protected:
	double d;
	...
```

Bclass.h

```
...

class B : public A
{
private:
	int b;

public:
	B(int val = 0): b(val) { };
	B(int bval, int aval, double dval ) :
		A(aval, dval), b(bval)
	{
		d = 17;
		a = 27;
		...
```

main1.cpp

```
...

int main( void )
{
	A aobj(1);
	A *aptr;
	B bobj(2);
	B *bptr;
	...
	 aobj.d = 17.5;
	...
	aptr->setb(15);
	...
	A a5(5);
	bobj = a5;
	...

}
```

Exercise 11-3 (part 2)

```
Aclass.h
...

class A
{
private:
        int a;

protected:
        double d;
        ...
```

```
Bclass.h
...

class B : public A
{
private:
        int b;

public:
        B(int val = 0): b(val) { };
        B(int bval, int aval, double dval ) :
                A(aval, dval), b(bval)
        {
                d = 17;
                a = 27;
                ...
```

```
main1.cpp
...

int main( void )
{
        A aobj(1);
        A *aptr;
        B bobj(2);
        B *bptr;
        ...
         aobj.d = 17.5;
        ...
        aptr->setb(15);
        ...
        A a5(5);
        bobj = a5;
        ...

}
```

Exercise 11-3 (part 2)

Aclass.h

```
...

class A
{
private:
	int a;

protected:
	double d;
	...
```

Bclass.h

```
...

class B : public A
{
private:
	int b;

public:
	B(int val = 0): b(val) { };
	B(int bval, int aval, double dval ) :
		A(aval, dval), b(bval)
	{
		d = 17;
		a = 27;
		...
```

main1.cpp

```
...

int main( void )
{
	A aobj(1);
	A *aptr;
	B bobj(2);
	B *bptr;
	...
	 aobj.d = 17.5;
	...
	aptr->setb(15);
	...
	A a5(5);
	bobj = a5;
	...

}
```

Exercise 11-3 (part 2)

Aclass.h

```
...

class A
{
private:
	int a;

protected:
	double d;
	...
```

Bclass.h

```
...

class B : public A
{
private:
	int b;

public:
	B(int val = 0): b(val) { };
	B(int bval, int aval, double dval ) :
		A(aval, dval), b(bval)
	{
		d = 17;
		a = 27;
		...
```

main1.cpp

```
...

int main( void )
{
	A aobj(1);
	A *aptr;
	B bobj(2);
	B *bptr;
	...
	 aobj.d = 17.5;
	...
	aptr->setb(15);
	...
	A a5(5);
	bobj = a5;
	...

}
```

Exercise 11-3 (part 4)

Aclass.h

```
...

class A
{
	...
public:
	void show() { std::cout << "A is " << a << std::endl; test(); }
	...
```

Bclass.h

```
...

class B
{
	...
public:
	void show() { A::show(); std::cout << "B is " << b << std::endl; test(); }
};
	...
```

main1.cpp

```
...
main( void )
{
	A aobj(1);
	A *aptr;
	B bobj(2);
	B *bptr;
	...
	bptr = &bobj;
	aptr = bptr;
	aptr->seta(3);
	aptr->show();
	...
```

```
>> ./main
...
A is 3
test A
...
```

Exercise 11-3 (part 4)

Aclass.h

```
...

class A
{
	...
public:
	virtual void show() { std::cout << "A is " << a << std::endl; test(); }
	...
```

Bclass.h

```
...

class B
{
	...
public:
	void show() { A::show(); std::cout << "B is " << b << std::endl; test(); }
};
	...
```

main1.cpp

```
...
main( void )
{
	A aobj(1);
	A *aptr;
	B bobj(2);
	B *bptr;
	...
	bptr = &bobj;
	aptr = bptr;
	aptr->seta(3);
	aptr->show();
	...
```

```
>> ./main
...
A is 3
test A
B is 2
test B
...
```

Outline

- Exercise 11-3
- Dynamic dispatch
- Function hiding and abstract classes
- Virtual destructors
- Review questions

Inheritance (casting)

- We can convert from a derived class back to its base
  - The compiler casts to the derived class

account.h

```
#include <string>
class Account
{
public:
	…
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
class CheckingAccount : public Account
{
public:
	…
};
```

main.cpp

```
#include <iostream>
#include "account.h"
using namespace std;
void PrintBalance( const Account acct )
{
	cout << "Balance: " << acct.balance() << endl;
}
int main( void )
{
	Account acct( 1000 );
	CheckingAccount cAcct( 5000 );
	PrintBalance( acct );
	PrintBalance( cAcct );
	return 0;
}
```

```
>> ./a.out
Balance: 1000
Balance: 5000
>>
```

Inheritance (slicing)

- We can convert from a derived class back to its base
  - The compiler "slices out" the derived class

account.h

```
#include <string>
class Account
{
public:
	…
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
class CheckingAccount : public Account
{
public:
	…
};
```

main.cpp

```
#include <iostream>
#include "account.h"
using namespace std;
void PrintBalance( const Account &acct )
{
	cout << "Balance: " << acct.balance() << endl;
}
int main( void )
{
	Account acct( 1000 );
	CheckingAccount cAcct( 5000 );
	PrintBalance( acct );
	PrintBalance( cAcct );
	return 0;
}
```

```
>> ./a.out
Balance: 1000
Balance: 5000
>>
```

Inheritance (slicing)

- We can convert from a derived class back to its base
  - The compiler "slices out" the derived class

account.h

