12. Abstract Classes and **`virtual`**

12.1. Abstract Classes and **`virtual`**

Abstract classes are classes that cannot be instantiated and has one or more *pure virtual* (or abstract) methods: `virtual void print() = 0;`

A pure virtual method needs to be overridden by a concrete (i.e., non-abstract) derived class and is indicated in the declaration with the syntax `= 0` behind the method's declaration.

Abstract classes cannot be used as parameter types, as function return types, or as explicit conversion types. Pointers and references to abstract classes can be declared.

12.1. Abstract Classes and virtual

Abstract.cpp

```
#include <iostream>

class AbstractClass {  // Class has pure virtual method:
 public:
  virtual void printName() const = 0;
 protected:
  std::string name = "myName";   // default initialization since C++11
};

class DerivedClass : public AbstractClass {
 public: // printName is overridden and implemented here:
  virtual void printName() const override { std::cout << name << "\n"; }
};

int main() {
  // This would fail:  AbstractClass myObject;
  DerivedClass myDerivedObject;  // The abstract class forces the
  myDerivedObject.printName();   // implementation of printName
}
```

12.1. Abstract Classes and `virtual`

`virtual` functions or methods can be overridden in derived classes. The overriding is preserved, even if the actual type of the class is not known at compile-time (i.e., when the derived class is handled using a pointer or reference to the base class).

`override` (since C++ 11) can be mentioned after the method declaration, to explicitly show intent to override a method. The compiler can this way stop at programmer's mistakes (for instance, when the method's name was mistyped).

`final` can be mentioned after the method declaration, to explicitly signal that no further subclasses can override this method anymore.

12.1. Abstract Classes and **virtual -- override**

```
#include <iostream>

class BaseClass {
 public:
  virtual void print() const {
   std::cout << "Base Class. \n";
  }
};

class DerivedClass : public BaseClass {
 public:
  virtual void print() const override {
    std::cout << "Derived Class. \n";
  }
};
```

```
int main() {

  BaseClass base; DerivedClass derived;

  BaseClass & bref = base;
  BaseClass & dref = derived;
  bref.print();  // "Base Class."
  dref.print();  // "Derived Class."

  BaseClass * bpnt = &base;
  BaseClass * dpnt = &derived;
  bpnt->print();  // "Base Class."
  dpnt->print();  // "Derived Class."

  bref.BaseClass::print(); // "Base Class."
  dref.BaseClass::print(); // "Base Class."

}
```

12.1. Abstract Classes and `virtual` -- `final`

Final.cpp

```
class AbstractClass {  // Class has pure virtual method:
 public:
  virtual void printName() const = 0;
};

class DerivedClass : public AbstractClass {
 public:  // printName is overridden and implemented here:
  virtual void printName() const override { std::cout << "name. \n"; }
};

class FinalClass : public DerivedClass {
 public:  // printName is final-overridden and re-implemented here:
  virtual void printName() const final { std::cout << "Name. \n"; }
};

class AnotherClass : public FinalClass {
 public:  // printName was final in FinalClass, cannot be overridden:
// virtual void printName() const { std::cout << "NAME. \n"; }
};
```

12.1. Abstract Classes and **virtual**

The following code shows that a destructor is not inherited, so objects that are freed in this way do not call the derived class' destructor:

```
#include <iostream>

class BaseClass {  // ~BaseClass illustration:
 public:
  ~BaseClass() { std::cout << " BaseClass resources freed \n"; }
};

class DerivedClass : public BaseClass {  // ~DerivedClass illustration:
 public:
  ~DerivedClass() { std::cout << "DerivedClass resources freed \n"; }
};

int main() {
  BaseClass * base = new DerivedClass;
  delete base;   // " BaseClass resources freed \n"
}
```

