# 600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

# Announcement

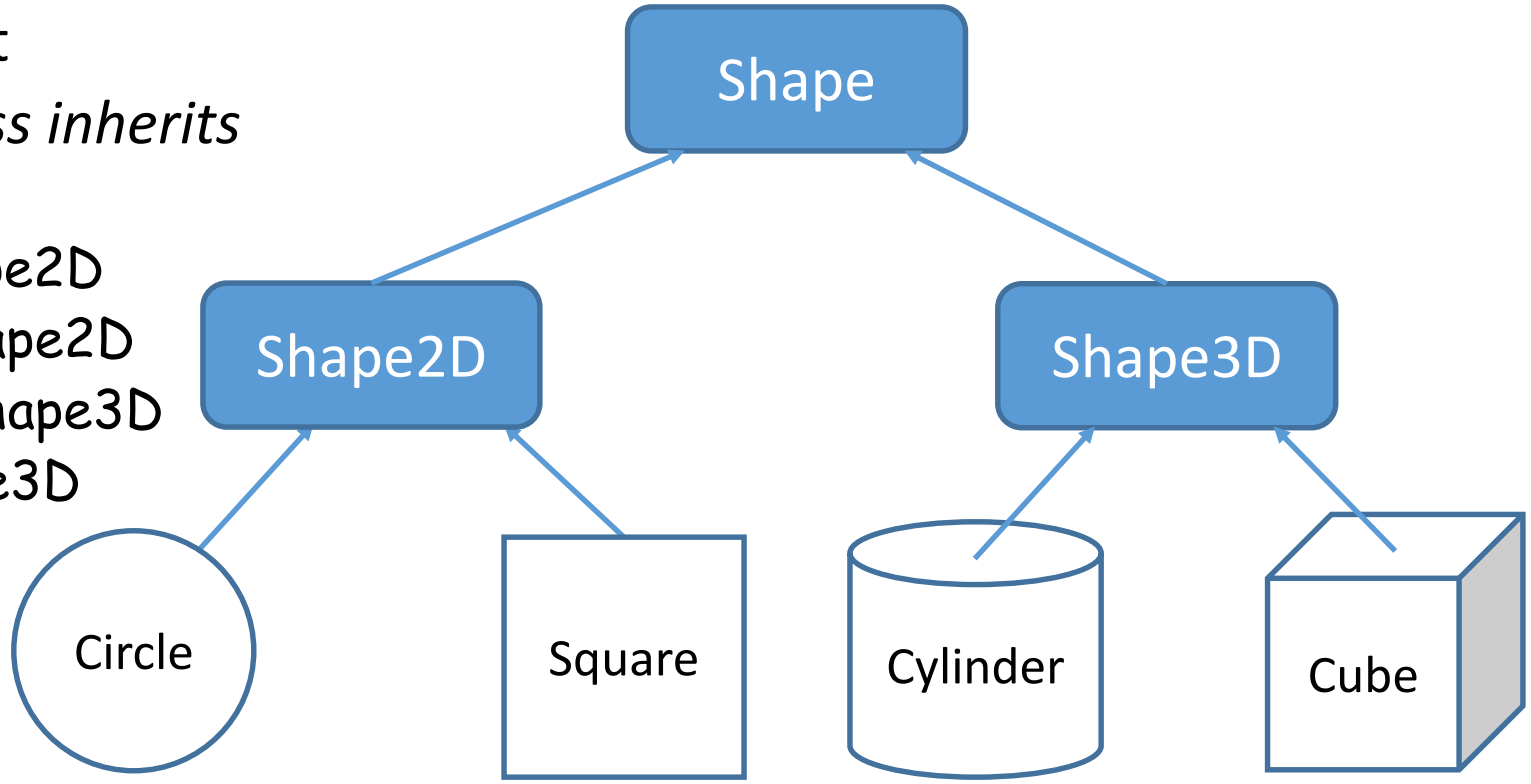
- Class cancelled:
  - Monday (4/10)
  - Wednesday (4/12)
  - Monday (4/17)

# Outline

- Inheritance

# Inheritance

- A class can describe a particular type of a more general class
  - Cylinders and cubes are types of 3D shape
  - Circles and squares are types 2D shapes
  - The relationships can nest
  - We say that *a derived class inherits from a base class*:
    - `Circle` inherits from `Shape2D`
    - `Square` inherits from `Shape2D`
    - `Cylinder` inherits from `Shape3D`
    - `Cube` inherits from `Shape3D`



# Inheritance

<u>Notation</u>:

To disambiguate, we will write:

<class name>::<member/method name>

to indicate that a member / method belongs to a particular class

- Account::balance specifies the public balance method of class Account
- Account::_balance specifies the private _balance method of class Account

```
class Account
{
public:
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

# Inheritance

- We specify that a derived class inherits from a base class when we declare the derived class

  class <derived class> : <inherit how> <base class>

  - <inherit how>: In general, inheritance is public

```
class Account
{
public:
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

```
class CheckingAccount : public Account
{
public:
	...
};
```

# Inheritance

Q: What is inherited?

