Exercise Session — Computer Science — 09

Structs, Classes, Operator overloading, Iterators

Overview

Today's Plan

Follow-up
Classes and Operator Overloading
Exercise "Tribool"
Iterators
Exercise "Find Max"
Recursion

`n.ethz.ch/~iopopa`

↗ Link to Webpage

↗ Send an e-Mail

1. Follow-up

Follow-up from last session

- I hope you managed to finish the Power Set exercise on your own.
- For those who liked recursion, check out the "Towers of Hanoi" exercise from last week’s slides.
- Don’t be scared of the "Towers of Hanoi", since most probably nothing as hard will come up in the exam.

2. Feedback regarding **code** expert

General things regarding code expert

- Nothing from my side this week :)

Any questions regarding **code** expert on your part?

3. Classes and Operator Overloading

Differentiating between functions

Differentiating between functions

It is possible for two functions to have the same name, as long as the compiler has another way to differentiate between them. The only possible criteria for distinguishing functions are:

Differentiating between functions

It is possible for two functions to have the same name, as long as the compiler has another way to differentiate between them. The only possible criteria for distinguishing functions are:

- Names of the functions
- Numbers of function arguments
- Types of function arguments

Putting the *Fun* in *Function* I

Will this produce a compiler error?

```
int fun1(const int a){
    // ...
}

int fun1(const int a, const int b){
    // ...
}
```

Putting the *Fun* in *Function* I

Will this produce a compiler error?

```
int fun1(const int a){
    // ...
}

int fun1(const int a, const int b){
    // ...
}
```

Answer: No, because

Putting the *Fun* in *Function* I

Will this produce a compiler error?

```
int fun1(const int a){
    // ...
}

int fun1(const int a, const int b){
    // ...
}
```

Answer: No, because the two functions have a different numbers of arguments (1 vs 2)

Putting the *Fun* in *Function* II

Will this produce a compiler error?

```
int fun2(const int a){
    // ...
}

int fun2(const float a){
    // ...
}
```

Putting the *Fun* in *Function* II

Will this produce a compiler error?

```
int fun2(const int a){
    // ...
}

int fun2(const float a){
    // ...
}
```

Answer: No, because

Putting the *Fun* in *Function* II

Will this produce a compiler error?

```
int fun2(const int a){
    // ...
}

int fun2(const float a){
    // ...
}
```

Answer: No, because the two functions have a different parameter types (`int` vs `float`)

Putting the *Fun* in *Function* III

Will this produce a compiler error?

```
int fun3(const int a){
    // ...
}

int fun3(const int b){
    // ...
}
```

Putting the *Fun* in *Function* III

Will this produce a compiler error?

```
int fun3(const int a){
    // ...
}

int fun3(const int b){
    // ...
}
```

Answer: Yes, because

Putting the *Fun* in *Function* III

Will this produce a compiler error?

```
int fun3(const int a){
    // ...
}

int fun3(const int b){
    // ...
}
```

Answer: Yes, because the two functions don’t have different numbers or types of arguments

Putting the *Fun* in *Function* III

Will this produce a compiler error?

```
int fun3(const int a){
    // ...
}

int fun3(const int b){
    // ...
}
```

Answer: Yes, because the two functions don’t have different numbers or types of arguments

Notice: The names of the function parameters are irrelevant to the compiler!

Putting the *Fun* in *Function* IV

Will this produce a compiler error?

```
int fun4(const int a){
    // ...
}

double fun4(const int a){
    // ...
}
```

Putting the *Fun* in *Function* IV

Will this produce a compiler error?

```
int fun4(const int a){
    // ...
}

double fun4(const int a){
    // ...
}
```

Answer: Yes, because

Putting the *Fun* in *Function* IV

Will this produce a compiler error?

```
int fun4(const int a){
    // ...
}

double fun4(const int a){
    // ...
}
```

Answer: Yes, because the two functions don’t have different numbers or types of arguments

Putting the *Fun* in *Function* IV

Will this produce a compiler error?

```
int fun4(const int a){
    // ...
}

double fun4(const int a){
    // ...
}
```

Answer: Yes, because the two functions don’t have different numbers or types of arguments

Notice: The return types of the functions are irrelevant to the compiler!

Putting the *Fun* in *Function* V

Will this produce a compiler error?

```
int fun5(const int a){
    // ...
}

int fun6(const int a){
    // ...
}
```

Putting the *Fun* in *Function* V

Will this produce a compiler error?

```
int fun5(const int a){
    // ...
}

int fun6(const int a){
    // ...
}
```

Answer: No, because

Putting the *Fun* in *Function* V

Will this produce a compiler error?

```
int fun5(const int a){
    // ...
}

int fun6(const int a){
    // ...
}
```

Answer: No, because the two functions carry different names

Just my Type

```cpp
void out(const int i){
  std::cout << i << " (int)\n";
}
void out(const double i){
  std::cout << i << " (double)\n";
}

int main(){
  out(3.5);
  out(2);
  out(2.0);
  out(0);
  out(0.0);
  return 0;
}
```

What’s the output going to be?