```
#include <string>
class Account
{
public:
	…
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
class CheckingAccount : public Account
{
public:
	…
};
```

main.cpp

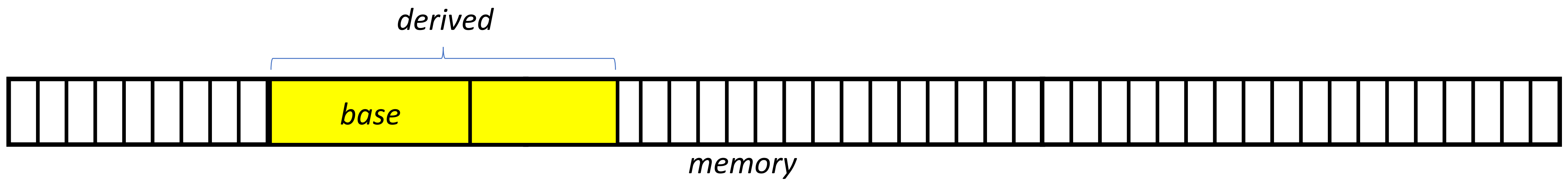
```
#include <iostream>
#include "account.h"
using namespace std;
void PrintBalance( const Account *acct )
{
	cout << "Balance: " << acct->balance() << endl;
}
int main( void )
{
	Account acct( 1000 );
	CheckingAccount cAcct( 5000 );
	PrintBalance( &acct );
	PrintBalance( &cAcct );
	return 0;
}
```

```
>> ./a.out
Balance: 1000
Balance: 5000
>>
```

Inheritance

<u>Under the hood</u>:

When the compiler lays out a derived object in memory, it puts the data of the base class first

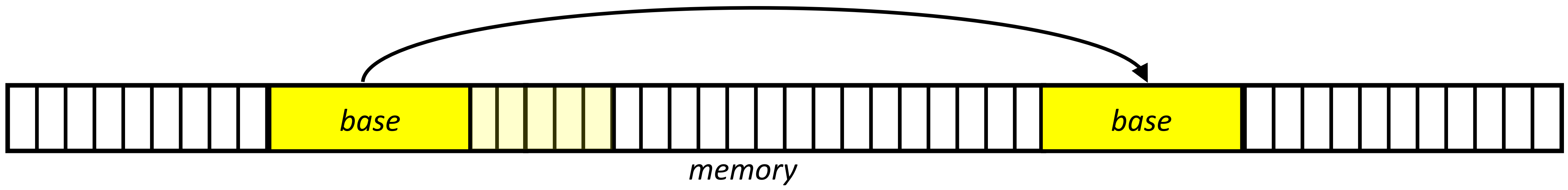


Inheritance (casting)

<u>Under the hood</u>:

When the compiler lays out a derived object in memory, it puts the data of the base class first

- To cast to the derived class, the compiler copies the contents of the base and ignores the contents of memory past the base data



Inheritance (slicing)

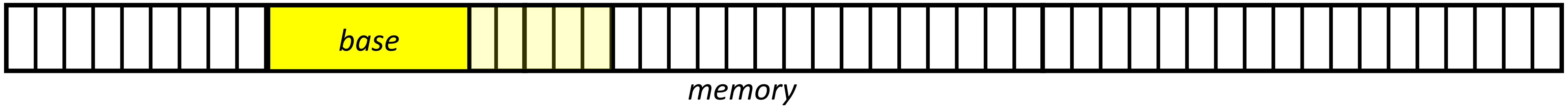
Under the hood:

When the compiler lays out a derived object in memory, it puts the data of the base class first

- To cast to the derived class, the compiler copies the contents of the base and ignores the contents of memory past the base data
- To slice out the derived class, the compiler ignores the contents of memory past the base data

⇒ The address of the derived object is the same as the address of the base

⇒ A reference to the derived object is a reference to the base



Inheritance (dynamic dispatch)

- We can tell the compiler to determine the "true" type of a class as it invokes certain methods, and use the implementation of that class

account.h

```
#include <string>
class Account
{
public:
	…
	virtual std::string type( void ) const { return "generic"; }
};
class CheckingAccount : public Account
{
public:
	…
	std::string type( void ) const { return "checking"; }
};
```

main.cpp

```
#include <iostream>
#include "account.h"
void PrintType( const Account& a )
{
	std::cout << "Type: " << a.type() << std::endl;
}
int main( void )
{
	Account acct( 1000 );
	CheckingAccount cAcct( 5000 );
	PrintType( acct );
	PrintType( cAcct );
	return 0;
}
```

