LECTURE 7

Derived classes - example

A base class

```
1  class UniversityMember {
2  public:
3    UniversityMember(std::string sFirstName, std::string sSurName);
4    virtual ~UniversityMember();
5    std::string getName() const;
6    virtual std::ostream & printDetails(std::ostream & os) const;
7
8  private:
9    std::string sFirstName_;
10   std::string sSurName_;
11 };
```

- This looks like a simple class with a constructor, destructor, two other `public` member functions and two `private` data members...
- ...except for the two occurrences of the keyword `virtual`.
- Using the keyword `virtual` indicates that one intends to derive other classes from this (that's what makes this a **base class**).
- For proper behaviour the destructor of base classes must be declared `virtual`; this guarantees that both the base and derived classes are destroyed properly when needed (see the structure later, also see slides 22-23).
- The function `printDetails()` is also declared `virtual`, which makes it possible to override (assign different behaviour) in derived classes.

Derived classes - example

A class publicly derived from `UniversityMember`

```
1  class Student : public UniversityMember {
2  public:
3    Student(std::string sFirstName, std::string sSurName,
4       std::string sCollege, std::string sCourse);
5    std::ostream & printDetails(std::ostream & os) const;
6
7  private:
8    std::string sCollege_;
9    std::string sCourse_;
10
11 };
```

- "publicly derived" means that `Student` **is a** `UniversityMember` with some additional members: **the base class' members are inherited by the derived class**.
- Only the additional members are declared in the body of `Student`.
- The non-`virtual` function `getName()` can be called from both the base and derived classes, and it'll execute the implementation given in the base class
- Both the base and the derived classes provide the `virtual` member `printDetails()`, however **when it's called from the derived class it'll do something** `Student`**-specific**.

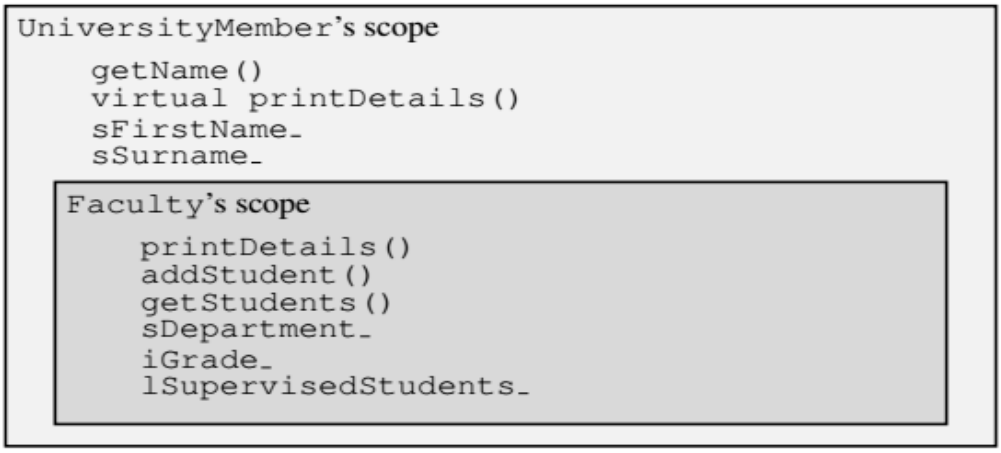
Derived classes - example

Another class publicly derived from `UniversityMember`

```
1  class Faculty : public UniversityMember {
2  public:
3    Faculty(std::string sFirstName, std::string sSurName,
4       std::string sDepartment, unsigned int iGrade);
5    std::ostream & printDetails(std::ostream & os) const;
6    void addStudent(Student &);
7
8  private:
9    std::list<Student> getStudents() const;
10   std::string sDepartment_;
11   unsigned int iGrade_;
12   std::list<Student> lSupervisedStudents_;
13 };
```

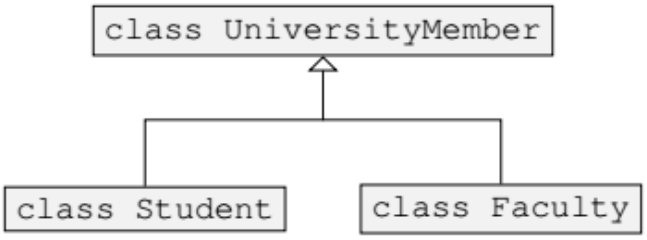
- `Faculty` **is a** `UniversityMember` but not a `Student`, i.e. the common properties of the different `UniversityMember`'s is included in the declaration of the base class
- The derive classes contain the derive class specific members (i.e. `sDepartment_` and `iGrade_`, `lSuperviseStudents`, `addStudent()`, `getStudents()`)
- The `Faculty`-specific members are only accessible from `Faculty` (`friend`'s of faculty, and maybe the classes derived from `Faculty`)