Just my Type

```
void out(const int i){
  std::cout << i << " (int)\n";
}
void out(const double i){
  std::cout << i << " (double)\n";
}

int main(){
  out(3.5);
  out(2);
  out(2.0);
  out(0);
  out(0.0);
  return 0;
}
```

What's the output going to be?

- `3.5 (double)`

Just my Type

```
void out(const int i){
  std::cout << i << " (int)\n";
}
void out(const double i){
  std::cout << i << " (double)\n";
}

int main(){
  out(3.5);
  out(2);
  out(2.0);
  out(0);
  out(0.0);
  return 0;
}
```

What’s the output going to be?

- `3.5 (double)`
- `2 (int)`

Just my Type

```
void out(const int i){
  std::cout << i << " (int)\n";
}
void out(const double i){
  std::cout << i << " (double)\n";
}

int main(){
  out(3.5);
  out(2);
  out(2.0);
  out(0);
  out(0.0);
  return 0;
}
```

What's the output going to be?

- `3.5 (double)`
- `2 (int)`
- `2 (double)`

Just my Type

```
void out(const int i){
  std::cout << i << " (int)\n";
}
void out(const double i){
  std::cout << i << " (double)\n";
}

int main(){
  out(3.5);
  out(2);
  out(2.0);
  out(0);
  out(0.0);
  return 0;
}
```

What’s the output going to be?

- 3.5 (double)
- 2 (int)
- 2 (double)
- 0 (int)

Just my Type

```cpp
void out(const int i){
  std::cout << i << " (int)\n";
}
void out(const double i){
  std::cout << i << " (double)\n";
}

int main(){
  out(3.5);
  out(2);
  out(2.0);
  out(0);
  out(0.0);
  return 0;
}
```

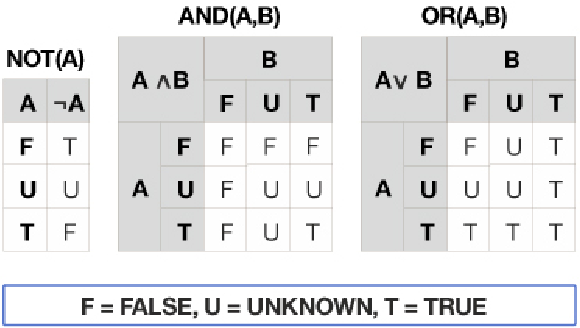
What's the output going to be?

- 3.5 (double)
- 2 (int)
- 2 (double)
- 0 (int)
- 0 (double)

Questions?

4. Exercise "Tribool"

Tribool as a Logic Object



NOT(A)

A	¬A
F	T
U	U
T	F

AND(A,B)

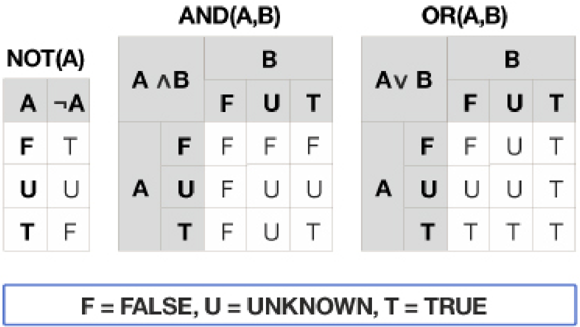
A ∧B		B		
		F	U	T
A	F	F	F	F
	U	F	U	U
	T	F	U	T

OR(A,B)

A∨ B		B		
		F	U	T
A	F	F	U	T
	U	U	U	T
	T	T	T	T

F = FALSE, U = UNKNOWN, T = TRUE

Tribool as a Logic Object



NOT(A)

A	¬A
F	T
U	U
T	F

AND(A,B)

A ∧B		B		
		F	U	T
A	F	F	F	F
	U	F	U	U
	T	F	U	T

OR(A,B)

A∨ B		B		
		F	U	T
A	F	F	U	T
	U	U	U	T
	T	T	T	T

F = FALSE, U = UNKNOWN, T = TRUE

- How could we implement this in C++?
- What operations and values do we need?

