Programming in C++

https://fan1x.github.io/cpp21.html
tomas.faltin@matfyz.cuni.cz

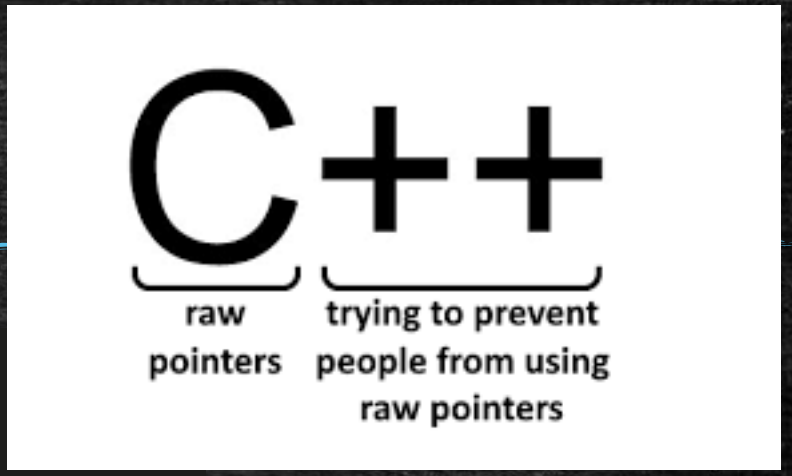
Programming in C++ - lab 7

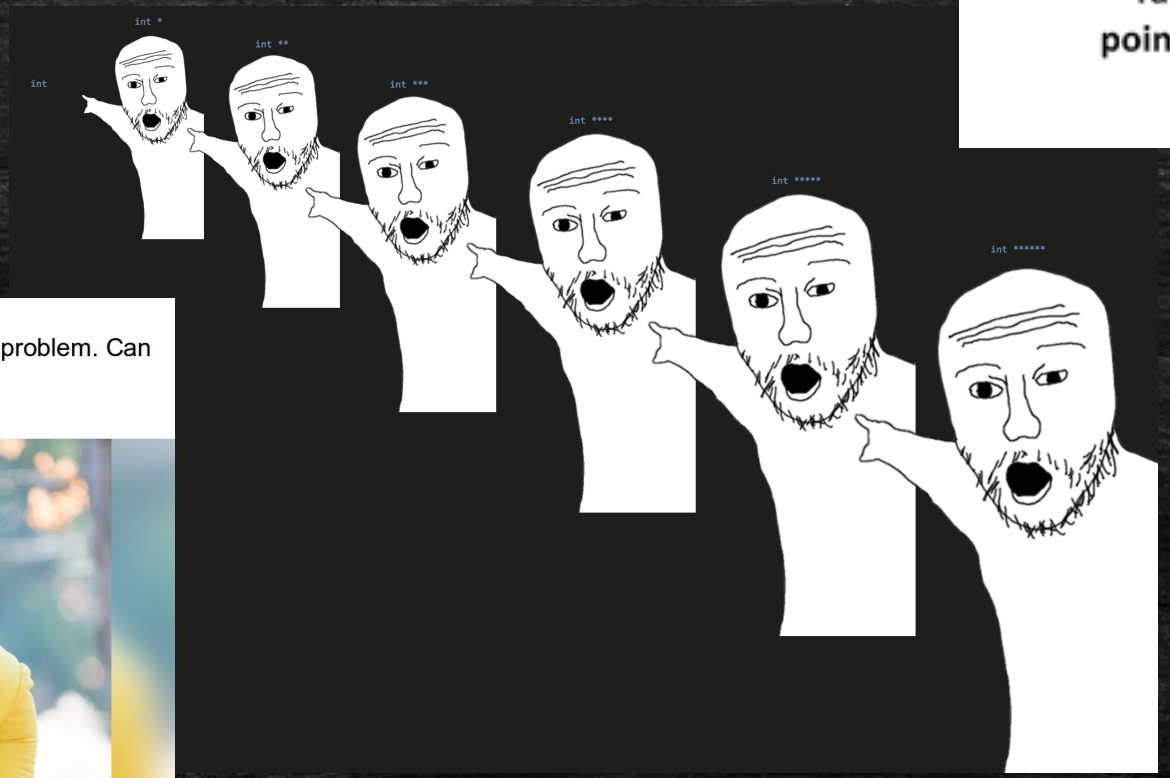
https://fan1x.github.io/cpp21.html
tomas.faltin@matfyz.cuni.cz