A: Everything

- All members and methods, whether **public**, **protected**, or **private**
  - As though all of the base class's members/methods were declared right before the derived class's members/methods are declared

```
class Account
{
public:
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

```
class CheckingAccount : public Account
{
public:
	...
};
```

# Inheritance

Q: What can be accessed?

A: All base-class members/methods marked **public** or **protected** can be accessed from derived class

- **private** members/methods are there and can be accessed by the base class, but not by the derived class

```
class Account
{
public:
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

```
class CheckingAccount : public Account
{
public:
	void printBalance( void ) const
	{
		cout << balance() << endl;
	}
	...
};
```

# Inheritance

- **protected** is an access level between **public** and **private**:
  - Derived classes have access to protected methods/members in the base class but nobody else does

```
class Account
{
public:
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

```
class CheckingAccount : public Account
{
public:
	void printBalance( void ) const
	{
		cout << balance() << endl;
	}
	...
};
```

# Inheritance

*account.h*

```
class Account
{
public:
	Account( void ) : _balance(0.0) { }
	Account( double b ) : _balance( b ) { }
	void credit( double amt ) { _balance += amt; }
	void debit( double amt ) { _balance -= amt; }
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

- Constructors:
  - Default constructor sets `Account::_balance` member to 0
  - Non-default constructor sets `Account::_balance` member according to the argument
- `Account::_balance` is private:
  - users modify it via the `Account::credit` / `Account::debit` methods
  - users get a copy of its value via the `Account::balance` methods

# Inheritance

*account.h*

```
class Account
{
public:
	Account( void ) : _balance(0.0) { }
	Account( double b ) : _balance( b ) { }
	void credit( double amt ) { _balance += amt; }
	void debit( double amt ) { _balance -= amt; }
	double balance( void ) const { return _balance; }
private:
	double _balance;
};
```

*main.cpp*

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

int main( void )
{
	Account acct( 1000.0 );
	acct.credit( 1000.0 );
	acct.debit( 100.0 );
	cout << "Balance is: $ " << acct.balance() << endl;
	return 0;
}
```

```
>> ./a.out
Balance is $1900
>>
```

- Constructors:
  - Default constructor sets `Account::_balance` member to 0
  - Non-default constructor sets `Account::_balance` member according to the argument
- `Account::_balance` is private:
  - users modify it via the `Account::credit` / `Account::debit` methods
  - users get a copy of its value via the `Account::balance` methods

# Inheritance

*account.h*

```
class Account
{
public:
	Account( void ) : _balance(0.0) { }
	Account( double b ) : _balance( b )
	void credit( double amt ) { _balanc
	void debit( double amt ) { _balance
	double balance( void ) const { retu
private:
	double _balance;
};
…
```

*account.h*

```
…
class CheckingAccount : public Account
{
public:
	CheckingAccount( void ) : Account() , _totalFees( 0.0 ) { }
	CheckingAccount( double b ) : Account( b ) , _totalFees( 0.0 ) { }
	void cashWithdrawal( double amt )
	{
		_totalFees += _ATMFee;
		debit( amt + _ATMFee);
	}
	double totalFees( void ) const { return _totalFees; }
private:
	static const double _ATMFee = 2.00;
	double _totalFees;
};
```

- CheckingAccount inherits from Account
  - CheckingAccount::cashWithrdawl calls the Account::debit (which modifies the private member Account::_balance)

# Inheritance

*account.h*

```
class Account
{
public:
	Account( void ) : _balance(0.0) { }
	Account( double b ) : _balance( b )
	void credit( double amt ) { _balanc
	void debit( double amt ) { _balance
	double balance( void ) const { retu
private:
	double _balance;
};
…
```

*account.h*

```
…
class CheckingAccount : public Account
{
public:
	CheckingAccount( void ) : Account() , _totalFees( 0.0 ) { }
	CheckingAccount( double b ) : Account( b ) , _totalFees( 0.0 ) { }
	void cashWithdrawal( double amt )

priv

};
```

*main.cpp*

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

int main(void)
{
	Account acct( 1000.0 );
	acct.credit( 1000.0 );
	acct.debit( 100.0 );
	cout << "Balance is: $" << acct. balance() << endl;
	return 0;
}
```

- CheckingAccount inherits f
  - CheckingAccount::cashWit
    (which modifies the private member Account::_

```
>> ./a.out
Balance is $1900
>>
```