Derived classes - structure

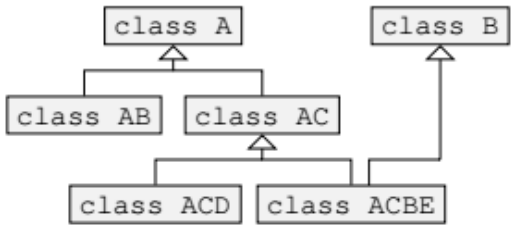


- When an instance of the derived class is created, an instance of the base class is also created in the background (see the implementation of derived constructor)
- **References to base and pointers to base can refer to/point to instances of derived. In such a case the derived specific `virtual` functions are used.** (see the main code of the example).
- Non-`virtual` members of the base can be redefined in the derived class, however this yields an unexpected behaviour (see Effective C++ by Scott Meyers for details).

Derived classes - class hierarchy



- In graphical illustrations, the arrow ⟶ indicates the base-derived relationship (the arrow pointing from the derived to the base).
- More complicated hierarchies are allowed in C++, e.g. one can derive from derived classes, classes can be derived from multiple base classes, etc., e.g.:



Derived classes - example

Implementation of the members of `UniversityMember`

```
1  UniversityMember::UniversityMember(std::string sFirstName,
2                 std::string sSurName)
3    : sFirstName_(sFirstName), sSurName_(sSurName){}
4
5  UniversityMember::~UniversityMember(){}
6
7  std::string UniversityMember::getName() const {
8    return "Name: " + sFirstName_ + " " + sSurName_;
9  }
10
11 std::ostream &
12   UniversityMember::printDetails(std::ostream & os) const {
13   os << getName();
14   return os;
15 }
```

- `getName()` returns a string constructed from the names.
- `printDetails()` copies the `getName()` generated name into an output stream.
- If a derived class does not implement `printDetails()` the implementation of the base is used.

Derived classes - example

Implementation of the members of `Student`

```
1  Student::Student(std::string sFirstName,
2                   std::string sSurName,
3                   std::string sCollege,
4                std::string sCourse)
5    : UniversityMember(sFirstName, sSurName),
6      sCollege_(sCollege),
7      sCourse_(sCourse) {}
8
9  std::ostream & Student::printDetails(std::ostream & os) const {
10   os<< getName() << ", College: " << sCollege_
11     << ", Course: " << sCourse_;
12   return os;
13 }
```

- The constructor of the derived class initializes the base's data members, however the base's private members are not accessible directly. (If it's not done explicitly, the compiler will call the default constructor of the base.)
- Note how the derived class' constructor initializes the base part, using its constructor.
- The derived's members are initialized the usual way.
- The member `printDetails()` is overridden, i.e. a derived specialized version is implemented.

Derived classes - example

Implementation of the members of `faculty`

```
1  Faculty::Faculty(std::string sFirstName,
2                   std::string sSurName,
3                std::string sDepartment,
4                unsigned int iGrade)
5    : UniversityMember(sFirstName, sSurName),
6      sDepartment_(sDepartment),
7      iGrade_(iGrade) {}
8
9
10 std::ostream & Faculty::printDetails(std::ostream & os) const {
11   os<< getName() << ", Department: " << sDepartment_
12     << ", Grade: " << iGrade_;
13   return os;
14 }
```

- `Faculty`'s constructor initialises the base part using its constructor.
- `printDetails()` is overridden.

