#include <iostream> —> if you want to print something on the screen
↳ directive for the preprocessor to read user input from the keyboard
↳ source code files for the compilation

using namespace std —> std::cout —> cout
↳ enclose to

```
import math
math.sqrt(4.0)
```

—>

```
from math import *
sqrt(4.0)
```

↳ don't overuse it —> Student::name
or
Course::name

A variable must always be declared before it can be used
↳ ASCII-letters, digits and underscore (_). The first character must be a letter

The data type that manipulates text is implemented by a library
—> #include <string>
Input and output operators => std::cin and std::cout
>> ↳ skips empty characters and line breaks
<<
↳ takes the printable string and returns the user-given string

getline
↳ takes —> input stream
↳ variable which the input value will be assigned to

Variable declaration

1) type of the variable } compulsory
2) name " "
3) initialization " " —> uninitialized variables can generate errors difficult to track

Local variables are defined inside a block (a pair of curly brackets {})

Data types → int, double, bool, string, char, float
↳ double precision ↳ single precision

Constant definitions → in the beginning of the script and in capital letters

```
const double PI = 3.14
```

Type conversion

```
float f = 0.123
double d = f
```

```
double d = 3
```
implicit conversions

```
Integer to a character (ASCII)
char ch = 113
```

```
int i = 0
unsigned int ui = 0
if (ui == i) {

}
```
Warning here

explicit conversion

```
if (ui == static_cast<unsigned int>(i)) {

}
```
Warning is vanished

```
cout << "Input a character: ";
char ch = ' ';
cin >> ch;
int ascii_value_of_ch = static_cast<int>(ch);
cout << "ASCII value of " << ch << " is " << ascii_value_of_ch << endl;
```

If a block has a single command, is possible to drop the curly brackets

```
while (i<10) {
  ++i;
}
```
→
```
while (i<10)
  ++i;
```

Styles

```
if(x) {
  ...
}
```

```
if(x)
{
  ...
}
```

C++ operators

==, !=, <, <=, >, >=, +=, -=, *=, /=

C++ logical operators

not a = !a

a and b = a && b

a or b = a || b

bitwise operator ^ for xor (exclusive or)

Variable types

int : normal integer that deals with positive and negative values → if 32 bits $\in [-2^{32}, 2^{32}]$

- unsigned int : natural numbers only
- long int : allow larger area of presentation than the usual int type but only if the processor architecture supports it
- unsigned long int :

C++ ROUND 1

Python

```
a = 42
b = a
c = 42
```

One data element which can have one or more names

(Reference Semantics)

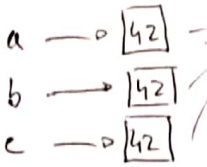
↳ different names for the same data

C++

```
int a = 42;
int b = a;
int c;
c = 42;
```

The processed data is copied during initialization and assignment

(Value Semantics)



memory cells

References &

Assign different names to the same variable (like in Python)

```
target_type& reference_name = target_variable;
```

Functions

```
function_name ( comma_seperated_parameters)
```

Python → C++

relational operators in chain

```
if (a<=b<=c)
```

```
if (a<=b and b<=c)
```

A function can be completely defined before calling it or can be declared before calling it

```
double average (double n1, double n2){
  return (n1 + n2)/2;
}
int main(){
  cout << average (2.0, 5.0) << endl;
}
```

```
double average (double n1, double n2);
int main (){
  cout << average (2.0, 5.0) << endl;
}
double average (double n1, double n2) {
  return (n1+n2)/2;
}
```

If you need a function which does not return any value

→ sub routine / sub program → void

```
void average (double n1, double n2) {
    cout << (n1 + n2)/2 << endl;
}
int main () {
    average (2.0, 5.0);
}
```

C++ uses static typing → the compiler must know the parameter types during compilation

2.0 → float

2 → int

return

EXIT_SUCCESS = 0 } constants from the library cstdlib

EXIT_FAILURE = 1 }

When creating a variable, it is possible to create a semine variable or just to reference it to an existing variable

```
void value_param_func (int number) {
    number = 2 * number;
}
int main() {
    int variable = 7;
    cout << variable << endl; // 7
    value_param_func (variable);
    cout << variable << endl; // 7
}
```

```
void refer_param_func (int & number) {
    number = 2 * number;
}
int main () {
    int variable = 7;
    cout << variable << endl; // 7
    refer_param_func (variable);
    cout << variable << endl; // 14
}
```

EFFICIENCY

When you assign a value using semantics, you copy the value but in reference semantics you simply change the target of the reference to a different data element

Imagine a very large data structure or container, copying the whole data element by element is a hard task compared to just taking the reference

If you don't want to change the original not even by accident just declare constant the reference

ions parameters are usually references

Strings and characters

string is an immutable data type

you can change an existing string except if it is declared as constant

```
string text = "abcdefg"
cout << text.at(3) << endl; // d
text.at(3) = 'x';
cout << text << endl; // abcxefg
```

the indexing operation on a string
text.at(index) will return a char type value

Methods on strings — operations that target strings

- text.length() : return number of characters
- text.find("abc") : returns at which point you can find the first "abc"
- if these numerical values (amount of characters, indexes, ...) are to be saved in a variable, the type must be string::size_type

```
int main(){
  string name = " ";
  string::size_type temp = 0;
  cout << "input your name:";
  getline(cin, name);
  temp = name.length();
  cout << "your name has " << temp << " characters " << endl;
  temp = name.find("ren");
  if(temp == string::npos){
     cout << "your name doesn't contain \"ren\"." << endl;
  }else{
     cout << "Combination \"ren\" is located starting from " << temp << endl;
  }
}
```

find returns the value string::npos if cannot find any combination

Methods

```
string :: size_type len = 0;
len = text.length();
```

```
char letter;
letter = text.at(5);
text.at(2) = 'x';
```

for replacement

text.erase(index)

destroys all characters starting at index

text.erase(index, len)

destroys from the index onwards until len position

```
string :: size_type loc = 0;
loc = text.find("abc", 5);
if (loc == string::npos){
  // could not find "abc" from
  // index 5 onwards
}else{
  // "abc" was found
}
```

```
string text = "abcdefg";
string result;
result = text.substr(3); // "defg"
result = text.substr(3,3); "def"
```

string substr(index)
string substr(index, len)

string::size_type rfind(target)
string::size_type rfind(target, index)