# Inheritance

*account.h*

```
class Account
{
public:
	Account( void ) : _balance(0.0) { }
	Account( double b ) : _balance( b )
	void credit( double amt ) { _balanc
	void debit( double amt ) { _balance
	double balance( void ) const { retu
private:
	double _balance;
};
…
```

*account.h*

```
…
class CheckingAccount : public Account
{
public:
	CheckingAccount( void ) : Account() , _totalFees( 0.0 ) { }
	CheckingAccount( double b ) : Account( b ) , _totalFees( 0.0 ) { }
	void cashWithdrawal( double amt )

priv

};
```

*main.cpp*

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

int main(void)
{
	CheckingAccount acct( 1000.0 );
	acct.credit( 1000.0 );
	acct.cashWithrdawl( 100.0 );
	cout << "Balance is: $" << acct. balance() << endl;
	return 0;
}
```

- CheckingAccount inherits f
  - CheckingAccount::cashWit
    (which modifies the private member Account::_

```
>> ./a.out
Balance is $1898
>>
```

# Inheritance (casting)

- 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 (casting)

- 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 (casting)

- 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 <vector>
#include "account.h"
int main( void )
{
	CheckingAccount cAcct( 5000 );
	Account acct = cAcct;
	return 0;
}
```

# Inheritance (casting)

- 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 <vector>
#include "account.h"
int main( void )
{
	std::vector< Account > accounts;
	CheckingAccount cAcct( 5000 );
	accounts.push_back( cAcct );
	return 0;
}
```

# Inheritance (method overriding)

- A derived class can override inherited methods by declaring its own version of the method

*account.h*

```
#include <string>
class Account
{
public:
        …
        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"
using namespace std;

int main( void )
{
        Account acct();
        CheckingAccount cAcct();
        cout << "Type: " << acct.type() << endl;
        cout << "Type: " << cAcct.type() << endl;
        return 0;
}
```

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

# Inheritance (method overriding)

Q: What happens if we cast and override? (Whose method is called?)

*account.h*

```
#include <string>
class Account
{
public:
	...
	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"
using namespace std;
void PrintType( const Account& acct )
{
	cout << "Type: " << acct.type() << endl;
}
int main( void )
{
	Account acct( 1000 );
	CheckingAccount cAcct( 5000 );
	PrintType( acct );
	PrintType( cAcct );
	return 0;
}
```

# Inheritance (method overriding)

Q: What happens if we cast and override? (Whose method is called?)

A: The method of the base class

- When PrintType is called, the CheckingAccount part of cAcct is sliced out and only the Account part remains

*main.cpp*

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

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

# Inheritance (dynamic dispatch)

- We can tell the compiler to determine the "true" type of a class <u>as it invokes certain methods</u>, and use the implementation of that class
  - Use the keyword `virtual` to indicate that a method *may* be overridden by a derived class and that the derived class's method should be used

# Inheritance (dynamic dispatch)

- We can tell the compiler to determine the "true" type of a class <u>as it invokes certain methods</u>, 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)

- We can tell the compiler to determine the "true" type of a class <u>as it invokes certain methods</u>, and use the implementation of that class
  - Use the keyword **virtual** to indicate that a method *may* be overridden by a derived class and that the derived class's method should be used
  - If the function is pass-by-value (instead of pass-by-reference) the object is copied and information about the derived class is lost

*main.cpp*

```
#include <iostream>
#include "account.h"
void PrintType( 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: generic
>>
```

# Inheritance (dynamic dispatch)

- We can tell the compiler to determine the "true" type of a class <u>as it invokes certain methods</u>, and use the implementation of that class
  - To call the method of the base class, we can cast to a non-reference type
    - This creates an object which stores a copy of the base's data

*main.cpp*

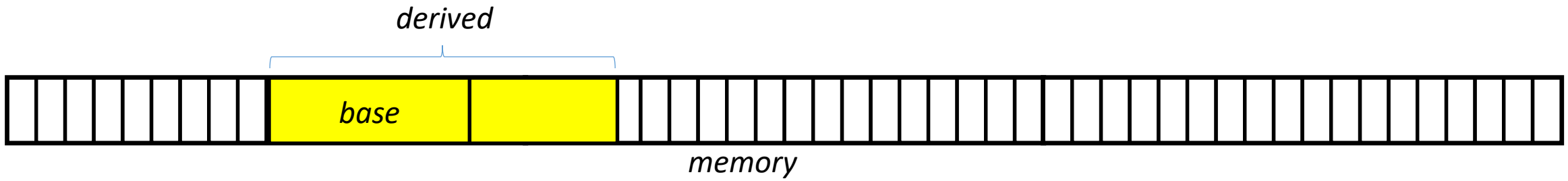
```
#include <iostream>
#include "account.h"
using namespace std;