12.1. Abstract Classes and virtual

Yet, a *virtual* destructor from a base class is always overridden by derived destructors, allowing the following:

```
#include <iostream>

class BaseClass {  // ~BaseClass call is virtual => calls ~DerivedClass
 public:
  virtual ~BaseClass() { std::cout << " BaseClass resources freed \n"; }
};

class DerivedClass : public BaseClass {  // ~DerivedClass afterwards calls ~BaseClass,
 public:                                 // following the typical destructor order
  virtual ~DerivedClass() { std::cout << "DerivedClass resources freed \n"; }
};

int main() {
  BaseClass * base = new DerivedClass;
  delete base;   // "DerivedClass resources freed \n BaseClass resources freed \n"
                 //    ^-- note that now both are called
}
```

12.2. The Non-Virtual Interface Idiom

Remember from Chapter 8: Polymorphism in C++ relies on methods from a base class being declared as [**virtual**](#) .

When **Dog** and **Fish** are classes that inherit from **Animal**, objects from these classes have a custom **print()** method, overloading from Animal's **virtual print()** method:

```
Animal * animal;
animal = new Dog("Scooby");
animal->print();  // prints out: I am Scooby. Bark!
animal = new Fish("Salmon");
animal->print();  // prints out: I'm Salmon (fish)
```

12.2. The Non-Virtual Interface Idiom

polyDemo.cpp

```
#include <iostream>
#include <cstdlib>

class Animal {  // Animal class stores the species and prints this in print()
 protected:
  std::string _species;
 public:
  Animal(std::string species) { _species = species; }
  virtual void print() const { std::cout << "I'm " +  _species << '\n'; }
};

class Dog : public Animal {  // Dogs inherit species from Animal and have a name
 protected:
  std::string _name;
 public:
  Dog(std::string name) : Animal("dog"), _name(name) {}
  void print() const override { std::cout << "I am " << _name << ". Bark!\n"; }
};
```

12.2. The Non-Virtual Interface Idiom

polyDemo.cpp

```
class Fish : public Animal {  // Fishes have species and subspecies
 protected:
  std::string _subspecies;
 public:
  Fish(std::string subspecies) : Animal("fish"), _subspecies(subspecies) {}
  void print() const override { std::cout << "I'm " << _subspecies << " (fish)\n"; }
};

int main() {
  Animal * animals[4] = { new Dog("Snowy"), new Fish("Salmon"),
                          new Dog("Scooby"), new Animal("some animal")};
  for (auto i=0; i<15; i++) {
    Animal * a = animals[rand() % 4];  // a is a polymorph variable: its
    a->print();                        // print's behavior depends on the
  }                                    // object that a points to
}
```

12.2. The Non-Virtual Interface Idiom

`Animal`'s **`virtual print()`** method is public. Problems that could occur here are:

- Sub classes do repeat code: The only part that changes is the string to print, but each class needs **`std::cout << ... << std::endl;`** code
- The base class **`Animal`** cannot make guarantees about what the **`print()`** does: Sub-classes may do something completely different as originally intended

This can be fixed by using a **non-virtual interface** that is supplemented by a **private virtual function** that allows polymorphic behaviour. The private virtual methods are called by public non-virtual methods: See next slide

12.2. The Non-Virtual Interface Idiom

polyNVIDemo.cpp

```
#include <iostream>
#include <cstdlib>

class Animal {  // Animal class stores the species and prints this in print()
 protected:
  std::string _species;
 public:
  Animal(std::string species) { _species = species; }
  void print() const { std::cout << getSound() << std::endl; }
 private:
  virtual std::string getSound() const { return "I'm " + _species; };
};

class Dog : public Animal {  // Dogs inherit species from Animal and have a name
 protected:
  std::string _name;
 public:
  Dog(std::string name) : Animal("dog"), _name(name) {}
 private:
  std::string getSound() const override { return "I am " + _name + ". Bark!"; }
};
```

