

Intermediate Programming

Day 32

Outline

- Exercise 31
- Inheritance
- Review questions

Exercise 31: `int_node.h` \rightarrow `my_node.h`

int_node.h

```
#ifndef INT_NODE_H
#define INT_NODE_H

class int_node
{
private:
    int    data; //the payload stored in this node
    int_node* next; //the pointer to node after this one

public:
    //constructors
    int_node(int value): data(value), next(nullptr) { }
    int_node(int value, int_node* ptr): data(value), next(ptr) { }

    //getters
    int get_data() const { return data; }
    int_node* get_next() const { return next; }

    //setters
    void set_data(int value) { data = value; }
    void set_next(int_node* ptr) { next = ptr; }
};
#endif
```

my_node.h

```
#ifndef MY_NODE_H
#define MY_NODE_H

template< typename T >
class my_node
{
private:
    T    data; //the payload stored in this node
    my_node* next; //the pointer to node after this one

public:
    //constructors
    my_node(T value): data(value), next(nullptr) { }
    my_node(T value, my_node* ptr): data(value), next(ptr) { }

    //getters
    T get_data() const { return data; }
    my_node* get_next() const { return next; }

    //setters
    void set_data(T value) { data = value; }
    void set_next(my_node* ptr) { next = ptr; }
};
#endif
```

Exercise 31: `int_set.h` \rightarrow `my_set.h`

int_set.h

```
#ifndef INT_SET_H
#define INT_SET_H

#include <iostream>
#include "int_node.h"

class int_set
{
private:
    int_node* head;
    int      size;

public:
    ...
    bool add(int new_value);
    ...

    friend std::ostream& operator <<
        (std::ostream& os, const int_set& s);
};

#endif
```

my_set.h

```
#ifndef MY_SET_H
#define MY_SET_H

#include <iostream>
#include "my_node.h"

template< typename T >
class my_set
{
private:
    my_node< T >* head;
    int          size;

public:
    ...
    bool add(T new_value);
    ...

    template< typename _T >
    friend std::ostream& operator <<
        (std::ostream& os, const my_set< _T >& s);
};

#include "my_set.inc"
#endif
```

Exercise 31: `int_set.cpp` → `my_set.inc`

Exercise 31: `int_set.cpp` → `my_set.inc`

`int_set.cpp`

```
int_set::int_set( const int_set& orig ) : head(nullptr)
{
    for( const int_node *n=orig.head ; n ; n=n->get_next() )
        add( n->get_data() );
}

int_set::~~int_set( void )
{
    clear();
}
...
```

`my_set.inc`

```
template< typename T >
my_set< T >::my_set(const my_set& o) : head(nullptr)
{
    for( const my_node< T > *n=o.head ; n ; n=n->get_next() )
        add( n->get_data() );
}

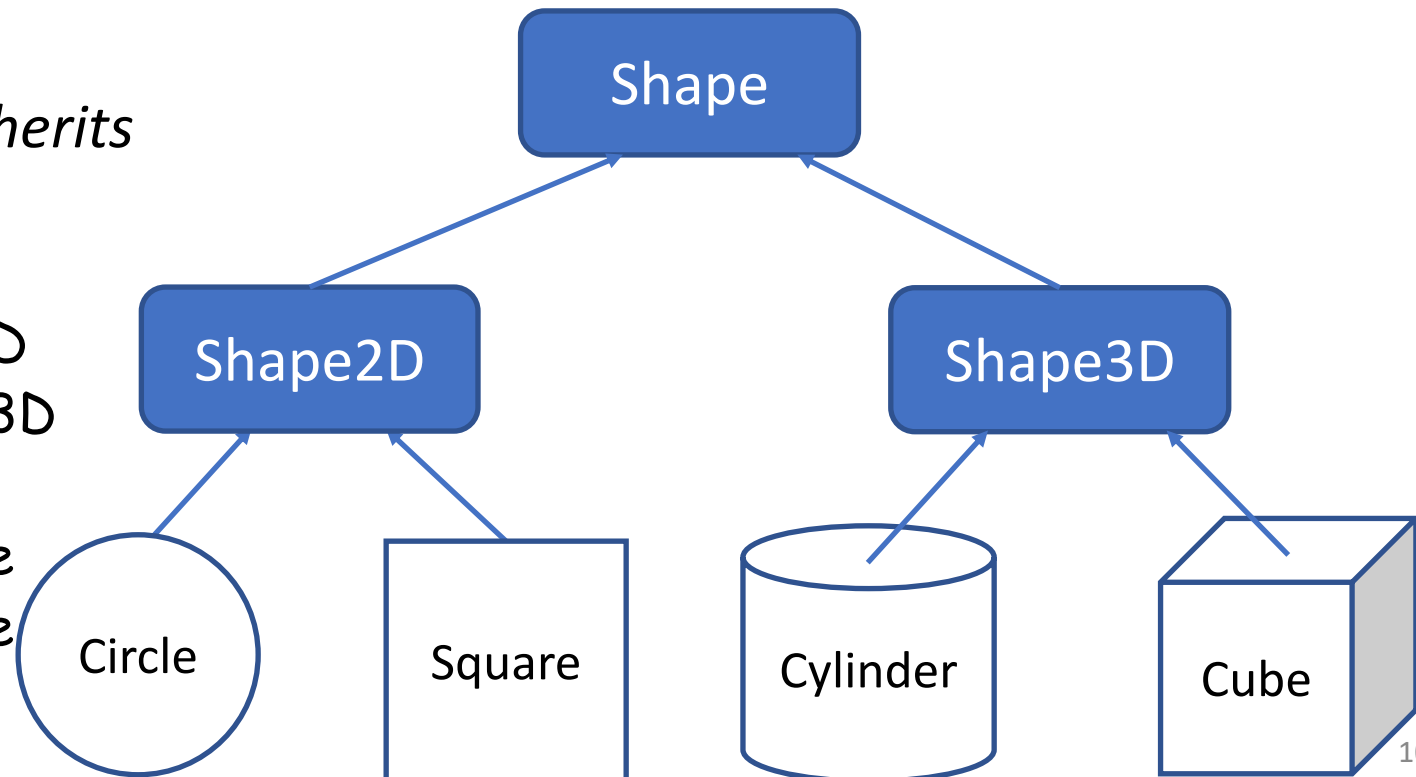
template< typename T >
my_set< T >::~~my_set( void )
{
    clear();
}
...
```