int main( void )
{
	CheckingAccount cAcct();
	cout << "Type: " << cAcct.type() << endl;
	cout << "Type: " << ( (Account)cAcct ).type() << endl;
	return 0;
}
```

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

# Inheritance (dynamic dispatch)

<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 (dynamic dispatch)

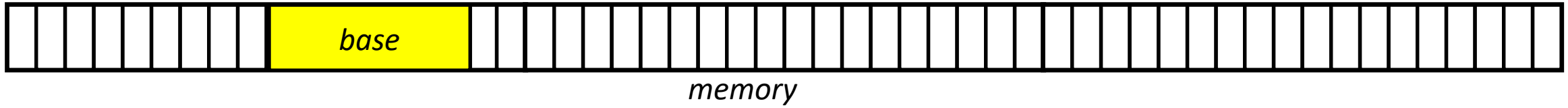
<u>Under the hood</u>:

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

- To 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)

<u>Under the hood</u>:

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

*baseDerived.h*

```
class Base
{
public:
	int* b;
};
class Derived : public Base
{
public:
	int* d;
};
```

*main.cpp*

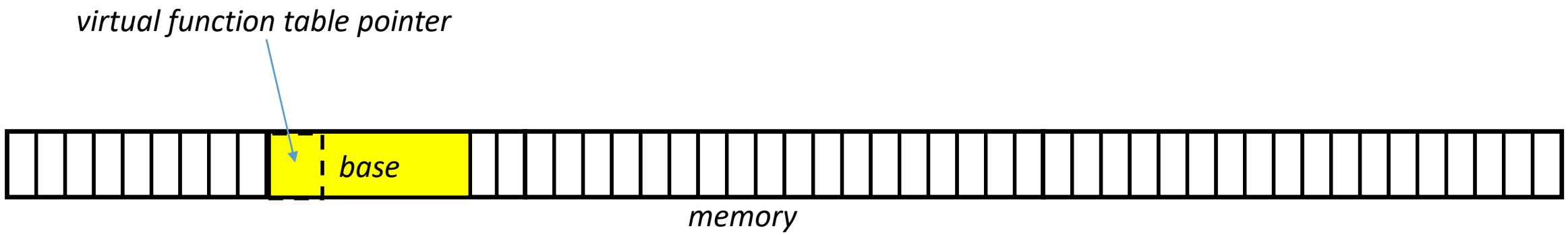
```
#include <iostream>
#include "baseDerived.h"
int main( void )
{
	Derived d;
	size_t bOff = (size_t)&d.b - (size_t)&d;
	size_t dOff = (size_t)&d.d - (size_t)&d;
	std::cout << bOff << " : " << dOff << std::endl;
	return 0;
}
```

```
>> ./a.out
0 : 8
>>
```

# Inheritance (dynamic dispatch)

<u>Under the hood</u>:

When a class has virtual methods, its objects store an additional (function table) pointer which tells it where to find the appropriate implementation



# Inheritance (dynamic dispatch)

<u>Under the hood</u>:

When a class has virtual methods, its objects store an additional (function table) pointer which tells it where to find the appropriate implementation

*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)

<u>Under the hood</u>:

When a class has virtual methods, its objects store an additional (function table) pointer which tells it where to find the appropriate implementation

*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 dispatch)

- Dynamic dispatch / virtual functions enable C++ *polymorphism*
  - With virtual functions, an object can “looks like” (i.e. its declared type is) the base class, but behave like a derived class (i.e. when you call its members, you’re sometimes calling a virtual function defined in the derived class)

# Multiple Inheritance

- C++ allows a derived class to inherit from multiple base classes

myFile.h

```
#include <cstdio>
class RFile
{
	FILE** _fp;
public:
	RFile( FILE** fp ) : _fp( fp ){ }
	FILE* fp( void ) { return *_fp; }
	char read( void );
};
class WFile
{
	FILE**_fp;
public:
	WFile( FILE** fp ) : _fp( fp ){ }
	FILE* fp( void ) { return *_fp; }
	void write( char );
};
class RWFile : public RFile , public WFile
{
	FILE* _fp;
public:
	RWFile( const char* name) : RFile( &_fp )  , WFile( &_fp )
	{
		_fp =fopen( name , "rw" );
	}
};
```

# Multiple Inheritance

- C++ allows a derived class to inherit from multiple base classes
  - The derived class can access members / methods from either base class

*myFile.h*

```
#include <cstdio>
class RFile
{
	FILE** _fp;
public:
	RFile( FILE** fp ) : _fp( fp ){ }
	FILE* fp( void ) { return *_fp; }
	char read( void );
};
class WFile
{
	FILE**_fp;
public:
	WFile( FILE** fp ) : _fp( fp ){ }
	FILE* fp( void ) { return *_fp; }
	void
};
class RW
{
	FILE
public:
	RWF
	{