Derived classes - "is-a" relationship demonstrated

```
1  std::ostream & operator<<(std::ostream & os,
2            const UniversityMember & um) {
3    return um.printDetails(os);
4  }
```

```
1  void testMembers() {
2    UniversityMember um("John","Doe");
3    Student st("Howard","Wolowitz", "St Hugh's",
4          "MSc Engineering");
5    Faculty fa("Sheldon","Cooper","Physics",8);
6
7    cout << "um: " << um << endl;
8    cout << "um.getName(): " << um.getName() << endl << endl;
9    cout << "st: " << st << endl;
10   cout << "st.getName(): " << st.getName() << endl << endl;
11   cout << "fa: " << fa << endl;
12   cout << "fa.getName(): " << fa.getName() << endl << endl;
13 }
```

- The function `operator<<()`'s second argument is a reference to a `UniversityMember`.
- In the implementation of `operator<<()` the `virtual` member of `UniversityMember` is called.
- In the `main` `operator<<()` is called with base and derived arguments. See output on next slide.

Derived classes - "is-a" relationship demonstrated

```
um: Name: John Doe
um.getName(): Name: John Doe

st: Name: Howard Wolowitz, College: St Hugh's,
Course: MSc Engineering
st.getName(): Name: Howard Wolowitz

fa: Name: Sheldon Cooper, Department: Physics, Grade: 8
fa.getName(): Name: Sheldon Cooper
```

- **Functions taking the base class by reference can take reference to derived instances.**
- **In such case, if the virtual member is called, the one belonging to the derived class is looked up and executed.**
- This lookup results in a bit of performance overhead (see exercise).
- When the non-`virtual` base member `getName()` is called on any of the base or derived instances, the one implemented in the base (the only one anywhere implemented) is executed.

Derived classes - slicing demonstrated

```
1  vector<UniversityMember> vec1;
2  vec1.push_back(um);
3  vec1.push_back(st);
4  vec1.push_back(fa);
5  cout << "Slicing: .printeDetails() called on values\n";
6  for(unsigned int i=0; i<vec1.size(); ++i)
7    (vec1[i].printDetails(cout) )<< endl;
```

```
Slicing: .printeDetails() called on values
Name: John Doe
Name: Howard Wolowitz
Name: Sheldon Cooper
```

- **The `push_back()` member of `vector` copies elements by value, and in this case, the derived part of the elements is sliced.**
- When calling the `virtual` member, the one in the base is executed (note: for each instance only the name is displayed, the other details are lost).
- Recall that function objects are passed by value to algorithms. Function objects are required to be monomorphic due to avoid slicing.

Derived classes - polymorphism demonstrated

```
1  typedef sptr::shared_ptr<UniversityMember> UMPtr;
2  vector<UMPtr> vec2;
3  vec2.push_back(UMPtr(new UniversityMember(um)));
4  vec2.push_back(UMPtr(new Student(st)));
5  vec2.push_back(UMPtr(new Faculty(fa)));
6  cout << "\nPolymorphism: ->printeDetails() called on pointers\n";
7  for(unsigned int i=0; i<vec2.size(); ++i)
8    (vec2[i]->printDetails(cout) ) << endl;
```

```
Polymorphism: ->printeDetails() called on pointers
Name: John Doe
Name: Howard Wolowitz, College: St Hugh's,
Course: MSc Engineering
Name: Sheldon Cooper, Department: Physics, Grade: 8
```

- In this example, we store pointers to the base class in a `vector`.
- Copying the pointer by value does not change the object it points to.
- **A pointer of type "pointer to base" can actually be the address of an instance of the derived class.**
- **Calling the `virtual` member through pointers to base (using the operator ->) executes the version of the derived class, i.e. it's polymorphic.**

Derived classes as function arguments

```
1  void testFunction1(UniversityMember um) {
2  //argument taken by value, derived part sliced
3    um.printDetails(cout);
4    cout << endl;
5  }
6  void testFunction2(UniversityMember & um) {
7  //argument taken by reference, derived part kept
8    um.printDetails(cout);
9    cout << endl;
10 }
```

```
1   cout << "\nPolymorphism via reference\n";
2   cout << "testFunction1(um): " ;
3   testFunction1(um);
4   cout << "testFunction1(st): " ;
5   testFunction1(st);
6   cout << "testFunction1(fa): " ;
7   testFunction1(fa);
8   cout << "testFunction2(um): " ;
9   testFunction2(um);
10  cout << "testFunction2(st): " ;
11  testFunction2(st);
12  cout << "testFunction2(fa): " ;
13  testFunction2(fa);
```

Derived classes as function arguments

```
Polymorphism via reference
testFunction1(um): Name: John Doe
testFunction1(st): Name: Howard Wolowitz
testFunction1(fa): Name: Sheldon Cooper
testFunction2(um): Name: John Doe
testFunction2(st): Name: Howard Wolowitz, College: St Hugh's,
Course: MSc Engineering
testFunction2(fa): Name: Sheldon Cooper, Department: Physics,
Grade: 8
```

- Functions taking base type as argument can actually take instances of derived.
- Derived **is a** base, therefore functions taking base **by value**, will use the base part of the instances of derived (derived part is sliced).
- If the function takes base **by reference**, it will keep the derived part of the derived objects and will access the polymorphic behaviour.