↳ like find but it will start the search at the end of the string towards the beginning

string insert(index, addition)

```
string text = "abcd";
text.insert(2, "xy"); // "abxycd"
```

text1 (<, <=, >, >=) text2
compares alphabetical order

string replace(index, len, replacement)

```
string text = "ABCDEF";
text.replace(1, 3, "xy"); // "AxyEF"
```

```
str += addition
       ↳ can be a string or character
string result = "Qt";
result += ' ';        // "Qt "
result += "Creator"; // "Qt Creator"
```

text1 + text2

```
string test1 = "Qt";
string test2 = "Creator";
string result;
result = test1 + " " + test2; // Qt Creator
```

getline(stream, line)
or
cin

reads one line of text from a file or keyboard and saves the string in the parameter line

int stoi (text) changes the parameter text to an integer	double stod (text) changes the parameter text to a real number
string text = "123"; int number; number = stoi (text); //123	string text = "123.456"; double number; number = stod (text); //123.456

The fact that the character is an integer

```
for (char letter = 'a'; letter < 'z'; ++letter)
  cout << letter;
cout << endl;
```

#include <cctype> allows the use of:

- islower (character): checks if you have a lowercase letter (boolean)
- isupper (character): " " " " an uppercase " "
- isdigit (character): " " " " a digit
- tolower (character): converts an uppercase into a lowercase
- toupper (character): " a lowercase " an uppercase

Interfaces and object-oriented programming

- Interface means an arrangement that limits the programmer's direct access a of part of a programm. That is, the programmer does not need to know how something has been implemented, yet they are still able to use its services
  string type is an example. An average programmer does not know how it works in the backstage yet he can use operations that are the public interface of the string type
- Interfaces are an essential part of object-oriented programming
- Object-oriented programming operates on objects communicating with each other via their public interfaces. The control flow of a program is based on passing messages between objects and the objects reacting to these messages

Classes and objects

A class is a data type and object is a value/variable whose data type is a class

```
class Person {
  public: // methods (member functions), by which the object of the class can be operated
    Person(string name, int age);
    string get_name() const;
    void celebrate_birthday(int next_age);
    void print() const;
  private: // attributes, that describe the concept implemented as a class. They cannot be directly accessed
           (member variables)
    string name_;
    int age_;
}; // note the semicolon here

int main() {
    Person pal("Matt", 18);
    cout << pal.get_name() << endl;
    pal.print();
    pal.celebrate_birthday(19);
    pal.print();
}

Person::Person(string name, int age):
    name_(name), age_(age) {
}
string Person::get_name() const {
    return name_;
}
void Person::celebrate_birthday(int next_age){
    age_ = next_age;
}
void Person::print() const {
    cout << name_ << ":" << age_ << endl;
}
```

Methods:

```
AClass::AClass(param1, param2):
  attribute1_(param1), attribute2_(param2){
}
```

- Constructor: always named after the class. It has no return. Always called when you create a new object. It initialize it.
- Selector: They just examine but do not change the state of the object. They have const after (). This will prevent any attempt to change the state of the object
- Mutator: they are allowed to change the state of an object
- destructor: called when an object dies

Except the constructor and destructor, methods are called like:

`object.method(parameters)`

The call of a constructor takes places automatically behind scenes every time you need to initialize a new object

```
bool Fraction::operator==(const Fraction& other) const {
    return numerator_ == other.numerator_ && denominator_ == other.denominator_;
```

```
Fraction f1(2, 3);
Fraction f2(3,4);
if (f1 == f2) ...
```

Class implementation in a separate file

main.cpp

```
#include <iostream>
#include "person.hh"
using namespace std

int main() {
    Person pal("Matt", 18);
    cout << pal.set_name() << endl;
    pal.print();
    pal.celebrate_birthday(19);
    pal.print();
}
```

person.hh

```
using namespace std;
class Person {
  public:
    Person(string name, int age);
    string get_name() const;
    void celebrate_birthday(int next_age);
    void print() const;
  private:
    string name_;
    int age_;
}; // note the semicolon
```

person.cpp

```
#include <iostream>
#include <string>
#include "person.hh"
using namespace std;
Person::Person(string name, int age):
    name_(name), age_(age) {
}
string Person::set_name() const {
    return name_;
}
void Person::celebrate_birthday(int next_age){
    age_ = next_age;
}
void Person::print() const{
    cout << name_ << ":" << age_ << endl;
}
```

- When you use the C++ libraries with the directive include you use angle brackets (< >)
- When you use your own files, (" ")

```
bool Fraction::operator==(const Fraction& other) const {
    return numerator_ == other.numerator_ && denominator_ == other.denominator_;
```

```
Fraction f1(2, 3);
Fraction f2(3,4);
if (f1 == f2) ...
```

Class implementation in a separate file

main.cpp

```
#include <iostream>
#include "person.hh"
using namespace std

int main() {
    Person pal("Matt", 18);
    cout << pal.set_name() << endl;
    pal.print();
    pal.celebrate_birthday(19);
    pal.print();
}
```

person.hh

```
using namespace std;
class Person {
  public:
    Person(string name, int age);
    string get_name() const;
    void celebrate_birthday(int next_age);
    void print() const;
  private:
    string name_;
    int age_;
}; // note the semicolon
```

person.cpp

```
#include <iostream>
#include <string>
#include "person.hh"
using namespace std;
Person::Person(string name, int age):
    name_(name), age_(age) {
}
string Person::set_name() const {
    return name_;
}
void Person::celebrate_birthday(int next_age){
    age_ = next_age;
}
void Person::print() const{
    cout << name_ << ":" << age_ << endl;
}
```

- When you use the C++ libraries with the directive include you use angle brackets (< >)
- When you use your own files, (" ")

Pointers

In C++ assigning something to a reference will change the value of a target

```
int integer = 42
int & integer_ref = integer
integer_ref = 100 // integer = 100
```

The reference still points to the variable integer since references in C++ are constant. You cannot change a reference to point to a new object but it will always point to the same place

Pointers also need you to define the type of variable

You can initialize the pointer by referencing the variable that will be the pointers target. So &variable will create a pointer that points to the variable in question.

```
target_type * variable = & target_variable
```

```
int integer = 42;
int * integer_ptr = & integer;
```

A pointer is a variable that stores a reference
Because a pointer is a variable, you can change the value it points to.

```
Person person1 ("Kalle", 6);
Person person2 ("Ville", 5);
Person * winner = & person1      — initialize
winner = & person2
winner = null ptr                → to point nowhere
```

