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

```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
  return 0;
}
```

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

`virtual` functions or method can be overridden in derived classes. The overriding is preserved, even 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`

```cpp
#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";
  }
};
```

```cpp
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."

  return 0;
}
```

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"
  return 0;
}
```

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"
  return 0;      //   ^-- 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];
  animals[0] = new Dog("Snowy");       // animals[] has pointers
  animals[1] = new Fish("Salmon");     // to objects of different
  animals[2] = new Dog("Scooby");      // subclasses (Dog, Fish, etc.)
  animals[3] = new Animal("some animal");  // or the animal base class
  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
  return 0;
}
```

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];
  animals[0] = new Dog("Snowy");       // animals[] has pointers
  animals[1] = new Fish("Salmon");     // to objects of different
  animals[2] = new Dog("Scooby");      // subclasses (Dog, Fish, etc.)
  animals[3] = new Animal("some animal");  // or the animal base class
  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
  return 0;
}
```

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 can inherit in C++ from multiple base classes. Classes can thus inherit from multiple abstract classes, using multiple pure abstract classes with only pure virtual public methods and static const attributes (similar to interfaces in Java).

InterfaceExample.cpp

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

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

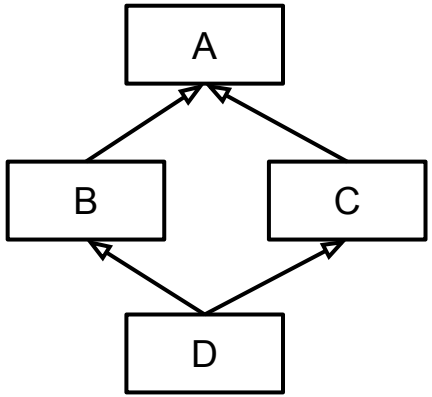
```cpp
#include <iostream>

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;  // note: this calls first A's, then B's constructor
  return 0;
}
```

12.3. Multiple Inheritance and the Diamond Problem

The *diamond problem* occurs when two superclasses of a class (class B and class C on the right) have a common base class (D).



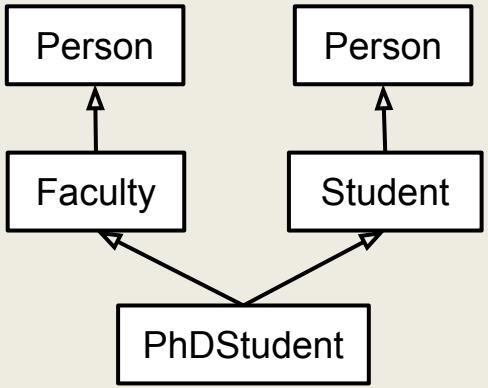
This will lead to the constructor of A being called twice (once via B and once via C). Similarly, the destructor of class A is called twice as well, and objects of class D have two copies of all of A's attributes and methods.

When B and C both inherit a 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

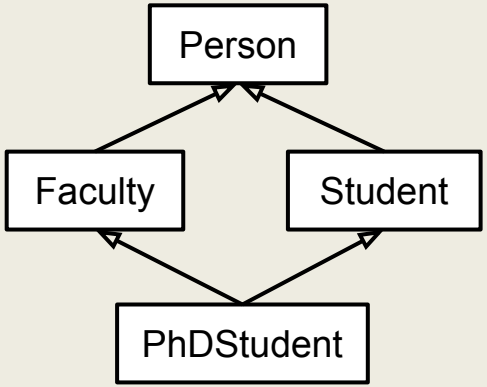
```
class Person {  // Person:
public:
 Person() { std::cout << "Person constructed.\n"; };
};
class Faculty : public Person {  // Faculty is a Person
 public:
  Faculty() { std::cout << "Faculty constructed.\n"; };
};
class Student : public Person {  // Student is a Person
 public:
  Student() { std::cout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

int main() {
  PhDStudent phd;  // note: this object will contain two copies of a person
  return 0;
}
```



12.3. Multiple Inheritance and the Diamond Problem

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



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

12.4. Templated Interfaces

```
#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() {
  // 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();
  return 0;
}
```

12.4. Templated Interfaces

```
int main() {  // class template argument deduction (since 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();

  return 0;
}
```

13. Enumerators, `struct` and `union`

13.1. Basics of `enum`

An enumerator (`enum`) creates a data type that can take the value in a set of named integral constants. By default, the first will be `0`, the second `1`, etc.