Outline

- Exercise 31
- **Inheritance**
- Review questions

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 of 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
 - Shape2D inherits from Shape
 - Shape3D inherits from Shape



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>`

```
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, whether **public**, **protected**, or **private** **except constructors, destructors, and assignment operators**
 - As though all of the base class's members were declared right before the derived class's members 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 marked **public** or **protected** can be accessed from the derived class

- **private** members 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 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 data to 0
 - Non-default constructor sets **Account::_balance** member data according to the argument
- **Account::_balance** is private:
 - users modify it via the **Account::credit** / **Account::debit** member functions
 - users get a copy of its value via the **Account::balance** member functions

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

Inheritance

```
account.h
class Account
{
public:
    Account( void ) : _balance(0.0) { }
    Account( double b ) : _balance( b )
    void credit( double amt ) { _balance
    void debit( double amt ) { _balance
    double balance( void ) const { return
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**
 - The **CheckingAccount** constructor calls the **Account** constructor (and initializes its own data)

Inheritance

```
account.h
class Account
{
public:
    Account( void ) : _balance(0.0) { }
    Account( double b ) : _balance( b )
    void credit( double amt ) { _balance
    void debit( double amt ) { _balance
    double balance( void ) const { retur
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**
 - The **CheckingAccount** constructor calls the **Account** constructor (and initializes its own data)
 - **CheckingAccount::cashWithdrawal** calls the **Account::debit** (which modifies the private member data **Account::_balance**)

Inheritance

account.h

```
class Account
{
public:
    Account( void ) : _balance(0.0) { }
    Account( double b ) : _balance( b )
    void credit( double amt ) { _balance
    void debit( double amt ) { _balance
    double balance( void ) const { retur
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 )
    {
```

main.cpp

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

private:

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 from **Account**
 - The **CheckingAccount** constructor (and initializes its own data)
 - **CheckingAccount::cashWithdrawal** (which modifies the private member data **Account::_balance**)

