Seminar 11-12
Polymorphism

1. Virtual functions.

- A **virtual function** is a member function (method) declared in a base class and marked by the identifier `virtual`. Derived classes also inherit the virtual methods, and they keep their virtuality in the derived classes and can be overridden by giving them a new definition.

  Note: When overriding a method, it is good practice to mark it by adding the `override` *identifier to the end of the method's declaration.*

- Virtuality is inherited by **all** the derived classes and cannot be stopped.

- Adding the `final` identifier to the end of a method's declaration in the base class will stop its derived classes from being able to override the method.

- Adding a virtual member inside a class will create a **virtual table** and will add a **virtual pointer** as a member field inside that class and its derived classes.

Source: GeeksForGeeks.org

Rules for Virtual Functions

1. *Virtual functions* ***cannot*** *be static and* ***cannot*** *be a friend function of another class.*
2. *Virtual functions should be accessed using a* ***pointer*** *of a base class type to achieve* ***runtime polymorphism****.*
3. *The prototypes of virtual functions in all derived classes must be the same as in the base class.*
4. *Virtual functions are always defined in a base class and can be overridden in a derived class. It is not mandatory for derived classes to override (or re-define the virtual function), in that case the base class version of the function is used.*
5. *A class may have a* ***virtual destructor****, but it cannot have a virtual constructor.*

2. Polymorphism.

- Polymorphism is the ability of an object to take on many forms. The abstract concept of polymorphism can be valid only in cases where different objects have the same abilities (methods) that are implemented in different ways. This way no matter what the object type is there is a common "interface" for the whole polymorphic hierarchy.

- Compile-time polymorphism.
  - A simple example of a **compile-time polymorphism** is overloading of functions and operators:

    Example:
    ```
    void foo(int num) { ... }
    void foo(double real) { ... }
    ```

- In this example during the **compilation** process depending on what parameter is used to call the function foo, a different implementation is used i.e. the function foo() takes on different forms during compilation.

- Runtime polymorphism
  - **Runtime polymorphism** is usually achieved through **pointers** of base classes and **virtual functions**.

    Example:
```
class Base
{
public:
    virtual void fun() const
        { cout << "I'm a base object" << endl; }
};

class Derived : public Base
{
public:
    void fun() const override
        { cout << "I'm a derived object" << endl; }
};
```
  - In this example polymorphism can be achieved by running this code:
```
Derived dObj;
Base* ptr = &dObj;
ptr->fun();     // Even though ptr is of type Base* the result
                // will be "I'm a derived object" printed on the
                // console since fun() is a virtual function
                // inherited and overridden from class Base
```

- Virtual destructors
  - If any of the derived classes need to implement a destructor (for example as a part of the Big Four) then the base class **must** declare its destructor as **virtual** so that when the base pointer is deleted the destructors of the derived classes are also called.

- There are **no virtual constructors** as creating an object can only be done by explicitly calling its constructor and thus specifying an object of which type we would like to create.

3. Polymorphic (heterogeneous) containers.

- **Storing** polymorphic objects can be done by creating a container that holds pointers to the base class of the polymorphic hierarchy.

  Example: `std::vector<Base*> vec;`

- **Adding** objects to a polymorphic container.
  Example: `vec.push_back(new Derived(...));`

- **Accessing** polymorphic methods.
  Example: `vec[0]->fun();`

- The polymorphic container must define an appropriate function, method, or destructor to **clear out all the dynamic memory** used in the container.
  Example:
  ```
  for (size_t i = 0; i < vec.size(); i++) {
      delete vec[i];
  }
  ```

- **Erasing** an object includes deleting its allocated memory before removing it from the container.
  Example:
  ```
  size_t index = 3;
  delete vec[index];
  vec.erase(vec.begin() + index);
  ```

- **Copying** a polymorphic container includes **cloning** all its objects.
  Note: More info about cloning polymorphic objects in the next point.