```
>> ./a.out
Type: generic
Type: checking
>>
```

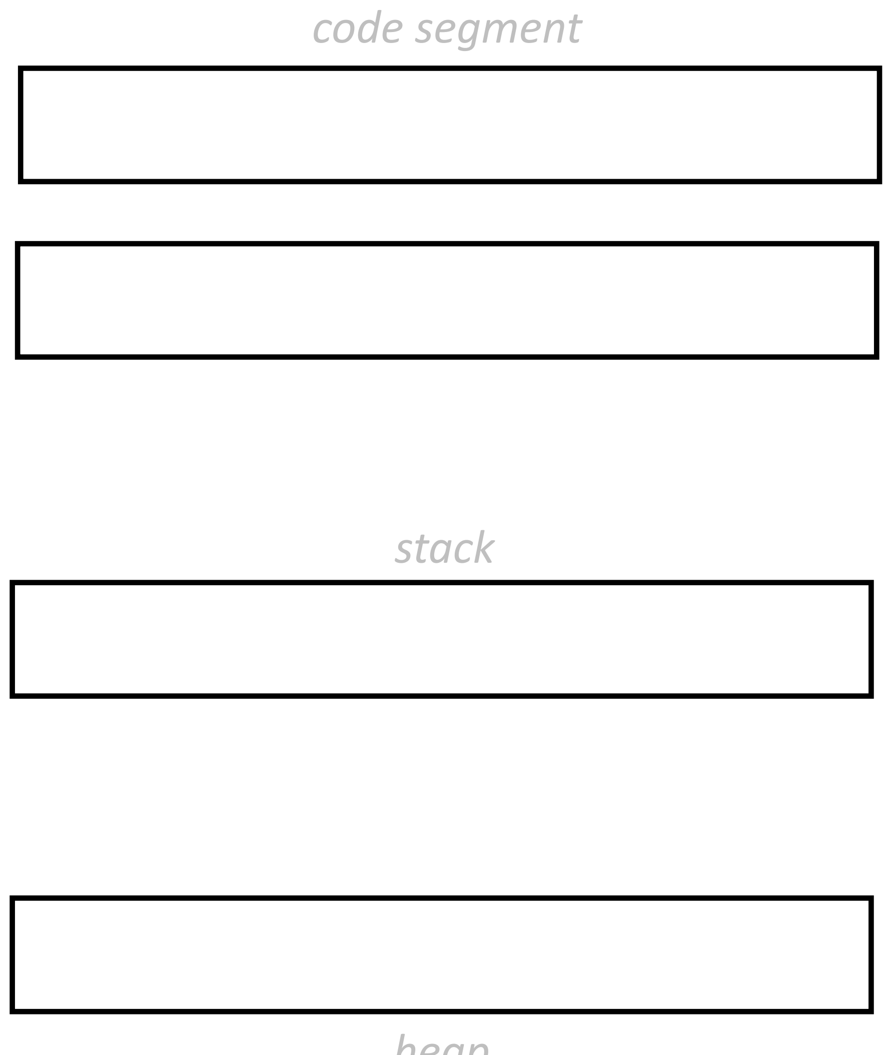
Inheritance (dynamic dispatch)

<u>Under the hood</u>:

When we previously talked about the memory layout, we talked about the *stack* and the *heap*.

This was a little simplified. There is also the *code segment*. This is where the code resides in memory.

(As opposed to the stack and heap, the size of the code segment is known at compile time.)



```
                    main.cpp (part 1)
#include <iostream>

class Base
{
        double _b;
public:
        void hi(){ std::cout << "hi(base)" << std::endl; }
        void bye() { std::cout << "bye(base)" << std::endl; }
};

class Derived : public Base
{
        double _d;
public:
        void hi(){ std::cout << "hi(derived)" << std::endl; }
};
```

c dispatch)

ut the memory

ck and the *heap*.

e is also the

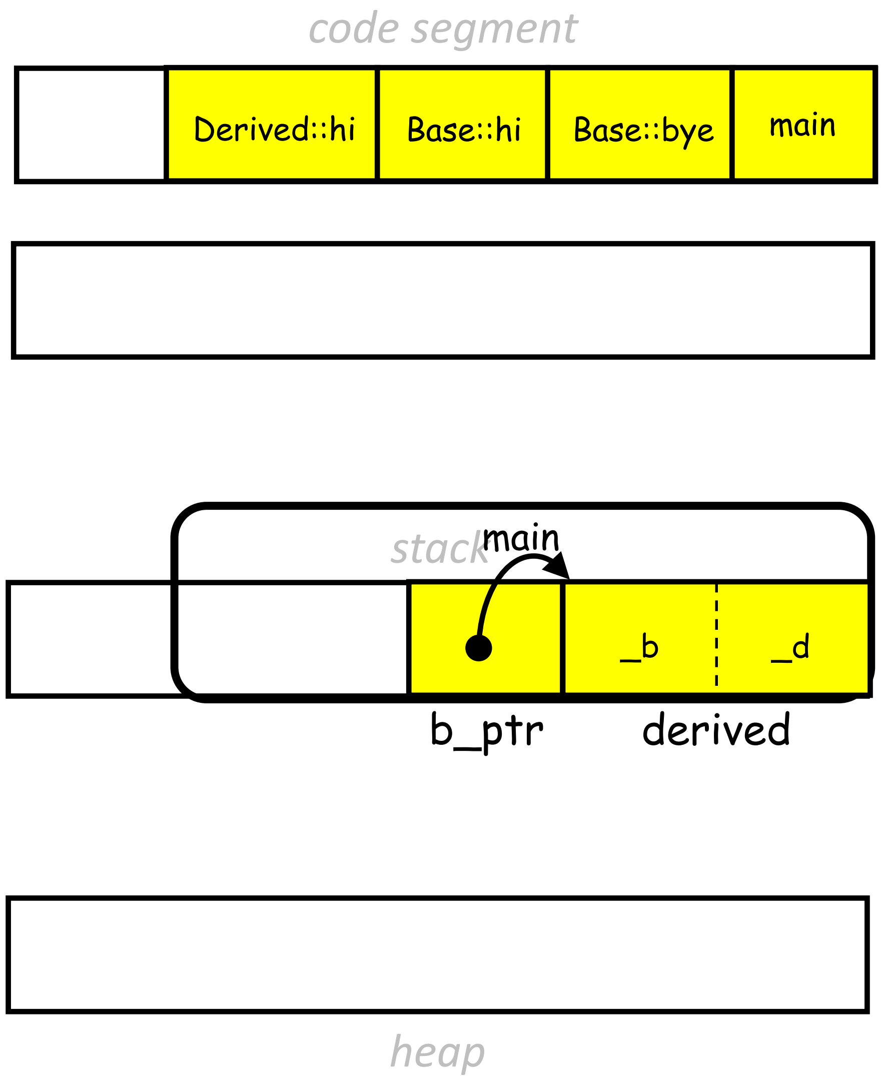
code segment. This is where the code resides

eap, the size of

ompile time.)