Derived classes - another example

```
1  class Base {
2  public:
3    virtual ~Base(){}
4    virtual void memberFn(Base & base) {
5      std::cout << "virtual member of Base" << std::endl;
6    }
7  };
8
9  class Derived : public Base
10 {
11 public:
12   //overridden virtual function
13   void memberFn(Base & base) {
14     std::cout << "overridden virtual of Derived" << std::endl;
15   }
16   //non-virtual member, overloads memberFn(),
17   //feature in Derived only
18   void memberFn(Derived & derived) {
19     std::cout << "additional feature of Derived" << std::endl;
20   }
21 };
```

Derived classes - another example

```
1   Base b;
2   Derived d;
3   Base & rb1 = b;
4   Base & rb2 = d;
5   Derived & rd = d;
6   rb1.memberFn(rb1);
7   rb1.memberFn(rb2);
8   rb1.memberFn(rd);
9   rb2.memberFn(rb1);
10  rb2.memberFn(rb2);
11  rb2.memberFn(rd);
12  rd.memberFn(rb1);
13  rd.memberFn(rb2);
14  rd.memberFn(rd);
```

Derived classes as function arguments

```
virtual member of Base
virtual member of Base
virtual member of Base
overridden virtual of Derived
overridden virtual of Derived
overridden virtual of Derived
overridden virtual of Derived
overridden virtual of Derived
additional feature of Derived
```

- `rb1.memberFn(...)` always calls the member of `Base`
- `rb2` is referred to by a reference to `Base`, therefore only the features declared in `Base` are accessible, that is the additional overloaded method is not visible. `rb2` is recognised to be an instance of `Derived`, the overridden version of the polymorphic feature is called.
- The additional overloaded feature of `Derived` is only called when the variable is explicitly of type `Derived`.

Why do `virtual` destructors matter?

```
1  class BaseWithDefect {
2  public:
3    BaseWithDefect(std::string name) : m_name(name) {}
4    ~BaseWithDefect(){
5      std::cout << "BaseWithDefect with name " << m_name
6          << " is being destroyed." << std::endl;
7    }
8
9  private:
10   std::string m_name;
11 };
12
13 class DerivedFromBWD : public BaseWithDefect {
14 public:
15   DerivedFromBWD(std::string name1, std::string name2) :
16     BaseWithDefect(name1), m_name2(name2) {}
17   ~DerivedFromBWD(){
18     std::cout << "DerivedFromBWD with name " << m_name2
19         << " is being destroyed." << std::endl;
20   }
21
22 private:
23   std::string m_name2;
24 };
```

Why do `virtual` destructors matter?

```
1  BaseWithDefect * bptr1 = new BaseWithDefect("Zero BWD.");
2  BaseWithDefect * bptr2 = new DerivedFromBWD("First BWD.","First D.");
3  DerivedFromBWD * dptr = new DerivedFromBWD("Second BWD.","Second D.");
4  delete bptr1;
5  delete bptr2;
6  delete dptr;
```

```
BaseWithDefect with name Zero BWD. is being destroyed.
BaseWithDefect with name First BWD. is being destroyed.
DerivedFromBWD with name Second D. is being destroyed.
BaseWithDefect with name Second BWD. is being destroyed.
```

Note: The destructor of the base class is not `virtual`, therefore the derived part of `bptr2` is not destroyed.

Derived classes - Summary

- polymorphism can be implemented via derived classes (`public` inheritance)
- the base class defines the common framework (`virtual` and non-`virtual` functions)
- the derived class specific behaviour is implemented via overriding virtual functions
- if the virtual function is not implemented in the derived class, the implementation in base will be used (exception: pure virtual function, see next lecture)
- the base class defines the common behaviour (non-`virtual` member functions and data members), these are also properties of the derived classes
- derived classes can have properties not defined in the base's framework
- classes can be derived from multiple base classes
- polymorphism can be exploited via references or pointers