Solution cpp endterm

Konzepte der C++ Programmierung (Technische Universität München)

Computational Imaging and Inverse Problems
Department of Informatics
Technical University of Munich

Esolution
Place student sticker here

Note:

- During the attendance check a sticker containing a unique code will be put on this exam.
- This code contains a unique number that associates this exam with your registration number.
- This number is printed both next to the code and to the signature field in the attendance check list.

Concepts of C++ Programming

Exam:	IN2377 / Endterm	**Date:**	Monday 8th August, 2022
Examiner:	PD Dr. Tobias Lasser	**Time:**	10:45 – 12:15

	P 1	P 2	P 3	P 4	P 5	P 6	P 7	P 8	P 9	P 10
I										

Working instructions

- This exam consists of **12 pages** with a total of **10 problems**.
  Please make sure now that you received a complete copy of the exam.
- The total amount of achievable credits in this exam is 69 credits.
- Detaching pages from the exam is prohibited.
- Allowed resources:
  - one **analog dictionary** English ↔ native language
- Subproblems marked by * can be solved without results of previous subproblems.
- **Answers are only accepted if the solution approach is documented.** Give a reason for each answer unless explicitly stated otherwise in the respective subproblem.
- Do not write with red or green colors nor use pencils.
- Physically turn off all electronic devices, put them into your bag and close the bag.

Left room from _____ to _____ / Early submission at _____

Problem 1 Warmup Questions (single-choice) (6 credits)

In the following questions, please mark the correct answer. Only *one* answer is correct (single-choice).

Mark correct answers with a cross ☒
To undo a cross, completely fill out the answer option ■
To re-mark an option, use a human-readable marking ✕■

a)* What does *static typing* in C++ mean?

- ☐ expressions always have the same type
- ☐ expression types cannot be converted
- ☐ expression types are not known at compile-time
- ☒ expression type compatibility checks are done at compile-time

b)* What is the C++ preprocessor?

- ☒ source code adjustment through text replacement
- ☐ a superset definition of C++ to allow more language features
- ☐ optimization step to improve code performance
- ☐ validation of C++ source code for correctness

c)* What does a build system like CMake do?

- ☐ it checks the code for correctness and does static analysis
- ☐ it creates reproducible, platform-independent builds for you
- ☒ simplifies compilation of project sources and its dependencies
- ☐ automatically creates continuous integration tests for your code

d)* What does a typical C++ compiler do?

- ☒ translate and optimize valid C++ source code to platform-specific machine code
- ☐ combine C++ source code files to platform-independent programs
- ☐ it creates an executable file that references all its dependencies
- ☐ it analyzes C++ sources for static code analysis and bug detection

e)* What does a typical C++ linker do?

- ☐ it bundles together multiple programs to a self-contained installation program
- ☒ it resolves symbol addresses from binary translation units
- ☐ when running a program, it maps its memory regions and dependency libraries
- ☐ it discovers program dependencies on the system

f)* Your code is violating the *one definition rule* (ODR). Which tools will tell you about it?

- ☒ some ODR violations will not be caught be either compiler or linker
- ☐ the compiler alone will catch all ODR violations
- ☐ the compiler and linker together will catch all ODR violations

Problem 2 C++ basics (10 credits)

a)* State three *programming paradigms* that are supported by C++.

0 1 2 3

- procedural programming
- object-oriented programming
- generic programming

b)* Write down a one-line code example each of: a *declaration*, a *definition*, and an *initialization*. Mark *clearly* which is which.

0 1 2 3

- declaration: `int foo();`
- definition: `int i;`
- initialization: `int i{42};`

c)* Write down a one-line code example of *safe initialization*. Explain briefly why you should use it.

0 1

- safe initialization: `int i{42};`
- safe initialization gives a compiler error if an implicit conversion potentially results in loss of information (narrowing)

d)* Explain briefly the difference between a `struct` and a `class`.

0 1

They are the same, except that `struct` has `public` as default access modifier, while `class` has `private` as default.

e)* Explain briefly the difference between `const` and `constexpr`.

0 1 2

- `constexpr` denotes an expression that can be evaluated at compile-time
- `const` annotates an object that cannot be modified

Problem 3 Value Categories (6 credits)

0 1 2 3

a)* Aside from *lvalue* and *rvalue*, there are *three* more value categories of expressions in modern C++. List those three additional value categories, and provide an example for each in form of an expression. *Clearly* mark which is which.