```
                    main.cpp (part 2)
int main( void )
{
        Derived derived;
        Base *b_ptr = &derived;
        b_ptr->hi();
        b_ptr->bye();
        return 1;
}
```

```
>> ./a.out
hi(base)
bye(base)
>>
```

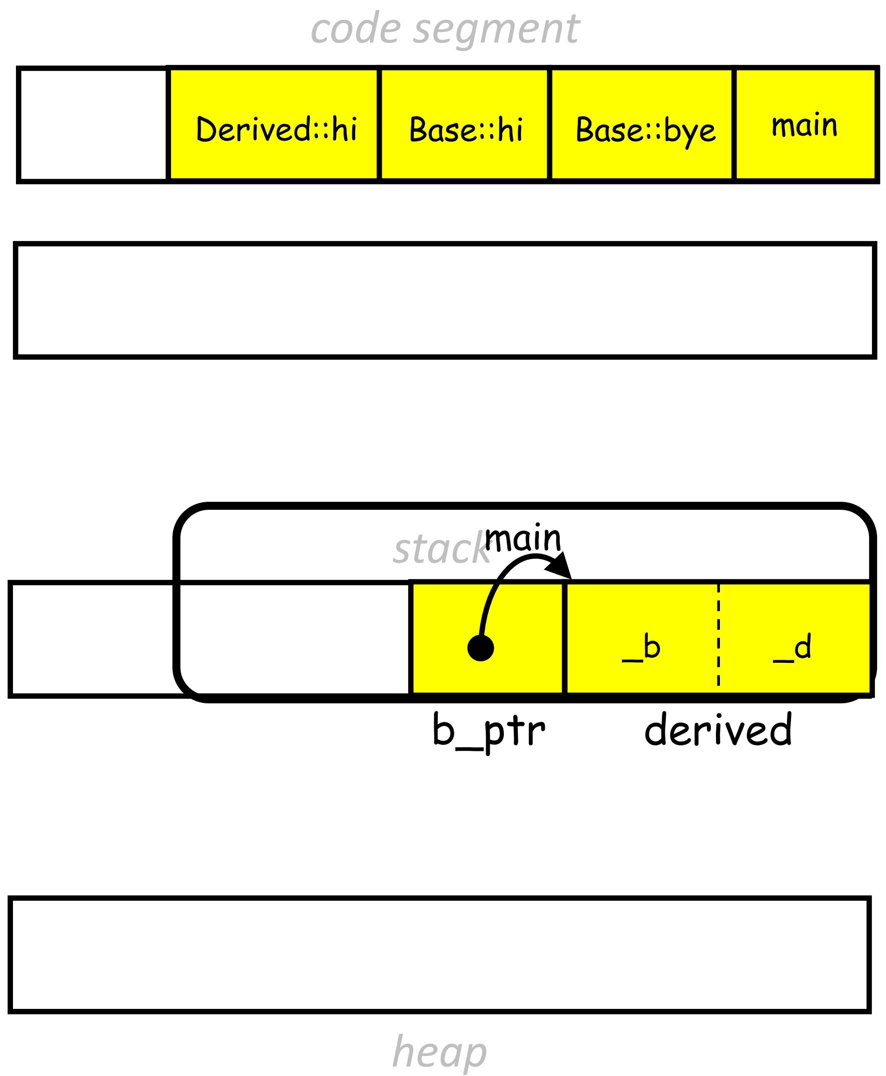


Inheritance (dynamic dispatch)

Under the hood:

When a class has virtual member functions:

1. The compiler creates a virtual function table for the class listing the addresses of its most derived virtual functions
2. The compiler adds a (hidden) member pointing to the class's virtual function table
3. When an object is created, the pointer points to the class's virtual function table



main.cpp (part 1)

```
#include <iostream>

class Base
{
        double _b;
public:
        virtual void hi(){ std::cout << “hi(base)” << std::endl; }
        virtual void bye() { std::cout << “bye(base)" << std::endl; }
};

class Derived : public Base
{
        double _d;
public:
        void hi(){ std::cout << “hi(derived)” << std::endl; }
};
```

c dispatch)

ber functions:

l function table

sses of its most

member pointing

table

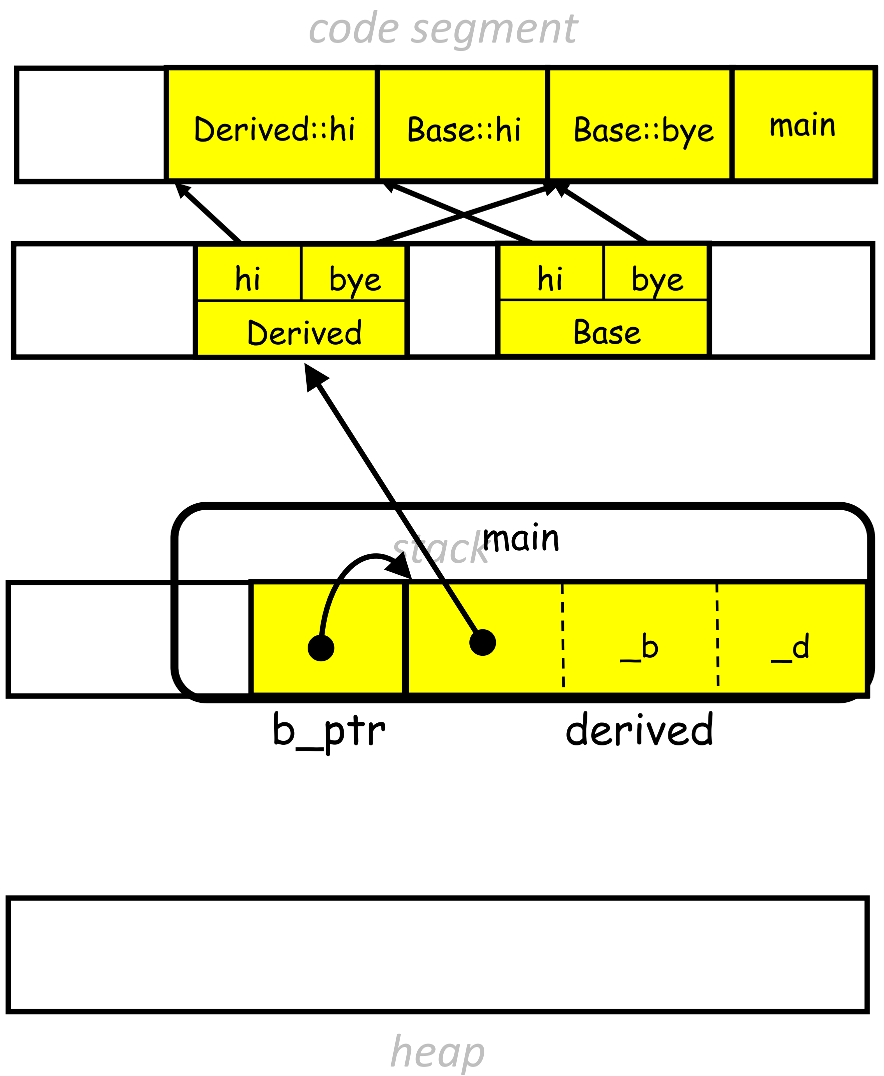
ne pointer points

table

main.cpp (part 2)