ROUND 4

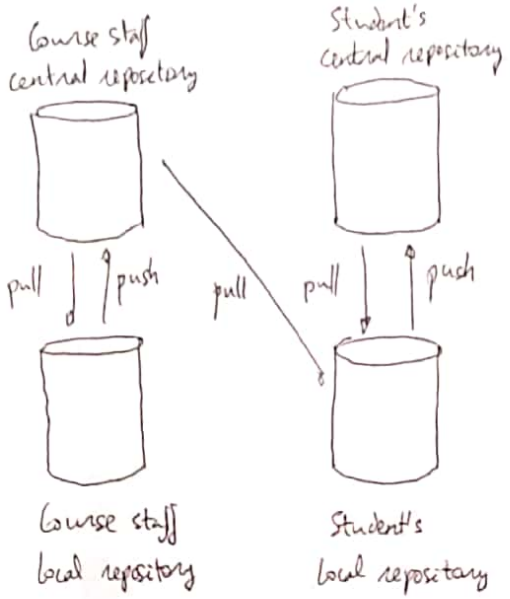
Towards C++ data structures

C++ does not include the more complicated data structures and their operations in the language itself. Instead, they are implemented in libraries called

- STL (Standard Template library) which is a combination of:
  - Containers
  - Iterators: kind of bookmarks where you save the location of a single data element within a container
  - Algorithms: ready-to-use mechanisms for executing basic operations to containers (sort the elements of a container in the order of your choice for example)
- extremely large collection of libraries

Git

Distributed version control system → it is possible to retrieve updates from several central repositories to a local repository



Qt creator → Tools > Git > Remote Repository >

> Manage Remotes > Fetch

↓

To update packages

STL's vector

All elements of a vector must be of the same type

```
#include <vector>

vector<element_type> variable_name;
```

```
myvector = {1,2,3}
myvector.push_back(4)
myvector = {1,2,3,4}
```

Adds new elements one by one with a certain value (or not)

If you want to save the return value of the method size:

```
vector<int>::size_type n_scores = scores.size();
n_scores = scores.size();
```

```
vector.pop_back()
```

The last element is removed

If the vector is empty, when pop_back is executed an exception occurs. Nonetheless, the right way is:

```
if (scores.size() != 0) {
  scores.pop_back();
} else {
  // error, you cannot remove anything from an
  // empty vector
}
```

You can access the first and last elements by:

```
cout << "first: " << scores.front() << endl
     << "last: " << scores.back() << endl;
```

Or any element by:

Indexing starts by 0 and the last element is n-1

```
cout << scores.at(3) << endl;
scores.at(index) = new_score;
```

If you use a vector as a parameter of a function it can be:

```
void function1(vector<double> measurements)
```
→ value parameter

```
void function2(vector<double>& measurements
```
→ reference parameter

```
void function3(const vector<double>& measurements
```
→ best performance reference parameter

const makes impossible to change the target of the reference → safer

Optionally, it is possible to set the amount of elements in the vector during its definition

```
vector<int> numbersA(20);
```
20 uninitialized integers in the vector numbersA

```
vector<double> numbersB(20, 5.3);
```
20 initialized to 5.3 real numbers in the vector numbersB

```
vector<string> namesA(5);
vector<string> namesB(10, "unknown");
vector<string> namesC = {"Matti", "Maija"};
```
10 times unknown

STL vector keeps the elements in the order they were stored → sequences

deque is very similar to vector but less efficient

Lo you can, unlike with vector, add elements to its beginning with push_front()
and remove them from the beginning with pop_front()

C++'s for loop

3 ways:

1) - Going through the elements in a container

```
vector<int> numeric_vector;
for (int vector_element : numeric_vector)
    cout << vector_element << endl;
}
```
variable in which will be stored each vector element

2) - Going thrugh the interval of chosen integer values

```
for (int number = 5; number < 10; ++number) {
    cout << 9 * number << endl;
}
```

3) - Creatin infinite loop

```
for (;;) {
}
```
=
```
while (true) {
}
```

1) Going through the elements of a container

```
for (int vector_element : numeric_vector) {
    cout << vector_element << endl;
}
```

vector_element is kind of a parameter
Passing by value is the default and changing the vector_element does not change the value stored in the container

If you want to modify the elements in the container

```
for (int& vector_element : numerical_vector) {
    vector_element = vector_element * 2;
}
```

Passing by reference
In a for loop it is not possible to add or remove elements from the container

```
int i = 42;
auto i = 42;
```
→ it deducts the data type automatically

```
vector<int> numerical_vector;
for (auto vector_element : numerical_vector) {
    cout << vector_element << endl;
}
```

2) Going through a chosen interval

```
for (initialization; condition; increment) {
    body
}
```

If you want to go over a certain sequence

```
for (int i : {2, 3, 5, 7, 11, 13, 17}) {

}
```
initialization list

Data vs Control (struct and enum types)

Record type (struct in C++)

Define your own data type which can group different types of data elements
It is like a class but without interface function. Nevertheless, all the fields can be accessed directly

```
struct Product {
    string product_name;
    double price;
};
```

```
Product item = {"soap", 1.23};
item.price = 0.9 * item.price; // discount 10%
cout << item.product_name << ": " << item.price << endl;
item = {"orange", 0.45};
```

You can use struct as
- a value that you store in the STL container
- a parameter of a function (a value or a reference)
- a return value

Enumeration type

When you need a variable that only can have certain values
In an enumeration all the possible values (elements) are defined by the programmer

```
enum Type_name {element 1, ..., element n};
enum Book_status {ON_THE_SHELF, BORROWED, RESERVED, LOST};
```

It is possible to use string or int but they would allow many other values
The elements of the enumeration types are integer constants defined automatically elem 1 = 0, elem 2 = 1, ... But it is possible to customize them if whished

```
enum Type_name {element 1 = value 1, ..., element n = value n};
```

Data driven programming

A computer program consists of data and the various commands targeting it

The idea is to design data structures within a program that reduce the amount of data processing commands, that is, to replace certain commands by data

Benefits:

- Shorter programs but still clean
- Smaller possibility of making errors
- Programs easier to expand and maintain

A sign of a need for data driven programming within a program is an if (or switch) structure that grows out of proportion

```
#include<string>
enum PostalAbbreviation {AL, AK, AZ, AR, ERROR_CODE};
PostalAbbreviation name_to_abbreviation(std::string name) {
  if(name == "Alabama"){
    return AL;
  }else if(name == "Alaska"){
    return AK;
  }else if(name == "Arizona"){
    return AZ;
  }else if(name == "Arkansas"){
    return AR;
  }else{
    return ERROR_CODE;
  }
}
```

Non-data driven program
A lot of "if"s

```
#include <string>
#include <vector>
enum PostalAbbreviation {AL, AK, AZ, AA, ERROR_CODE};

struct StateInfo {
  std::string name;
  PostalAbbreviation abbreviation;
};
const std::vector<StateInfo> STATES = {
    {"Alabama", AL},
    {"Alaska", AK},
    {"Arizona", AZ},
    {"Arkansas", AR}.
};
PostalAbbreviation name_to_abbreviation (std::string name) {
   for (auto s: STATES){
      if (name == s.name){
        return s.abbreviation;
      }
   }
   return ERROR_CODE;
}
```