```
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  level_t danger = HIGH;  // note: HIGH == 2
  std::cout << danger << "\n";
  return 0;
}
```

The integer values that represent the constants can also be set and controlled explicitly. They are numbered in increasing order:

```
enum battery_t { FULL = 100, ADEQUATE = 50, EMPTY = 0 };
enum level_t { LOW = -100, MEDIUM, HIGH };  // MEDIUM == -99, HIGH == -98

enum day_t { MON, TUE, WED, THU, FRI, SAT, SUN = 7};
// MON == 1, TUE == 2, WED == 3, THU == 4, FRI == 5, SAT == 6
```

13.1. Basics of **enum**

The advantage of **enum** is that the code is readable easier to maintain variables that can take any of a given set of variables :

```
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  level_t humidity = LOW;
  std::string s;
  switch (humidity) {
    case LOW: s = "low"; break;
    case MEDIUM: s = "medium"; break;
    case HIGH: s = "high"; break;
  }
  std::cout << "The humidity is " << s << "\n";
  return 0;
}
```

13.1. Basics of **enum**

Since the values are mapped to integers, this might lead to problems:

```
#include <iostream>
int main() {
  enum level_t { LOW, MEDIUM, HIGH };
  enum battery_t { FULL, ADEQUATE, EMPTY };  // LOW cannot be reused here
  level_t status1 = LOW;      // multiple types' values are basically integers,
  battery_t status2 = EMPTY;  // the following will lead to a warning only:
  std::cout << ( status1 == status2 ) << "\n";
  return 0;
}
```

Typically, **enum**s are used when values are unlikely going to change, such as week days, months, colors, card values.

13.2. Scoped enumeration `enum class`

Since C++11, a *scoped* enumeration (`enum class`) data type is a type-safe enumerator (not a class) that is not implicitly convertible to an integer.

```
enum class Level { LOW, MEDIUM, HIGH };
Level status1 = Level::LOW;  // "status1 = LOW" would lead to error

enum class Battery { FULL, ADEQUATE, EMPTY };
Battery status2 = Battery::EMPTY;  // "status2 = EMPTY" would lead to error

// the following will lead to an "invalid operands" error:
std::cout << ( status1 == status2 ) << "\n";
```

The scoped enumeration's underlying data type can be set explicitly:

```
enum class Choice : int8_t { YES, MAYBE, NO, UNKNOWN };
```

13.2. Scoped enumeration `enum class`

A scoped enumeration cannot be compared to, or use the constants as, integers.

`enum class Level { LOW, MEDIUM, HIGH };` will cause:

`Level l = Level::MEDIUM; std::cout << level;` to result in an error.

The need for scope might also make code less terse and longer:

```
enum class Level { LOW, MED, HIGH };
Level humidity = Level::LOW;
std::string s;
switch (humidity) {
  case Level::LOW: s = "low"; break;
  case Level::MEDIUM: s = "med"; break;
  case Level::HIGH: s = "high"; break;
}
```

Since C++20, `using enum` can import enumerators in the local scope:

```
enum class Level { LOW, MED, HIGH };
Level humidity = Level::LOW;
std::string s;
switch (humidity) {
  using enum Level;
  case LOW: s = "low"; break;
  case MEDIUM: s = "med"; break;
  case HIGH: s = "high"; break;
}
```

13.3. `typedef` and `struct`

Structures (preceded by `struct`) historically combine variables of different types, similar to how arrays combine variables of the same type.
Structures are semantically *very* similar to classes, but by default do not set attributes or methods to private (plus more).

```
struct anonExamEntry {
 long studentId = 0;  // initialization here
 float grade = 1.0;   // possible since C++11
};

anonExamEntry entry1;
entry1.studentId = 17017491;
entry1.grade = 2.3;
```

13.3. **typedef** and **struct**

typedef is a keyword that is used to assign a new name to any existing data-type:

```
typedef int integer; integer i = 9;
```

In C, new variables of a particular structure type would need the keyword **struct** to be added each time:

```
struct examEntry {
  long studentId;
  float grade;
};
struct examEntry entry;  // "anonExamEntry entry;" not possible in C
```

Which is why **typedef** is used to provide a new name instead:

```
typedef struct examEntry examEntry;  // "struct examentry" = "examEntry"
```

13.4. **union** and **std::variant**

A **union** is a special class type that can hold only one of its non-static attributes.
It is at least as big as needed to store the largest attribute, but is usually not larger.
Unions can have non-virtual methods, but cannot be involved in inheritance.

```
#include <iostream>

union Number {   // a Number has:
  long l;        // either a long,
  unsigned u;    // or an unsigned,
  double f;      // or a double
};
```

```
int main() {
  Number n;
  n.l = 7l;
  std::cout << n.l << '\n';
  n.f = 8.9;
  std::cout << n.f << '\n';
  std::cout << sizeof(n) << '\n';
  return 0;
}
```

13.4. **union** and **std::variant**

Since C++17, STL includes **std::variant**, which replaces many uses of unions and union-like classes:

```
#include <iostream>
#include <variant>

int main() {
  std::variant<char, std::string> s;  // s stores a char or a string
  s = 'a';
  std::visit([](auto x){ std::cout << x << '\n';}, s);  // visit since C++17
  s = "string!";
  std::visit([](auto x){ std::cout << x << '\n';}, s);  // (more info here)
  return 0;
}
```