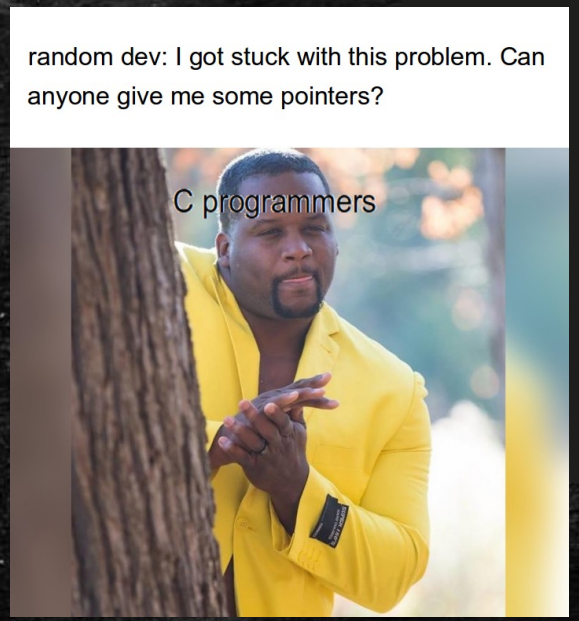
Reminder

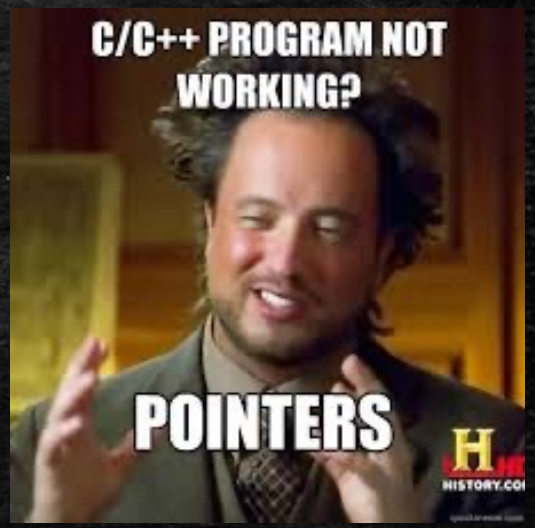
- Two large homeworks in ReCodex (total 40 points)
  - Points are included in the final score from the course
  - Smaller HW – 15 points, 15/11 → 5/12
  - Larger HW – 25 points, most probably 29/11 → 23/12
    - Ping me if you want to start sooner
- Software project
  - Topic must be approved by 27/11/2022
  - POC: 18/12/2022
  - First submission: 02/04/2023
  - Final submission: 28/05/2023

Pointers


C++
raw pointers
trying to prevent people from using raw pointers

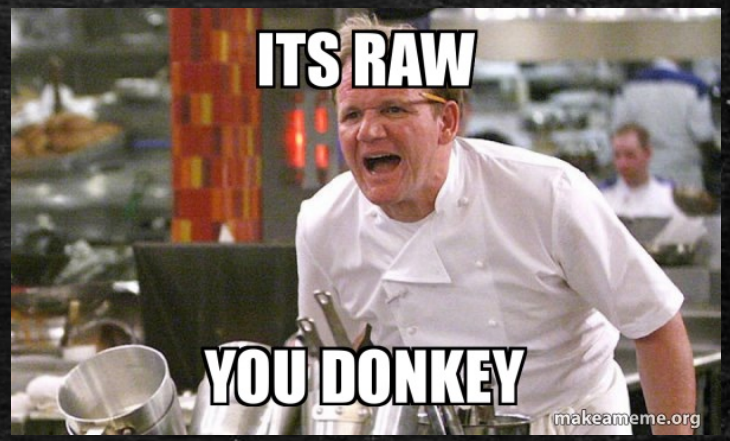

int
int *
int **
int ***
int ****
int *****
int ******


random dev: I got stuck with this problem. Can anyone give me some pointers?
C programmers


C/C++ PROGRAM NOT WORKING?
POINTERS

Dynamic Allocation

- Use smart pointers (no raw pointers, i.e., **no new/new[]**)
- **Single owner**
  - Most common case
  - Passing the ownership - move only, no copy
  - `unique_ptr<T>`
- Shared ownership (multiple owners)
  - `shared_ptr<T>`
  - `weak_ptr<T> // to break the cycle`
- Creation: make_unique<T>, make_shared<T>
- Allocation of consecutive memory (~array)
  - `make_unique<int[10]>()`

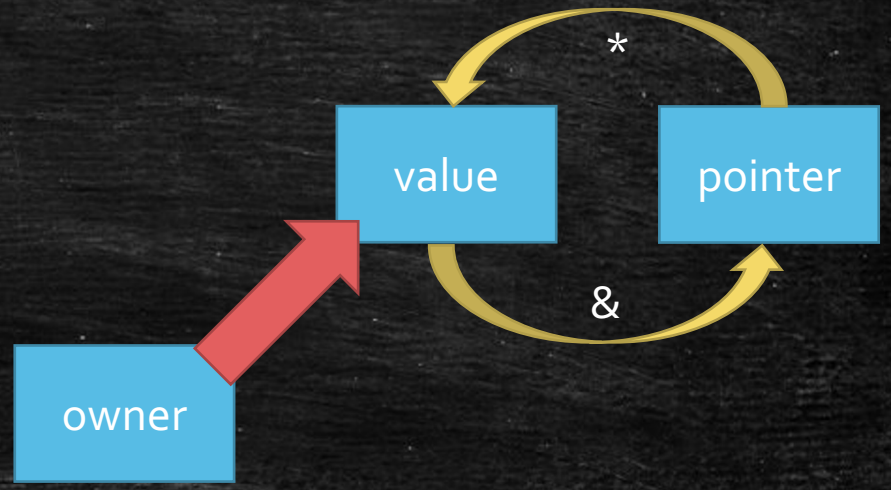


Observers

- Working with the pointer with **no changes to ownership**
- Returned type is a (const) pointer
  - Smart pointers - `get()`
  - Observer ~ Address
    - Getting an address: `&x`
- To access the values through a pointer
  - operator`*`(), operator`->`()

Owners And Observers: Quick Rules

- Owner → `unique_ptr<T>, shared_ptr<T>`
- Modifying observer → `T *` (pointer)
- Read-only observer → `const T *`



- Getting an observer
  - smart pointers → `get()`
  - otherwise → & (observer ~ address)
- Accessing the value
  - access the value directly → *
  - access member attribute → `->` (same as doing `(*var).`
- Pointing to nothing → `nullptr`

Pointers In Memory

```
int main() {
  int i = 2;
  int *pi = &i;
  int **ppi = &pi;
  cout << i;      // 2
  cout << pi;     // ..00
  cout << *pi;    // 2
  cout << ppi;    // ..04
  cout << *ppi;   // ..00
  cout << **ppi; // 2
}
```

Address	Value	(Variable)
....		
..00	2	i
..02		
..04	..00	pi
..06		
..08	..04	ppi
..0a		
....		

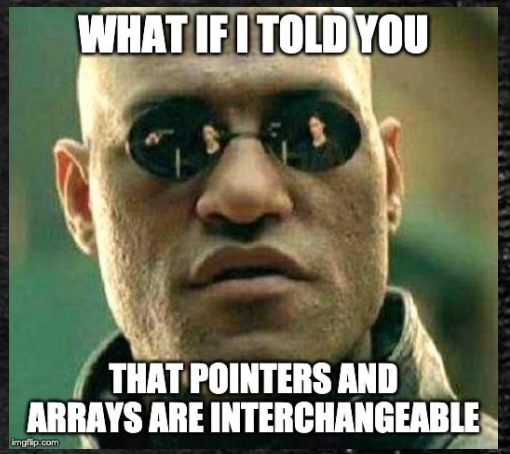
Q: What if we call `cout << *i`?