	}
};
```

*main.cpp*

```
#include <cstdlib>
#include "myFile.h"
int main( void )
{
	RWFile file( "foo.txt" );
	char c = file.read( );
	file.write( c );
	return 0;
}
```

# Multiple Inheritance

- C++ allows a derived class to inherit from multiple base classes
  - The derived class can access members / methods from either base class
  - What happens when the base classes have a method / member with the same name?

*myFile.h*

```
#include <cstdio>
class RFile
{
	FILE** _fp;
public:
	RFile( FILE** fp ) : _fp( fp ){ }
	FILE* fp( void ) { return *_fp; }
	char read( void );
};
class WFile
{
	FILE**_fp;
public:
	WFile( FILE** fp ) : _fp( fp ){ }
	FILE* fp( void ) { return *_fp; }
	void
};
class RW
{
	FILE
public:
	RWF
	{

	}
};
```

*main.cpp*

```
#include <cstdlib>
#include "myFile.h"
int main( void )
{
	RWFile file( "foo.txt" );
	fseek( file.fp() , 0 , SEEK_END );
	file.write( 'a' );
	return 0;
}
```

# Multiple Inheritance

- C++ allows a derived class to inherit from multiple base classes
  - The derived class can access members / methods from either base class
  - What happens when the base classes have a method / member with the same name?

*myFile.h*

```
#include <cstdio>
class RFile
{
    FILE** _fp;
public:
    RFile( FILE** fp ) : _fp( fp ){ }
    FILE* fp( void ) { return *_fp; }
    char read( void );
}
c
{

p

    void
};
class RW
{
    FILE
public:
    RWF
    {

    }
};
```

```
>>  g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp:5:15: error: request for member fp is
ambiguous
   fseek( file.fp() , 0 , SEEK_END );
                ^~
>>
```

*main.cpp*

```
#include <cstdlib>
#include "myFile.h"
int main( void )
{
    RWFile file( "foo.txt" );
    fseek( file.fp() , 0 , SEEK_END );
    file.write( 'a' );
    return 0;
}
```

# Multiple Inheritance

- C++ allows a derived class to inherit from multiple base classes
  - The derived class can access members / methods from either base class
  - When the member / method is ambiguous, we can cast to a reference of the base class to disambiguate

*myFile.h*

```
#include <cstdio>
class RFile
{
	FILE** _fp;
public:
	RFile( FILE** fp ) : _fp( fp ){ }
	FILE* fp( void ) { return *_fp; }
	char read( void );
};
class WFile
{
	FILE**_fp;
public:
	WFile( FILE** fp ) : _fp( fp ){ }
	FILE* fp( void ) { return *_fp; }
	void
};
class RW
{
	FILE
public:
	RWF
	{

	}
};
```

*main.cpp*

```
#include <cstdlib>
#include "myFile.h"
int main( void )
{
	RWFile file( "foo.txt" );
	fseek( ( (Wfile&)file ).fp() , 0 , SEEK_END );
	file.write( 'a' );
	return 0;
}
```

# Multiple Inheritance

- C++ allows a derived class to inherit from multiple base classes

*main.cpp*

```
#include <iostream>
class Base1 { public: size_t b; };
class Base2 { public: size_t b; };
class Derived: public Base1, public Base2 { public: size_t d; };

using namespace std;

int main( void )
{
	cout << sizeof(Base1) << " : " << sizeof(Base2) << " : ";
	cout << sizeof(Derived) << endl;
	return 0;
}
```

```
>>  ./a.out
8 : 8 : 24
>>
```

# Multiple Inheritance

- C++ allows a derived class to inherit from multiple base classes
  - Casting is trickier because we cannot put both base classes at the beginning of the derived class

*main.cpp*

```
#include <iostream>
class Base1 { public: size_t b; };
class Base2 { public: size_t b; };
class Derived: public Base1, public Base2 { public: size_t d; };

using namespace std;