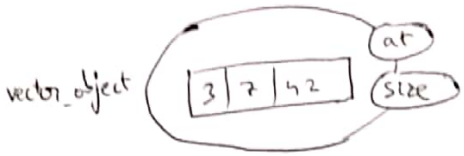
```
class Square {...}; -> class Square; // forward declaration
std::vector<std::vector<Square>> board;
using Board = std::vector<std::vector<Square>>; -> Board board;
```

↳ you avoid from repeating it

~~class Square~~

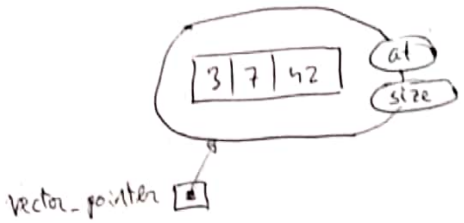
object. method (parameters);
object -> method (parameters);

- we handle the object directly
- we handle the object through a pointer



When we use a named object itself

vector_object.size()



When we use an object through a pointer

vector_pointer -> size()

or

*vector_pointer.size()

Object-oriented programming style (good practices)

- Each ~~file~~ header file (.hh) contains a single public interface (class)
  Header files contain only definitions (no implementations)
- include directives can be used to include only header files. If you include several files, you must list your own files first and after the library files
- the implementation of an interface is written in the implementation file .cpp in the same order as they are in the corresponding header file .hh
- Classes are defined with a good public interface and all the operations are conducted here
- Write the public part before the private one
- The public part has no member variables
- The private part contains no useless variables
- No operations of a class are implemented outside the member functions
- The member variables of a class must be initialized in their definition or in the initialization list of the constructor in the same order as the header file
- When possible, define a member function as const

Other good practices

- The scope of a variable must be as small as possible (e.g. block, function, class)
- Each variable must be defined separately (avoiding int a, b;)
- Variables must be initialized
- The characters for pointers and references (* and &) are written immediately after the type without an empty space
- The return type of main is always int
- The names of the parameter of a function must be given in both the declaration and definition of a class, and they must be identical in both places
- If you want to pass an object as a parameter for a function, use a reference to the object instead of the object itself
- The default value parameters of a function must be given in the declaration of the function and you must not give more of them in the definition of the function
- A function must not return a reference or a pointer to its local data
- Records must NOT have member functions, but operators are allowed
- Do not index vectors with brackets but with at function

File management

```
#include <iostream>
#include <string>
#include <fstream> // required library for file management
using namespace std;
int main () {
    string filename = "";
    cout << "Input file name : ";
    getline (cin, filename);
    ifstream file_object (filename);
    if (not file_object) {
        cout << "Error" << endl;
    } else {
        int sum = 0
        string line;
        while (getline (file object, line)) {
            sum += stoi (line);
                    └→ converts string to int
        }
        file_object.close ();
        cout << "sum : " << sum << endl;
    }
}
```

read write
ifstream / ofstream

ifstream >> variable
ofstream << variable

cout << output
output_stream << output
cin >> variable
input_stream >> variable

getline (input_stream, line) ← char type
getline (input_stream, line, separator)
└→ string variable

Read a char from the input stream
input_stream.get (ch)

```
// read the file one character at a time
char input_char;
while (file_object.get (input_char)) {
    cout << "character : " << input_char << endl;
}
```

```
string text_line = "";
// one line of text from keyboard
getline (cin, text_line);
// until the next colon. It reads several lines at the same time
getline (file_object, text_line, ':');
```

Checking the success of stream operations

The compiler interprets the stream as bool type

```
ifstream file_object ("file.txt");
if (file_object) {
    // opening successful
} else {
    // error
}
```

```
while (getline (file_object, line)) {
  // you will enter the loop structure if reading a line
  // from the stream succeeded
}
// the loop structure ends when you cannot read any
// more lines, that is, when the whole file has been read
```

STL iterators

Used to examine and modify the elements stored in a container

Lo bookmark that remembers the location of one element in a container

An iterator does not point to the element of a container but the space between two elements

Elements in certain containers cannot be indexed

One iterator: mark the location of an element

Two iterators: show a range of elements in the container

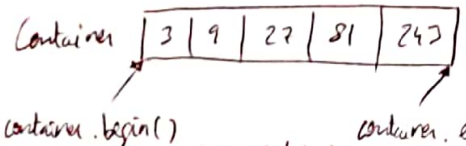
Lo every element between two elements

Iterator types:

- container_type < element_type > :: iterator
- container_type < element_type > :: const_iterator
  - examine the elements but not edit them
  - needed if container variable has been defined as constant

```
const vector<string> MONTHS = ...
```

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
  vector <double> numbers = {1.0, 1.5 2.0, 2.5};
  vector <double> :: iterator vec_iter;
  vec_iter = numbers.begin();
  while (vec_iter != numbers.end()){
    cout << *vec_iter << " ";
    ++vec_iter;
  }
  cout << endl;
}
```



The element of a container can be handled (pointed to by an iterator) by targeting the * operator at it

If you want to target with methods at the object the iterator is pointing to, use -> operator

++ makes the iterator point to the element that is next in line

-- " " " " " " previous " "

== } test if two iterators are pointing to the same element or different ones
!=

If you just want to go through all the elements within the container:

```
int main() {
    vector<double> numbers = {1.0, 1.5, 2.0, 2.75};
    for (auto element : numbers) {
        cout << element << " ";
    }
    cout << endl;
}
```

It uses iterators behind the scenes

If you want to change container's elements in the body of for structure:

```
int main() {
    vector<double> numbers = {1.0, 1.5, 2.0, 2.75};
    for (auto &element : numbers) {
        element *= 2;
    }
}
```

If you add or delete elements elements from a container, the values of the iterators that were set to point to its original elements will not be valid anymore

```
auto amount = 0;  // int
auto temperature = 12.1  // double
auto iter = numbers.begin(); // vector<double>::iterator
```

STL algorithms

#include <algorithm>

It provides common operations you need to conduct on elements within a container

They are able to operate with any container

Every function has at least two parameters

sort elements of the vector into an ascending order

```
vector<double> double_vector;
...
sort(double_vector.begin(), double_vector.end());
```

```
vector<double>::iterator range_begin = double_vector.begin();
vector<double>::iterator range_end = double_vector.end();
++range_begin; // pointing now to the second element
--range_end; // pointing now to the last element
// sort all the elements excluding the first and the last one
sort(range_begin, range_end);
```

The operations in the algorithm library don't affect on the target element of the second iterator parameter. The last element of the vector was not sorted because range_end was pointing to it.

