C++ Programming I

Basics of Object-Oriented Programming
Inheritance

C++ Programming
March 29, 2018

Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- Basics of Inheritance
- Implementing Inheritance
  - Base Class Initialization
  - Overriding Base Class's Methods
  - Order of Construction of Derived Class
- Private and Protected Inheritance
- Multiple Inheritance
- Avoiding Inheritance
- Exam

Basics of Inheritance

Bern University
of Applied Sciences

Basic Inheritance

Introduction

- In programming complex problems are often split into smaller and less complex sub-problems. A common approach is to structure the program into function, thus called **procedural programming**
- More modern programming techniques split the program into multiple classes which are adapted to its respective data and thus called **Object-Oriented Programming**
- The benefit of classes is its simple **reuse** in same or an other application
- Inheritance aims to

  1. **Minimize** the amount of **duplicated code**
  2. Increase the **reusability** of code form the base class
  3. Increase **extensibility**, *i.e.* extend the base class logic
  4. Provide additional **data hiding** capabilities
  5. Provide an **function overriding** mechanism, to adapt the derived classes accordingly
  6. and finally enables **abstract classes**, **interfaces** and **dynamic binding**, thus **polymorphism** (next lesson)

Basic Inheritance

Introduction

- In general, there are two ways of connecting classes:
  1. **Composition:** Classes are built from other classes. Objects are said to have a **“has-a”** relationship, *e.g.* library & book from the exercise
  2. **Inheritance:** Classes form a hierarchy. Objects are said to have a **“is-a”** relationship, *i.e.* features and behaviour are passed to the derived class

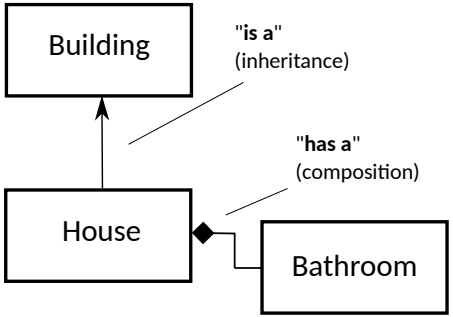
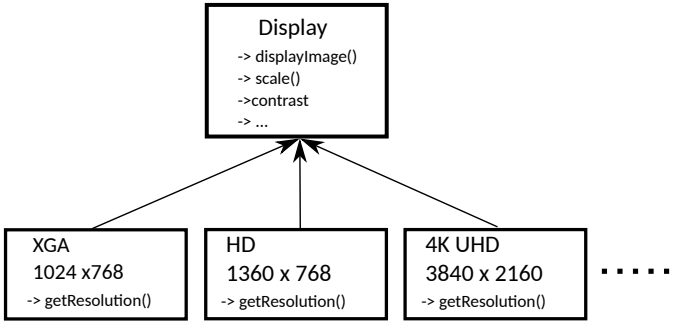


Figure: UML-style diagram.

Basic Inheritance

Example using Display Class



- No redundant code, functionality of display is implemented in one class!
- Easy to extend, *i.e.* add new display generation
- Easy to maintain and fix bugs

Implementing Inheritance

Bern University
of Applied Sciences

Basic Inheritance

C++ Syntax of Derivation

- The general syntax to derive class Base from class Derived is:

```
1  class Base
2  {
3      // ...base class members
4  };
5
6  class Derived : access-specifier Base
7  {
8      // ...derived class members
9  };
```

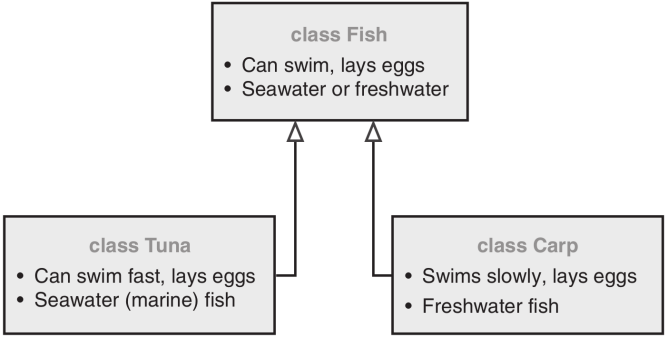
- The access-specifier can be
  1. `public`, where a derived class **is a** base class relationship (most frequently used)
  2. `protected`, where a derived class **has a** base class relationship
  3. `private`, where a derived class **has a** base class relationship
- Note: 2. and 3. are similar to composition
- Let's do a example!