int main( void )
{
	Derived d;
	size_t dAddr = (size_t)&d;
	size_t b1Addr = (size_t)(Base1*)&d;
	size_t b2Addr = (size_t)(Base2*)&d;
	cout << b1Addr-dAddr << " : " b2Addr-dAddr << endl;
	return 0;
}
```

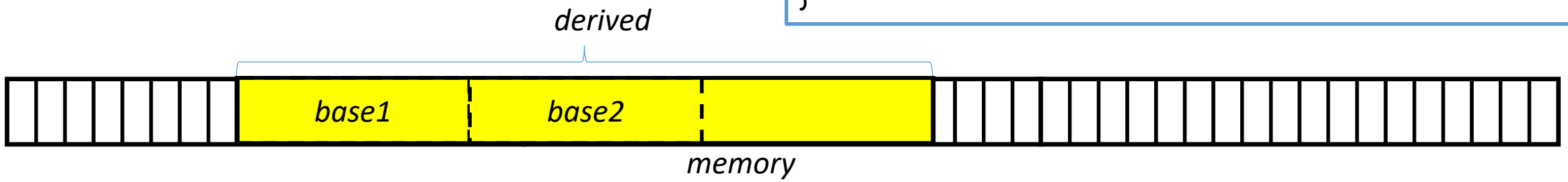
```
>>  ./a.out
0 : 8
>>
```

# Multiple Inheritance

- C++ allows a derived class to inherit from multiple base classes
  - Casting is trickier because we cannot put both base classes at the beginning of the derived class
  - The compiler must appropriately offset the address when casting to a base class

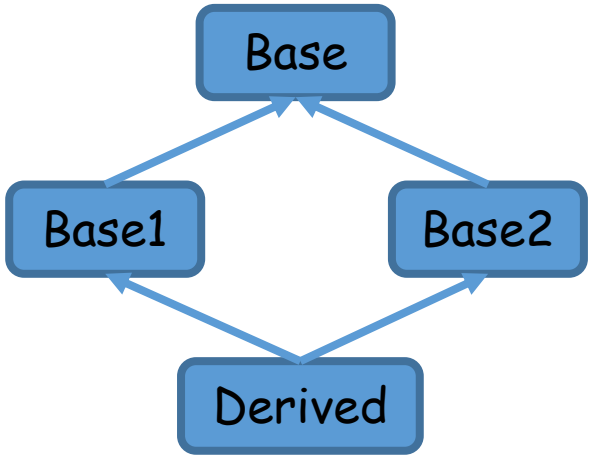
*main.cpp*

```
#include <iostream>
class Base1 { public: size_t b; };
class Base2 { public: size_t b; };
class Derived: public Base1, public Base2 { public: size_t d; };

using namespace std;

int main( void )
{
	Derived d;
	size_t dAddr = (size_t)&d;
	size_t b1Addr = (size_t)(Base1*)&d;
	size_t b2Addr = (size_t)(Base2*)&d;
	cout << b1Addr-dAddr << " : " b2Addr-dAddr << endl;
	return 0;
}
```



# Multiple Inheritance

- Diamond problem
  - A derived class can inherit from two base classes, both of which inherit from a single base class
    - There will be two instances of the **Base** class stored with a **Derived** object



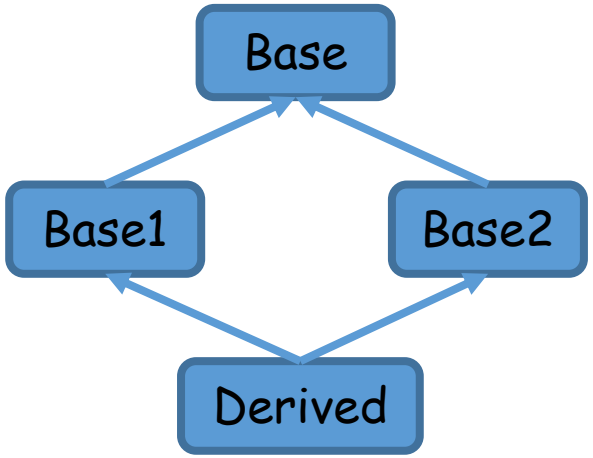

*main.cpp*

```
#include <iostream>
class Base { public: size_t b; };
class Base1 : public Base { public: size_t b1; };
class Base2 : public Base { public: size_t b2; };
class Derived: public Base1, public Base2 { public: size_t d; };

using namespace std;
int main( void )
{
	cout << sizeof(Base) << " : ";
	cout << sizeof(Base1) << " : " << sizeof(Base2) << " : ";
	cout << sizeof(Derived) << endl;
	return 0;
}
```

```
>> ./a.out
8 : 16 : 16 : 40
>>
```

# Multiple Inheritance

- Diamond problem
  - A derived class can inherit from two base classes, both of which inherit from a single base class
    - Access to the members of the **Base** object are necessarily ambiguous




*main.cpp*

```
#include <iostream>
class Base { public: size_t b; };
class Base1 : public Base { public: size_t b1; };
class Base2 : public Base { public: size_t b2; };
class Derived: public Base1, public Base2 { public: size_t d; };

using namespace std;
int main( void )
{
	Derived d;
	cout << d.b << endl;
	return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In function int main():
main.cpp:11:13: error: request for member b is ambiguous
   cout << d.b << endl;
             ^
>>
```

# Multiple Inheritance

- Diamond problem
  - A derived class can inherit from two base classes, both of which inherit from a single base class
    - Access to the members of the **Base** object are necessarily ambiguous
      - We cannot cast to the **Base** class because we don't know which of the two instances we want

*main.cpp*

