C++ Test

Author: Adrian Michałek
Email: devmichalek@gmail.com
Difficulty : All levels

Task 0. **(1P)**

This is an example task of how the further tasks will look like. Below you can see a part of code to examine. All the tasks contain description and questions to answer. Answers to these questions are always one page further so there is no reason to scroll down and search for them. You can write your answers on the paper and quickly check if you are right by going to the next page. Each task contains available number of points that you can get. For instance if you correctly answer the question for this task you get **(1P) - 1 point**. There is no reason to test your knowledge if you feel exhausted with the test. Feel free to finish this test at any moment and check how many points you've got. Let's start with the first task.

What is the output of the executed code? **(1P)**

```
#include <iostream>

using namespace std;

int main()
{
      cout << "Hello World";
      return 0;
}
```

Correct answer **(1P)**:

Hello World

Task 1. **(2P)**

With the code given below answer the questions.

```
void print(int&& number);

//void f1()
//{
//      print(0);
//}

static void f2()
{
      print(0);
}

int main()
{
      return 0;
}
```

1. Will compiler report any errors for the given code? **(1P)**
2. What will happen if we uncomment the function f1? **(1P)**

1. It’s compiler dependent. Compiler may or may not report error. Obviously we don’t have function print body. Some compilers may see that function f2 is not used anywhere. It can make this assumption because f2 is static, so f2 can be used anywhere only in the main.cpp. Since it’s not used anywhere there is no problem with the code and there are no errors. **(1P)**

2. This case is different. Compiler cannot assume that f1 is not used, that’s why linker will take this function into consideration, f1 can be used in other cpp file that’s why it is crucial to have function print body. Linker will definitely report undefined reference error for function print that is inside function f1.

Task 2. **(2P)**

With the code given below answer the questions.

```
#include <stdio.h>

int do_a()
{
	printf("A");
	return 0;
}

char do_b()
{
	printf("B");
	return 0;
}

class A
{
	int a = do_a();
public:
	A(int a = 0)
	{
		printf("C");
	};

private:
	char b = do_b();
};

void do_d(int x, int y, A z)
{
	printf("D");
}

int main()
{
	do_d(do_a(), do_b(), A());
	return 0;
}
```

1. What is the output for the executed code? **(1P)**
2. What is the order of the evaluation of arguments passed to the function in C++? **(1P)**

1.
According to the documentation there is no quarantee that the evaluation of the function arguments is from the left to the right or from the right to the left. We cannot tell if function do_a() will be called first.

What we know surely is the order of the class fields initialization. After constructor call A() definitely letters **ABC** will be printed. According to the documentation:

„First, and only for the constructor of the most derived class ass described below, virtual base classes shall be initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where „left-to-right" is the order of appearance of the base class names in the derived class base-specifier-list."

„Then, direct base classes shall be initialized in declaration order as they appear in the base-specifier-list (regardless of the order of the mem-initializers)."

„Then, nonstatic data members shall be initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers).„

„Finally, the body of the constructor is executed. [Note: the declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization."

Definitely we can also tell that letter **D** will be printed at the end.

Possible outputs are:
ABCBA**D**
ABCAB**D**
B**ABC**A**D**
A**ABC**B**D**
AB**ABCD**
BA**ABCD**

(1P)

2. Like it was said previously according to the documentation there is no quarantee that the evaluation of the function arguments is from the left to the right or from the right to the left. Order is undefined. **(1P)**

Task 3. **(3P)**

The code below has two classes A and B. Non of them are perfect. The idea was to not repeat the code for constructors in other words the code that initializes fields. The idea is to initialize fields in the constructor with a parameter (second constructor in both classes) the same way as the constructor with no parameters. Class A has a good approach, constructor with a parameter calls init function the same as constructor with no parameter. But if we change first constructor we break the rule that the constructor with a parameter initializes fields in the same way. In this case we will have to add changes to init function. Class B has even better approach, no matter what changes are applied to first constructor of class B the second constructor will behave the same. But there are other issues with class B.

```
class A
{
	void init() {}

public:
	A()
	{
		init();
		// If we add some changes here they need to be applied in the init function - not ideal
	}

	A(int x)
	{
		init();
	}
};

class B
{
	void init() {}

public:
	B()
	{
		init();
		// If we add some changes here they don't have to be applied in the init function – but there are still some issues
	}

	B(int x)
	{
		B(); // Hint: There are issues with this call
	}
};
```

1. Give at least one issue (there are at least two) that is with a class B. Remember that second constructor have to initialize fields in the same manner as first constructor does. **(1P)**

2. Write class C with the presented idea with no flaws. **(2P)**

1. Class B has a wrong approach because while calling second constructor we call another constructor which creates again new object so we have two calls instead of one. Second issue is that first constructor may not be called first. There may be a situation when we want to use initializer list in a second constructor but the initialization have to happen after first constructor call. It is impossible with the approach showed in class B. **(1P)**