```
>> ./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 ) { _balance
    void debit( double amt ) { _balance
    double balance( void ) const { retur
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 )
    {
```

main.cpp

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

private:

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

- **CheckingAccount** inherits from **Account**
 - The **CheckingAccount** constructor (and initializes its own data)
 - **CheckingAccount::cashWithdrawal** (which modifies the private member data **Account::balance**)

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

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>`: Describes the `public` / `private` / `protected` status of inherited members

- `public`: members stay as is
- `protected`: `public` members become `protected`, everything else stays as is
- `private`: everything becomes `private`

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( 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

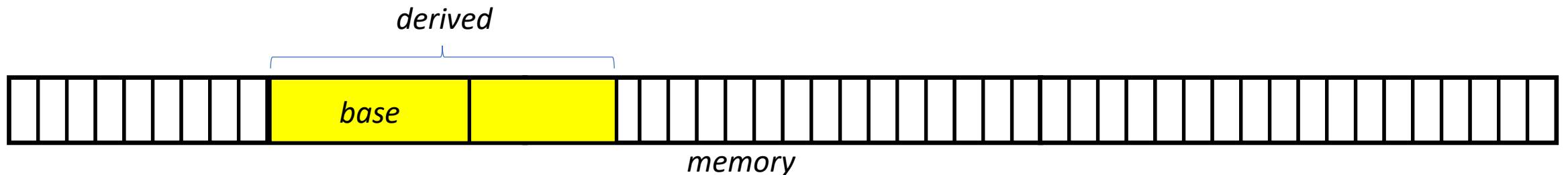
```
#include <iostream>
#include "account.h"
using namespace std;
void PrintBalance( const Account *acct )
{
    cout << "Balance: " << acct->balance() << endl;
}
int main( void )
{
    Account acct( 1000 );
    CheckingAccount cAcct( 5000 );
    PrintBalance( &acct );
    PrintBalance( &cAcct );
  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

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


Inheritance

Under the hood:

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

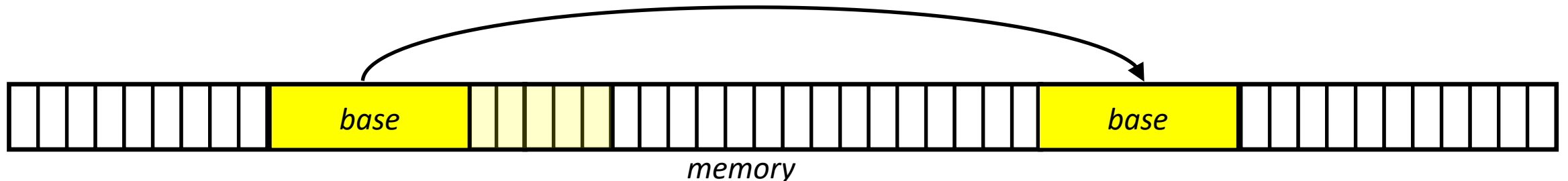


Inheritance (casting)

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



Inheritance (slicing)

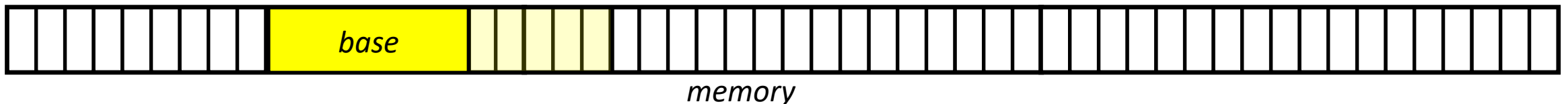
Under the hood:

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 (slicing)

Under the hood:

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

baseDerived.h

```
class Base
{
public:
    char b;
};
class Derived : public Base
{
public:
    char 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 : 1
>>
```

Inheritance (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 (overriding)

Q: What happens if we slice 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

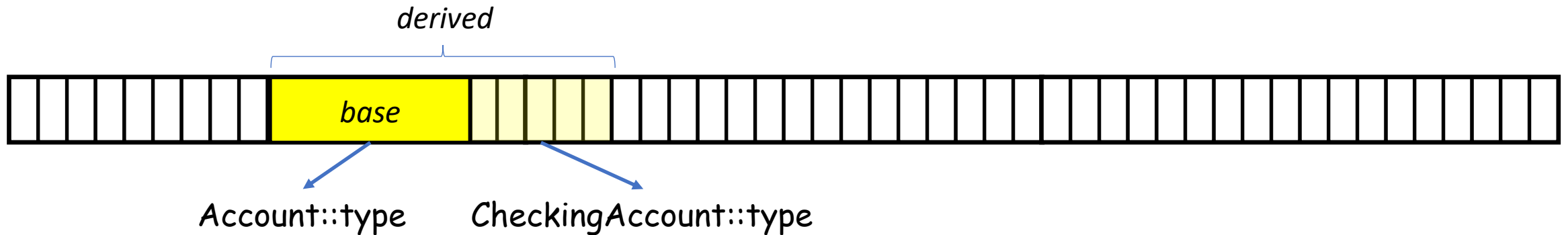
```
#include <iostream>
#include "account.h"
using namespace std;
void PrintType( const Account &acct )
{
    cout << "Type: " << acct.type() << endl;
}
int main( void )
{
    Account acct( 1000 );
    CheckingAccount cAcct( 5000 );
    PrintType( acct );
    PrintType( cAcct );
    return 0;
}
```