```
int main( void )
{
        Derived derived;
        Base *b_ptr = &derived;
        b_ptr->hi();
        b_ptr->bye();
        return 1;
}
```

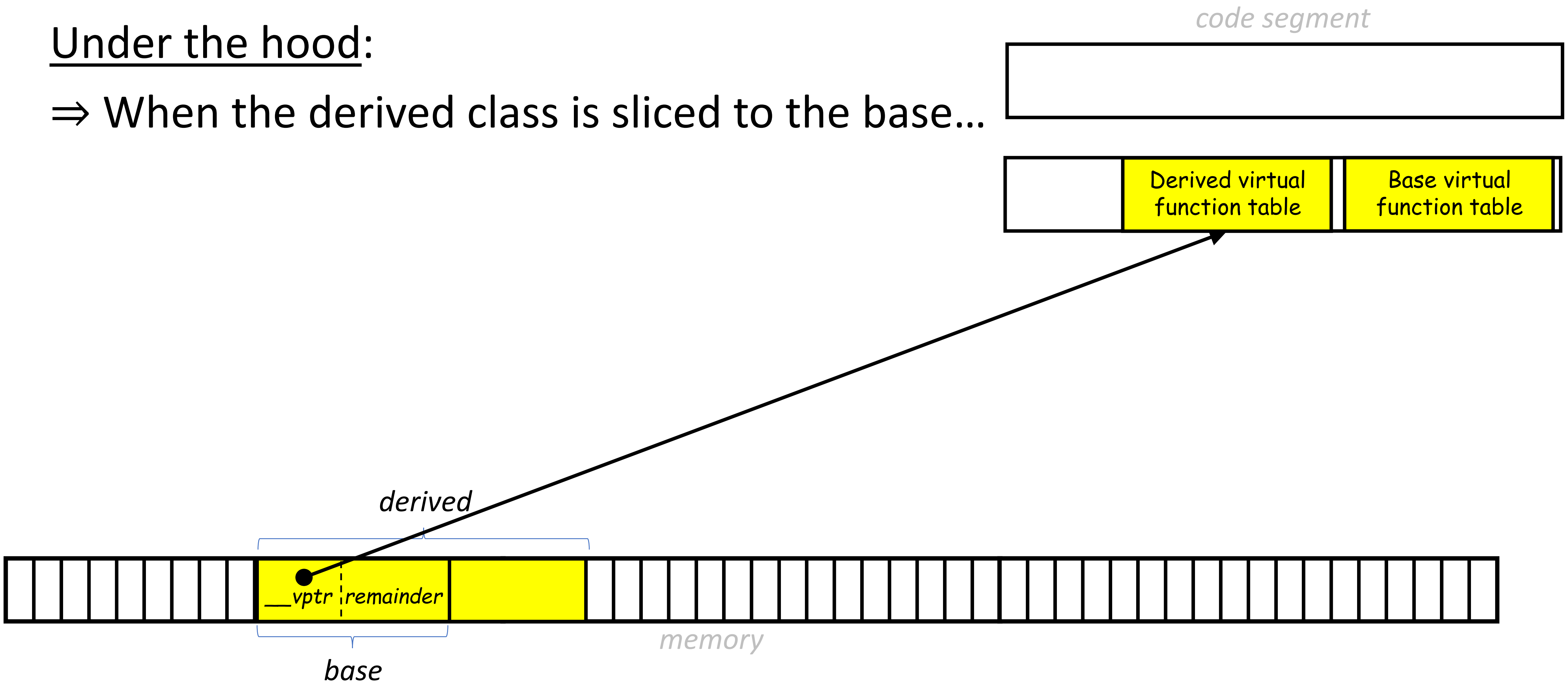
```
>> ./a.out
hi(base)
bye(derived)
>>
```



Inheritance (dynamic dispatch)

<u>Under the hood</u>:

⇒ When the derived class is sliced to the base…

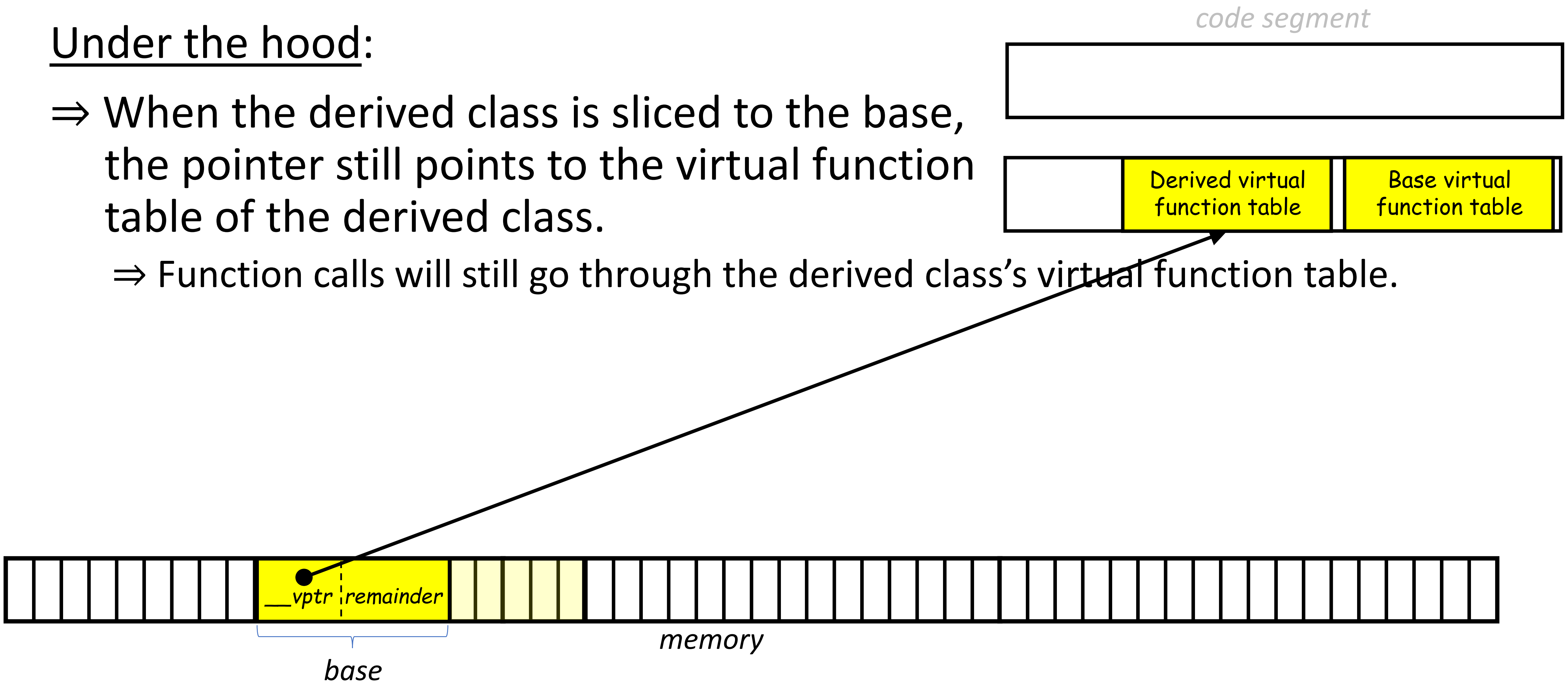


Inheritance (dynamic dispatch)

<u>Under the hood</u>:

⇒ When the derived class is sliced to the base, the pointer still points to the virtual function table of the derived class.

⇒ Function calls will still go through the derived class's virtual function table.



Inheritance (dynamic dispatch)

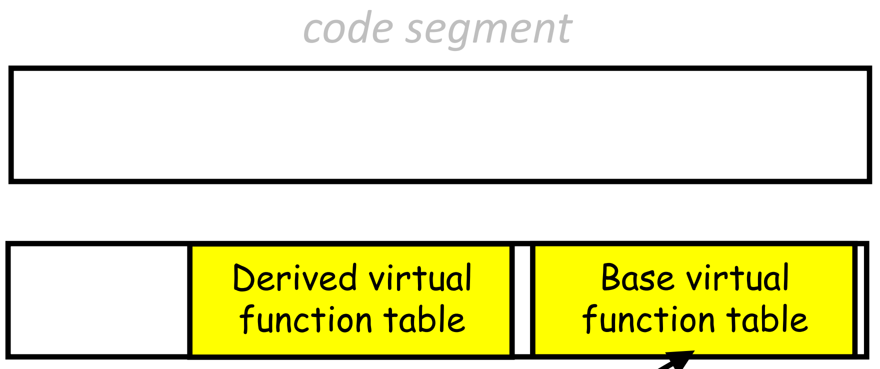