2. C++11 standard provides solution for that:

```
class C
{
	void init() {}
public:
	C()
	{
		init();
	}

	C(int x) : C()
	{

	}
};
```

Calling class C second constructor is safe because we use functionality of the first constructor as we want without second call. Calling class C second constructor is even better because we can now use initializer list and make sure that first constructor is called first. **(2P)**

Task 4. **(1P)**

The following code gives an error: „No appropriate default constructor available" or (depends on compiler) „No matching function for call to Dog::Dog()"

```
class Dog
{
	int age;
public:
	Dog(int age) { this->age = age; }
	/* implementation goes here */

	/* implementation ends here*/
};

int main()
{
	Dog dog;
	return 0;
}
```

Implement a default constructor using C++11 feature for defining default constructors. **(1P)**

The solution is:

```
class Dog
{
        int age;
public:
        Dog(int age) { this->age = age; }
        /* implementation goes here */
        Dog() = default;
        /* implementation ends here*/
};

int main()
{
        Dog dog;
        return 0;
}
```

(1P)

Task 5. **(1P)**

With the code given below add a solution.

```
#include <iostream>

#include <vector>

template<typename func>
void filter(func f, std::vector<int> arr)
{
        for (auto i : arr)
        {
                if (f(i))
                        std::cout << i << " ";
        }
}

int main()
{
        std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
        filter(/*implementation*/, v);
        return 0;
}
```

Write lambda function (by replacing the `/*implementation*/` comment) that takes one integer as an argument and returns true if integer is greater than 3. **(1P)**

The solution is:

```
#include <iostream>

#include <vector>

template<typename func>
void filter(func f, std::vector<int> arr)
{
	for (auto i : arr)
	{
		if (f(i))
			std::cout << i << " ";
	}
}

int main()
{
	std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
	filter([](int x) { return (x > 3); }, v);
	return 0;
}
```

(1P)

Task 6. **(1P)**

With the code given below write solution.

```
#include <stdio.h>

int main()
{
        int a = 0, b = 1, c = 2, d = 3;
        auto f1 = [/*pass variable 'a' by reference*/]() {
                a = 1;
                printf("%d\n", a);
        };

        auto f2 = [/*pass all variables by const copy*/]() {
                printf("%d\n", a + b + c + d);
        };

         auto f3 = [/*pass all variables by reference except the variable
'b'*/]() {
                a = c = d;
                printf("%d\n", b);
        };

        return 0;
}
```

Replace comments that are present in the code with a proper captures according to what is written in the comment. **(1P)**

The solution is:

```
#include <stdio.h>

int main()
{
	int a = 0, b = 1, c = 2, d = 3;
	auto f1 = [&a]() {
		a = 1;
		printf("%d\n", a);
	};

	auto f2 = [=]() {
		printf("%d\n", a + b + c + d);
	};

	auto f3 = [&, b]() {
		a = c = d;
		printf("%d\n", b);
	};

	return 0;
}
```

(1P)

Task 7. **(1P)**

```
#include <iostream>

class MyVector
{
	unsigned size;
	double* arr;

public:
	MyVector(int newSize)
	{
		size = newSize;
		arr = new double[size];
		for (unsigned i = 0; i < size; ++i)
			arr[i] = i;
	}

	MyVector(MyVector &&rhs)
	{
		size = rhs.size;
		arr = rhs.arr;
	}

	~MyVector()
	{
		delete[] arr;
	}
};

int main()
{
	MyVector v1(10);
	MyVector v2(std::move(v1));
	return 0;
}
```

There is the bug in the following code. Find a bug and give a solution. Do not add new functions, variables and changes in function main. **(1P)**

There is a problem with move constructor. Without `rhs.arr = nullptr;` the destructor of the object passed with move semantics (in this case `v1`) may throw an exception.

```
#include <iostream>

class MyVector
{
	unsigned size;
	double* arr;

public:
	MyVector(int newSize)
	{
		size = newSize;
		arr = new double[size];
		for (unsigned i = 0; i < size; ++i)
			arr[i] = i;
	}

	MyVector(MyVector &&rhs)
	{
		size = rhs.size;
		arr = rhs.arr;
		rhs.arr = nullptr;
	}

	~MyVector()
	{
		delete[] arr;
	}
};

int main()
{
	MyVector v1(10);
	MyVector v2(std::move(v1));
	return 0;
}
```

(1P)

Task 8. **(2P)**

With the C++17, `std::shared_ptr` can be used to manage a dynamically allocated array. According to the documentation the following code will compile with no errors.

```
#include <memory>

struct Dog
{
};

int main()
{
	std::shared_ptr<Dog> p1 = std::make_shared<Dog>();
	std::shared_ptr<Dog> p2 = std::shared_ptr<Dog>(new Dog());
	std::shared_ptr<Dog> p3(new Dog[3]);
	std::shared_ptr<Dog> p4 = nullptr;
	return 0;
}
```

Is the given code safe? If not, answer why and give a solution. **(2P)**

The given code is not safe, due to the following line:

```
std::shared_ptr<Dog> p3(new Dog[3]);
```

There are memory leaks wen this shared pointer is out of scope and has reference counter equal to 0. Only first element is deleted correctly, the other elements are lost.

