Mechatronic Engineering

Object Oriented Programming and Software Engineering
Laboratory instruction 10
C++ introduction

AGH Kraków, 2021

Materials created for educational purposes.
Dedicated for students attending Software Engineering course.
Author would appreciate any feedback regarding errors of any kind found in the instruction script.
Please report those to the following email address: danielt@agh.edu.pl

Table of contents

1. Virtual functions

Polymorphism is a feature of object-oriented programming that allows different behaviour of the same virtual functions while running the program. Virtual function must be a class member. When using pointers or references in the program, the use of virtual methods may be useful for full control of the program.

Example:

```
#include <iostream>
using namespace std;

class base {
public:
    void hat() {
          cout << "Random empty hat" << endl;
    }
};

class derived
    : public base {
public:
    void hat() { //hiding base class method
          cout << "Hat with magical rabbit inside!" << endl;
    }
};

int main() {
    base empty;
    derived full;

    empty.hat();
    full.hat();

    base * ptr = & full;
    ptr->hat();
}
```

The above program presents the situation without the use of virtual methods. After the execution of the program, we can observe that the last pointer, despite pointing to the derived class object, uses the base class method. This is because the pointer is of the base class type.

Example:

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void hat() {
        cout << "Random empty hat" << endl;
  }
};

class derived
  : public base {
public:
  virtual void hat() { //hiding base class method
        cout << "Hat with magical rabbit inside!" << endl;
  }
};

int main() {
  base empty;
  derived full;

  empty.hat();
  full.hat();

  base * ptr = & full;
  ptr->hat();
}
```

As we can see from the way this program works, using virtual methods allows you to use the functions from the pointed object class, regardless of the type of pointer used.

Only a method can be a virtual function. It is associated with inheritance, which is a class attribute. Another important information is the need to use the word virtual only in the function declaration. In the definition it is not required, the compiler automatically guesses that it is dealing with a virtual function.

Method	Names	Arguments	Range
Virtual	same	same	varies (members of different classes)
Overloaded	same	varies	same (members of one class)

Table 1: Source: J. Grebosz, *Symfonia C++ Standard*

The virtual function calling mechanism should not be confused with function overloading. The following table summarizes the differences between these two features of the function:

2. Abstract classes

An abstract class is a class with purely virtual functions. Such functions have only declarations, and the class itself is used as a template for inheriting classes. The virtual methods should be pure virtual method. To declare a pure virtual method, one must use following syntax:

```
virtual function_type function_name(arg_type arg_name) = 0;
```

Bellow a practical example of an abstract class usage:

```
#include<iostream>
#include<string>

using namespace std;

class vehicle{
protected:
  int wheels;
  float engineCapacity;
  string engineType;
  string vehicleName;

public:
  virtual ~vehicle(){};
  virtual void engineStart()=0;
  virtual void engineStop()=0;
  virtual void displayStats()=0;

};
```

```
class car: public vehicle{
  float trunkCapacity;
public:
  car(int w, float ec, float tc, string et, string vn){
     wheels = w;
     engineCapacity = ec;
     engineType = et;
     vehicleName = vn;
     trunkCapacity = tc;

  }
  ~car(){
       cout<<"Car destroyed!"<<endl;
  }
  void engineStart(){
       cout<< "Checking engine..."<<endl;
       cout << engineCapacity <<"L "<< engineType << "
   roars loudly!"<<endl;
  }
  void engineStop(){
       cout<< engineType << " modestly stops"<<endl;
  }
  void displayStats(){
       cout << "Car name: " <<vehicleName << endl;
       cout << "Engine type: "<<engineType<<", capacity:
            "<<engineCapacity<<"L"<<endl;
       cout << "Trunc capacity: "<<trunkCapacity<<endl;
       cout << "This vehicle has "<<wheels<<"
       wheels"<<endl;
  }
  void car_Name(){
       cout<< vehicleName;
  }
};

class motobike: public vehicle{

public:
  motobike(int w, float ec, string et, string vn){
       wheels = w;
       engineCapacity = ec;
       engineType = et;
       vehicleName = vn;