<u>Under the hood</u>:

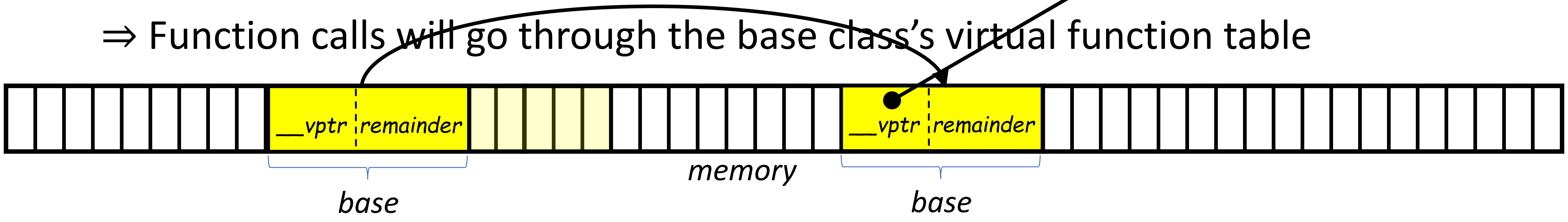
⇒ When the derived class is sliced to the base, the pointer still points to the virtual function table of the derived class.

⇒ Function calls will still go through the derived class's virtual function table.

⇒ When the derived class is cast to the base, the pointer is set to the virtual function table of the base class and the member data is copied (e.g. using the overloaded assignment operator).

⇒ Function calls will go through the base class's virtual function table





Inheritance (dynamic dispatch)

Under the hood:

When a class has virtual methods, the compiler adds a (hidden) member pointing to the class's virtual function table

baseDerived.h