12.2. The Non-Virtual Interface Idiom

polyNVIDemo.cpp

```
class Fish : public Animal {  // Fishes have species and subspecies
 protected:
  std::string _subspecies;
 public:
  Fish(std::string subspecies) : Animal("fish"), _subspecies(subspecies) {}
 private:
  std::string getSound() const override { return "I'm " + _subspecies + " (fish)"; }
};

int main() {
  Animal * animals[4] = { new Dog("Snowy"), new Fish("Salmon"),
                          new Dog("Scooby"), new Animal("some animal")};
  for (auto i=0; i<15; i++) {
    Animal * a = animals[rand() % 4];  // a is a polymorph variable: its
    a->print();                        // print's behavior depends on the
  }                                    // object that a points to
}
```

12.2. The Non-Virtual Interface Idiom

Thus: Non-Virtual Interfaces decouple a class' public interface (e.g., **`Animal`**s **`print()`**) by making it non-virtual, from functions (e.g., **`getSound()`**) that are providing customization points for sub-classes (e.g., **`Dog`** or **`Fish`**).

As an idiom, Non-Virtual Interface is a programming guideline, implementing the Template Method design pattern (not to be confused with C++ templates).

A complete treatise on virtuality can be read in "Virtuality", by Herb Sutter (in C/C++ Users Journal, 19(9), September 2001).

12.3. Multiple Inheritance and the Diamond Problem

Classes often inherit from multiple interfaces (as abstract classes, using multiple pure abstract classes with only pure virtual public methods and static const attributes).

InterfaceExample.cpp

```
class MyInterface {
 public:
  virtual int getFormulaWithX() const = 0;
  virtual ~MyInterface() {};
 public:
  static const int X = 7;
};
```

Classes can in fact also inherit in C++ from multiple base classes in general by separating them with a comma, e.g.:

```
class DerivedClass : public ClassA, public ClassB {
```

In that case, constructors of inherited classes are called in the same order in which they are inherited. The destructors are called in reverse order of the constructors.

12.3. Multiple Inheritance and the Diamond Problem

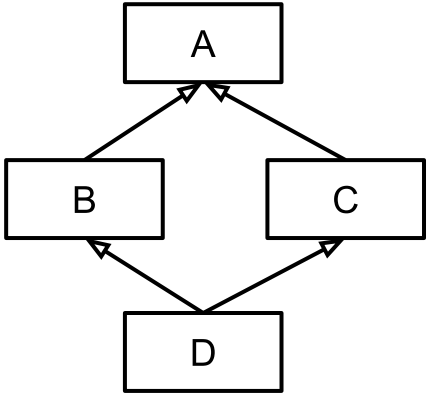
```cpp
#include <iostream>

class ClassA {
 public:
  ClassA() { std::cout << "Class A constructed.\n"; };
};
class ClassB {
 public:
  ClassB() { std::cout << "Class B constructed.\n"; };
};
class DerivedClass : public ClassA, public ClassB {
 public:
  DerivedClass() { std::cout << "Derived Class constructed.\n"; };
};

int main() {
  DerivedClass myDerivedObject;  // this calls first A's, then B's constructor
}
```

12.3. Multiple Inheritance and the Diamond Problem

The *diamond problem* occurs when two parent classes of a class (class B and class C on the right) have a common child class (D) that inherits from both, and a common parent class (A).



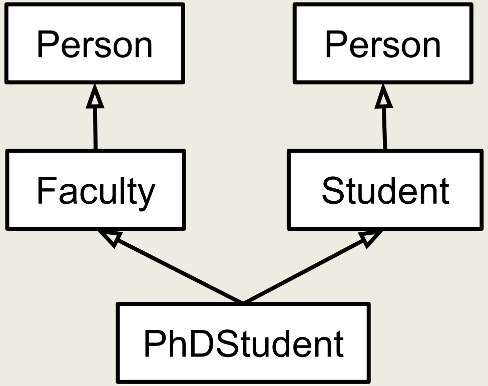
This will lead to the constructor of A being called twice (once via B and once via C). Similarly, the destructor of class A is called twice as well, and objects of class D have two copies of all of A's attributes and methods.

When B and C both inherit a function/method **`m()`** from A, which is then meant when an object **`d`** from Class D calls **`d.m()`** ?

