Student Exercises for Intermediate C++ course.

[1. Objects 2](#_Toc376438507)

[1.1. Singleton 3](#_Toc376438508)

[1.2. Operator overloading 4](#_Toc376438509)

[2. Inheritance 4](#_Toc376438510)

[2.1. Constructors and Destructors call order 5](#_Toc376438511)

[3. Polymorphism 5](#_Toc376438512)

[4. Casting 6](#_Toc376438513)

[5. Exceptions 7](#_Toc376438514)

[6. Roulette [Home exercise] 7](#_Toc376438515)

**Introduction**

This document contains exercise description for the course ”Intermediate C++”. It is required that users are equipped with laptop with C++ compiler.

# Objects

In this exercise you will get familiar with class usage. Remember that in C++ structures are almost the same as classes with the following exception: The default access specifier for class is private and for structure it is public.

1. Write the following program:
   1. Create the following class:

class Person

{

public:

Person(const string& firstName, const string& surname);

Person(const Person&);

string fullName();

int age();

void operator++();

private:

string \_firstName;

string \_surname;

int \_age;

};

* 1. Person(const string& firstName, const string& surname) – constructor that sets the private parameters (use initialization list, set \_age to 0).
  2. Person(const Person&); – copy constructor that copies the private members (but sets the \_age to 0).
  3. string fullName(); – method that returns full name: \_firstName + \_surname.
  4. int age(); – method that returns the private field \_age.
  5. void operator++(); – overloaded operator of pre-incrementation. It shall increase the \_age of Personality by 1. In order to define post-incrementation operator, add one parameter of type int.
  6. In main(), create an instance of Person, then create another one that is a copy of the first one.
  7. Use the public methods you have defined. Try to display private members directly.
  8. Move constructors to the private section. What happened?

In the next exercise you will use the vector template from STL. All elements required to conduct the exercise will be shown when needed.

1. Modify the program from point 1:
   1. Make sure all constructors are public again.
   2. Modify existing constructors so that they announce themselves to the cout.
   3. Add public constructor Person(); that sets all the members to their default value and announces itself to the cout.
   4. Add public destructor ~ Person (); that announces itself to the cout.
   5. In the program #include <vector>
   6. Write the code presented below in the main(). Tray to explain the output:

int main()

{

vector <Person> v;

Person John("John", "Smith");

cout << "-------------------" << endl;

v.push\_back(John);

cout << "-------------------" << endl;

Person Tim(John);

v.push\_back(Tim);

cout << "-------------------" << endl;

v.clear();

cout << "-------------------" << endl;

return 0;

}

# Singleton

In [software engineering](http://en.wikipedia.org/wiki/Software_engineering), the singleton is a [design pattern](http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29) that restricts the [instantiation](http://en.wikipedia.org/wiki/Instantiation_%28computer_science%29) of a class to one [object](http://en.wikipedia.org/wiki/Object-oriented_programming). This is useful when exactly one object is needed to coordinate actions across the system (like logging).

1. Create a class Singleton.
2. Add private: constructor without parameters, copy constructor, operator= and destructor (to ensure no one can call those methods).
3. Add public method: static Singleton& instance(); that takes no arguments, declares static instance of Singleton and returns the reference to that instance.

class Singleton

{

public:

static Singleton& instance()

{

static Singleton singleton;

return singleton;

}

private:

Singleton() {}

Singleton(const Singleton&) {}

void operator= (const Singleton&) {}

~Singleton() {}

};

1. Add public method void print(const string&); that prints the received text to the cout.
2. Use the class in main(): Singleton::instance().print(“Test!”).

Thanks to the use of static method, we are able to use it even if no instance of Singleton class exists. Then the method upon being called creates the static instance of Singleton (it can use the private constructor).

# Operator overloading

In this exercise you will overload several operators for a given class.

1. Create a structure Vector:

struct Vector

{

double dx, dy;

};

1. Add a public method: Vector add(const Vector&); that returns a Vector that is a sum of the given argument and the Vector the method was called upon.
2. Use the method in main().
3. Change the name of the method to operator+. Use the new way of calling this method.
4. Overload the operator++ that will increase the length of each component (dx, dy).
5. Choose and overload an operator for class Vector that will return the given vector turned by 180 degree.

# Inheritance

Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own.