Exercise "Tribool"

```
class Tribool {
private:
    // 0 means false, 1 means unknown, 2 means true.
    unsigned int value;  // INV: value in {0, 1, 2}.
public:
    // ...
};
```

Exercise "Tribool"

```cpp
class Tribool {
private:
    // ...
public:
    // Constructor 1 (passing a numerical value)
    // PRE: value in {0, 1, 2}.
    // POST: tribool false if value was 0, unknown if 1, and true if 2.
    Tribool(unsigned int value_int);
    // TODO: add the definition in tribool.cpp

    // Constructor 2 (passing a string value)
    // PRE: value in {"true", "false", "unknown"}.
    // POST: tribool false, true or unknown according to the input.
    // TODO: add declaration here and the definition in tribool.cpp
    // ...
};
```

Exercise "Tribool"

```
class Tribool {
private:
    // ...
public:
    // ...
    // Member function string()
    // POST: Return the value as string
    // TODO: add declaration here and the definition in tribool.cpp

    // Operator && overloading
    // POST: returns this AND other
    // TODO: add declaration here and the definition in tribool.cpp
};
```

Exercise "Tribool"

Where do we even start?

1. First (`int`) Constructor
2. Second (`std::string`) Constructor
3. Implement `string()` method
4. Implement logical AND as an operator

Exercise "Tribool"

Where do we even start?

1. First (`int`) Constructor
2. Second (`std::string`) Constructor
3. Implement `string()` method
4. Implement logical AND as an operator

Where to put all this?

- Declarations into `Tribool.h`
- Definitions into `Tribool.cpp`
  - Using Out-of-Class definitions using the Scope Resolution Operator (`::`)

Let’s Code (together)!

- Open "Tribool" on **code** expert

Let’s Code (together)!

- Open "Tribool" on **code** expert
- We’re doing a live coding session

Exercise "Tribool" Concepts

We encountered the following concepts and keywords while solving this task:

Exercise "Tribool" Concepts

We encountered the following concepts and keywords while solving this task:

- Classes and Structs
- Visibility
- Operator Overloading
- Declaration vs Definition
- Out-of-Class-Definitions
- `const` Functions
- Constructors ("C-tors")
- Member Initializer Lists
- ...

Questions?

5. Iterators

What even are Iterators?

- Iterators are used iterate (or move) through elements in a Container

[1] https://en.cppreference.com/w/cpp/container

What even are Iterators?

- Iterators are used iterate (or move) through elements in a Container
- What are Containers then?
    - Containers are objects that are used to store collections of elements
    - Some common C++ containers include

[1] https://en.cppreference.com/w/cpp/container

What even are Iterators?

- Iterators are used iterate (or move) through elements in a Container
- What are Containers then?
  - Containers are objects that are used to store collections of elements
  - Some common C++ containers include
    - `std::vector`
    - `std::set`
    - `std::list`

[1] https://en.cppreference.com/w/cpp/container

What even are Iterators?

- Iterators are used iterate (or move) through elements in a Container
- What are Containers then?
  - Containers are objects that are used to store collections of elements
  - Some common C++ containers include
    - `std::vector`
    - `std::set`
    - `std::list`
  - A complete list of the containers of the C++-standard library can be found here,[1] but most are not of relevance for us now

[1] `https://en.cppreference.com/w/cpp/container`

Using Iterators on Containers

Very easy and by design always the same!

Given: a container named `c`

[2] Very useful for unwieldy return types

[3] PTE: Past-the-End

Using Iterators on Containers

Very easy and by design always the same!

Given: a container named `C`

- `auto`[2] `it = C.begin()`

[2] Very useful for unwieldy return types

[3] PTE: Past-the-End

Using Iterators on Containers

Very easy and by design always the same!

Given: a container named `C`

- `auto`[2] `it = C.begin()`

  Iterator pointing to first element
- `auto it = C.end()`

[2] Very useful for unwieldy return types

[3] PTE: Past-the-End

Using Iterators on Containers

Very easy and by design always the same!

Given: a container named `C`

- `auto`[2] `it = C.begin()`
  Iterator pointing to first element
- `auto it = C.end()`
  Iterator pointing to first element *past the end*[3]
- `*it`

[2] Very useful for unwieldy return types

[3] PTE: Past-the-End

Using Iterators on Containers

Very easy and by design always the same!

Given: a container named `C`

- `auto`[2] `it = C.begin()`
  Iterator pointing to first element
- `auto it = C.end()`
  Iterator pointing to first element *past the end*[3]
- `*it`
  Access (and maybe modify) current element
- `++it`

[2] Very useful for unwieldy return types

[3] PTE: Past-the-End

Using Iterators on Containers

Very easy and by design always the same!