```
class Base
{
public:
	int *b;
	int base( void ){ return 0; }
};
class Derived : public Base
{
public:
	int *d;
};
```

main.cpp

```
#include <iostream>
#include "baseDerived.h"
int main( void )
{
	std::cout << sizeof(Base) << " : " << sizeof(Derived) << std::endl;
	return 0;
}
```

```
>> ./a.out
8 : 16
>>
```

Inheritance (dynamic dispatch)

Under the hood:

When a class has virtual methods, the compiler adds a (hidden) member pointing to the class's virtual function table

baseDerived.h

```
class Base
{
public:
	int *b;
	virtual int base( void ){ return 0; }
};
class Derived : public Base
{
public:
	int *d;
};
```

main.cpp

```
#include <iostream>
#include "baseDerived.h"
int main( void )
{
	std::cout << sizeof(Base) << " : " << sizeof(Derived) << std::endl;
	return 0;
}
```

```
>> ./a.out
16 : 24
>>
```

Inheritance (dynamic dispa

Warning:

To override, a base class's function the signatures (function's name, arguments, and **const** designators) have to match. Otherwise you may end up calling the base method when you meant to call the derived.

baseDerived.h

```
#include <string>

class Base
{
public:
        virtual std::string type( void ) const { return "base"; }
};

class Derived : public Base
{
public:
        std::string type( void ) { return "derived"; }
};
```

main.cpp

```
#include <iostream>
#include "baseDerived.h"

int main( void )
{
        Derived derived;
        Base &base = derived;
        std::cout << base.type() << std::endl;
        return 0;
}
```

```
>> ./a.out
base
>>
```

Inheritance (dynamic dispa

baseDerived.h

```
#include <string>

class Base
{
public:
	virtual std::string type( void ) const { return "base"; }
};

class Derived : public Base
{
public:
	std::string type( void ) override { return "derived"; }
};
```

main.cpp

```
#include <iostream>
#include "baseDerived.h"

int main( void )
{
	Derived derived;
	Base &base = derived;
	std::cout << base.type() << std::endl;
	return 0;
```

<u>Warning</u>:

To override, a base class's function the signatures (function's name, arguments, and **const** designators) have to match. Otherwise you may end up calling the base method when you meant to call the derived.

You can protect your code by specifying that the derived method should **override** the base method.

```
>> g++ main.cpp ...
In file included from foo.cpp:2:
baseDerived.h:12:15: error: 'std::string Derived::type()' marked 'override', but does not override
   12 |   std::string type( void ) override { return "derived"; }
      |               ^~~~
>>
```

Outline

- Exercise 11-3
- Dynamic dispatch
- Function hiding and abstract classes
- Virtual destructors
- Review questions

Function hiding

- When a derived class defines a member function with the same name, the base class's member function becomes hidden, even if it's the better match.

main.cpp

```
#include <iostream>

using namespace std;

class Base
{
public:
	void foo( int ){ cout << "base" << endl; }
};

class Derived : public Base
{
public:
	void foo( double ){ cout << "derived" << endl; }
};

int main( void )
{
	Derived d;
	d.foo( 1 );
	d.foo( 1. );
	return 0;
}
```

```
>> ./a.out
derived
derived
>>
```

Function hiding

- When a derived class defines a member function with the same name, the base class's member function becomes hidden, even if it's the better match.
- In fact, the base class's member function becomes hidden, even if the derived class cannot match the argument list.

main.cpp

```
#include <iostream>

using namespace std;

class Base
{
public:
	void foo( int , int ){ cout << "base" << endl; }
};

class Derived : public Base
{
public:
	void foo( double ){ cout << "derived" << endl; }
};

int main( void )
{
	Derived d;
	d.foo( 1 , 1 );
	return 0;
}
```

Function hiding

- When a derived class defines a member function with the same name, the base class’s member function becomes hidden, even if it’s the better match.
- In fact, the base class’s member function becomes
  if the derived class
  the argument list.

main.cpp

```
#include <iostream>

using namespace std;

class Base
{
public:
	void foo( int , int ){ cout << "base" << endl; }
};

class Derived : public Base
{
public:
	void foo( double ){ cout << "derived" << endl; }
};
```

```
>> g++ main.cpp ...
main.cpp: In function ‘int main()’:
main.cpp:20:15: error: no matching function for call to ‘Derived::foo(int, int)’
   20 |  d.foo( 1 , 1 );
      |               ^
main.cpp:14:8: note: candidate: ‘void Derived::foo(double)’
   14 |   void foo( double ){ cout << "derived" << endl; }
      |        ^~~
main.cpp:14:8: note:   candidate expects 1 argument, 2 provided
>>
```

Inheritance (pure **virtual** functions)

- You can declare a function to be pure virtual by setting it "=0"
- This makes the class abstract because it has undefined function members

  ⇒ You cannot create an object of the base type because it will be abstract.

main.cpp

```
#include <iostream>
class Base
{
public:
	virtual void print( void ) const = 0;
};
class Derived : public Base
{
public:
	void print( void ) const
	{ std::cout << "derived" << std::endl; }
};
int main( void )
{
	Base b;
	return 0;
}
```

Inheritance (pure **virtual** functions)

- You can declare a function to be pure virtual by setting it "=0"
- This makes the class abstract because it has undefined function members

  ⇒ You cannot create an object of the base type because it will be abstract

main.cpp

```
#include <iostream>
class Base
{
public:
	virtual void print( void ) const = 0;
};
class Derived : public Base
{
```