Q: Is there a difference between static and dynamic allocation?

Pointer Arithmetic

- Takes type into account

```
unique_ptr<int[]> int_arr = make_unique<int[]>(10);
for (size_t i = 0; i < 10; ++i) { int_arr[i] = 0; }
int *ptr = int_arr.get();
(*ptr) = 1;
*(ptr + 1) = 2;
cout << *ptr << ' ' << *(ptr + 1) << endl; // 1 2
ptr += 1;
cout << *ptr++ << endl; // 2
cout << *ptr << ' ' << *(ptr - 1) << endl; // 0 2
*ptr-- = 3;
cout << ptr[-1] << ' ' << ptr[0] << ' ' << ptr[1] << endl; // 1 2 3
```



Notes On Dynamic Allocation

- Prefer static/automatic storage
  - ☺ `complex c;`
  - ☹ `auto c = make_unique<complex>();`
- Prefer containers
  - ☺ `vector<int> vi(10);`
  - ☹ `auto ia = make_unique<int[]>(10);`
- Dynamic allocation is slow
- Use only when necessary
  - Object lifetime doesn't correspond to function invocations
  - Polymorphism

Linked List Example

```
struct node {
  unique_ptr<node> next;
  int value;

  node(int value, unique_ptr<node> &&next) :
    value(value), next(std::move(next)) {}
};

class linked_list {
  unique_ptr<node> first_node;
public:
  node *front() {
    return first_node.get(); // Observer
  }

  const node *back() const {
    node *ptr = first_node.get(); // Observer
    if (ptr != nullptr) {
      while (ptr->next != nullptr) {
        ptr = (*ptr).next.get(); // Equivalent to ->
      }
    }
    return ptr;
  }
```

```
  void push_front(int value) {
    auto new_node = std::make_unique<node>(value, std::move(first_node));
    first_node = std::move(new_node);
  }

  void pop_front() {
    auto first = std::move(first_node);
    first_node = std::move(first->next);
  } // automatic deallocation of first
```

Operator overloading

- Implement your own operators
  - +, -, ->, /, [], …
- Keep the semantic!
- https://en.cppreference.com/w/cpp/language/operators

Homework1: Finish the LL

- size(), print(), push_back(), pop_back()
- ctor(), ctor(init_size, default_value), dtor
- operator[]

Homework2: int vector

- Implement your own integer vector
- Mandatory operations
  - default ctor, ctor(size_t, value_type), copy/move ctor/assignment
  - size(), capacity(), reserve(), push_back(), operator[]()
- Use allocations of arrays, no LL
- **Q:** How many owners does it need?

Large homework – Data Aggregation

- In ReCodex
- Deadline: 5/12 (Monday) 23:59
- 15 points (10p + 5p)
  - Functionality: max 10 points
  - Code culture: max 5 points
    - ~ `points_for_functionality/2`
  - -5 points per each week

Programming in C++ - lab 6

https://fan1x.github.io/cpp21.html
tomas.faltin@matfyz.cuni.cz

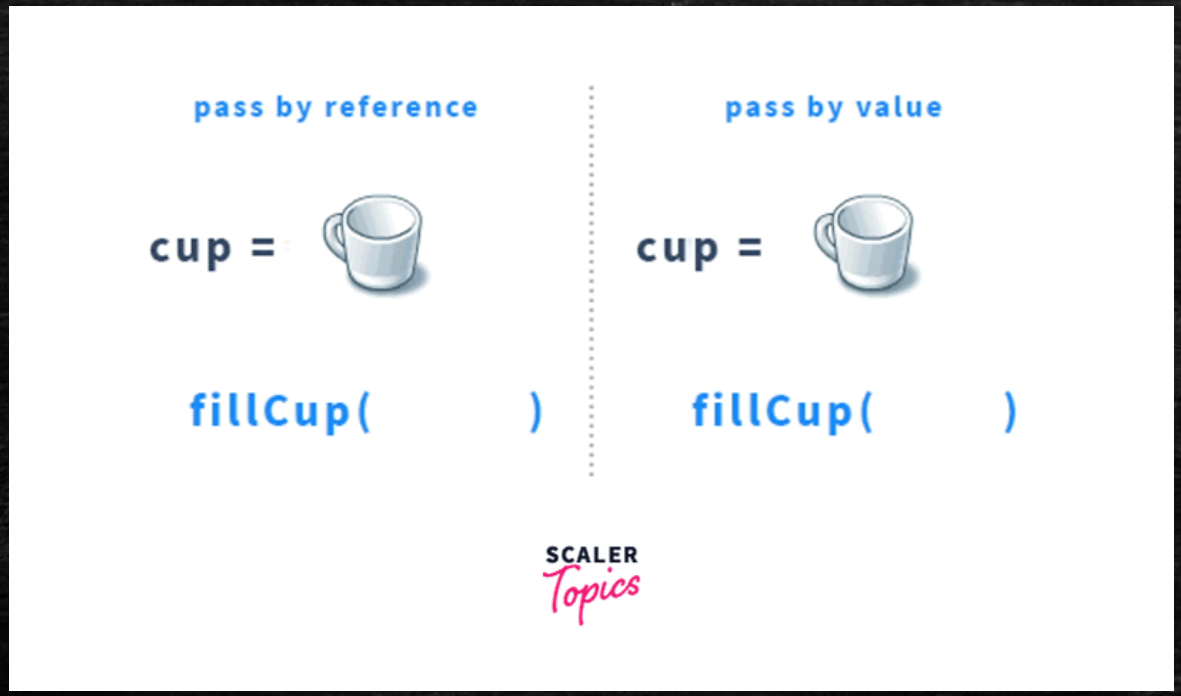
Reminder

- Two large homeworks in ReCodex (total 40 points)
  - Points are included in the final score from the course
  - Smaller HW – 15 points, assigned 15/11
  - Larger HW – 25 points, ~December
- Software project
  - Topic must be approved by 27/11/2022
  - POC: 18/12/2022
  - First submission: 02/04/2023
  - Final submission: 28/05/2023