(1P)

To prevent this we have to replace previous line with:

```
std::shared_ptr<Dog> p3(new Dog[3], [](Dog* pDog) {delete[] pDog; });
```

Prior to C++17 std::shared_ptr could not be used to manage dynamically allocated arrays. By default, std::shared_ptr will call **delete** on the managed object when no more references remain to it. However, when you allocate using **new**[] you need to call **delete**[] to free the resource.

(1P)

Task 9. **(1P)**

```
#include <iostream>

int* i = new int (0);

int& f()
{
      return *i;
}

int main()
{
        int&(*h)() = f;
        int& i = (*h)();
        (*h)() = 1;
        ++i;
        ++(*h)() += i;
        std::cout << (*h)() << std::endl;
        delete &i;
        return 0;
}
```

The above code was compiled and executed. What was printed to the output? **(1P)**

Value:

6

was printed to the output. **(1P)**

Task 10. (2P)

```cpp
#include <iostream>
#include <stdexcept>
#include <mutex>
#include <thread>


int unsigned_max(int a, int b)
{
        if (a < 0 || b < 0)
                throw std::invalid_argument("received negative value");
        return a > b ? a : b;
}

std::mutex mutualExclusion;
void print(int& i, int j)
{
        mutualExclusion.lock();
        printf("%d\n", unsigned_max(i, j));
        mutualExclusion.unlock();
}

void __()
{
        for (int i = 10; i >= -5; --i)
                print(i, i);
}

int main()
{
        std::thread t(__);
        for (int i = 25; i >= 10; --i)
                print(i, i);
        t.join();
        return 0;
}
```

1. Function `print` is mutually exclusive but still have some issues. What's wrong with `print` function? (1P)

2. Repair the `print` function. (1P)

1. The function is mutually exclusive but not safe in case when exception is thrown. When exception is thrown one of the thread may not unlock the mutex which may cause the second thread waiting forever. **(1P)**

2. To solve this problem we have to make sure that mutex is unclocked in case of exception:

```
std::mutex mutualExclusion;
void print(int& i, int j)
{
	std::lock_guard<std::mutex> locker(mutualExclusion);
	printf("%d\n", unsigned_max(i, j));
}
```

(1P)

Task 11. **(2P)**

A derived class should be able to do everything the base class can do. A polymorphism means a different classes of objects react to the same API differently.

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

struct Dog
{
        void bark(int age)
        {
                std::cout << "I am " << age << " years old." << std::endl;
        }

        virtual void bark(std::string msg = "just a")
        {
                std::cout << "I am " << msg << " dog" << std::endl;
        }
};

struct YellowDog : public Dog
{
        virtual void bark(std::string msg = "yellow")
        {
                std::cout << "I am " << msg << " dog" << std::endl;
        }
};

int main()
{
        YellowDog* yd = new YellowDog;
        yd->bark();
        yd->bark(2);
        delete yd;
        return 0;
}
```

1. What is the output of the executed code? **(1P)**
2. Is there are any risks with the above code? If yes, give the solution. **(1P)**

1. This code won't compile. When compiler sees function `void bark(int age)` it will first search in the `YellowDog` class. If the compiler didn't find the function in the derived class it will keep searching in the base class. Since `YellowDog` have a bark function with different argument still with the matching function name the compiler will stop searching right there.
Function `void bark(std::string msg = "yellow")` shadows function `void bark(int age)`.

(1P)

2. The problem is described above. The solution is:

```
#include <iostream>

#include <string>

struct Dog
{
        void bark(int age)
        {
                std::cout << "I am " << age << " years old." << std::endl;
        }

        virtual void bark(std::string msg = "just a")
        {
                std::cout << "I am " << msg << " dog" << std::endl;
        }
};

struct YellowDog : public Dog
{
        using Dog::bark;
        virtual void bark(std::string msg = "yellow")
        {
                std::cout << "I am " << msg << " dog" << std::endl;
        }
};

int main()
{
        YellowDog* yd = new YellowDog;
        yd->bark();
        yd->bark(2);
        delete yd;
        return 0;
}
```

(1P)

Task 12. **(1P)**

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

struct Dog
{
	void bark()
	{
		std::cout << "I am just a dog." << std::endl;
	}
};

struct YellowDog : public Dog
{
	void bark()
	{
		std::cout << "I am yellow dog." << std::endl;
	}
};

Dog* getNewDog()
{
	return new YellowDog();
}

int main()
{
	Dog* dog = getNewDog();
	dog->bark();
	delete dog;
	return 0;
}
```

Execution of the above code gives:

I’m just a dog.

on the output, even tough the object is of `YellowDog` class.

