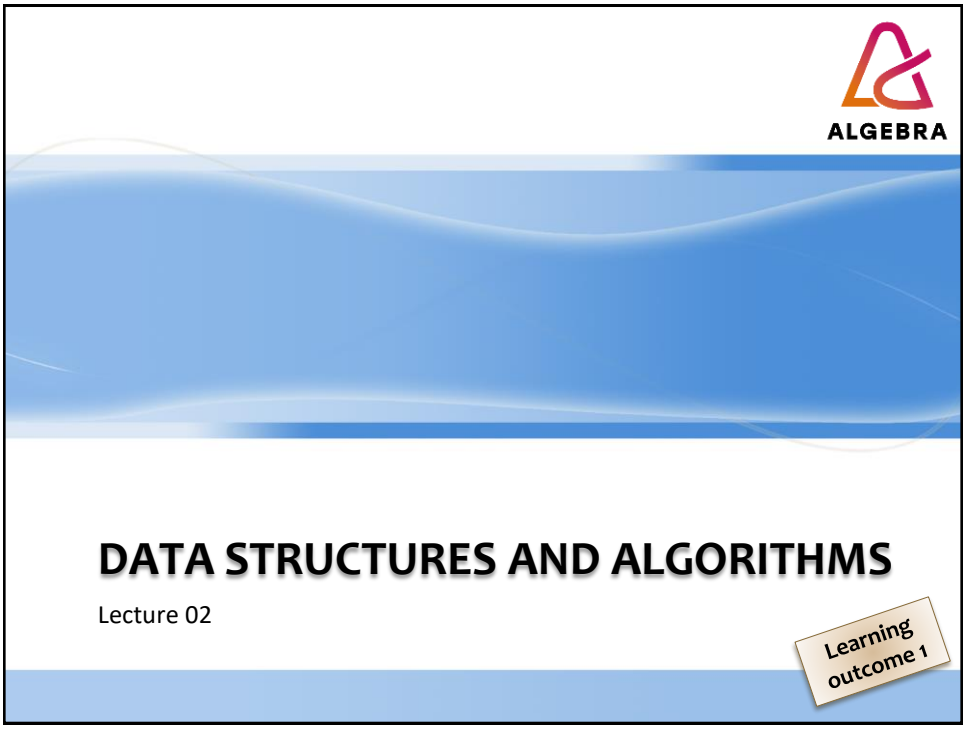


Lets repeat

- Write a program that defines a complex number and enables its correct displaying, multiplication with a scalar and addition with another complex number. Demonstrate the operation of all operations in the main program.
  - Questions we have to ask ourselves:
    - Will we choose a structure or a class?
    - Which members will be private and which will be public?
    - If our variables are private, how do we set them up?

ALGEBRA

Solution

```
#pragma once
class ComplexNumber {
private:
    int x;
    int y;
public:
    void initialize(int px, int py);
    void multiply(int n);
    void add(ComplexNumber k);
    void display();
};
```

ComplexNumber.h

```
#include "ComplexNumber.h"

int main() {
    ComplexNumber k1;
    k1.initialize(5, 5);
    k1.multiply(2);
    k1.display();

    ComplexNumber k2;
    k2.initialize(5, 2);

    k1.add(k2);
    k1.display();
    return 0;
}
```

Source.cpp

```
#include <iostream>
#include "ComplexNumber.h"
using namespace std;

void ComplexNumber::initialize(int px, int py) {
    x = px;
    y = py;
}
void ComplexNumber::multiply(int n) {
    x *= n;
    y *= n;
}
void ComplexNumber::add(ComplexNumber other) {
    x += other.x;
    y += other.y;
}
void ComplexNumber::display() {
    cout << x << " + " << y << "i" << endl;
}
```

ComplexNumber.cpp

ALGEBRA

CONSTRUCTORS AND DESTRUCTORS

ALGEBRA

Method overloading (more on OOP)

- Each method can have multiple versions
  - We say it is overloaded
  - They must differ in the number and / or types of parameters
  - Parameter names and return values are not important
- Lets try calling all three functions – how does compiler know which one to call?

```
void display(int a) {
      cout << "A: " << a << endl;
}
void display (int a, int b) {
      cout << "B: " << a << " " << b << endl;
}
void display (double a, double b) {
      cout << "C: " << a << " " << b << endl;
}
```


Problem

- Lets try this:

```
int main() {
      ComplexNumber k1;
      k1.display();

      return 0;
}
```

- Why do we get such a result?
- Does it make sense to have a complex number for which x and y are not defined?
- How can we prevent the use of such uninitialized complex numbers?