```
vector<string> enemies;
// how many people by the name of Aki are there in the enemies vector?
cout << count(enemies.begin(), enemies.end(), "Aki") << "enemies by the name
        of Aki!" << endl;
```

min_element and max_element search the container for the element with the smallest or greatest value and return the iterator pointing at each element:

```
vector<int> amounts;
...
vector<int>::iterator smallest_it;
smallest_it = min_element(amounts.begin(), amounts.end());
cout << "Smallest amount: " << *smallest_it << endl;
```

find searches the container for the chosen value and returns the iterator into the first element it finds or otherwise the end iterator of the container

```
vector<string> patients;
...
vector<string>::iterator iter;
iter = find(patients.begin(), patients.end(), "Kai");
if (iter == patients.end()) {
  // no Kai patient found
} else {
  //Kai was found and is located where iter points at
  //print it and delete it from the queue
  cout << *iter << endl;
  patients.erase(iter);
}
```

String, set and map types have the method function find which does not depend on #include <algorithms> so it is several times faster

replace replaces all chosen values with a new value

```
replace(text_vector.begin(), text_vector.end(), "TUT", "TUNI");
```

it does not work with set or map structures

reverse reverses the order of the given range

```
vector<Player> order_of_turns;
...
// on the next round, the players take their turns in the opposite order
reverse(order_of_turns.begin(), order_of_turns.end());
```

shuffle shuffles the elements and puts them in a random order

```
vector<Card> deck;
minstd_rand gen; // a random number generator called gen is created
...
shuffle(deck.begin(), deck.end(), gen);
```

it does NOT work with set or map structures

Copy copies the elements of the container into another container

```
string word = "...";
...
// initialize char_vector to have as many elements as there are characters in the word
vector<char> char_vector(word.length());
copy(word.begin(), word_end(), char_vector.begin());
```

You must have free space at the target location of the copy action

```
class Student {
 public:
    Student(string name, int student_id);
    string fetch_name() const;
    int fetch_student_id() const;
    void print() const;
 private:
    string name_;
    int id_;
};
bool compare_ids(const Student& stud1, const Student& stud2){
  if(stud1.fetch_student_id() < stud2.fetch_student_id()){
    return true;
  }else{
    return false;
  }
}
bool compare_names(const Student& stud1, const Student& stud2){
  if(stud1.fetch_name() < stud2.fetch_name()){
    return true;
  }else{
    return false;
  }
}
int main(){
  vector<Student> students = {
    {"Teekkari, Tiina", 12 11 21}, {"Aikkari, Antti", 111222}, ... };
  // let's order and print in the increasing order of students ids
  sort(students.begin(), students.end(), compare_ids);
  for(auto stud : students)
    stud.print();
  cout << string(30, '-') << endl;
  // let's order in the increasing order of names
  sort(students.begin(), students.end(), compare_names);
  for(auto stud : students) stud.print();
}
```

You can search and print the student with greatest id number

```
vector<Student>::iterator iter;
iter = max_element(students.begin(),
                   students.end(),
                   compare_ids);
// because the target of the
// next print method is the
// object the iterator is pointing
// to, you will use ->
// operator instead of normal .
```

Associative containers STL

They store elements in a way that is as fast as possible to search

The searches are conducted with a (search) key

Set

```
#include <set>
```

```
set<string> ship_has_been_loaded;
```
after the definition, it is empty

```
set<int> lottery_numbers_1;
...
set<int> lottery_numbers_2 (lottery_numbers_1);
...
lottery_numbers_1 = lottery_numbers_2;
```
It can be initialized with another set of the same type
Another set of the same type can be assigned to it

```
set<int> prime_numbers = {2, 3, 5, 7, 11, 13, 17};
...
set<string> friends;
...
friends = {"Matti", "Maija", "Teppo"};
```
You can initialize and assign manually with curly brackets

```
ship_has_been_loaded.insert("dogs")
```
You can add new elements
But it won't do anything if that same element is already there

```
// amount of friends
cout << friends.size() << endl;
```

```
// the word is not included in the set, let's add it
if (ship_has_been_loaded.find(word) == ship_has_been_loaded.end()) {
    ship_has_been_loaded.insert(word);
    cout << "OK!" << endl;
// the word was already included in the set
} else {
    cout << "You lost," << word << "has already been loaded!" << endl;
}
```

```
// Teppo is not a friend anymore
friends.erase("Teppo");
```

```
ship_has_been_loaded.clear()
```
removes all elements

```
if (lottery_numbers_1 == lottery_numbers_2) {
  // both of the sets include the same elements
  }else{
  // the contents of the sets differ
}
```

```
if (not friends.empty()){
  // there is at least one friend
}
```

map

#includes <map>

It includes both the types of the key and the mapped value.

```
map<string, double> prices;
```
After definition, unitialized map is empty

```
map<string, string> dictionary_1;
...
map<string, string> dictionary_2( dictionary_1);
...
dictionary_1 = dictionary
```

```
map<string, double> prices_2 = {
  {"milk", 1.05},
  {"cheese", 4.95},
  {"ghe", 3.65},
};
...
prices_2 = {
  {"pepper", 2.10},
  {"cream", 1.65},
  {"chocolate", 1.95},
};
```

the data related to a search key can be accessed with the method at:

```
cout << prices_2.at("cream") << endl; //1.65
prices_2.at("cream") = 1.79;
```

If the key cannot be fount in the map, the program will crash

You CANNOT add new elements to map with the method at

You can avoid this by checking before

```
if (dictionary_3.find(word) != dictionary_3.end()) {
  // the word was found in the map
  cout << dictionary_3.at(word) << endl;
}else{
  // the word was not found in the map. Error
}
```

You can remove a key-value pair from the map by giving the search key you want to be removed as a parameter to the method erase

```
if ( prices_2.erase("chocolate")) {
  // the erasing was successful, "chocolate" is not in the pricelist anymore
} else {
  // the erasing was not successful, "chocolate" was not in the pricelist
```

Not necessary, since it does not cause error

You can add a new key-value pair with the insert method

```
dictionary_1.insert({word, word_in_finnish});
```

```
Cout<< "Dictionary contains " << dictionary_2.size() << "pairs of words" << endl;
```

→ The action of going through the elements of map has also been implemented with iterators but it is not as straightforward as with other STL containers

Because now each element contains a key and a mapped value then it has been implemented by making the elements of map structs

```
map<int, string> students = {
    // id, name
    {123456, "Teekari, Teemu"},
    ...
};
```

the elements stored in the structure would be

```
struct {
  int first; // key value
  string second; // mapped value
};
```

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main(){
  map<int, string> students = {{20000, "Tiekari, Tiina"}, {123456, "Teekari, Teemu"}, ...};
  map<int, string>:: iterator iter;
  iter = students.begin();
  while (iter != students.end()) {
    cout << iter -> first << " " << iter -> second << endl;
    ++ iter;
  }
}
```