How to repair this bug? **(1P)**

You have to add virtual keyword before function bark declaration inside the base class:

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

struct Dog
{
	virtual void bark()
	{
		std::cout << "I am just a dog." << std::endl;
	}
};

struct YellowDog : public Dog
{
	void bark()
	{
		std::cout << "I am yellow dog." << std::endl;
	}
};

Dog* getNewDog()
{
	return new YellowDog();
}

int main()
{
	Dog* dog = getNewDog();
	dog->bark();
	delete dog;
	return 0;
}
```

The virtual keyword means that function may be overriden by derived class.

(1P)

Task 13. (2P)

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

struct Dog
{
	virtual void bark(std::string msg = "just a")
	{
		std::cout << "I am " << msg << " dog." << std::endl;
	}
};

struct YellowDog : public Dog
{
	void bark(std::string msg = "Yellow") override
	{
		std::cout << "I am " << msg << " dog." << std::endl;
	}
};

Dog* getNewDog()
{
	return new YellowDog();
}

int main()
{
	Dog* dog = getNewDog();
	dog->bark();
	delete dog;
	return 0;
}
```

The above code was compiled and executed. What wa printed to the console output?
Explain why. (2P)

The output is:

I’m just a dog.

Virtual function is bound at runtime. However, the default value for function parameter is bound at compile time. Even tough dog is of class YellowDog when dog barks it will invoke the virtual function of YellowDog as we expected but it will pickup a default value for message from Dog::bark function. Never override a default value of overridden function.

(2P)

Task 14. **(1P)**

```
int main()
{

}
```

Is it possible to have main function without return statement? **(1P)**

It is possible. If no return statement is present, the main function (and thus, the program itself) return 0 by default. **(1P)**

Task 15. **(1P)**

The following code was compiled and executed:

```cpp
#include <iostream>

void func(int* x)
{
	*x = 7;
}

int main()
{
	const int a = 3;
	int* ptr;
	ptr = const_cast<int*>(&a);
	std::cout << "A=" << a << std::endl;
	*ptr = 5;
	std::cout << "A=" << a << std::endl;
	*((int*)(&a)) = 6;
	std::cout << "A=" << a << std::endl;
	func((int*)(&a));
	std::cout << "A=" << a << std::endl;
	return 0;
}
```

What was printed to the output? **(1P)**

It depends. Based on compiler the answer may differ. Most modern compiler do not allow to change value of the const int even when using **const_cast**. The most possible output is:
A=3
A=3
A=3
A=3

Most compilers will optimize const variables that's why in this case **const_cast** is undefined. The variable 'a' may not really exist in the compiled code.

(1P)

Task 16. **(1P)**

```
int main()
{
        int i;
        const int* p1 = &i;                    // 1.
        int* const p2 = &i;                    // 2.
        const int* const p3 = &i;              // 3.
        int const* p4 = &i;                    // 4.
        return 0;
}
```

What is `const`, data or pointer? Give an answer for each line. **(1P)**

1. Data is `const`
2. Pointer is `const`
3. Data and pointer is `const`
4. Data is `const`

The general rules are:
If `const` is on the left side of * - data is `const`
If `const` is on the right side of * - pointer is `const`

(1P)

Task 17. **(1P)**

The following code was compiled and executed:

```cpp
#include <iostream>

int main()
{
	int i = 2;
	const int& ref = i;
	const int* ptr = &i;
	const_cast<int&>(ref) = 3;
	*const_cast<int*>(ptr) = 4;
	std::cout << i << ' ' << ref << ' ' << *ptr;
	return 0;
}
```

What was printed to the output? **(1P)**

The text printed to the output is:
4 4 4

The **const_cast** was used to cast away constness of the objects ref and ptr. Both ptr and ref point to **non-const** variable which can be modified.

(1P)

Task 18. **(1P)**

The following code compiled with errors:

```cpp
#include <iostream>

class A
{
        inline static int field = 0;
public:
        static int getField() const
        {
                return field;
        }
};

int main()
{
        std::cout << A::getField() << std::endl;
        return 0;
}
```

Explain why. **(1P)**

Answer:
When you apply the const qualifier to a nonstatic member function, it affects the this pointer. For a const-qualified member function of class **c**, the **this** pointer is of type **C const***, whereas for a member function that is not-const-qualified, the **this** pointer is of type **C***.

A static member function does not have a this pointer (such a function is not called on a particular instance of a class), so const qualification of a static member function doesn't make any sense.