Constuctor

- Each structure and class has one or more constructors
  - Constructor is a method that is automatically called when creating (constructing) an object
    - Constructing an object on a stack:

      ```
      ComplexNumber k;
      ```

    - Constructing an object on a heap:

      ```
      ComplexNumber* k1 = new ComplexNumber;
      ```

  - The constructor is most often used to set the initial state of an object
    - Instead of initialize() method
    - E.g. receives some parameters and copies them into member variables

Constructor design

- The constructor name must always be the same as the structure or class name
- It has no return value (it doesn't even have a void)
- We can define as many constructors as we want
  - Constructor overload (because the constructor is also a method)
- A constructor without parameters is called the default constructor
  - If we do not define any constructor, it is automatically created by the compiler
  - If we create any constructor, the default constructor will not be created automatically!
    - If we need it, we have to create it

Using constructors

- The constructor is called automatically when creating the object
- If a structure/class has more than one constructor defined, the parameters we pass determine which will be called
  - As with any function call, the parameters are enclosed in parentheses
  - Exception: if we want to use the default constructor, we must not write parentheses
- Which constructor will be called:

```
ComplexNumber k1;
ComplexNumber k2(4);
ComplexNumber k3(6, 8);
ComplexNumber k4[5];
ComplexNumber* k5 = new ComplexNumber(4, 6);
```


Pointer this

- A special pointer `this` is always available in structure or class methods
  - Points to the object to which the method is called
  - More details: OOP
- What would happen if we didn't use `this`:

```
class Square {
public:
      Square(int n) {
            this->n = n;
      }
      void display() { cout << n << endl; }
private:
      int n;
};
```


Two short questions

- What is wrong with the following code :

```
class Rectangle {
public:
      Rectangle(int a) {}
};
int main() {
      Rectangle p;
      return 0;
}
```

- Enable compiling the following code from the main:

```
Rectangle p1[10];
Rectangle p2(10, 3);
Rectangle* p3 = new Rectangle(12);
```


ALGEBRA

Solving the problem of an uninitialized object

- Can we prevent the use of uninitialized complex numbers?
  - Yes, by properly defining the constructor

```
ComplexNumber::ComplexNumber(int px, int py) {
    x = px;
    y = py;
}
…

int main() {
    ComplexNumber k1(5, 5);
    k1.multiply(2);
    k1.display();
    …
    return 0;
}
```

The only way for someone to create an object of type ComplexNumber is to define both x and y

ALGEBRA

Destructor

- Every structure and class can have a destructor
  - The method called when destroying an object
    - End of function (for stack objects)
    - Call `delete` (for objects on the heap)
  - Name equal to the name of the structure or class with a tilde in front, no return value, no parameters

```
class Rectangle {
public:
    ~Rectangle() {
        cout << "Destructor executing" << endl;
    }
};
```

  - Used to release resources (for example, if new goes to the constructor, then `delete` goes to the destructor)

Question

- What the next code will print:

```
class Square {
public:
      Square(int n) {
            cout << "Constructor, n=" << n << endl;
            this->n = n;
      }
      ~Square() { cout << "Destructor, n=" << n << endl; }
private:
      int n;
};
int main() {
      Square k1(4);
      Square* k2 = new Square(5);
      delete k2;
      Square k3[2] { Square(6), Square(7) };
      return 0;
}
```


*OPTIONAL WAY TO USE THE CONSTRUCTOR

Array initialization

- Two alternative ways of writing:

```
int a[] = { 10, 20, 30, 40, 50 };
int b[] { 10, 20, 30, 40, 50 };
```


Initialization of variables

- Standard way:

```
int a = 10;
char b = 'M';
float c = 2.2f;
```

- C ++ 11 specific way
  - Taken from array initialization
  - Objective: to unify the initializations of variables, arrays, and objects

```
int a{ 10 };
char b{ 'M' };
float c{ 2.2f };
```


Initialization of objects

```
class Point {
private:
    int x;
    int y;
public:
    Point(); // sets x and y to 0
    Point(int x, int y); // sets x and y to x and y
};
```

- Standard way:

```
Point k1;
Point k2(5, 3);
```

- C++11 specific way:

```
Point k1{};
Point k2{ 5, 3 };
```


ADDITIONAL C++ TOPICS

ALGEBRA

C++ versions

- C++ is a programming language that has been expanded with new functionalities throughout its history
- Every compiler knows how to work with certain language versions
- Some of the versions:
  - C++ 98
  - C++ 11
  - C++ 14
  - C++ 17
- For example, push_back method on vectors is present since C++ 98, while the shrink_to_fit method was introduced only in C++ 11

ALGEBRA

Exceptions

- Sometimes we have to inform the method caller that an error has occurred
  - For example, there is no file we want to open
- There are two approaches to do this:
  - Traditional, returning true for success or false for failure
    - Alternatively, using an integer
  - Modern, using the try / catch block
    - The method throws an exception if an error occurs
    - The caller catches the exception and processes it
    - More details: OOP
- There will be no emphasis on this topic at the DSA
  - If you need, you can use any approach you like