Note:

This lesson starts with `public` inheritance to understand the concept of inheritance and the most frequent form of inheritance before moving on to `private` or `protected` inheritance

Basic Inheritance

Hierarchical Relationship: Fish and subspecies



- `Fish` is a **base class**
- `Tuna` **is a** `Fish`
- `Carp` **is a** `Fish`
- `Tuna` and `Carp` **inherit from** or **derive from** the **base** or **super class**

Basic Inheritance

Hierarchical Relationship: Fish and subspecies

```
1  #include <iostream>
2  using namespace std;
3
4  class Fish
5  {
6  public:
7      bool m_isFreshWaterFish;
8
9      void swim()
10     {
11         if (m_isFreshWaterFish)
12             cout << "swims in lake" << endl;
13         else
14             cout << "swims in sea" << endl;
15     }
16 };
17
18 class Tuna : public Fish // Tuna inherits from Fish
19 {
20 public:
21     Tuna()
22     {
23         m_isFreshWaterFish = false; // Specialisation of
               Fish!
24     }
25 };
26 // --> continuing
```

Basic Inheritance

Hierarchical Relationship: Fish and subspecies

```
1  class Carp : public Fish // Carp inherits from Fish
2  {
3  public:
4      Carp()
5      {
6          m_isFreshWaterFish = true; // Specialisation of
               Fish!
7      }
8  };
9
10 int main()
11 {
12     Carp carpFish;
13     Tuna tunaFish;
14
15     cout << "Carp ";
16     carpFish.swim(); // -> Carp swims in lake
17
18     cout << "Tuna ";
19     tunaFish.swim(); // -> Tuna swims in sea
20
21     return 0;
22 }
```

Warning! Main can change base member

Tuna.m_isFreshWaterFish = true; // but Tuna isn't a fresh water fish!

Basic Inheritance

Access Specifier Keyword `protected`

- A better class `Fish` using the `protected` Keyword to expose its member attribute only to the derived classes

```
1  class Fish
2  {
3  protected: // accessible only to derived classes
4      bool m_isFreshWaterFish;
5
6      void swim()
7      {
8          if (m_isFreshWaterFish)
9              cout << "swims in lake" << endl;
10         else
11             cout << "swims in sea" << endl;
12     }
13 };
14
15 class Tuna : public Fish // Tuna inherits from Fish
16 {
17 public:
18     Tuna()
19     {
20         m_isFreshWaterFish = false; // set protected
               member
21     }                               // of base class
22 };
23 ..
24 .
```

- Ensuring that derived classes can safely inherit base class attributes!

Basic Inheritance

Base Class Initialization - Overloaded Constructor

- To enforce proper initialization of base classes by the derived classes an overloaded constructor is provided by the base class

```
1  #include <iostream>
2  using namespace std;
3
4  class Fish
5  {
6  protected:
7      bool m_isFreshWaterFish; // accessible only to
           derived
8                               // classes
9  public:
10     // Fish constructor forcing derived class to set
           member
11     Fish(bool isFreshWater) :
           m_isFreshWaterFish(isFreshWater){}
12
13     void swim()
14     {
15         if (m_isFreshWaterFish)
16             cout << "Swims in lake" << endl;
17         else
18             cout << "Swims in sea" << endl;
19     }
20 };
```

- Note: The derived classes are not accessing the `protected` member, so we could set it to `private` to enhance security

Basic Inheritance

Base Class Initialization - Initialization Lists

- **Initialization lists** invoke the appropriate base class constructor via the constructor of the derived class

```
1  class Tuna : public Fish
2  {
3  public:
4      Tuna() : Fish(false) {} // Calling base constructor
5  };                          // in initialisation list
6
7  class Carp: public Fish
8  {
9  public:
10     Carp() : Fish(true) {} // Calling base constructor
11 };                         // in initialisation list
12
13
14 int main()
15 {
16     Carp carpFish;
17     Tuna tunaFish;
18
19     cout << "Carp ";
20     carpFish.swim(); // -> Carp swims in lake
21
22     cout << "Tuna ";
23     tunaFish.swim(); // ->Tuna swims in sea
24     return 0;
25 }
```