If you want to avoid direct use of iterators, you can use for to go through the elements of the map:

```
int main(){
    map<int, string> students = {{20000}, "Teemari, Tiina"}, {123456, "Teevuori, Teemu"}, ...};
    for(auto data_pair : students) {
        cout << data_pair.first << " " << data_pair.second << endl;
    }
}
```

Because data_pair is not an iterator in the for loop but a struct located in map, we have the usual operator. to handle the fields first and second

Pair

```
#include <utility>
pair<int, char> pair1;
pair1.first = 1;
pair1.second = 'a';
pair<int, char> pair2 (1,'a');
pair<int, char> pair3;
pair3 = make_pair(1,'a');
auto pair4 = make_pair(1,'b');
```

map and set have the property that their elements are ordered according to the key values. A new element is inserted to such a location that the order is preserved

This can be inefficient for some operations

↳ solution? unordered_map and unordered_set

↳ when there is no need to keep elements in order and you don't need to traverse the whole container element by element. But just to search for a certain element at a time

ROUND 6

Recursion

Defining something in terms of the subject itself

You present the solution in the same format as the original problem but as its sub-problems with a simpler solution, After that, you apply the recursive rule over and over again onto the received sub-problems, until the problem is simple enough for you to see the solution directly

The goal is to divide and conquer

A recursive function is self-defined, meaning it calls itself

```
unsigned int factorial (unsigned int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial (n-1);
}
```

- A function is recursive if it calls itself (directly or indirectly)
- During the execution, as many instances of the recursive function are "on" as there are unfinished recursive calls
- Each instance has its own values for the parameters and local variables

Two characteristics

- It has to have a terminating condition (or several) that recognizes the trivial cases of the problem and reacts to them without having to make a new recursive call
- Each recursive call has to simplify the problem in question in order to finally reach the trivial case

Recursion creates repetition

- Any loop can be replaced with a recursion
- The best use of recursion is solving problems that are, by nature, recursive
- Recursive solutions are slightly slower and uses more memory that the loop structure

Types of recursion

- Direct recursion : when a function calls itself in its own body
- Indirect / Mutual recursion : the func_a calls the func_b and viceversa

There can be more functions (circle of calls)

Try to avoid it, since is difficult to understand

- Tail recursion : the return value of a recursive call becomes the return value of the calling instance without any additional operations

The recursive call is located in such a point in the function that after the call, there are no statements to execute or expressions to evaluate

```
unsigned int factorial (unsigned int n, unsigned int result) {
  if (n == 0) {
    return result;
  } else {
    return factorial (n-1, n * result);
  }
}
```

```
cout << factorial (5, 1) << endl;
```

When the execution of a function is finished, all the local variables of the function are removed from the memory of the computer

So by using a reference called by the main program, the reference becomes a dangling (colgado) pointer because when the function is finished, its local variables are removed from the memory of the computer.

A dangling pointer can mean either a reference or a pointer targeting a variable that no longer exists

Mistake: Returning a reference that points to a local variable

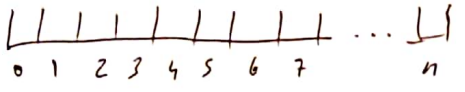
Until now, all used variables have been automatic which means the compiler would find a storage place for them in the memory and take care of destroying them.

But with more complex programs it is better that the programmer is able to determine exactly when the variable is destroyed

Pointers are relevant when it comes manual memory management and are important when implementing the higher-level structure of a program (indirect pointing and sharing access/ownership)

Memory and memory addresses

All the initial data is stored in the main memory of the computer



The processor can store a small amount of data into each cell (memory location) and its index is the memory address

A memory location can contain a byte of data which is composed by 8 bits
Usually the memory is handled in parts that are the size of multiples of a byte

One memory location can store a data element that can be presented with 8 bits (e.g. a "char")
In case you need to store data that is presented with a larger numbers of bits, you have to use a suitable amount of consecutive memory locations

If the element is made of 32 bits (like an int) you can save such element into 4 (32/8) consecutive memory locations. If it is made of 64 bits (like a double) then it would require 8 " " ".

As data is presented as bits in depths of the computer, you cannot find out what type of data is stored in a memory location only by looking at its content. That's why data needs to have a type in order to be handled correctly.

- each memory location is identified with a unique memory address
- the memory location stores the actual data
- the variable is the name for the value stored in the memory location
- each variable is connected to a memory address

Pointers

Pointer types are data types with a value set consisting of the memory address of data elements of a certain type. You can store a memory address into a pointer.

```
target_type * variable
```

When necessary, you can find out the memory address of any of the program's variables by using &. With * you can access the contents of the storage location at the memory address which is stored in the pointer

```
int main () {
    int i;
    int * ip = nullptr;
    i = 5;
    ip = &i;
    *ip = 42;
    std::cout << i << ", "
              << *ip << ", "
              << ip << ", "
              << &ip << std::endl;
```

- Creating the integer variable i, i.e. allocating space for it in a free location in the memory (0x000002)
- Creating a pointer to the integer ip; i.e. allocating space for it in a free location in the memory (0x000008)
- Assigning the value 5 to i,
- Finding out the address of the variable i (0x000002) by using & and storing it in the variable ip, that is, in (0x000008)
- Assigning the binary form of 42 into the memory locations starting at the address stored in ip (0x000002), i.e., the same locations where i is stored, therefore the value of i is changed

- Because the pointer ip is a variable, its location within the memory can be found with the & operator

Result: 42, 42, 0x000002, 0x000008

ip [0] ; [42]

By depicting the pointer with an arrow, we can show where it points to without repeating the index of the storage location, which is irrelevant.