Given: a container named `C`

- `auto`[2] `it = C.begin()`
  Iterator pointing to first element
- `auto it = C.end()`
  Iterator pointing to first element *past the end*[3]
- `*it`
  Access (and maybe modify) current element
- `++it`
  Advance iterator by one element

[2] Very useful for unwieldy return types

[3] PTE: Past-the-End

6. Exercise "Find Max"

Exercise "Find Max"

Exercise "Find Max"

```
// PRE:  i < j <= v.size()
// POST: Returns the greatest element of all elements
//         with indices between i and j (excluding j)
int find_max(const std::vector<int>& v, int i, int j) {
  int max_value = 0;

  for (; i < j; ++i) {
    if (max_value < v[i]) {
      max_value = v[i];
    }
  }

  return max_value;
}
```

Exercise "Find Max"

Exercise "Find Max"

- Open "Find Max" on **code** expert

Exercise "Find Max"

- Open "Find Max" on **code** expert
- Think about how you would approach the problem with pen and paper

Exercise "Find Max"

- Open "Find Max" on **code** expert
- Think about how you would approach the problem with pen and paper
- Implement a solution (optionally in groups)

Exercise "Find Max" (Solution)

Exercise "Find Max" (Solution)

```
// PRE:  (begin < end) && (begin and end must be valid iterators)
// POST: Return the greatest element in the range [begin, end)
int find_max(std::vector<int>::iterator begin,
             std::vector<int>::iterator end) {
  int max_value = 0;

  for(; begin != end; ++begin) {
    if (max_value < *begin) {
      max_value = *begin;
    }
  }

  return max_value;
}
```

Questions?

The `algorithm` Library

- Surely somebody smarter already implemented all the common algorithms for us, right?

The `algorithm` Library

- Surely somebody smarter already implemented all the common algorithms for us, right?
- Yes! The `algorithm` library

The `algorithm` Library

- Surely somebody smarter already implemented all the common algorithms for us, right?
- Yes! The **`algorithm`** library
- These functions are designed to work with various containers like vectors, arrays, lists, etc., and help in performing tasks efficiently without the need to write the algorithms from scratch each time

The `algorithm` Library

- Surely somebody smarter already implemented all the common algorithms for us, right?
- Yes! The **`algorithm`** library
- These functions are designed to work with various containers like vectors, arrays, lists, etc., and help in performing tasks efficiently without the need to write the algorithms from scratch each time
- Don't forget to **`#include <algorithm>`**

Exercise "The algorithm Library"

Exercise "The algorithm Library"

- Open "The algorithm Library" on **code** expert

Exercise "The algorithm Library"

- Open "The algorithm Library" on **code** expert
- Think about how you would approach the problem

Exercise "The algorithm Library"

- Open "The algorithm Library" on **code** expert
- Think about how you would approach the problem
- Implement a solution (optionally in groups)

Exercise "The algorithm Library" (Solution)

Exercise "The algorithm Library" (Solution)

```
// ...

int largest_element = *std::max_element(vec.begin(), vec.end());

// ...

std::sort(vec.begin(), vec.end());

// ...
```

Questions?

7. Recursion

Exercise "Recursion to Iteration 1"

Exercise "Recursion to Iteration 1"

- Open "Recursion to Iteration 1" on **code** expert

Exercise "Recursion to Iteration 1"

- Open "Recursion to Iteration 1" on **code** expert
- Think about how you would approach the problem

Exercise "Recursion to Iteration 1"

- Open "Recursion to Iteration 1" on **code** expert
- Think about how you would approach the problem
- Implement a solution (optionally in groups)

Exercise "Recursion to Iteration 1" (Solution)

Exercise "Recursion to Iteration 1" (Solution)

```
// PRE: n >= 0
int f_it(const int n) {
  if (n <= 2) {
    return 1;
  }
  int a = 1;                        // f(0)
  int b = 1;                        // f(1)
  int c = 1;                        // f(2)
  for (int i = 3; i < n; ++i) {
    const int a_prev = a;           // f(i-3)
    a = b;                          // f(i-2)
    b = c;                          // f(i-1)
    c = b + 2 * a_prev;             // f(i)
  }
  return c + 2 * a;                 // f(n-1) + 2 * f(n-3)
}
```

Exercise "Recursion to Iteration 2"

Exercise "Recursion to Iteration 2"

- Open "Recursion to Iteration 2" on **code** expert

Exercise "Recursion to Iteration 2"

- Open "Recursion to Iteration 2" on **code** expert
- Think about how you would approach the problem

Exercise "Recursion to Iteration 2"

- Open "Recursion to Iteration 2" on **code** expert
- Think about how you would approach the problem
- Implement a solution (optionally in groups)

Exercise "Recursion to Iteration 2" (Solution)

Exercise "Recursion to Iteration 2" (Solution)

```
// PRE: n >= 0
int f_it(const int n) {
  if (n == 0) {   // special case
    return 1;
  }

  std::vector<int> f_values(n+1, 0);
  f_values[0] = 1;

  for (int i = 1; i <= n; ++i) {
    f_values[i] = f_values[i-1] + 2 * f_values[i / 2];
  }

  return f_values[n];
}
```

Questions?

8. Outro

General Questions?

See you next time

Have a nice week!