Basic Inheritance

Overriding Base Class's Methods

- Note: `isFreshWaterFish` became private in the base class `Fish`

```
1  #include <iostream>
2  using namespace std;
3
4  class Fish
5  {
6  private:
7      bool m_isFreshWaterFish; // not accessible by
8                               // derived class
9  public:
10     // Fish constructor
11     Fish(bool isFreshWater) :
           m_isFreshWaterFish(isFreshWater){}
12
13     void swim() // base class method
14     {
15         if (m_isFreshWaterFish)
16             cout << "Swims in lake" << endl;
17         else
18             cout << "Swims in sea" << endl;
19     }
20 };
```

Basic Inheritance

Overriding Base Class's Methods

- If the derived classes implements the same methods with the same signatures as in the base class it inherits from, it **overrides** those methods

```
1  class Tuna : public Fish
2  {
3  public:
4      Tuna() : Fish(false) {}
5
6      void swim() // Overriding base class method
7      {
8          cout << "Tuna swims fast" << endl;
9      }
10 };
11
12 class Carp : public Fish
13 {
14 public:
15     Carp() : Fish(true) {}
16
17     void swim() // Overriding base class method
18     {
19         cout << "Carp swims slow" << endl;
20     }
21 };
```

Basic Inheritance

Overriding Base Class's Methods

```
1  int main()
2  {
3      Carp carpFish;
4      Tuna tunaFish;
5
6      carpFish.swim(); // -> Carp swims slow
7      tunaFish.swim(); // -> Tuna swims fast
8
9      return 0;
10 }
```

- The method `swim()` of the appropriate base class is called
- The only way to invoke `Fish::Swim()` is by having main() use the scope resolution operator `(::)` in explicitly invoking `Fish::Swim()`

```
1  Tuna tunaFish;
2
3  tunaFish.swim();         // will invoke Tuna::swim()
4
5  tunaFish.Fish::swim(); // invokes Fish::swim()
6                         // using instance of Tuna
```

Basic Inheritance

Order of Construction & Destruction

```
1  #include <iostream>
2  using namespace std;
3
4  class FishMember
5  {
6  public:
7     FishMember(){cout << "FishMember constructor" <<
          endl;}
8     ~FishMember(){cout << "FishMember destructor" <<
          endl;}
9  };
10
11 class Fish // is base class
12 {
13 protected:
14    FishMember m_fishMember; // composition with
          FishMember class
15
16 public:
17    // Fish constructor
18    Fish(){cout << "Fish constructor" << endl;}
19    ~Fish(){cout << "Fish destructor" << endl;}
20 };
```

- This example show the order of construction and destruction when **inheritance and composition** is involved

Basic Inheritance

```
1  class TunaMember // member of derived
2  {
3  public:
4      TunaMember(){cout << "TunaMember constructor" <<
           endl;}
5      ~TunaMember(){cout << "TunaMember destructor" <<
           endl;}
6  };
7
8  class Tuna: public Fish // derives from base
9  {
10 private:
11     TunaMember m_tuneMember;
12
13 public:
14     Tuna(){cout << "Tuna constructor" << endl;}
15     ~Tuna(){cout << "Tuna destructor" << endl;}
16 };
17
18 int main()
19 {
20     Tuna tuna;
21 }
22
23 //  FishMember constructor
24 //  Fish constructor          --> base class finished
25 //  TunaMember constructor
26 //  Tuna constructor          --> dervice class finished
27 //  Tuna destructor
28 //  TunaMember destructor     --> derived class destructed
29 //  Fish destructor
30 //  FishMember destructor     --> base class destructed
```

Private and Protected Inheritance

Basic Inheritance

Order of Construction & Destruction

- So far we have always used the most common access specifier *public* to to derive from base class, thus called *public*-inheritance
- Recap **Access Levels of components:**
  1. **public:** accessible everywhere
  2. **private:** accessible only in methods of the own class
  3. **protected:** accessible only in methods of the own class or derived class
- Using inheritance the **access levels** or visibility of the derived components in the derived classes can be changed:
  1. `class A: public B`
     access level not changed
  2. `class A: protected B`
     access level changed from `public` to `protected`
  3. `class A: private B`
     access level `public` and `protected` changed to `private`

```
1  class Base
2  {
3      // ... base class members and methods
4  };
5  class Derived : private Base // or protected Base
6          // private inheritance
7  {
8      // ... derived class members and methods
9  };
```