With * you can find out the actual pointed data from the pointer

A null pointer points nowhere. If you have stored the memory address of a class-type or struct-type value, the methods and fields of the pointed data are handled by ->

If you try to access data through the null pointer → error

```
1 int x = 3;
2 int* ptr = &x;
3 cout << ptr << endl;
4 cout << *ptr << endl;
```

3: hexadecimal number

4: 3

```
1 int x;
2 int* ptr = &x;
3 x = 4;
4 cout << ptr << endl;
5 cout << *ptr << endl;
```

4: hexadecimal number

5: 4

```
1 int x;
2 int* ptr = &x;
3 *ptr = 4;
4 cout << ptr << endl;
5 cout << *ptr << endl;
```

4: hexadecimal number

5: 4

```
1 int x;
2 int* ptr = &x;
3 x = 5;
4 cout << ptr << endl;
5 cout << *ptr << endl;
6 cout << x << endl;
7 cout << &x << endl;
```

4: hexadecimal number

5: 5

7: hexadecimal number

C++ arrays

Vector: several values of the same type need to be stored and the elements then accessed by their index number

Array: simple static data structure (fixed number of elements) which you can access by [ ]. Inherited by C

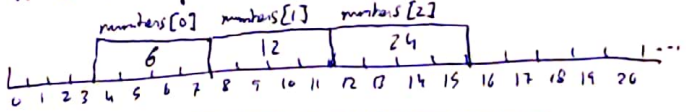
You don't need to use this primitive version of vector

`int numbers[3]` An array that has space for 3 integers

The compiler allocates enough consequent memory locations to store the three int values

```
numbers[0] = 6
numbers[1] = 12;
numbers[2] = 24;
cout << numbers[0] + numbers[2] << endl;
// 30
```

For efficiency reasons, the array type has been implemented so that it can be presented as a constant pointer to its first element




```
cout << numbers << endl; // 0x000004
```

Since the array is understood as an address to its first element, it is a primitive data type:

- An array variable cannot be assigned into another array variable with =. You cannot initialize it with another array either
- If you give an array as a parameter to a function, it will always act like a reference parameter
- It cannot be directly in a STL container

```
int * array_ptr = nullptr;
array_ptr = numbers;
while ( array_ptr < numbers +3) {
    cout << *array_ptr << endl;
    ++ array_ptr;
}
```

You can go through the array with a pointer

```
cout << numbers + 3 << endl;
// prints memory address 16
```

```
cout << number[2] << endl;
number[1] = 99;
```

=

```
cout << *(numbers +2) << endl;
*(numbers +1) = 99;
```

C level code is intended for microprocessors, machine-level programming, ...

Dynamic memory allocation

Until now we have used automatic variables, that is, allocating memory for them and then deallocating (releasing) it was automated:

- At variable definition, the compiler has taken care of finding the necessary amount of memory somewhere and allocated it
- When the variable reached the end of its lifetime (when the execution of the program leaves the block where the variable was defined. Also when the lifetime of an object ends, also do their member variables), the compiler deallocates the memory which is no needed anymore.

However, sometimes the programmer wants to control the lifetimes of the variables

Ex dynamic data structures → dynamic variables ← dynamic memory management

C++ commands → allocate → new
→ deallocate → delete

When calling "new" if there is not enough space → error

new:

- a dynamic variable does not have a name, but "new" returns a pointer, the value of which reveals where the new dynamic variable is located within the main memory
- if the variable is a class-type, new will take care of calling the constructor

delete:

- if the variable is a class-type, "delete" will call the destructor

Because the compiler does not automate anything when handling a dynamic variable, it is important to remember to deallocate the manually allocated memory when you no longer need it

If you forget it → memory leak: the program continues to keep memory locations allocated even though it no longer needs them

```
int main() {
    int* dyn_variable_address = nullptr;
    dyn_variable_address = new int(7);
    cout << "Address: " << dyn_variable_address << endl;
    cout << "Start: " << *dyn_variable_address << endl;
    *dyn_variable_address = *dyn_variable_address * 4;
    cout << "End: " << *dyn_variable_address << endl;
    delete dyn_variable_address;
}
```

- Create the pointer variable so that we have a storage place for the address of the variable we are going to create in the next line
- Allocating a new dynamic variable and initializing it to 7 and store its address for later use

- Using the dynamic variable with its memory address and the *
- When you no longer need the dynamic variable, you must deallocate it

```
Address: 0x1476010
Start: 7
End: 28
```

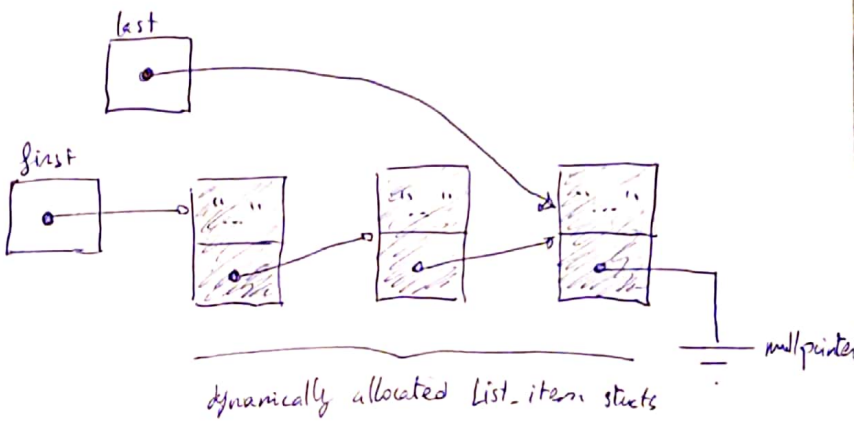
Linked list

It's a basic implementation of the dynamic data structures

They are composed of struct-type elements, each separately allocated by "new". In addition to the actual value stored in the list, each element includes a pointer, by which the next list element can be found. We also need a separate pointer-type ~~element~~ variable where we store the location of the list's first element. It is efficient to have a second point variable to store the address to the last element in the list.

If we want to store text in a list so that each line of an email is a separate element of the list we'd need:

```
struct List_item {
    string text_line;
    List_item * next;
};
List_item * first;
List_item * last;
```



When inserting or removing elements, we have to update the pointer values so that they store the correct memory addresses

A good way to implement list structures is hiding the abovementioned struct type and pointers pointing to the first and last elements of the list in the private part of the class and implementing the rest in the class methods

Function pointers

You can pass a function as a parameter to another function, pass it as a return value of a function or store it into a variable.

↳ you can do this by using function pointer

A function taking another function as its parameter or returning another function is called a high-order function. Functions that are not higher-order ones are called first-order functions.

Numeric integration

The idea is to approximate the area between the graph of a function and the x-axis by dividing the area into narrow, rectangular areas and calculating the sum of such areas

~~We calculate~~ The height of a rectangle which is defined by the value of the function at the center of the rectangle

```
double integrate ( Func f , double left, double right, int number_partitions = 500);
```

```
using Func = decltype ( &polynomial);
```

decltype reveals the data type of any variable or function

- Func f is the function f we want to integrate

```
using Func = double (*) (double);
```

Func is a pointer to a function that returns a double and also gets a double as a parameter.

The first double is the return value,
(*) means the function pointer and
within the parenthesis, there is a list of parameters

```
double integrate ( double (*f) (double), double left, double right, int number_partitions = 500);
```

```
cout << integrate ( sin, 0, 2) << endl;
..      ..      (cos, ..  ..
..      ..      (sqrt, ..  ..
..      ..      ( polynomial, ..  ..
```

Function pointers are useful for:
- We want to choose the functionality to be executed in a program but we do not want to include a very long if structure, with each block containing one of the possible functionalities
- We are implementing graphical user interfaces and want to bind a functionality to a component of the user interface

Smart pointers

Although all the actions with the dynamic memory can be implemented with C++ pointers and the commands new and delete, using them is often messy. Especially when handling complicated dynamic structures, we often end up in a situation where we allocate memory but do not remember to deallocate it.