Homework1 Solution

Returning A (Const) Reference

- To give an access to data inside an object
  - E.g., getters, operator=(), ...
- The returned data lives even after end of the function



Returning A (Const) Reference - Example

```
class complex_double {
  double im;
  double re;
public:
  explicit complex_double(double re, double im = 0.0) {
    this->im = im;
    this->re = re;
  }

  double &real_part() {
    return re;
  }
  double real_part() const {
    return re;
  }

  double &imaginary_part() {
    return im;
  }
  double imaginary_part() const {
    return im;
  }
};
```

```
int main() {
  complex_double cd(1, 2);
  cout << '{' << cd.real_part()
       << ',' << cd.imaginary_part() << '}' << endl;

  cd.imaginary_part() = 0;
  cout << '{' << cd.real_part()
       << ',' << cd.imaginary_part() << '}' << endl;
}
```

Containers

- **std::vector<Type>** - dynamic array
  - my_vec[idx] = value, push_back(), …
- **std::array<Type, Size>** - fixed size array
  - my_array[idx] = value, …
- **std::deque<Type>** - double ended dynamic queue/array
  - push_back(), push_front(), back(), front(), …
- **std::list<Type>** - linked list
- **std::map<Key,Value>**, **std::unordered_map<Key, Value>** - map
  - my_map[key] = value, find(), insert(), …
- **std::set<Key>**, **std::unordered_set<Key>** - set
  - contains(), insert(), find(), …

Homework1: Dictionary

```
// An example of API
class Dictionary {

  // Insert a new language and returns its ID
  size_t add_language(const string &name);

  // Insert new words into a dictionary
  bool add_vocabulary(size_t words1_language_id, const string &word1,
      size_t words2_language_id, const string &word2);

  // Translate a given text with the given language into the output language
  string translate(size_t input_language_id, const string &text,
      size_t output_language_id) const;

  // Automatically translate a given text into a given language
  string translate(const string &text, size_t output_language_id) const;

  // Return all vocabularies for a given language
  const vector<string> &all_vocabulary(size_t language_id) const;
};
```

Homework2: Simple People Database

- In Recodex: https://recodex.mff.cuni.cz/
- Deadline: Monday 13:00
- NOT the official large homework
  - Just to test your knowledge and the access to the ReCodex

Programming in C++ - lab 5

https://fan1x.github.io/cpp21.html
tomas.faltin@matfyz.cuni.cz

Reminder

- Two large homeworks in ReCodex (total 40 points)
  - Points are included in the final score from the course
  - Smaller HW – 15 points, ~November
  - Larger HW – 25 points, ~December
- Software project
  - Topic must be approved by 27/11/2022
  - POC: 18/12/2022
  - First submission: 02/04/2023
  - Final submission: 28/05/2023

Homework1 Solution

Declaration/Definition

```
// file: my_class.hpp

#ifndef MY_CLASS_HPP
#define MY_CLASS_HPP

void fn(int x);

class my_class {
public:
  my_class();
  int exec(int x);

private:
  double d;
  static size_t i;
};

#endif // MY_CLASS_HPP
```

```
// file: my_class.cpp

#include “my_class.hpp”
#include <iostream>

void fn(int x) {
  cout << “fn()”;
}

my_class::my_class() : d(1.0) {
  cout << “ctor”;
}

int my_class::exec(int x) {
  for(int i=0; i < x; ++i) { … }
}

size_t my_class::i = 0;
```

Declaration/Definition

```
// file: my_class.hpp

#ifndef MY_CLASS_HPP
#define MY_CLASS_HPP

void fn(int x);

class my_class {
public:
  my_class();
  int exec(int x);

private:
  double d;
  static size_t i;
};

#endif // MY_CLASS_HPP
```

guards

declarations

```
// file: my_class.cpp

#include “my_class.hpp”
#include <iostream>

void fn(int x) {
  cout << “fn()”;
}

my_class::my_class() : d(1.0) {
  cout << “ctor”;
}

int my_class::exec(int x) {
  for(int i=0; i < x; ++i) { … }
}

size_t my_class::i = 0;
```

#include header

definitions

Homework1: Summing Program

- Implement special methods only
  - ctors, dtor, operators, …
- You can add O(1) attributes into C
  - E.g., cannot add a vector
- Use C::`print()` for printing
  - Cannot use anything else for printing
- Example (
  - Input: `5 7`
  - Output:

```
Summing numbers:
5
6
7
Preparing...
Sum of the numbers:
18
```

```
class C {
  /* CAN ADD MORE ATTRIBUTES */
  const int value;
  /* USE THIS FUNCTION FOR PRINTING */
  void print() const {
    cout << value << "\n";
  }

public:
  /* IMPLEMENT SPECIAL METHODS ONLY */
};

class D {
  std::vector<C> cs;
  /* CANNOT ADD MORE ATTRIBUTES */
public:
  /* IMPLEMENT SPECIAL METHODS ONLY */
};

int main(int argc, char *argv[]) {
  int first, last;
  cin >> first >> last;
  cout << "Summing numbers:\n";
  D d(first, last); // prints number first, first+1, ..., last
  cout << "Preparing...\n";
  D d2 = d;
  cout << "Sum of the numbers:\n";
  d2 = d; // prints sum of numbers first..last
}
```

Homework2: Piškvorky for 2 players

- For 2 players only
  - Set the names at the beginning
- Game ends when one of the player has 5 in a row
  - Write who is the winner
- Validate user inputs