```
d::endl; }
```

```
>> g++ main.cpp ...
main.cpp: In function ‘int main()’:
main.cpp:14:6: error: cannot declare variable ‘b’ to be of abstract type ‘Base’
   14 | Base b;
      |      ^
main.cpp:2:7: note:   because the following virtual functions are pure within ‘Base’:
    2 | class Base
      |       ^~~~
main.cpp:5:14: note:     ‘virtual void Base::print() const’
    5 | virtual void print( void ) const = 0;
      |              ^~~~~
>>
```

Inheritance (pure **virtual** functions)

- You can declare a function to be pure virtual by setting it "=0"
- This makes the class abstract because it has undefined function members
  - ⇒ You cannot create an object of the base type because it will be abstract.
  - ⇒ You can create a derived object **if** the derived class defines the method

main.cpp

```
#include <iostream>
class Base
{
public:
        virtual void print( void ) const = 0;
};
class Derived : public Base
{
public:
        void print( void ) const
        { std::cout << "derived" << std::endl; }
};
int main( void )
{
        Derived d;
        return 0;
}
```

Inheritance (pure **virtual** functions)

- You can declare a function to be pure virtual by setting it "=0"
- This makes the class abstract because it has undefined function members

  ⇒ You cannot create an object of the base type because it will be abstract.

  ⇒ You can create a derived object if the derived class defines the method

  ⇒ You can have pointers and references to the base object

main.cpp

```
#include <iostream>
class Base
{
public:
        virtual void print( void ) const = 0;
};
class Derived : public Base
{
public:
        void print( void ) const
        { std::cout << "derived" << std::endl; }
};
int main( void )
{
        Derived d;
        Base &b = d;
        return 0;
}
```

Inheritance (pure **virtual** functions)

- You can declare a function to be <u>pure virtual</u> by setting it "=0"
- This makes the class <u>abstract</u> because it has undefined function members
- You can also make the class abstract by making its constructor **protected**.

main.cpp

```
#include <iostream>
class Base
{
protected:
	Base( void ){ std::cout << "base" << std::endl; }
};
class Derived : public Base
{
public:
	Derived( void ) : Base()
	{
		std::cout << "derived" << std::endl;
	}
};
int main( void )
{
	Base b;
	return 0;
}
```

Inheritance (pure **virtual** functions)

- You can declare a function to be pure virtual by setting it "=0"
- This makes the class abstract because it has undefined function members
- You can also make the class abstract by making its constructor **protected**.

main.cpp

```
#include <iostream>
class Base
{
protected:
	Base( void ){ std::cout << "base" << std::endl; }
};
class Derived : public Base
{
public:
	Derived( void ) : Base()
	{
		std::cout << "derived" << std::endl;
	}

t main( void )

	Base b;
	return 0;
```

```
>> g++ main.cpp ...
main.cpp: In function 'int main()':
main.cpp:17:6: error: 'Base::Base()' is protected within this context
   17 | Base b;
      |      ^
main.cpp:5:2: note: declared protected here
    5 |  Base( void ){ std::cout << "base" << std::endl; }
      |  ^~~~
>>
```

Inheritance (pure **virtual** functions)

- You can declare a function to be pure virtual by setting it "=0"
- This makes the class abstract because it has undefined function members
- You can also make the class abstract by making its constructor **protected**.

main.cpp

```
#include <iostream>
class Base
{
protected:
	Base( void ){ std::cout << "base" << std::endl; }
};
class Derived : public Base
{
public:
	Derived( void ) : Base()
	{
		std::cout << "derived" << std::endl;
	}
};
int main( void )
{
	Derived d;
	return 0;
}
```

```
>> ./a.out
base
derived
>>
```

Outline

- Exercise 11-3
- Dynamic dispatch
- Function hiding and abstract classes
- Virtual destructors
- Review questions

Virtual destructors

- When you slice a derived class to a base class, it is the base class's destructor that is invoked when the object is deleted.

main.cpp

```
#include <iostream>
using namespace std;
class Base
{
public:
        Base( void ){ cout << "base" << endl; }
        ~Base( void ) { cout << "~base" << endl; }
};
class Derived : public Base
{
public:
        Derived( void ){ cout << "derived" << endl; }
        ~Derived( void ){ cout << "~derived" << endl; }
};
int main( void )
{
        Base *b = new Derived();
        delete b;
        return 0;
}
```

```
>> ./a.out
base
derived
~base
>>
```

Virtual destructors

- When you slice a derived class to a base class, it is the base class's destructor that is invoked when the object is deleted.
- You can declare destructor of the base to be `virtual` to force the derived destructor to be used (e.g. if the derived classes needs to release resources when it is destroyed.)

main.cpp

```
#include <iostream>
using namespace std;
class Base
{
public:
        Base( void ){ cout << "base" << endl; }
        virtual ~Base( void ) { cout << "~base" << endl; }
};
class Derived : public Base
{
public:
        Derived( void ){ cout << "derived" << endl; }
        ~Derived( void ){ cout << "~derived" << endl; }
};
int main( void )
{
        Base *b = new Derived();
        delete b;
        return 0;
}
```

```
>> ./a.out
base
derived
~derived
~base
>>
```

Virtual destructors

<u>Rule of thumb</u>:
If a class has virtual member functions, it should also have a virtual destructor.

Virtual member functions

$\Downarrow$

The derived class could have unforeseen functionality

$\Downarrow$

The derive class could acquire resources that need to be released

Outline

- Exercise 11-3
- Dynamic dispatch
- Function hiding and abstract classes
- Virtual destructors
- Review questions

Review questions

1. Explain what object slicing is in C++.

When a pointer/reference to a base class is used to point to/reference a derived object, the compiler “squints” and only looks at the base’s subset of the information.

Review questions

2. What is the keyword `override` in C++?

A way to indicate that a function in a derived class is supposed to override one in a base class

Review questions

3. Explain what function hiding is in C++?

When a function in a derived class has the same name but different parameters than one in its base class

Review questions

4. In C++, how do you make an abstract class?

Include a pure virtual function, or do not provide a public constructor

Review questions

5. Can we create an object from an abstract class?

No

Exercise 12-1

- Website -> Course Materials -> ex12-1