- *prvalue:* pure temporaries. Example: `42`
- *glvalue:* generalized lvalue, i.e. anything with referable names. Example: `int a;`
- *xvalue:* anything with referable name that can be moved. Example: `std::move(a)` (where `int a;`)

0 1 2 3

b)* Given the following code, write down the console output when `main()` is executed.

```
#include <iostream>

struct Argument {};

struct SomeClass {
    void function(Argument&)              { std::cout << "a "; }
    void function(Argument&&)             { std::cout << "b "; }
    void function(Argument&)        const { std::cout << "c "; }
    void function(Argument&&)       const { std::cout << "d "; }
    void function(const Argument&)        { std::cout << "e "; }
    void function(const Argument&&)       { std::cout << "f "; }
    void function(const Argument&)  const { std::cout << "g "; }
    void function(const Argument&&) const { std::cout << "h "; }
};

int main() {
    Argument arg{};
    const Argument& carg = arg;

    SomeClass obj{};
    const SomeClass& cobj = obj;

    obj.function(carg);
    cobj.function(std::move(arg));
    obj.function(arg);
    cobj.function(carg);
    cobj.function(Argument{});
    obj.function(std::move(carg));
}
```

e d a g d f

Problem 4 Rvalue References (5 credits)

a)* Consider the following code snippet. Is the expression `a` in line 2 an *rvalue* or an *lvalue*? Explain your decision briefly.

0 1

```
1 bool feed(Animal&& a) {
2     Feeder::enqueue(a);
3 }
4
5 void feed_all() {
6     Animal cat{};
7     feed(std::move(cat));
8 }
```

`a` is an lvalue, as it has a name.

b) Explain briefly why it is difficult to introduce a single wrapper function that retains value categories of its arguments ("forwarding problem").

0 1

The difficulty is having one wrapper function, while still supporting both rvalue and lvalue arguments. As seen in a), rvalue arguments turn into lvalues in the function body of the wrapper.

c) You are given the following declarations:

0 1 2 3

```
1 struct arg;
2 void api_function(arg& arg);
3 void api_function(arg&& arg);
```

Now, write down the code for a single function called `wrapper` (taking an argument `arg`) that does the following:

- it wraps the call to `api_function()`
- before calling the `api_function()`, it shall invoke the method `arg.update()`
- when it invokes `api_function()`, the argument value category of the parameter `arg` must be preserved

```
1 template <typename T>
2 void wrapper(T&& arg) {
3     arg.update();
4     api_function(std::forward<T>(arg));
5 }
```

Problem 5 Resource Management (8 credits)

0 1 2

a)* Explain briefly what the concept of *RAII* (Resource Acquisition Is Initialization) means. Outline briefly how you typically implement RAII in C++.

- RAII means the lifetime of a resource (e.g. memory, socket, mutexes) is bound to the lifetime of an object. This guarantees that the resource is available during the object lifetime, and released when the object lifetime ends.
- Implementation typically works by encapsulating the resource in a class, using the constructors and destructor to guarantee initialization and release of the resource.

0 1 2 3 4

b)* Write down the declarations of the following *special member functions* of `class Foo`: copy constructor, move constructor, move assignment operator, copy assignment operator. *Clearly* mark which is which.

- copy constructor: `Foo(const Foo& other);`
- move constructor: `Foo(Foo&& other);`
- copy assignment: `Foo& operator= (const Foo& other);`
- move assignment: `Foo& operator= (Foo&& other);`

0 1

c)* Define the *"Rule of 0"* concisely.

Classes not involved in resource management should not define any of the special member functions: destructor, copy/move constructor, copy/move assignment.

0 1

d)* Explain briefly why `shared_ptr` is less performant than a `unique_ptr`.

- `shared_ptr` uses reference counting to track the number of active `shared_ptr` objects to the managed memory.
- This reference counting has overhead in both space and time.

Problem 6 Object-oriented Programming (6 credits)

You are executing the following code:

```cpp
#include <iostream>
#include <memory>

struct Vehicle {
    Vehicle()  { std::cout << "create Vehicle\n" ; }
    ~Vehicle() { std::cout << "destroy Vehicle\n"; }
    virtual void notify() const = 0;
};

struct Bike : Vehicle {
    Bike()  { std::cout << "create Bike\n"; }
    ~Bike() { std::cout << "destroy Bike\n"; }
    void notify() const override { std::cout << "ring ring\n"; }
};

struct Car : Vehicle {
    Car()  { std::cout << "create Car\n"; }
    ~Car() { std::cout << "destroy Car\n"; }
    void notify() const override { std::cout << "honk honk\n"; }
};

int main() {
    std::unique_ptr<Vehicle> b = std::make_unique<Bike>();
    std::unique_ptr<Vehicle> c = std::make_unique<Car>();
    b->notify();
    c->notify();
}
```

a)* Given this code, what console output does executing `main()` produce?

```
create Vehicle
create Bike
create Vehicle
create Car
ring ring
honk honk
destroy Vehicle
destroy Vehicle
```

0 1 2 3 4

b) The code above contains a mistake. What is the mistake? How can it be fixed?

No bike or car is destructed!

The destructor of `~{}Vehicle` has to be marked `virtual`.

0 1 2

Problem 7 Lambdas (7 credits)

0 1

a)* Explain in *one* (short!) sentence what a lambda expression in C++ is.

It is a simplified notation for anonymous function objects.