Programming in C++ - lab 4

https://fan1x.github.io/cpp21.html
tomas.faltin@matfyz.cuni.cz

Homework Feedback

- Use const functions for read-only functions
  - `print() const, get_matrix() const, get_vector() const`
- Use `class` or `using` to create new types
  - Decomposition!
  - `using` can be anywhere (inside the class as well)

Argument Passing - Recap

- By copy/value
  - `int max(int x, int y);`
- By const-reference:
  - `Matrix sum(const Matrix &m1, const Matrix &m2);`
- By reference
  - `void find_zero_matrix(const vector<Matrix> &ms,`
    `Matrix &zero_matrix);`

Argument Passing – By R-value Reference (&&)

- To transfer an ownership
- Moves the object into a function
  - the object no longer lives outside the function
- Typical usage
  - a single owner (`std::unique_ptr`)
  - moving large objects
- Use `std::move()` on the caller side

```
vector<unique_ptr<int>>::push_back(unique_ptr<int> &&new_obj);
vector<unique_ptr<int>> vector_of_ints;
vector_of_ints.push_back(move(make_unique<int>(x));
```

Static With Classes

- Attribute/method belongs to a class (not an object/instance)
- Need to share attribute/method among the objects/instances
- Most things belong to an object

```
class Verbose {}; // class

int main()
{
  Verbose v1; // object/instance
  Verbose v2(2); // object/instance
}
```

Static With Class

```
class CountingClass {
  static size_t num_instances;

  static void inc_num_instances() {
    ++num_instances;
  }

  static void dec_num_instances() {
    --num_instances;
  }

public:
  static bool has_instance() {
    return num_instances > 0;
  }

  static size_t get_num_instances() {
    return num_instances;
  }

  CountingClass() { inc_num_instances(); }

  CountingClass(const CountingClass &) {
    inc_num_instances();
  }

  ~CountingClass() { dec_num_instances(); }
};
```

```
size_t CountingClass::num_instances = 0;

void f() {
  cout << CountingClass::get_num_instances() << endl; // 0
  CountingClass cc1;
  cout << CountingClass::get_num_instances() << endl; // 1
  CountingClass cc2 = cc1;
  cout << CountingClass::get_num_instances() << endl; // 2
  std::vector<CountingClass> ccs(10);
  cout << CountingClass::get_num_instances() << endl; // 12
}

int main() {
  cout << CountingClass::get_num_instances() << endl; // 0
  f();
  cout << CountingClass::get_num_instances() << endl; // 0
}
```

Special Methods In Classes

```
class Verbose {
  int x;
public:
  Verbose() {
    cout << "default ctor\n";
    this->x = 1;
  }

  Verbose(const Verbose &v) {
    cout << "copy ctor\n";
    this->x = v.x;
  }

  Verbose(Verbose &&v) {
    cout << "move ctor\n";
    this->x = v.x;
    v.x = 0;
  }

  ~Verbose() {
    cout << "dtor\n";
  }

  Verbose(int x) {
    cout << "user ctor\n";
    this->x = x;
  }

  Verbose &operator=(const Verbose &v) {
    cout << "copy assignment\n";
    this->x = v.x;
    return *this;
  }

  Verbose &operator=(Verbose &&v) {
    cout << "move assignment\n";
    this->x = v.x;
    return *this;
  }
};

int main()
{
  Verbose v1; // default ctor
  Verbose v2(2); // user ctor
  Verbose v3{3}; // user ctor
  Verbose v4(v2); // copy ctor
  Verbose v5 = v3; // copy ctor
  Verbose v6(std::move(v1)); // move ctor
  Verbose v7 = std::move(v4); // move ctor
  v1 = v2; // copy assignment
  v2 = std::move(v3); // move assignment
} // Calls destructors
```

Homework1: Implement class C

- Finish program so it writes: `1,2,3,…,16`
- Touch **only** class C, nothing else
  - Nothing can be into `main()` or `fn_XXX()`
- Don't use `exit()`, `break`, `goto`, …
- Hint: which methods are called?

```
class C { /* implement me */ };

// Don't touch anything below!!!
void fn_copy(C) {}
void fn_cref(const C&) {}
void fn_rref(C&&) {}

int main(int argc, char* argv[])
{
  cout << "1\n";
  C c1;
  cout << "3\n";
  C c2(c1);
  cout << "5\n";
  C c3 = c2;
  cout << "7\n";
  fn_copy(c1);
  cout << "9\n";
  fn_cref(c1);
  fn_copy(std::move(c1));
  fn_rref(std::move(c2));
  cout << "11\n";
  c3 = c2;
  cout << "13\n";
  c2 = std::move(c1);
  cout << "15\n";
}
```

Homework2: Finish 3DMatrix For Integers

- Correct all issues in the previous HW
- Implement correctly all special methods
- Show usage/test

Programming in C++ - lab 3

https://fan1x.github.io/cpp21.html
tomas.faltin@matfyz.cuni.cz

Down to operator

```
void op_downto(int x) {
  while (x --> 0) {
    cout << x;
  }
}

op_downto(10); // prints 9,8,7,…,1,0
```

Homework Feedback

- Use `const &` for large objects
- Only source codes and project/config files to GIT
  - No binaries (they can be compiled from the source codes)
- Use STL functions
  - `isdigit(), stoi(), …`
- Prefer C++ strings to C-style strings
  - `std::string, std::string_view`

`Class/Struct - Recap`

- Put all related things (data, functions) together
- No real difference except for default visibility, inheritance, …
  - `class` – by default everything `private`
  - `struct` – by default everything `public`
- Internal things → `private`
  - `protected` if need access from a child
- Read-only functions → `const`
  - const-correctness
- Special methods (**constructor**, destructor, …)

Defining your own types - using

- Use `using` (or `typedef` in old C/C++)
- Can be used together with templates (later)

```
using my_int = int;
using int_pair_t = std::pair<my_int, my_int>;
using my_string = std::vector<char>;
using int_vector_t = std::vector<int>;

my_int x = 3;
int_pair_t p{10, 20};
my_string str = {‘a’, ‘b’, ‘c’};
int_vector_t vi(10, 0);
```