}
~motobike(){
  cout<<"Bike destroyed!"<<endl;
```

```
}
void engineStart(){
  cout << "Engine roars loudly!"<<endl;
}
void engineStop(){
  cout<< "Engine modestly stops"<<endl;
}
void displayStats(){
  cout << "Bike name: " <<vehicleName << endl;
  cout << "Engine type: "<<engineType<<", capacity:
  "<<engineCapacity<<"L"<<endl;
  cout << "This vehicle has "<<wheels<<" wheels"<<endl;
}
void bike_Name(){
  cout<< vehicleName;
}

};

int main(){
  car camaro(4,6.5,257.68,"L78 V8","1969 Camaro SS 396'");
  motobike rocket(2,2.3,"In-line three, four-stroke",
  "Triumph Rocket III Roadster");

  int a;
  vehicle *choice;

  do{
       cout<<"Pick your ride: "<<endl<<"1.
       ";camaro.car_Name();cout<<"."<<endl;
       cout<< "2. ";rocket.bike_Name();cout<<"."<<endl;
       cin>>a;
  }while (a!=1&&a!=2);

  if (a==1)choice=&camaro;
  else choice=&rocket;

  cout <<"You have chosen your ride!"<<endl;
  choice->displayStats();
  choice->engineStart();
  cout<<"Shame it's just a virual vehicle :("<<endl;
  choice->engineStop();


  return 0;
}
```

3. Virtual destructor

In order to prevent the launch of the wrong destructor so-called Virtual destructor is used. With its use, the destructor will be run from the object's origin class rather than the pointer-type class. To declare a virtual destructor:

```
class random {
public:
        virtual ~random(){};
};
```

4. Programs composed of several files

In the previous instruction, the header files were mentioned. They are very useful when it comes to organizing function and class declarations in one's code. This section presents a method of practical use of header files to create multi-source source code. You might ask why split the source code of the program into many files, isn't this an unnecessary complication? Sometimes, however, it is useful to complicate your work to some extent to be able to simplify your further work.
Below are examples of several header files containing class declarations, the definitions of which are contained in the corresponding cpp files (filenames are presented as comments in first line of each code:

```
//class.hpp
#ifndef class_hpp
#define class_hpp

class figure {
public:
  virtual double area()=0;
  virtual double circumference()=0;
  virtual ~figure(){};

};

#endif


//circle.hpp
#ifndef circle_hpp
```

```
#define circle_hpp

#include "class.hpp"

class circle : public figure {
public:
  circle(double);
  ~circle();
  double area();
  double circumference();
private:
  double radius;
};

#endif

//rectangle.hpp
#ifndef rectangle_hpp
#define rectangle_hpp

#include "class.hpp"

class rectangle : public figure {
public:
  rectangle(double, double);
  ~rectangle();
  double area();
  double circumference();
private:
  double side_a, side_b;
};

#endif

//square.hpp
#ifndef square_hpp
#define square_hpp

#include "class.hpp"

class square : public figure {
public:
  square(double);
  ~square();
  double area();
  double circumference();
private:
```

```
  double side_a;
};

#endif



//circle.cpp
#include <iostream>
#include "circle.hpp"
const float pi = 3.14159;

circle::circle(double r){
  radius=r;
}
circle::~circle(){
  std::cout <<"circle destroyed"<<std::endl;
}
double circle::area(){
  return (pi*radius*radius);
}
double circle::circumference(){
  return (pi*radius*2);
}

//rectangle.cpp
#include <iostream>
#include "rectangle.hpp"

rectangle::rectangle(double a, double b){
  side_a=a; side_b=b;
}
rectangle::~rectangle(){
  std::cout <<"rectangle destroyed"<<std::endl;
}
double rectangle::area(){
  return (side_a*side_b);
}
double rectangle::circumference(){
  return ((2*side_a) +(2*side_b));
}

//square.cpp
#include <iostream>
#include "square.hpp"

square::square(double a){
```

```
  side_a=a;
}
square::~square(){
  std::cout <<"rectangle destroyed"<<std::endl;
}
double square::area(){
  return (side_a*side_a);
}
double square::circumference(){
  return (4*side_a);
}