Inheritance (overriding)

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



Inheritance (overriding)

Q: What happens if we cast/slice 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 (polymorphism)

- 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
 - 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 (polymorphism)

- 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
 - Use the keyword **virtual** to indicate that a

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 (polymorphism)

- Note:

This only works when you pass the values by reference or by pointer. If you pass by value, the implementation of base class is used.

foo.h

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

main.cpp

```
#include "foo.h"
void Print1( const Base& b ){ b.print(); }
void Print2( const Base* b ){ b->print(); }
void Print3( Base b ){ b.print(); }
int main( void )
{
    Base b;
    Derived d;
    Print1( b ); Print1( d );
    Print2( &b ); Print2( &d );
    Print3( b ); Print3( d );
    return 0;
}
```

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

Inheritance (polymorphism)

Q: When is polymorphism useful?

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 );
    virtual void draw( void );
};
...
```

Inheritance (polymorphism)

Q: When is polymorphism useful?

A: When we only know

shape.h

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

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 (polymorphism)

Q: When is polymorphism useful?

A: When we only know

shape.h

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

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 (polymorphism)

Q: When is polymorphism used?

A: When we only know the base class

shape.h

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

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 (polymorphism)

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

Q: When is polymorphism used?

A: When we only know the base class

shape.h

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

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 (polymorphism)

Q: When is polymorphism used?

A: When we only know the base class

shape.h

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

```
#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;
}
```

```
>> ./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
```

Inheritance (polymorphism)

Q: When is polymorphism used?

A: When we only know the base class

shape.h

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

```
#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;
}
```

```
>> ./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
```

Inheritance (polymorphism)

Q: When is polymorphism used?

A: When we only know the base class

shape.h

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

```
#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;
}
```

```
>> ./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
```

Inheritance (polymorphism)

Q: When is polymorphism used?

A: When we only know the base class

shape.h

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

```
#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;
}
```

```
>> ./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 )
>>
```

Inheritance (multiple)

- 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
>>
```

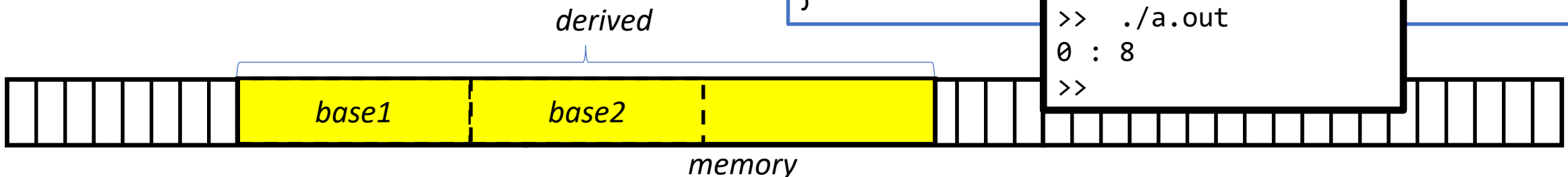
Inheritance (multiple)

- C++ allows a derived class to inherit from multiple base classes
 - Slicing is trickier because we can't 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;
}
```



Inheritance (multiple)

Note:

Watch out for ambiguity when accessing a base class's member!

```
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;
    cout << d.b << endl;
    return 0;
}
```

Inheritance (multiple)

Note:

Watch out for ambiguity when accessing a base class's member!

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;
    d.b << endl;
    return 0;
}
```

```
>> g++ -std=c++11 -Wall -Wextra main.cpp
main.cpp: In function 'int main()':
main.cpp:11:12: error: request for member 'b' is ambiguous
   11 |     cout << d.b << endl;
      |           ^
main.cpp:3:30: note: candidates are: 'size_t Base2::b'
     3 |     class Base2 { public: size_t b; };
      |                               ^
main.cpp:2:30: note: 'size_t Base1::b'
     2 |     class Base1 { public: size_t b; };
      |                               ^
>>
```


Inheritance (multiple)

Note:

Watch out for ambiguity when accessing a base class's member!

This can be resolved by slicing to the appropriate 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;
    cout << ( (Base1&)d ).b << endl;
    return 0;
}
```

Outline

- Exercise 31
- Inheritance
- Review questions

Review questions

1. Do derived classes inherit constructors?

No

Review questions

2. What does **protected** imply for a class field?

Only member functions of the class and member functions of derived class have access to the field.

Review questions

3. What is polymorphism?

When a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Review questions

4. What is the purpose of the `virtual` keyword?

To indicate that the behavior of a function can be overridden by a derived class.

Exercise 32

- Website -> Course Materials -> ex32