Constant values – constexpr/const

- Read only value that cannot be changed
- Naming values in code
  - ~ Every number in the code should be a named constant
- `constexpr` – constant value (potentially) evaluated in the compile time
  - Can be used as arguments to templates
- `const` – constant value
- Both can be used together with `static` (later)

```
constexpr double PI = 3.14;
constexpr size_t MAX_SIZE = 16 * 1024 * 1024;
```

Coding: 3D Matrix for Integers - API

- ctor(), ctor(width, length, heigth)
- set(x, y, z, value), get(x, y, z), print()
- set_width(), set_length(), set_heigth(), get_width(), get_length(), get_height()
- get_matrix(x), get_matrix(y), get_matrix(z)
- get_vector(x, y), get_vector(y, z), get_vector(x, z)
- clear() – set all values to 0 (zero)
- fill_with_value(value) – set all values to a given value
- num_zeros(), num_negatives(), num_positives();

Coding: 3D Matrix for Integers - Hints

- Think about the desing
  - array → matrix → 3D matrix → 4D matrix → … → XD matrix
  - Design simple first, then continue to the next level
- No need to focus too much on performance yet
- Focus:
  - Passing arguments: const-references, references, …
  - const functions
  - class design
    - Decomposition into functions
    - Function reusing
    - private/public

Coding: 3D Matrix - Improvements

- `print()`
- `sort_vector(x, y)`
  - Use `std::sort()`
- change underlying matrix container - `std::deque, std::list`
  - the change to different container must be only few lines of change
    - Hint: use `using`
- change underlying matrix container - `std::array`
  - Use large enough array
    - ! Use constants
  - Report error in case of overflow

Programming in C++ - lab 2

https://fan1x.github.io/cpp21.html
tomas.faltin@matfyz.cuni.cz

Recap

Homework Example

Class/Struct

- Put all related things (data, functions) together
  - Represents objects in OOP
  - almost everything should belong to a class
- No real difference except for default visibility, inheritance, …
  - `class` – by default everything `private`
  - `struct` – by default everything `public`
- Internal things → private
  - protected if need access from a child
- Read-only functions → const
  - const-correctness
- Special methods (**constructor**, destructor, …)

Class Example

```
class calculator {
  // by default everything is private
  void sum();
  void substract();

public:
  calculator() { /* default ctor */ }
  calculator(const std::string &str) () {
    /* ctor */
  }
  void calc(const std::string &str);
  void print_result() const;

private:
  void multiply();

protected:
  void init();

private:
};
```

can be used multiple times

semicolon at the end!

```
calculator c; // no need for new!
c.calc("1+2-3");
c.print_result();

// calling non-default ctor
calculator c2("1+2-3");
c.print_result();

// creating a vector
std::vector<calculator> calcs;
```

Class vs. Struct

- Use class if the class has an invariant; use struct if the data members can vary independently

```
struct coordinate {
  int x;
  int y;
  int z;

  coordinate();
  coordinate(int x);
  coordinate(int x, int y);
  coordinate(int x, int y, int z);

  void set(int x, int y, int z);
};
```

Dynamic Array - std::vector<T>

- Beware of time complexity
- vector<bool> optimization

```
#include <vector>
int main() {
  std::vector<int> vi{1, 2, 3, 4, 5, 6}; // [1, 2, 3, 4, 5, 6]
  std::vector<float> vf(5, 0.0f); // [0.0, 0.0, 0.0, 0.0, 0.0]
  std::cout << vi[3] << " " << vf.at(3) << std::endl; // access the 4th! element
  std::cout << vi.size();
  vi[3] = 100; vi.at(6) = 600; // access the 4th and 7th element
  vf.push_back(100.0f); vf.emplace_back(200.0f); // insert at the end
  vf.emplace_back(200.0f); // create element at the end
  vf.insert(3, 300.0f); vf.emplace(3, 300.0f); // insert at the specific place
  vf.emplace(3, 300.0f); // create element at the specific place
  vi.pop_back(); // erase the last element
  vf.erase(2); // erase the 3rd element
  vi.clear(); // clear whole container
  vi.reserve(10); // reserve space(=memory) for 10 elements
  vi.resize(10); // actually create 10 elements using default ctor
}
```

3D Matrix for Integers – minimal API

- ctor(), ctor(x, y, z)
- set(x, y, z, value), get(x, y, z), print()
- set_width(), set_length(), set_heigth(), get_width(), get_length(), get_height()
- get_matrix(x), get_matrix(y), get_matrix(z)
- get_vector(x, y), get_vector(y, z), get_vector(x, z)
- clear() – set all values to 0 (zero)
- fill_with_value(value) – set all values to a given value
- num_zeros(), num_negatives(), num_positives();

3D Matrix for Integers - Hints

- Think about the desing
  - array → matrix → 3D matrix → 4D matrix → … → XD matrix
  - Design simple first, then continue to the next level
- No need to focus too much on performance yet
- Focus:
  - Passing arguments: const-references, references, …
  - const functions
  - class design
    - Decomposition into functions
    - Function reusing
    - private/public

Programming in C++ - lab 1

https://fan1x.github.io/cpp21.html
tomas.faltin@matfyz.cuni.cz

Basic information

- Email: tomas.faltin@matfyz.cuni.cz
- Labs web: https://fan1x.github.io/cpp22.html
- Lecture web: https://www.ksi.mff.cuni.cz/teaching/nprg041-web/
- Mattermost
  - Invite link in SIS/Notice-board
  - Channel: `nprg041-cpp-faltin`
- Gitlab
  - https://gitlab.mff.cuni.cz/
  - https://gitlab.mff.cuni.cz/teaching/nprg041/2022-23/faltin