0 1

b)* The following code snippet contains a bug causing *undefined behavior*. Explain briefly why the bug occurs, and propose a fix.

```
1 auto number_generator(int base_number) {
2     return [&] (int input) { return input + base_number; };
3 }
4 int main() {
5     auto gen = number_generator(1234);
6     std::cout << gen(4321) << std::endl;
7 }
```

`base_number` is captured by reference, but is an automatic variable that no longer lives when the lambda is actually invoked.

Fix: capture it by value (e.g. using `[=]`).

0 1 2 3 4

c)* Assuming the types `type_a`, `type_b`, `type_c`, `type_r` are declared somewhere else, consider the following code snippet from inside a function body:

```
1 type_a a;
2 type_b b;
3 type_c c;
4 type_r r;
5
6 auto fun = [a, &b](type_c c) -> type_r {
7     return a + b + c;
8 };
9
10 r = fun(c);
```

Write code to define `fun` with the same behavior, but *without* using a lambda expression. In other words: replace lines 6-8 of the code so that line 10 (i.e. `r = fun(c);`) produces the same result.

```
1 struct lambdastruct {
2     type_r operator() (type_c c) {
3         return a + b + c;
4     }
5     type_a a;
6     type_b& b;
7 };
8
9 lambdastruct fun{a, b};
10 r = fun(c);
```

- `operator()`with return typeand body
- `type_a` member
- `type_b` reference member

Problem 8 Generic Code (9 credits)

a)* Describe briefly what a *variadic template* is.

0 1 2

Variadic templates are templates with at least one parameter pack.

Parameter packs are template parameters that accept zero or more arguments.

b)* Write a function `reduce_sum` using *variadic templates* that computes the sum of all of its arguments. The invocation has to work like this:

0 1 2 3

```
int main() {
    return reduce_sum<int>(1, 2, 3, 4);  // returns 10
}
```

```
template <typename R, typename... Args>
R reduce_sum(const Args&... args) {
    return (args + ...);
}
```

c)* Name *two* ways of restricting template parameters in C++20.

0 1 2

- C++20 concepts using e.g. `requires` clauses
- SFINAE with type traits and `std::enable_if`

d)* Specify a C++20 concept called `Equal` that is suitable to validate if a type supports equality checks using `operator==()` and `operator!=()`.

0 1 2

```
template <typename T>
concept Equal = requires (T a, T b) {
    { a == b } -> std::convertible_to<bool>;
    { a != b } -> std::convertible_to<bool>;
};
```

Problem 9 Compile-time Programming (5 credits)

0 1 2 3 4 5

Implement the calculation of the Fibonacci numbers at *compile-time*, using whatever compile-time technique you deem suitable. Make sure to provide an *example invocation* of your implementation.

The formula for the Fibonacci numbers is:

$$\begin{aligned} F(n) &= F(n-1) + F(n-2) \\ F(0) &= 0 \\ F(1) &= 1 \end{aligned}$$

TMP implementation:

```
template <int N>
struct fibonacci {
    static constexpr int value = fibonacci<N-1>::value + fibonacci<N-2>::value;
};

template <>
struct fibonacci<1> {
    static constexpr int value = 1;
};

template <>
struct fibonacci<0> {
    static constexpr int value = 0;
};

std::cout << fibonacci<10>::value << "\n";
```

example function implementation:

```
consteval int fibonacci(int n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}

std::cout << fibonacci(10) << "\n";
```

Problem 10 Wind-down Questions (single-choice) (7 credits)

In the following questions, please mark the correct answer. Only *one* answer is correct (single-choice).

Mark correct answers with a cross ☒
To undo a cross, completely fill out the answer option ■
To re-mark an option, use a human-readable marking ✕■

a)* What is a “friend” in C++?

- ☐ it is a public non member method accessing a class
- ☒ it is granted access to otherwise inaccessible members of a class
- ☐ it specifies that a class is directly related to another class

b)* Why do we need *polymorphic cloning*?

- ☐ to pass a variadic argument list without runtime overhead
- ☐ to duplicate an inheritance hierarchy into a new class
- ☒ to allow deep copying of an object in a inheritance hierarchy

c)* What is the guaranteed *time complexity* (in terms of input size n) of the typical operations of a `std::map`?

- ☒ $\mathcal{O}(\log n)$
- ☐ $\mathcal{O}(n)$
- ☐ $\mathcal{O}(1)$

d)* How would you declare a function `func`, in which you want to read-access a large object of type `T`?

- ☐ `void func(const T obj);`
- ☒ `void func(const T& obj);`
- ☐ `void func(const T&& obj);`

e)* Which modifier guarantees *compile-time* execution of a function?

- ☒ `consteval`
- ☐ `constexpr`
- ☐ `const`

f)* What is the purpose of expression templates?

- ☒ lazy evaluation of arithmetic expressions at compile-time
- ☐ specifying an arbitrary amount of template parameters
- ☐ restricting template parameters

g)* What can the *curiously recurring template pattern* (CRTP) be used for?

- ☐ provide dynamic polymorphism
- ☒ provide static polymorphism
- ☐ provide recursive polymorphism

Additional space for solutions–clearly mark the (sub)problem your answers are related to and strike out invalid solutions.