(1P)

Task 19. (3P)

Based on the code below answer the questions:

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

class Dog
{
	string m_name;

public:
	Dog()
	{
		m_name = "dummy";
		print();
	}

	const string& getName()
	{
		return m_name;
	}

	void print() const
	{
		cout << getName() << " const" << endl;
	}

	void print()
	{
		cout << getName() << " not const" << endl;
	}
};

int main()
{
	Dog d1;
	const Dog d2;
	d1.print();
	d2.print();
	return 0;
}
```

1. Repair the code so that it compiles with no errors (there are at least two solutions, give one). You can add or remove only one word! (1P)
2. After the code is repaired, compiled and executed what will be printed to the console? (1P)
3. Only one print function is called from the constructor. Can we change which one is called by changing the constructor body? Give a solution for that. (1P)

1. The code does not compile because print function with `const` qualifier have a function getName() which is not marked as `const`. In this case `const` qualifier tells the compiler that none of the fields are of the class will be changed. There are at least two solutions for that.

First solution is to add a `const` qualifier to getName() function:

```
const string& getName() const
{
	return m_name;
}
```

Second solution is to get rid of getName() function call inside print function by replacing it with the field:

```
void print() const
{
	cout << m_name << " const" << endl;
}
```

(1P)

2. The output is:

dummy not const

dummy not const

dummy not const

dummy const

(1P)

3. It is possible to have a call to print function with `const` qualifier by using the `const_cast`:

```
Dog()
{
	m_name = "dummy";
	const_cast<const Dog*>(this)→print();
}
```

(1P)

Task 20. **(1P)**

```
class C
{
        const int* const fun(const int* const& p) const;
};
```

Describe the function given above: the data type it returns, the arguments it takes etc. **(1P)**

The function is a member function of class C.
The function is const which means that fields of class C used in this function cannot be changed.
The function is private, function cannot be called outside the class C.
The function fun returns const pointer (the value of pointer cannot be changed) that points to const integer (the data the pointer points to cannot be changed).
The function takes one argument a reference const pointer that points to const integer. On function call argument won’t be copied, the pointer cannot be changed and data it points to cannot be changed.

(1P)

Task 21. **(2P)**

The following class won't compile:

```
#include <vector>

class BigArray
{
        std::vector<int> v;
        int accessCounter;

public:
        int getItem(int index) const
        {
                accessCounter++;
                return v[index];
        }
};
```

1. Explain why the code won't compile. **(1P)**

2. Repair the code so that the code compiles. Do not change getItem() function declaration and its functionality. Funtion getItem() must return vector element and increment accessCounter variable. **(1P)**

1. The code does not compile because of the keyword **const** that is at the end of getItem() declaration. In this case **const** denotes that no fields of the class **BigArray** can be changed.

(1P)

2. First solution is to add **`mutable`** keyword:

```
class BigArray
{
        std::vector<int> v;
        mutable int accessCounter;

public:
        int getItem(int index) const
        {
                accessCounter++;
                return v[index];
        }
};
```

Second solution is to cast away constness:

```
class BigArray

{
        std::vector<int> v;
        int accessCounter;

public:
        int getItem(int index) const
        {
                const_cast<BigArray*>(this)->accessCounter++;
                return v[index];
        }
};
```

(1P)

Task 22. **(1P)**

The following code was compiled and executed:

```
#include <stdio.h>

void fun(char** p)
{
        char* t;
        t = (p += sizeof(int))[-1];
        printf("%s\n", t);
}

int main()
{
        const char* argv[] = { "ab", "cd", "ef", "gh", "ij", "kl" };
        fun((char**)argv);
        return 0;
}
```

What was printed to the console? **(1P)**

The console output is:
gh

The expression (p += **sizeof**(int))[-1] can be written as (p += 3)

(1P)

Task 23. **(2P)**

The following code won't compile because of uninitialized static field m_type:

```
#include <iostream>

class Cat
{
	enum class Type
	{
		Unknown,
		Black,
		White
	};

	unsigned m_age;
	static Type m_type;

public:
	Cat(unsigned age) : m_age(age)
	{
		std::cout << age << " years old ";

		switch (m_type)
		{
			case Type::Black: std::cout << "black"; break;
			case Type::White: std::cout << "white"; break;
			default: std::cout << "transparent"; break;
		}

		std::cout << " cat was born\n";
	}
};

int main()
{
	Cat cat(5);
	return 0;
}
```

1. Add code that initializes static field m_type so that the type of cats is black. **(1P)**

2. Add code that initializes static field m_type using C++17 feature, type of cats must be white. **(1P)**

1. The solution is to add the following code outside the class:

```
Cat::Type Cat::m_type = Cat::Type::Black;
```

(1P)

2. The solution is to change m_type field declaration into:

```
static inline Type m_type = Type::White;
```

(1P)

Task 24. **(1P)**

The following code was compiled and executed:

```
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;
std::mutex mu;

struct A
{
	A()
	{
		this->operator()(0);
	}

	void operator()(int)
	{
		std::unique_lock<std::mutex> locker(mu, std::defer_lock);
		locker.lock();
		++counter;
		locker.unlock();
	}
};

int main()
{
	A a;
	std::thread t1(a, 0);
	std::thread t2(&A::operator(), a, 0);
	std::thread t3(A(), 0);
	std::thread t4(std::ref(a), 0);
	t1.join();
	t2.join();
	t3.join();
	t4.join();
	std::cout << counter << std::endl;
	return 0;
}
```

What was printed to the output? **(1P)**

The following number was printed to the output:
6

Explanation:

```
A a; // Counter was incremented (via constructor then operator)
std::thread t1(a, 0); // Counter was incremented (via operator)
std::thread t2(&A::operator(), a, 0); // The same as for t1
std::thread t3(A(), 0); // Counter was incremented twice, via constructor then operator and
via operator
std::thread t4(std::ref(a), 0); // Counter was incremented via operator
```

(1P)

Task 25. **(2P)**

Based on given code solve the task:

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

struct Human
{
	Human(std::string name) { std::cout << "My name is " << name << std::endl; }
};

int main(int argc, char** argv)
{
	Human h1(std::string("John"));
	Human h2(h1); // Copy constructor used
	return 0;
}
```

Do not allow to use copy constructor, give at least two different solutions. Do not change other bahaviors! **(2P)**

The first solution is to delete copy constructor:

```
Human(const Human&) = delete;
```

The second solution is to make copy constructor private:

```
private:
	Human(const Human&);
```

(2P)

Task 26. **(1P)**

There are memory leaks in the following code:

```
class Human
{
	int m_data;
	~Human()
	{
	}

public:
	Human() : m_data(123)
	{
	}
};

int main()
{
	Human* human = new Human;
	// delete human; // Won't compile
	return 0;
}
```

Get rid of memory leaks. Keep destructor private and allocation on heap inside function `main()`. You can modify `class Human` and function `main()`. **(1P)**

The solution is to add new function and call it externally from function `main()`:

```
class Human
{
	int m_data;
	~Human()
	{
	}

public:
	Human() : m_data(123)
	{
	}

	void clear()
	{
		delete this;
	}
};

int main()
{
	Human* human = new Human;
	human->clear();
	return 0;
}
```

The presented code has of course many flaws. For example in the case of exception memory would not be freed.

(1P)

Task 27. **(1P)**

With the following code answer the question:

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


struct Cat
{
	Cat()
	{
		std::cout << "A";
	}

	Cat(std::string input)
	{
		std::cout << input;
	}
};

struct Dog
{
	Dog(std::string input)
	{
		std::cout << input;
	}
};

int main(int argc, char** argv)
{
	Cat c1("B");
	Dog d1;
	Cat c2("A");
	Cat c3;
	Dog d2("C");
	return 0;
}
```

What's wrong with the above code? **(1P)**

Presented code won't compile as there is no default constructor. There will be an error because there is specified constructor that has one parameter. In case of user implemented constructors (at least one) the compiler will not generate additional constructors.

(1P)

Task 28. **(2P)**

The following code was compiled:

```
#include <memory>

class Dog
{
	int* m_data;

public:
	~Dog()
	{
		delete[] m_data;
	}
	Dog()
	{
		m_data = new int[10];
	}
};

class YellowDog : public Dog
{
	char* m_word;

public:
	~YellowDog()
	{
		delete[] m_word;
	}
	YellowDog()
	{
		m_word = new char[4];
	}
};

struct DogFactory
{
	static Dog* createYellowDog()
	{
		return new YellowDog();
	}
};
```

1. Is it possible to have a non-virtual destructor in the base class and be sure that data allocated by the derived class object was correctly deallocated when derived class object was destroyed? **(1P)**

2. Modify function `createYellowDog()` so that it is static and safe to use. Hint: use specific smart pointer. **(1P)**

1. Yes it is possible with the `std::shared_ptr`. When `std::shared_ptr` is created it stores a deleter object inside itself. This object is called when the `std::shared_ptr` is about to free the pointed resource. Since you know how to destroy the resource at the point of construction you can use `std::shared_ptr` with incomplete types. Whoever created the `std::shared_ptr` stored a correct deleter there.

(1P)

2. The solution as described previously is to use `std::shared_ptr`:

```
struct DogFactory

{
	static std::shared_ptr<Dog> createYellowDog()
	{
		return std::shared_ptr<YellowDog>(new YellowDog());
	}
};
```

(1P)

Task 29. **(1P)**

What are the main differences between malloc, calloc and realloc? **(1P)** malloc() - allocated memory on heap and returns pointer to the allocated memory

calloc() - does the same thing as malloc() does except it fills allocated memory with zeros (sometimes the allocated memory can be already filled with zeroes so the second step is not performed)

realloc() - changes the size of memory block on heap, frees if the new size is smaller

(1P)

Task 30. **(1P)**

Based on the following code solve the task:

```
class Collar
{
};

class Dog
{
	Collar* c;

public:
	Dog()
	{
		c = new Collar;
	}

	~Dog()
	{
		delete c;
	}

	Dog& operator=(const Dog& rhs)
	{
		delete c;
		c = new Collar(*rhs.c);
		return *this;
	}
};

int main()
{
	Dog a;
	a = a;
	return 0;
}
```

Correct copy assignment operator so that it is flawless (it has to perform deep copy). **(1P)**

Solution:

```cpp
class Collar

{
};

class Dog
{
	Collar* c;

public:
	Dog()
	{
		c = new Collar;
	}

	~Dog()
	{
		delete c;
	}

	Dog& operator=(const Dog& rhs)
	{
		if (this == &rhs)
			return *this;

		Collar* buffer = c;
		c = new Collar(*rhs.c);
		delete buffer;
		return *this;
	}
};

int main()
{
	Dog a;
	a = a;
	return 0;
}
```

(1P)

Task 31. **(1P)**

The following code was compiled and executed:

```c
#include <stdio.h>

#define R 10
#define C 20

int main()
{
	int* p[R][C];
	printf("%d\n", sizeof(p));
	printf("%d\n", sizeof(*p));
	printf("%d\n", sizeof(**p));
	return 0;
}
```

What was printed to the standard output? **(1P)**

The standard output should contain consecutively:

a) on x64 architecture (pointer size is 8)
1600
160
8

b) on x86 architecture (pointer size is 4)
800
80
4

(1P)

Task 32. **(2P)**

The following code was compiled:

```
#include <iostream>

int main()
{
	int i = 1;
	char c = *reinterpret_cast<char*>(&i);
	if (c == 1)
		std::cout << "First byte is equal to 1\n";
	else
		std::cout << "Last byte is equal to 1\n";
	return 0;
}
```

1. What the above code may test? **(1P)**

2. The code was executed. What was printed to the standard output? **(1P)**

1. The given code tests endianness – order or sequence of bytes of a word of digital data in computer memory.

(1P)

2. It depends on what kind of endianness the processor use.

In case of little endian (the most significant bit is last):
First byte is equal to 1

In case of big endian (the most significat bit is first):
Last byte is equal to 1

(1P)

Task 33. **(1P)**

Based on the following code solve the task:

```
#include <iostream>

int main()
{
	std::cout << "String\n";
	std::cout << "String" << std::endl;
	return 0;
}
```

What is the difference between these two `std::cout`? **(1P)**

The only difference is that `std::endl` flushes the output buffer, and `\n` doesn't. If you don't want the buffer flushed frequently, use `\n`. If you want to get all the output and the program is unstable use `std::endl`.

```
std::cout << "String" << std::endl;
```

Is equivalent to:

```
std::cout << "String\n" << std::flush;
```

(1P)

Task 34. **(2P)**

Assuming that the following code was compiled and executed on the machine with little endian bytes order and the size of `int` is 4 bytes:

```
#include <stdio.h>

struct Block
{
	int a[3] = { 1, 2, 3 };
	int b[3] = { 4, 5, 6 };
	unsigned char ch[4] = { 0, 1, 0, 0 };
	int c[3] = { 7, 8, 9 };
};

int main()
{
	Block block;
	int* ptr = (int*)(&block.b + 1);
	printf("%d %d %d", *(block.b + 1), *(ptr - 1), *ptr);
	return 0;
}
```

What was printed to the output? Explain why. **(2P)**

The following values were printed to the output:
5 6 256

Explanation:
1. `*(block.b + 1)` field b is dereferenced and incremented by 1, it is equivalent to `block.b[1]`
2. `ptr` - points to field `ch`, - `(int*)(&block.b + 1)` - firstly the address of field `b` is taken, the incrementation moves address by 12 bytes then the address is converted statically to `int*`
3. Char array `unsigned char ch[4] = { 0, 1, 0, 0 }` converted to `int` is 256 – (the 8th bit is set).
4. `*(ptr - 1)` - ptr is decreased by 4 bytes and dereferenced, the value hold by this memory address is 6

(2P)

Task 35. **(1P)**

Based on the code give below answer the question:

```
#include <iostream>

#define watch(x) std::cout << #x << " is equal to " << x << "\n";

int function()
{
	return 1;
}

int main()
{
	int i = 0xFF;
	int* j = &i;
	watch(i);
	watch(*j);
	watch(function());
	return 0;
}
```

What `#` prefix in the keyword `#x` does in the above define?**(1P)**

The `#x` keyword will be replaced with expression passed to watch define, so that the output for the executed code will look like this:

i is equal to 255
*j is equal to 255
function() is equal to 1

(1P)

Task 36. **(1P)**

The following code was compiled and executed without any errors:

```
class Foo

{
public:
	Foo(int data) : m_data(data)
	{
	}

	int get()
	{
		return m_data;
	}

private:
	int m_data;
};

void callme(Foo foo)
{
	int i = foo.get();
}

int main()
{
	callme(42);
	return 0;
}
```

Disallow compiler to perform implicit conversion to resolve the parameters to a function. The changes may be applied only to `Foo class`. **(1P)**

The solution is to add `explicit` keyword:

```
class Foo

{
public:
	explicit Foo(int data) : m_data(data)
	{
	}

	int get()
	{
		return m_data;
	}

private:
	int m_data;
};
```

Now in the `main` function we are forced to explicitly use constructor:

```
int main()

{
	callme(Foo(42));
	return 0;
}
```

(1P)

Task 37. (2P)

Based on the below code, solve the task:

```
#include <iostream>

namespace A
{
	struct X
	{
	};

	void g(X x)
	{
		std::cout << "calling A::g()\n";
	}
}

struct B
{
	void g(A::X x)
	{
		std::cout << "calling B::g()\n";
	}
};

class C : public B
{
public:
	void j()
	{
		A::X x;
		g(x);
	}
};

int main()
{
	A::X x;
	g(x);
	C c;
	c.j();
	return 0;
}
```

Does the above code compiles without any errors? What was printed to the output? Explain why.

(2P)

Code compiles without any errors.

The following strings were printed:
calling A::g()
calling B::g()

The first line of the output shows the Koenig Lookup example. If the compiler have not found the g() function it starts to search inside variable scope (A scope).

The second line of the output shows the priority class towards namespace. Class and its parents are always taken first into consideration by compiler.

(2P)

Task 38. **(1P)**

What's the difference between functions `e()` and `f()` ? **(1P)**

```
int e()
{
	return 0;
}

constexpr int f()
{
	return 0;
}

int main()
{
	int a = e();
	constexpr int b = f();
	return 0;
}
```

Both functions are not static, we can assume that they may be used outside of the main.cpp file. `constexpr` allows to compute expressions at compile time. Function `f()` will be computed at compilation so that the variable `b` will be initialized with 0 without function call.

(1P)

Task 39. **(1P)**

The following code was compiled and executed:

```c
#include <stdio.h>

int main(void)
{
	const volatile int local = 10;
	int* ptr = (int*)&local;

	printf("%d\n", local);

	*ptr = 100;

	printf("%d\n", local);
	return 0;
}
```

What was printed to the output? What does the keyword volatile amplify? **(1P)**

Output is:
10
100

`volatile` is a hint for a compiler to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation.

`volatile` tells the compiler: Hey compiler, I am `volatile` and, you know, I can be changed by some XYZ that you are not even aware of. That XYZ could be anything. Maybe some alien outside this planet called program. Maybe some lighting, some form of interrupt, volcanoes, etc can mutate me. Maybe. You never know who is going to change me! So O you ignorant, stop playing an all-knowing god, and do not dare touch the code where I am present.

(1P)

Task 40. **(2P)**

Based on the following code, solve tasks:

```
class Animal
{
	int m_age;

public:
	/* implementation starts here */
	Animal(int age)
	{
		m_age = age;
	}


	/* implementation ends here*/
};

int main()
{
	Animal a(3);
	Animal b(3.53f); // Should result in compiler error
	int number = 3;
	a = number; // Should result in compiler error
	return 0;
}
```

1. Disallow implicit conversion from float to integer. Give at least two solutions. **(1P)**

2. Disallow copying with assignment operator. Give at least two solutions. **(1P)**

1. Solutions:

Add new constructor and make it `private`:

```
private:
	Animal(float age);
```

Add deleted constructor:

```
Animal(float age) = delete;
```

(1P)

2. Solutions:

Add operator and make it `private`:

```
private:
	Animal operator=(const Animal& rhs);
```

Add deleted operator:

```
Animal operator=(const Animal& rhs) = delete;
```

(1P)

Task 41. **(2P)**

The given code won't compile:

```
#include <iostream>

int main()
{
	int lp = 0;
	auto Report = [lp](const char* msg)
	{
		std::cout << msg << ++lp << std::endl;
	};

	Report("Report: ");
	std::cout << "Main: " << lp << std::endl;
	Report("Report: ");
	std::cout << "Main: " << lp << std::endl;
	return 0;
}
```

1. Correct lambda function declaration so that is compiles without errors. Do not add any captures to lambda function declaration! Do not change lambda function body! **(1P)**

2. After code was repaired, compiled and executed what was printed to the output? **(1P)**

1. The only correct solution is to add `mutable` keyword:

```
auto Report = [lp](const char* msg) mutable
{
	std::cout << msg << ++lp << std::endl;
};
```

(1P)

2. Output is:

Report: 1
Main: 0
Report: 2
Main: 0

Explanation:

Passing values to lambda function not by reference means passing by const copy! Keyword mutable allows to change the value. Modification of variable is not visible outside of lambda function, but inside, the value is remembered. It is because lambda function is not function but object function.

(1P)

Task 42. **(2P)**

Task description

1. Question. **(1P)**

1. The solution is to add the following code outside the class:

(1P)

Task 43. **(2P)**

Task description

1. Question. **(1P)**

1. The solution is to add the following code outside the class:

(1P)

1. The solution is to add the following code outside the class:

(1P)

1. The solution is to add the following code outside the class:

(1P)

1. The solution is to add the following code outside the class:

(1P)

1. The solution is to add the following code outside the class:

(1P)