Communication is the key

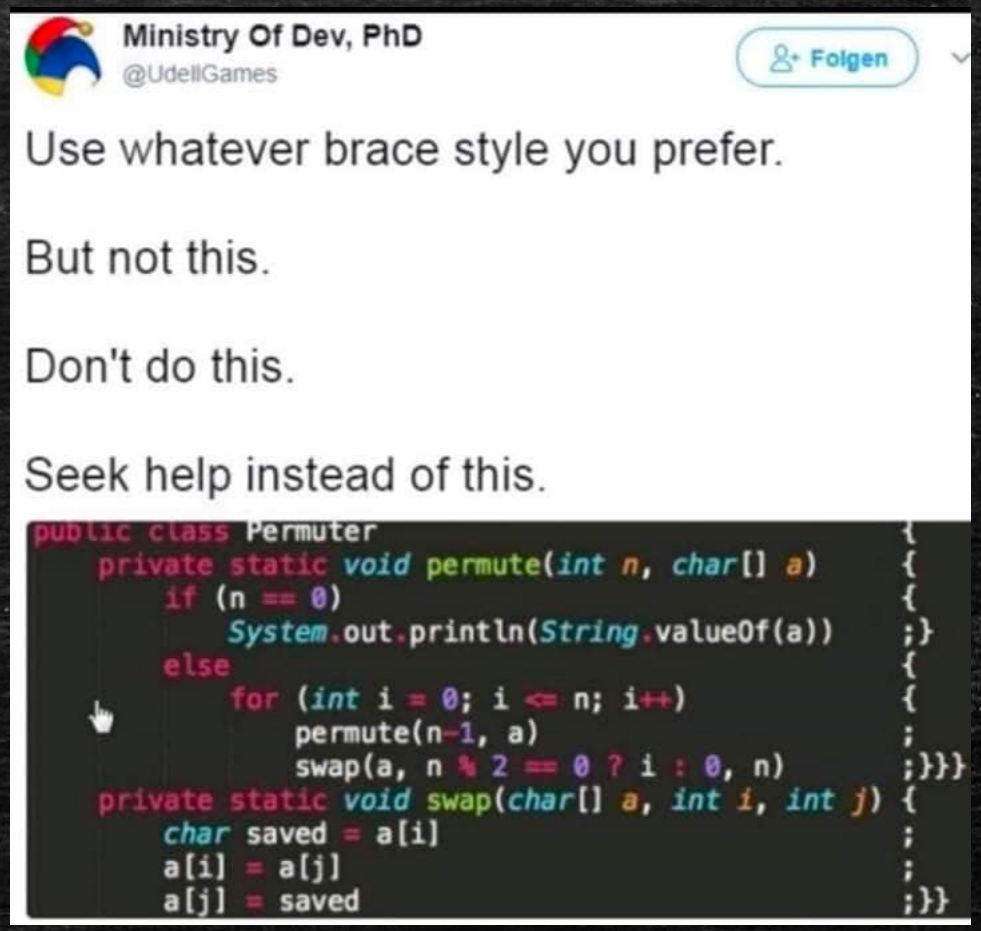
- Don’t be afraid to ask
- Be proactive
  - via email
  - on Mattermost (instant)
    - DM if related to you only
    - Into a channel if others can benefit from it
- If you struggle with something
- If you feel like you might miss a deadline

Labs credit

- Submitted homeworks before Sunday midnight (Sunday 23:59)
  - to Gitlab
  - Even if not attending!
  - Won't be graded, for feedback only
- Two large homeworks in ReCodex (total 40 points)
  - Points are included in the final score from the course
  - Smaller HW – 15 points, ~November
  - Larger HW – 25 points, ~December
- Software project
  - Topic must be approved by 27/11/2022
  - POC: 18/12/2022
  - First submission: 02/04/2023
  - Final submission: 28/05/2023
  - **All the steps typically mean multiple iterations within multiple days. If you wait for the last minute, there is a chance you won't make it**