Basic Inheritance

Access Specifier

- The following table summarizes the possible access level modifications

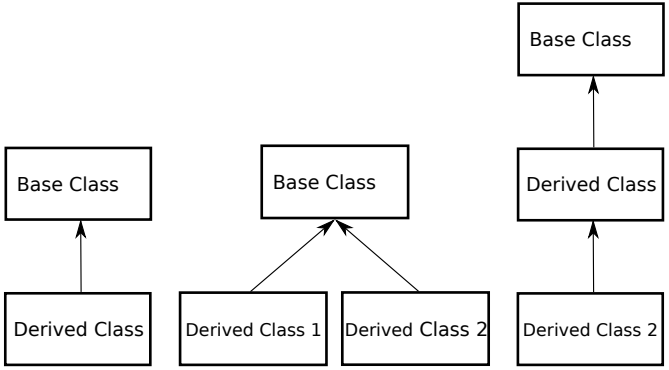
Access Specifier In base class	Access Specifier when inherited publicly	Access Specifier when inherited privately	Access Specifier when inherited protectedly
Public	Public	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible
Protected	Protected	Private	Protected

- `private` and `protected` inheritance describe a **has-a** relationship
- For simplicity, prefer composition over `private` and `protected` inheritance!

Single Inheritance

Single Base Classes

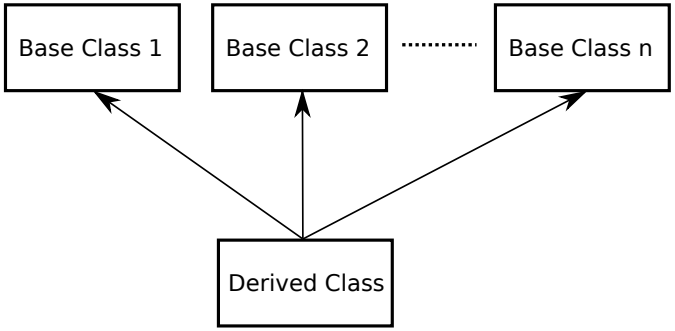
- Single, hierarchical and multilevel inheritance inherit from one single base class



Multiple Inheritance

Multiple Base Classes

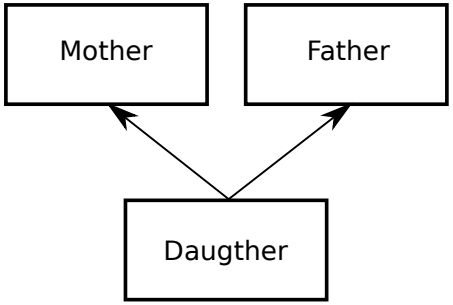
- Multiple Inheritance is a feature of C++ where a class can inherit from more than one class



Multiple Inheritance

Multiple Base Classes

- For example to describe genetic inheritance



Avoiding Inheritance

Avoiding Inheritance

The keyword `final`

- If you want to avoid inheritance at some point...

```
1  #include "mother.h"
2  #include "father.h"
3
4  class Daughter : public Mother, public Father
5  {
6  public:
7      void introduce()
8      {
9          std::cout << "My name is Amy" << std::endl;
10     }
11 };
```

- add the keyword `final` to ensure that the derived class can't be used as base class

```
1  class Daughter final : public Mother, public Father
2  {
3  public:
4      void introduce()
5      {
6          std::cout << "My name is Amy" << std::endl;
7      }
8  };
```

- In addition to classes, `final` can be used on member functions in controlling polymorphic behaviour (next lesson)

Exam

Exam
Information

General Information:

- The midterm exam takes place at **12.04.2018** at **Uni-S** in room **A022**
- Style: **Written exam on paper**
- Duration: **90 minutes**
- Open book or cheat sheet?
- **No laptop** or internet capable devices allowed

You have to solve:

- Multiple choice
- Find and fix bugs in given code
- Determine the output/result of given code
- Write code (orthographic errors are not counted!)
- Skill questions

Thank You
Questions

???