```
#include <iostream>
class Base { public: size_t b; };
class Base1 : public Base { public: size_t b1; };
class Base2 : public Base { public: size_t b2; };
class Derived: public Base1, public Base2 { public: size_t d; };

using namespace std;
int main( void )
{
	Derived d;
	Base& b = (Base&)d;
	return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In function int main():
main.cpp:11:20: error: Base is an ambiguous base of Derived
   Base& b = (Base&)d;
                    ^
>>
```

# Multiple Inheritance

- Diamond problem
  - A derived class can inherit from two base classes, both of which inherit from a single base class
    - Access to the members of the **Base** object are necessarily ambiguous
      - We cannot cast to the **Base** class because we don't know which of the two instances we want
      - We can cast to either **Base1** or **Base2** and then cast from there to **Base**

*main.cpp*

```
#include <iostream>
class Base { public: size_t b; };
class Base1 : public Base { public: size_t b1; };
class Base2 : public Base { public: size_t b2; };
class Derived: public Base1, public Base2 { public: size_t d; };

using namespace std;
int main( void )
{
	Derived d;
	Base1& b1 = (Base1&)d;
	Base& b = (Base&)b1;
	return 0;
}
```

# Inheritance (dynamic dispatch)

Q: When is dynamic binding useful?

# Inheritance (dynamic dispatch)

A: When we only know the object type at run-time

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) = 0;
	virtual void draw( void ) = 0;
};
...
```

# Inheritance (dynamic dispatch)

A: When we only know the object type at run-time

- When a virtual method is defined to be 0, it is *pure virtual*
  - Derived classes *must* override the method in order to be instantiatable

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) = 0;
	virtual void draw( void ) = 0;
};
...
```

# Inheritance (dynamic dispatch)

A: When we only know the object type at run-time

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) = 0
	virtual void draw( void ) = 0;
};
...
```

*shape.h*

```
...
class Circle : public Shape
{
	double x , y , r; // center and radius
	void readParameters( void )
	{
		cout << "Center and radius: " ; cin >> x >> y >> r;
	}
	void draw( void )
	{
		cout << "Drawing circle: ";
		cout << "( " << x << " , " << y << " ) " << r << endl;
		...
	}
};
...
```

# Inheritance (dynamic dispatch)

A: When we only know the object type at run-time

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) = 0
	virtual void draw( void ) = 0;
};
...
```

*shape.h*

```
...
class Rectangle : class Shape
{
	double x1 , y1 , x2 ,  y2; // corners
	void readParameters( void )
	{
		cout << "Bottom left / Top right: " ; cin >> x1 >> y1 >> x2 >> y2;
	}
	void draw( void )
	{
		cout << "Drawing rectangle: ";
		cout << "( " << x1 << " , " << y1 << " ) : " << "( " << x2 << " , " << y2 << " )" << endl;
		...
	}
};
```

# Inheritance (dynamic dispatch)

A: When we only kno

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) =
	virtual void draw( void ) = 0;
};
...
```

*main.cpp*

```
#include <iostream>
#include <vector>
#include <string>
#include "shape.h"
int main( void )
{
	std::string type;
	std::vector< Shape* > shapes;
	std:: cout << "Shape type [circle/rectangle]: ";
	while( std::cin >> type )
	{
		if( type=="circle" ) shapes.push_back( new Circle() );
		else if( type=="rectangle" ) shapes.push_back( new Rectangle() );
		else break;
		shapes.back()->readParameters();
		std::cout << "Shape type [circle/rectangle]: ";
	}
	for( size_t i=0 ; i<shapes.size() ; i++ ){ shapes[i]->draw() ; delete shapes[i]; }
	return 0;
}
```

# Inheritance (dynamic disp

```
>> ./a.out
Shape type [circle/rectangle]: circle
Center and radius: 5 6 7
```

A: When we only kno

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) =
	virtual void draw( void ) = 0;
};
...
```

*main.cpp*

```
#include <iostream>
#include <vector>
#include <string>
#include "shape.h"
int main( void )
{
	std::string type;
	std::vector< Shape* > shapes;
	std:: cout << "Shape type [circle/rectangle]: ";
	while( std::cin >> type )
	{
		if( type=="circle" ) shapes.push_back( new Circle() );
		else if( type=="rectangle" ) shapes.push_back( new Rectangle() );
		else break;
		shapes.back()->readParameters();
		std::cout << "Shape type [circle/rectangle]: ";
	}
	for( size_t i=0 ; i<shapes.size() ; i++ ){ shapes[i]->draw() ; delete shapes[i]; }
	return 0;
}
```

# Inheritance (dynamic disp