12.3. Multiple Inheritance and the Diamond Problem

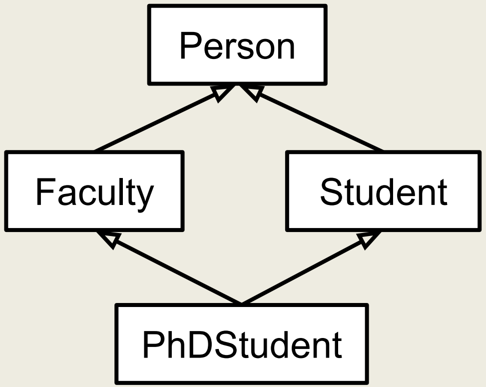
```
struct Person {  // Person:
  Person() { std::cout << "Person constructed.\n"; };
  void print() { std::cout << "in print()\n"; }
};
struct Faculty : public Person {  // Faculty is a Person
  Faculty() { std::cout << "Faculty constructed.\n"; };
};
struct Student : public Person {  // Student is a Person
  Student() { std::cout << "Student constructed.\n"; };
};
struct PhDStudent : public Faculty, public Student {  // PhDStudent is both
  PhDStudent() { std::cout << "PhDStudent constructed.\n"; };
};

int main() {
  PhDStudent phd;  // note: this object will contain two copies of a person
  // phd.print();  // error: member found in multiple base-class subobjects
}
```



12.3. Multiple Inheritance and the Diamond Problem

```
struct Person {  // Person:
   Person() { std::cout << "Person constructed.\n"; };
};
struct Faculty : virtual public Person {
   Faculty() { std::cout << "Faculty constructed.\n"; };
};
struct Student : virtual public Person {
   Student() { std::cout << "Student constructed.\n"; };
};
struct PhDStudent : public Faculty, public Student {
   PhDStudent() { std::cout << "PhDStudent constructed.\n"; };
};
```



Using *virtual inheritance*, C++ is told that there is one common Person base object for both Faculty and Student and their subclasses (PhDStudent).

12.4. Templated Interfaces

```cpp
#include <iostream>

template <class T>  // force subclasses to
class IMenuItem {   // implement printItem:
 public:
  IMenuItem(T item) : item(item) {}
  virtual void printItem() const = 0;
 protected:
  const T item;     // and hold item here, too
};

template <class T>
class Item : public IMenuItem<T>{
 public:  // implementation of printItem:
  Item(T item) : IMenuItem<T>(item) {}
  void printItem() const override {
    std::cout << "Choice: " << this->item << '\n';
  };
};
```

12.4. Templated Interfaces

```
int main() {  // see next slide for a cleaner version since C++17+
  // menu list where items are given an integer:
  std::array<IMenuItem<int> *, 3> menu
     = { new Item<int>(0), new Item<int>(1), new Item<int>(5)};
  for (auto i = 0; i < menu.size(); i++)
    menu[i]->printItem();
  // menu list where items are given a character:
  std::array<IMenuItem<char> *, 3> menu2
     = { new Item<char>('a'), new Item<char>('b'), new Item<char>('c')};
  for (auto i = 0; i < menu2.size(); i++)
    menu2[i]->printItem();
  // menu list where items are given a string:
  std::array<IMenuItem<std::string> *, 2> menu3
     = { new Item<std::string>(std::string("optionA")),
         new Item<std::string>(std::string("optionB"))};
  for (auto i = 0; i < menu3.size(); i++)
    menu3[i]->printItem();
}
```

12.4. Templated Interfaces

```
int main() {  // with class template argument deduction (C++17+) & range based loops
  // menu list where items are given an integer:
  std::array menu = { new Item(0), new Item(1), new Item(5) };
  for (auto i : menu) i->printItem();

  // menu list where items are given a character:
  std::array menu2 = { new Item('a'), new Item('b'), new Item('c') };
  for (auto i : menu2) i->printItem();

  // menu list where items are given a string:
  std::array menu3 = { new Item(std::string("optionA")),
                       new Item(std::string("optionB")) };
  for (auto i : menu3) i->printItem();

}
```

12.5. Strategy Pattern

The Strategy Design Pattern defines encapsulated algorithms (strategies) via an interface to make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it through a context.