An example of the traditional approach

```
bool divide(int a, int b, int& result) {
    if (b == 0) {
        return false;
    }
    result = a / b;
    return true;
}

int main() {
    int result;
    bool ok = divide(17, 0, result);

    if (!ok) { cout << "Division by zero" << endl; }
    else { cout << result << endl; }

    return 0;
}
```


An example of the modern approach

```
int divide(int a, int b) {
    if (b == 0) {
        throw exception("Division by zero");
    }
    return a / b;
}

int main() {
    try {
        cout << divide(17, 0) << endl;
    }
    catch (const exception& err) {
        cout << err.what() << endl;
    }

    return 0;
}
```


ALGEBRA

Stringstream class

- So far we have used the following streams:
  - `cin`, `cout`, `ifstream`, `ofstream`
- The stringstream class represents the input / output stream to the character buffer in memory:
  - Include the header: `#include <sstream>`
  - Create the object: `stringstream sstr;`
  - Write to stream: `sstr << "XY" << 22 << endl;`
  - Read:
    ```
    sstr >> broj;
    getline(sstr, ime);
    sstr.str();
    ```
  - Reset stream:
    ```
    sstr.str("");
    sstr.clear();
    ```


ALGEBRA

Stringstream class applications

- The stringstream class has two basic applications for us:
  1. Merge multiple strings into one larger string (concatenation)
  2. Convert data types (usually string => something)
     - Alternative to C-like functions atoi, atof, …
- Lets solve the following tasks :

1. Let us write a function that receives three strings and returns one concatenated string.
2. Let us write a function that receives a string with three numbers separated by commas and returns their sum.

Solutions

```
string concatenate(string s1, string s2, string s3) {
    stringstream sstr;
    sstr << s1 << " " << s2 << " " << s3;
    return sstr.str();
}
int sum(string s) {
    stringstream sstr;
    sstr << s;

    int total = 0;
    int n;
    while (sstr >> n) {
        total += n;
    }
    return total;
}
int main() {
    cout << concatenate("this", "is", "test") << endl;
    cout << sum("49 1 10") << endl;
    return 0;
}
```


INTRODUCTION TO C++ STANDARD TEMPLATE LIBRARY (STL)

ALGEBRA

Introduction

- The Standard Template Library (STL) is a set of:
  - Containers
    - The container stores more data of some type
  - Algorithms in form of functions
  - Iterators (lecture 04)
- The purpose of STL is to provide the developer with the functionality she needs on a daily basis
- Knowledge of STL is the first serious step in the development of any C ++ programmer
  - cplusplus.com/reference/stl/

ALGEBRA

Example of STL container: array<T, N> (1/3)

- `array<T,N>` is just a light wrapper around a plain array on a stack
  - https://cplusplus.com/reference/array/array/
- `array<T,N>` is a generic class and when creating an object of that class the programmer must provide two data:
  - `T`: the type of data to be stored in the array
  - `N`: the size of the array to be reserved on the stack
- For example, what each of the lines will do:

```
array<string, 50> p2;
array<Rectangle, 7> p3;
array<int, 5> p1 = { 11, 22, 33, 44, 55 };
```


Example of STL container: array<T, N> (2/3)

- Usage example:

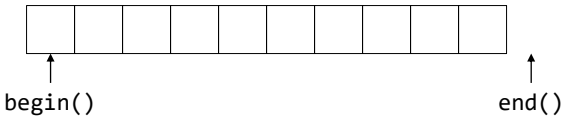
```
array<int, 5> p = { 11, 22, 33, 44, 55 };
for (unsigned i = 0; i < p.size(); i++) {
    cout << p[i] << endl;
}
```

- Isn't it easier to use a regular array?
  - Well, yes ☺
  - However, the `array` class offers an interface similar to other STL container classes so changing containers is easy:

```
vector<int> p = { 11, 22, 33, 44, 55 };
for (unsigned i = 0; i < p.size(); i++) {
    cout << p[i] << endl;
}
```


Example of STL container: array<T, N> (3/3)

- `array<T,N>` it also offers two methods that return "pointers":
  - `begin()` returns the "pointer" to the first element
  - `end()` returns the "pointer" to the first element after the end



  - Benefit: Most other containers offer the same methods

Example of STL algorithms (1/3)

- A large number of algorithms defined in the header `<algorithm>`
  - cplusplus.com/reference/algorithm/
- We will look at the following algorithm implementations:
  - `reverse(od, do)` – rearranges all elements in the range [from, to) from end to beginning

```
array<int, 5> p = { 11, 22, 33, 44, 55 };

reverse(p.begin(), p.end());

for (unsigned i = 0; i < p.size(); i++) {
    cout << p[i] << endl;
}
```