Code Requirements - Consistency

- Consistency
  - Be consistent within the code
  - keep a single code style

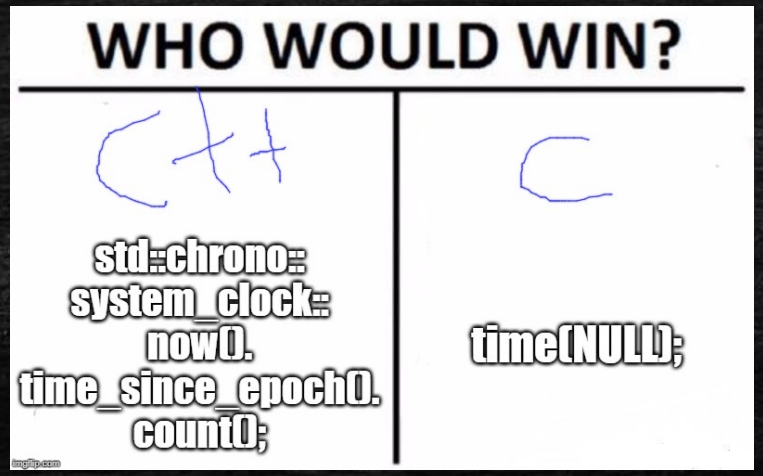


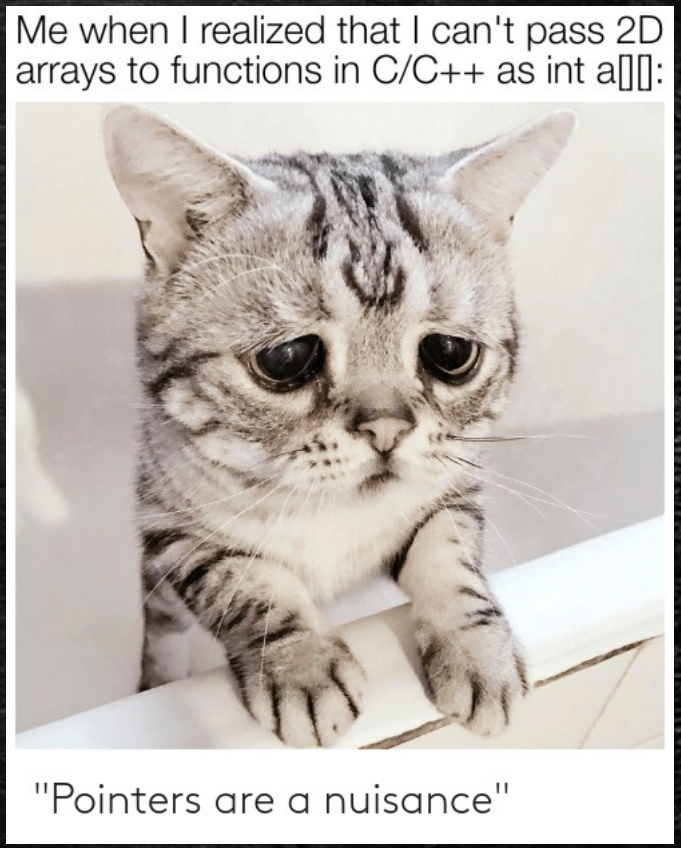
Code Requirements – Readability

- Code doesn't contain commented/dead parts
- Code should be readable on its own
- Comment complicated code

Code Requirements – Safe, Modern

- Prefer using modern constructs
- Additional safety
- Maybe performance
- E.g., prefer `std::vector<int>` to `new int[]`





Code Requirements – Working

- OFC, if the code is not working, all the above points are not that important
- they will help you with debugging at least ☺

Why C++

"C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg."
-- Bjarne Stroustrup

"It was only supposed to be a joke, I never thought people would take the book seriously. Anyone with half a brain can see that object-oriented programming is counter-intuitive, illogical and inefficient."
-- Stroustrup C++ 'interview' (https://www-users.cs.york.ac.uk/susan/joke/cpp.htm)

Working Environment

- Use anything you like ☺
- IDEs
  - Visual Studio
    - License for students at https://portal.azure.com/…
  - VS Code
  - Clion
  - Code::Blocks
  - Eclipse
  - …
- Compilers
  - MSVC, GCC, Clang+LLVM, ICC, …

C++ (interesting) links

- Reddit, Slack, …
- https://en.cppreference.com/w/
- http://www.cplusplus.com/
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- https://www.youtube.com/user/CppCon
- https://isocpp.org/
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/
- https://godbolt.org/
- …

Learning C++

- C++ in 100 seconds: https://youtu.be/MNeX4EGtR5Y
- C++ in 31h: https://youtu.be/8jLOx1hD3_0

Hello World

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

int main() {
  std::string name;
  std::cin >> name;
  std::cout << "Greetings from " << name << std::endl;
  return 0;
}
```

Hello World

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

int main() {
  std::string name;
  std::cin >> name;
  std::cout << "Greetings from " << name << std::endl;
  return 0;
}
```

Include the libraries which implements the used STL constructs (`string`, `cin`, `cout`)

The main entry point/function for all programs. The execution starts here

Declare a variable of type string

Read from standard input (keyboard)

Write to standard output (screen)

All the STL constructs live inside `std` namespace

Compilation

- `c++ --version`
  - c++ is a compiler, here GCC
- `c++ hello_world.cpp -o hello_world`
  - Compile program into `hello_world` executable (using default settings)
- `c++ -Wall -Wextra -Werror -O3 -std=c++2b hello_world.cpp -o hello_world`
  - Wall: Show all warnings
  - Wextra: Show additional extra warnings
  - Werror: Thread all warnings as errors
  - O3: level of optimizations
  - std=c++2b: Used C++ standard
- Or use IDE ☺

More Complex Program

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

using namespace std;

void pretty_print(const vector<string>& args) {
  // ... args[i]
}

int main(int argc, char** argv) {
  vector<string> args(argv, argv+argc);
  pretty_print(args);
  return 0;
}
```

More Complex Program

```cpp
#include <iostream>
#include <string>
#include <vector>

using namespace std;

void pretty_print(const vector<string>& args) {
  // ... args[i]
}

int main(int argc, char** argv) {
  vector<string> args(argv, argv+argc); // Wrap arguments
  pretty_print(args);
  return 0;
}
```

Include the whole std namespace

Passing the argument by (const) reference

Number of arguments

Arguments of the program on the command line

Transform "magically" the arguments into C++ array of strings

Functions And Parameters

```
int get_max(int v1, int v2) {
  return v1 > v2 ? v1 : v2;
}

int get_max1(const vector<int> &ints)       {
  int max = std::numeric_limits<int>::min();
  for (int x : ints) {
    max = get_max(x, max);
  }
  return max;
}
```

```
bool get_max2(const vector<int> &ints, int &max) {
  max = std::numeric_limits<int>::min();
  for (int x : ints) {
    max = get_max(x, max);
  }
  return !ints.empty();
}

std::tuple<bool, int> get_max3(const vector<int> &ints) {
  int max = std::numeric_limits<int>::min();
  for (int x : ints) {
    max = get_max(x, max);
  }
  return { !ints.empty(), max };
}
```

Functions And Parameters

- read-only input parameter
  - Most of the types (string, vector, …) → use const-reference - `const &`
    - `int get_max(const vector<int> &ints)`
  - For small numeric types (`int, float, double`, …) → use direct parameter
    - `int get_max(int v1, int v2)`
- output parameters
  - Single output parameter → use `return` value
    - `int get_max(const vector<int> &ints)`
  - Few output parameters → use tuple/pair/structure
    - `std::tuple<bool, int> get_max(const vector<int> &ints)`
  - Many output parameters → use reference - `&`
    - `bool get_max(const vector<int> &ints, int &max)`

Homeworks

1. Hello World
2. A greeting program (use names from arguments)
   - `hello.exe Adam Eve` → `Hello to Adam and Eve`
   - What is inside `args[0]`?
3. Summation of numbers from arguments
   - `sum.exe 1 2 3 4 5` → `15`
   - `stoi(), stod(), stoX()`
     - Functions for transformation from **s**tring **to** <something>
4. A simple calculator (only for operations +-)
   - `calc.exe 1+2+3-4` → `2`
   - to Gitlab
   - The previous programs are not needed, they should give you a lead