- The Strategy ***interface*** declares operations or methods common to all supported versions of some algorithm
- The Strategy ***context*** uses this interface to call the algorithm defined by concrete Strategies

Example: Imagine having to develop a shopping cart, for which the payment operation can be switched at runtime for each payment (PayPal, Credit Card, etc.).

12.5. Strategy Pattern

Strategy.cpp

```
#include <iostream>
template <class Currency>
struct PaymentStrategy {  // Strategy Interface
  virtual void pay(Currency amount) = 0;
  virtual ~PaymentStrategy() = default;  // use compiler-generated default
};
// Concrete payment strategies:
template <class Currency>
struct CreditCardPayment : public PaymentStrategy<Currency> {
  void pay(Currency amount) override {
    std::cout << "Paid " << amount << " Euro using Credit Card.\n";
  }
};
template <class Currency>
struct PayPalPayment : public PaymentStrategy<Currency> {
  void pay(Currency amount) override {
    std::cout << "Paid " << amount << " Euro using PayPal.\n";
  }
};
```

12.5. Strategy Pattern

Strategy.cpp

```
// The shopping cart class contains a unique strategy smart pointer, whic
// can be set to any of the above payment strategies. The checkout function
// will then call the strategies' pay method (polymorphism). This is the
// Strategy Context.
template <class Currency>
class ShoppingCart {
  std::unique_ptr<PaymentStrategy<Currency>> strategy;
 public:
  void setPaymentStrategy(std::unique_ptr<PaymentStrategy<Currency>> ps) {
    strategy = std::move(ps);
  }
  void checkout(Currency total) {
    if (strategy) strategy->pay(total);
    else std::cout << "No payment method selected.\n";
  }
};
```

12.5. Strategy Pattern

Strategy.cpp

```
int main() {
  ShoppingCart<double> cart;

  // std::make_unique<CreditCardPayment>() is a function (C++14 and above), which
  // creates a std::unique_ptr smart pointer to a new CreditCardPayment object:
  cart.setPaymentStrategy(std::make_unique<CreditCardPayment<double>>());
  cart.checkout( 123.45 );

  cart.setPaymentStrategy(std::make_unique<PayPalPayment<double>>());
  cart.checkout( 67.89 );
}
```

12.6. Type Traits and `if constexpr`

What if the output of the strategies in the payment example needs to be different (e.g., for **`int`** vs **`double`**), or if we want to avoid types (e.g., **`char`**)?

`if constexpr` (since C++17) enables *compile-time* branching with type traits and can thus avoid generating invalid code for types that don't match the condition, e.g.:

> **`if constexpr (std::is_integral<Currency>::value)`** ensures this branch only compiles for integral types like integers.
>
> **`if constexpr (std::is_floating_point<Currency>::value)`** ensures this branch only compiles for floating points types such as float or double.
> **`std::fixed`** and **`std::setprecision(2)`** can then apply monetary formatting.

12.6. Type Traits and if constexpr

What if the output of the strategies in the payment example needs to be different (e.g., for **int** vs **double**), or if we want to avoid types (e.g., **char**)?

StrategyTypeTraits.cpp

```
// Concrete Payment Strategies:
template <class Currency>
struct CreditCardPayment : public PaymentStrategy<Currency> {
  void pay(Currency amount) override {
    std::cout << "Paid ";
    if constexpr (std::is_integral<Currency>::value) {
      std::cout << amount << " (int) Euro";
    } else if constexpr (std::is_floating_point<Currency>::value) {
      std::cout << std::fixed << std::setprecision(2);  // set precision to 2 decimals
      std::cout  << amount << " (float) Euro";
    } else {
      std::cout << " something with an unsupported type";
    }
    std::cout " using Credit Card.\n";
  }
};
```