Example of STL algorithms (2/3)

- `count(od, do, val)` – returns the total number of elements in the range [from, to) that are equal to the `val`
  - Uses `operator==` to check equality

```
array<int, 7> p = { 11, 22, 33, 11, 44, 55, 11 };

int n = count(p.begin(), p.end(), 11);

cout << n << endl;
```

- What number would be printed if we replaced the middle line with:

```
int n = count(p.begin(), p.begin() + 3, 11);
```


Example of STL algorithms (3/3)

- `for_each(od, do, func)` – applies a specified function for each element in the range [from, to)

```
void multiply(int& number) {
    number *= 2;
}
void display(int& bla) {
    cout << bla << endl;
}
int main() {
    array<int, 5> p = { 11, 22, 33, 44, 55 };
    for_each(p.begin(), p.end(), multiply);
    for_each(p.begin(), p.end(), display);

    return 0;
}
```


PARSING FILES

ALGEBRA

Parsing files

- Parsing files often consists of the following steps :
  - Read the string to the specified character
    - If necessary, convert the string to another data type
    - Repeat for all data in that line
  - From the read data, construct an object and put it in a container (array, vector, linked list,…)
  - Repeat as long as there are lines in the file
- Possible approach :

```
while (true) {
    if (!getline(dat, str, ',')) {
        break; // I failed to read, so end of file.
    }
    ...
}
```


ALGEBRA

Example: parsing a file

- Task: write down which year the water level of the Huron River (Michigan) was the highest
  - File: LakeHuron.csv
  - Source: vincentarelbundock.github.io/Rdatasets/datasets.html
- Example of the first 5 lines:

```
"","time","LakeHuron"
"1",1875,580.38
"2",1876,581.86
"3",1877,580.97
"4",1878,580.8
```


Making decisions

- Prerequisite: understand what the data in the file means
- What columns do we need to complete the task?
  - time and LakeHuron
- What are the types of column data we need
  - time is int, LakeHuron is double
- What about the first line and the first column
  - Discard because we don't need it
- Which container will we use, an array or a vector?
  - Vector, because the number of lines can vary

Algorithm (1/3)

- How do we read and discard the first line?

```
"","time","LakeHuron"
"1",1875,580.38
"2",1876,581.86
"3",1877,580.97
"4",1878,580.8
```

- Then we read and discard everything until the first comma:

```
"","time","LakeHuron"
"1",1875,580.38
"2",1876,581.86
"3",1877,580.97
"4",1878,580.8
```

ALGEBRA

Algorithm (2/3)

- We read text to next comma and convert it to int

```
"","time","LakeHuron"
"1",1875,580.38
"2",1876,581.86
"3",1877,580.97
"4",1878,580.8
```

- We read text to the end of the line and convert it to double

```
"","time","LakeHuron"
"1",1875,580.38
"2",1876,581.86
"3",1877,580.97
"4",1878,580.8
```

ALGEBRA

Algorithm (3/3)

- Make an object from the read int and double and put it in the vector
- Repeat while there are lines
  - How do we know there are no more lines?
  - Reading to the next comma will return false

Solution (1/3)

- Structure:

```
struct water_level {
    int year;
    double level;
};
```

- Function main:

```
ifstream dat("LakeHuron.csv");
if (!dat) {
    cout << "Error opening file" << endl;
}

// parsing…

dat.close();
```


Solution (2/3)

- Parsing:

```
string temp;
getline(dat, temp);

stringstream sstr;
vector<water_level> entries;
water_level obj;
while (true) {
      if (!getline(dat, temp, ',')) {
            break;
      }

      getline(dat, temp, ',');
      sstr << temp;
      sstr >> obj.year;
      sstr.str("");
      sstr.clear();
```


Solution (3/3)

```
      getline(dat, temp);
      sstr << temp;
      sstr >> obj.level;
      sstr.str("");
      sstr.clear();

      entries.push_back(obj);
}
```


Better solution (1/2)

- What part of the code can we extract to a function?

```
int get_int(ifstream& dat, stringstream& sstr, string& temp, char s) {
    int res;
    getline(dat, temp, s);
    sstr << temp;
    sstr >> res;
    sstr.str("");
    sstr.clear();
    return res;
}
double get_double(ifstream& dat, stringstream& sstr, string& temp,
                                                              char s) {
    double res;
    getline(dat, temp, s);
    sstr << temp;
    sstr >> res;
    sstr.str("");
    sstr.clear();
    return res;
}
```


Better solution (2/2)

- Parsing in main now becomes:

```
while (true) {
      if (!getline(dat, temp, ',')) {
            break;
      }

      obj.year = get_int(dat, sstr, temp, ',');
      obj.level = get_double(dat, sstr, temp, '\n');

      entries.push_back(obj);
}
```