1. Create class A with private, protected and public member.
2. Create class B that derives from A publicly. Try to access the inherited fields using instance of class B.
3. Change the inheritance to protected and then to private. See what happens.

# Constructors and Destructors call order

1. Write a program with three classes: A, B and C, where C derives publicly from B and B derives publicly from A.
2. In each class define constructor without parameters that announces itself to the cout.
3. In main() create an instance of class A, B and C. Watch the order in which the constructors are called in each case.
4. In each class define destructor in the same manner as you have defined constructors. Use operator new and delete on a class C. Watch the order in which the destructors are called.

# Polymorphism

In this exercise you will see how the polymorphism works.

1. Write the following program:
   1. Create a class Animal:

struct Animal

{

virtual void *show*() = 0;

static Animal\* create(string type);

};

* 1. virtual void *show*() = 0; - pure virtual method that will have concrete implementation in classes that derive from Animal.
  2. static Animal\* create(string type); - static method that will be responsible for creating concrete animal types
  3. Create the following classes: Bird, Mammal and Reptile. All those classes shall derive publicly from class Animal. Below you can see the implementation of create method from Animal class.

Animal\*

Animal::create(string type)

{

if(type == "bird") return new Bird;

if(type == "mammal") return new Mammal;

if(type == "reptile") return new Reptile;

return NULL;

}

* 1. Use the function in main(). Then call method show() on the created Animal.
  2. Remember to free the memory once not needed using operator delete.

The structure you have created in this exercise is called Factory Pattern and it is used to create instances of concrete classes using interface located in the base class.

# Casting

In this exercise you will create a small application that will use the new casting mechanism provided by C++ that allows casting between instances of different classes.

The application will be used to create a signal hierarchy and a simple sending / receiving mechanism that will allow the easy creation of new signals.

1. Define the following, base Signal class:

class Signal

{

public:

int sigNo;

Signal(int \_sigNo) : sigNo(\_sigNo) {}

};

1. Define the following concrete signal:

class MessageOK : public Signal

{

public:

int param;

MessageOK(int \_param) : Signal(MESSAGE\_OK),

param(\_param) {}

};

1. Add a unique identifier for a signal: const int MESSAGE\_OK = 0;

This way you have created a simple way of adding new signals with multiple parameters. All you have to do is to create new class that derives publicly from Signal and to create a constructor for it.

1. Sending functions should look for example like this (this is just a stub, it has no body – just to show you how would the mechanism look like):

void send(Signal\* signal);

1. Thanks to inheritance, you can send any signal created for example like this:

Signal\* message = new MessageOK(2);

send(message);

1. Define a function that will receive the message void receive(Signal\* signal); The function shall display the parameters carried by the signal.
2. The first thing the function must do is to check the sigNo field and then using static\_cast cast the received signal to the correct message type.
3. The function must print the parameters of the signal to the cout.
4. Add new signal: OperationFailed with its own identification number, and three parameters of different types..
5. Modify receive function to handle new signal.

# Exceptions

In this exercise you will create and use your own exception.

1. Create class DivisionByZero with a public method string description(); that display the information about the exception.
2. Write a function double div(const double&, const double&); that returns the result of division of the given arguments. The function must throw an exception when zero is given as a second argument.
3. Try the function in main(), and handle the exception when needed.
4. Try to find the documentation about the exceptions thrown by the C++ standard library (std::bad\_alloc as an example).

# Roulette [Home exercise]

In this exercise you will write a simplified version of Roulette to be played using console.

1. Simplified Roulette rules:
   1. Each round one number in range 0-36 (37 possibilities total) is randomly chosen (#include <time> and use the following code to generate random number in range 0-36:

srand (time(NULL)); // initialize random seed

number = rand() % 37; // get number 0-36

* 1. Player can make the following bets (win ratio X – player gains X $ for each 1$ he bets):
     1. Straight Up – any single number (36)
     2. Dozen bet – any of the three blocks of numbers: 1-12, 13-24, 25-36 (3)
     3. Even / Odd – even or odd numbers (excluding 0) (2)

1. Write a program where:
   1. At the game start you may choose any number of players to participate
   2. Players should have their names
   3. Players make any number of bets (up to their bank balance) before the number is generated
   4. Current game status can be displayed
   5. Game history is stored
   6. Player statistics are stored (for example: number of bets, number of wins)
   7. Player may leave / join the game between rounds