//main.cpp
#include <iostream>
#include "class.hpp"
#include "circle.hpp"
#include "rectangle.hpp"
#include "square.hpp"
using namespace std;

int main() {
  int i;
  do {
        cout <<"\t\t\t\tChoose operation:" << endl << "1.
        Calculate area of a circle. \n2. Calculate
        circumference of a circle.
        \n3. Calculate area of a rectangle. \n";
        cout <<"4. Calculate circumference of a rectangle.
        \n5. Calculate area of a square. \n6. Calculate
        circumference of a square.\n 0. Exit\n";
      cin >> i;
      if(i >6 || i < 0) continue;
      switch(i){
        case 0: cout << "ending program"; break;
        case 1: {
          double r;
          cout << "What is the radius?: "; cin >> r;
          circle c1(r);
          cout << endl << "Area of a circle (radius = " <<
          r <<") equals: " << c1.area() << endl; break;}
        case 2: {
          double r;
          cout << "What is the radius?: "; cin >> r;
          circle c1(r);
          cout << endl << "Circumference of a circle
          (radius = " << r <<") equals: " <<
          c1.circumference() << endl; break;}
```

```
        case 3: {
            double a,b;
            cout << "How long is the side a?: ";
            cin >> a;
            cout <<endl<< "How long is the side b?: ";
            cin >> b;
            rectangle r1(a,b);
            cout << endl << "Area of a rectangle (a = " << a
            <<", b = " << b <<") equals: " << r1.area() <<
            endl; break;}
        case 4: {
            double a,b;
            cout << "How long is the side a?: ";
            cin >> a; cout <<endl<< "How long is the side
            b?: ";
            cin >> b;
            rectangle r1(a,b);
            cout << endl << "Circumference of a rectangle (a
            = " << a <<", b = " << b <<") equals: " <<
            r1.circumference() << endl; break;}
        case 5: {
            double a;
            cout << "How long is the side a?: ";
            cin >> a;
            square s1(a);
            cout << endl << "Area of a square (a = " << a
            <<") equals: " << s1.area() << endl; break;}
        case 6: {
            double a;
            cout << "How long is the side a?: ";
            cin >> a;
            square s1(a);
            cout << endl << "Circumference of a square (a =
            " << a <<") equals: " << s1.circumference() <<
            endl; break;}
      }
  }while(i!=0);
  return 0;
}
```

To compile the above code into a fully functional program, it will be important to compile with the -c flag:

```
g++ -c file_name.cpp
```

Information about what the -c flag does, copied from the compiler manual (man g ++):

> *[...] (g++) -c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file. By default, the object file name for a source file is made by replacing the suffix .c, .i, .s, etc., with .o. Unrecognized input files, not requiring compilation or assembly, are ignored. [...]*

To combine individual .o files, use the -o option and name all files compiled with the -c option:

```
g++ -o output_file_name input1_name.o input2_name.o input3_name.o
```

The compilation process can of course be accelerated with the use of a bash script:

```
#!/bin/bash
clear
g++ -c main.cpp
g++ -c circle.cpp
g++ -c rectangle.cpp
g++ -c square.cpp
g++ -o binary_file main.o circle.o rectangle.o square.o
echo 'compilation process finished!'
```

One might ask why bother and why is this a simplification? By dividing the code into smaller batches, you can compile those parts that have changed and combine them with the intact previously compiled fragments. In addition, it simplifies group work on the code, allowing you to divide individual fragments between project participants without worrying about accidental changes in the code that are already working.

Task

Based on the information provided in this manual, please improve the simple RPG character creation program.
Program requirements:

1. Create an abstract class interface that will store all the attributes that will be inherited by the hero and monster class.
2. Create an abstract class interface that will store the necessary methods that will be inherited by your profession classes.
3. Divide your code into separate files (one for templates, one for hero managing, one for monsters managing and one for main function)
4. Upload all previously created programs (including examples from the instructions) placed in appropriate folders (exercises, lab01, lab02, lab03 etc.) into your git repository.