4. Cloning polymorphic objects.

- Since polymorphism only works when **pointers** are used, copying polymorphic objects **cannot** be done ONLY through the standard copy constructor and so an additional **clone method** must be implemented to properly clone such objects.
  Example:
  ```
  class Base
  {
  public:
      ...
      virtual Base* clone() const = 0;     // Pure virtual method
  }

  class Derived : public Base
  {
  public:
      ...
      Base* clone() const override {
          return new Derived(*this);      // Uses the copy ctor
      }
  };

  // Running the following code will correctly clone the derived obj
  Derived* dObj = new Derived(...);
  ```

```
Base* ptr = dObj;
Base* clonedObj = ptr->clone(); // Copying the derived object

// This clone method should be called for each object when copying
// a polymorphic container.
```

5. Pure virtual functions and abstract classes.

- Declaring a **pure virtual** function can be done by adding “`= 0`” to the end of its prototype. These functions **do not have a definition** in the current class and can be defined in the derived class. Classes containing pure virtual functions are called **abstract classes** and **cannot be instantiated** i.e., we **cannot create objects** of these classes.
  Example: *In the previous example the clone function is a pure virtual function. As such it does not have a definition and we cannot create an object of type Base. The clone function is defined for the Derived class so we can create an object of type Derived.*

- Sometimes a class containing **only pure virtual functions** can be called an **Interface**. That name is usually used in **other programming languages**.

  In **C++** there are only **abstract classes** which have **at least one** pure virtual function but can also contain other methods and member fields.

6. Casting types depending on the situation.

- C-style casting
  Generally frowned upon because it can act like any of the other casts, depending on the situation. Prefer to use a specific cast instead of c-style.
  ```
  float real = 2.7f;
  cout << (int)real;
  ```

- static_cast<type>
  Safe general-purpose casting, always prefer this one before trying another cast.
  Example:
  ```
  float real = 2.7f;
  cout << static_cast<int>(real);
  ```

- reinterpret_cast<ptr_type>
  Telling the compiler to act as if the given variable/value is actually of the specified type, using pointers.
  Example:
  ```
  struct Example {
      int num1;
      int num2;
  };
  ```

```cpp
Example test = { 5, 10 };

// The following line will give a compile error:
//  Cannot convert Example* to int*
int* err = static_cast<int*>(&test);

// But with reinterpret_cast there’s no error
int* magic = reinterpret_cast<int*>(&test);

// Printing it on the console will result in: 5
cout << *magic;
```

- dynamic_cast<type>
  Used with polymorphic hierarchy. It safely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy.
  When casting to a pointer if the conversion is impossible nullptr will be given as a result.
  Example:

```cpp
struct Base {
    virtual void fun() { cout << “Base”; }
};

struct Derived1 : public Base {
    void fun1() { cout << “Fun1”; }
};
struct Derived2 : public Base {
    void fun2() { cout << “Fun2”; }
};

Base* bPtr = new Derived2;
Derived1* d1Ptr = dynamic_cast<Derived1*>(bPtr);
if (d1Ptr) {
    d1Ptr->fun1();  // Safe to call fun1()
} else {
    cout << “Cast ‘Derived2 to Derived1’ was not successful.\n”;
}

Derived2* d2Ptr = dynamic_cast<Derived2*>(bPtr);
if (d2Ptr) {
    d2Ptr->fun2();  // Safe to call fun2()
} else {
    cout << “Cast not successful.\n”;
}
```

```
// The output of this code is:
// Cast ‘Derived2 to Derived1’ was not successful.
// Fun2
```

- const_cast<type>
  Can be used for casting away (removing) or adding a const modifier of a pointer or a reference.
  Warning: Casting away constness of a variable can have undefined behaviour!

7. Additional information.

- typeid operator
  Returns an object of type std::type_info that cannot be copied and has an overloaded operator== to compare to another std::type_info.
  The object contains information about the exact type of the given argument.
  Example:

```
Base* bPtr = new Derived1;
typeid(*bPtr) == typeid(Derived1)   // returns true
typeid(*bPtr) == typeid(Base)       // returns false
```

- Each **class** that has a virtual method also has a **virtual table** that lists all the virtual functions (pointers to functions) and connects them to the **correct** definition of each virtual function that the class uses.

- If a virtual method has been overridden in the class, the pointer in the virtual table will point to that overridden definition.
  If a virtual method has **not** been overridden in a class, the pointer in the virtual table will point to the base class's definition of the function.

- Each class that has a virtual method also has a **virtual pointer** as a member field, i.e. each **object** of that class holds a **virtual pointer** which points to the correct virtual table for the certain object.

- This effectively increases the size of **each object** of that class by *8 bytes**.
  ** 8 bytes on a 64-bit system and 4 bytes on a 32-bit system.*

  Note: Due to member field alignment this additional size can be up to 15 bytes on a 64-bit system and up to 7 bytes on a 32-bit system.