```
>> ./a.out
Shape type [circle/rectangle]: circle
Center and radius: 5 6 7
Shape type [circle/rectangle]: rectangle
Bottom left / Top right: 1 3 2 5
```

A: When we only kno

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) =
	virtual void draw( void ) = 0;
};
...
```

```
#include <iostream>
#include <vector>
#include <string>
#include "shape.h"
int main( void )
{
	std::string type;
	std::vector< Shape* > shapes;
	std:: cout << "Shape type [circle/rectangle]: ";
	while( std::cin >> type )
	{
		if( type=="circle" ) shapes.push_back( new Circle() );
		else if( type=="rectangle" ) shapes.push_back( new Rectangle() );
		else break;
		shapes.back()->readParameters();
		std::cout << "Shape type [circle/rectangle]: ";
	}
	for( size_t i=0 ; i<shapes.size() ; i++ ){ shapes[i]->draw() ; delete shapes[i]; }
	return 0;
}
```

# Inheritance (dynamic disp

A: When we only kno

```
>> ./a.out
Shape type [circle/rectangle]: circle
Center and radius: 5 6 7
Shape type [circle/rectangle]: rectangle
Bottom left / Top right: 1 3 2 5
Shape type [circle/rectangle]: rectangle
Bottom left / Top right: -1 -5 90 399
```

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) =
	virtual void draw( void ) = 0;
};
...
```

```
#include <iostream>
#include <vector>
#include <string>
#include "shape.h"
int main( void )
{
	std::string type;
	std::vector< Shape* > shapes;
	std:: cout << "Shape type [circle/rectangle]: ";
	while( std::cin >> type )
	{
		if( type=="circle" ) shapes.push_back( new Circle() );
		else if( type=="rectangle" ) shapes.push_back( new Rectangle() );
		else break;
		shapes.back()->readParameters();
		std::cout << "Shape type [circle/rectangle]: ";
	}
	for( size_t i=0 ; i<shapes.size() ; i++ ){ shapes[i]->draw() ; delete shapes[i]; }
	return 0;
}
```

# Inheritance (dynamic disp

A: When we only kno

```
>> ./a.out
Shape type [circle/rectangle]: circle
Center and radius: 5 6 7
Shape type [circle/rectangle]: rectangle
Bottom left / Top right: 1 3 2 5
Shape type [circle/rectangle]: rectangle
Bottom left / Top right: -1 -5 90 399
Shape type [circle/rectangle]: done
```

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) =
	virtual void draw( void ) = 0;
};
...
```

```
#include <iostream>
#include <vector>
#include <string>
#include "shape.h"
int main( void )
{
	std::string type;
	std::vector< Shape* > shapes;
	std:: cout << "Shape type [circle/rectangle]: ";
	while( std::cin >> type )
	{
		if( type=="circle" ) shapes.push_back( new Circle() );
		else if( type=="rectangle" ) shapes.push_back( new Rectangle() );
		else break;
		shapes.back()->readParameters();
		std::cout << "Shape type [circle/rectangle]: ";
	}
	for( size_t i=0 ; i<shapes.size() ; i++ ){ shapes[i]->draw() ; delete shapes[i]; }
	return 0;
}
```

# Inheritance (dynamic disp

A: When we only kno

```
>> ./a.out
Shape type [circle/rectangle]: circle
Center and radius: 5 6 7
Shape type [circle/rectangle]: rectangle
Bottom left / Top right: 1 3 2 5
Shape type [circle/rectangle]: rectangle
Bottom left / Top right: -1 -5 90 399
Shape type [circle/rectangle]: done
Drawing circle: ( 5 , 6 ) 7
Drawing rectangle: ( 1 , 3 ) : ( 2 , 5 )
Drawing rectangle: ( -1 , -5 ) : ( 90 , 399 )
>>
```

*shape.h*

```
#include <iostream>
using namespace std;
class Shape
{
public:
	virtual void readParameters( void ) =
	virtual void draw( void ) = 0;
};
...
```

```
#include <iostream>
#include <vector>
#include <string>
#include "shape.h"
int main( void )
{
	std::string type;
	std::vector< Shape* > s
	std:: cout << "Shape type [circle/rectangle]: ";
	while( std::cin >> type )
	{
		if( type=="circle" ) shapes.push_back( new Circle() );
		else if( type=="rectangle" ) shapes.push_back( new Rectangle() );
		else break;
		shapes.back()->readParameters();
		std::cout << "Shape type [circle/rectangle]: ";
	}
	for( size_t i=0 ; i<shapes.size() ; i++ ){ shapes[i]->draw() ; delete shapes[i]; }
	return 0;
}
```