To make this easier → smart pointers

They are library data types that automate the deallocating of memory when nothing points to it anymore. You can use them like raw pointers

`#include <memory>` →
- shared_ptr
- unique_ptr
- weak_ptr

```
#include <iostream>
#include <memory>
using namespace std;
int main(){
    shared_ptr<int> int_ptr_1(new int(1));
    shared_ptr<int> int_ptr_2(make_shared<int>(9));
    cout << *int_ptr_1 << " " << *int_ptr_2 << endl;
    cout << int_ptr_1 << " " << int_ptr_2 << endl;
    cout << int_ptr_1.use_count() << " " << int_ptr_2.use_count() << endl*2;
   *int_ptr_2 = *int_ptr_2 -4;
    int_ptr_1 = int_ptr_2;
    cout << *int_ptr_1 << " " << *int_ptr_2 << endl;
    cout << int_ptr_1 << " " << int_ptr_2 << endl;
    cout << int_ptr_1.use_count() << " " << int_ptr_2.use_count() << endl;
}
```

Prints:

```
1  9
0x2878467   0x1234567
1  1
5  5
0x2878467   0x1234567
2  2
```

1° Defining a pointer variable of the type shared_ptr, into which we can store the memory address of a int-type variable, initializing it to point to the variable that is reserved dynamically with new and that has value of 1

2° The same action as on the line above, but we use the pointer formed by the function make_shared as the initial value of the shared pointer. The difference to the previous one is that this way faster

3° shared_ptr are used mostly in the same way as normal pointers

4° Calling the method use_count() to print the value of the reference counter for both pointers. It counts the number of shared_ptr pointing to the same memory location as the target object. When this value reaches 0, the memory allocated will be deallocated automatically

5° When we set int_ptr_1 to point to the same memory address as int_ptr_2, there are no shared_ptr-type pointers pointing to the allocated memory where int_ptr_1 originally pointed to: the allocated memory is automatically deallocated

6° The lifetime of the variables int_ptr_1 and int_ptr_2 ends, which means that the memory area they pointed to has no shared_ptr: the allocated memory is automatically deallocated

*, ->, comparison, printing → work with shared_ptr

++, -- → do NOT work " "

```
shared_ptr<double> shared_double_ptr(new double);
...
double *raw_double_ptr = nullptr;
...
raw_double_ptr = shared_double_ptr.get();
```

- You cannot use = to assign a normal pointer to a shared pointer
- You can assign with =, == a nullptr value to a shared_ptr

If you want to get the memory addres from a shared_ptr

- A shared_ptr cannot be directly compared to a normal pointer

```
if (raw_pointer == shared_pointer.get()) {
   ...
}
```

shared_ptr has a troublesome characteristic : if you create a loop of them, the memory will be never deallocated:

```
#include <iostream>
#include <memory>
using namespace std;
struct Test {
  ...
  shared_ptr<Test> s_ptr;
};
int main() {
  shared_ptr<Test> ptr1(new Test);
  shared_ptr<Test> ptr2(new Test);
  ptr1->s_ptr = ptr2;
  ptr2->s_ptr = ptr1;
```

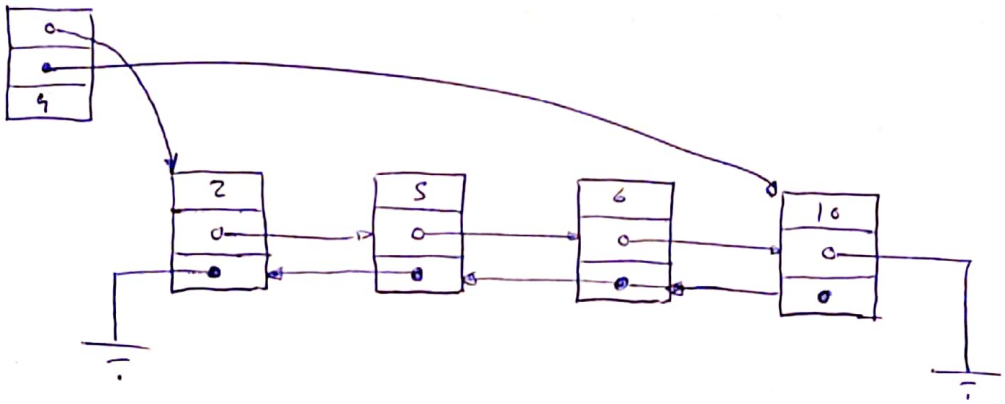
egg or chicken problem

The memory ptr1 points to cannot be deallocated because the pointer ptr2->s_ptr points to it.

Then again, the memory that ptr2 points to cannot be deallocate either because ptr1->s_ptr points to it

Doubly-linked list

Example: set of integers as a doubly-linked list in which the elements are stored in ascending order. If you attempt to add an already existing integer, the addition fails.
A set of 2, 5, 6 and 10 looks like:



- raw pointers
- shared_ptr

- The member variable pointing to the first element of the list as well as the field in each element that points to the next element are implemented with o so that you dont have to deallocated them manually
- If both pointers in the list element were shared, the memory would never be deallocated because consequent elements would point to each other

This is the weakness of shared pointers. Using it alone, you cannot implement structures that include pointer loops (A points to B, and B points to A, or a longer chain)

- The member variable pointing to the last element of the list is a normal pointer, because when you remove the last element from a non-empty list, that member variable must be set to point to the originally second-to-last element

```
last_ = last_ -> prev;
```

- You must not deallocate the normal pointer you receive from the method get with delete, because that will mess up the internal log of shared_ptr

ROUND 10

Modularity

It is a mechanism that divides a large program into small, more easily manageable parts when designing and implementing a program. If a program is modular, it has been divided into distinctly considered parts, and the result of their combined functioning.

Module is a whole consisting of cohesive programming structures. Each module is implemented as a separate source code file or a pair of source code files. They are similar to classes, they have a public and private interface. If a class forms a distinct part of the program, it is reasonable to implement it as a module

Example: a main program and a module containing geometric calculations

The main program module: calculator.cpp

```
#include "geometry.hh"
#include <iostream>
using namespace std;
int main() {
    double dimension = 0.0;
    cout << "Input the length of the side of a square";
    cin >> dimension;
    cout << "Perimeter: " << square_perimeter(dimension) << endl;
        << "Area:      " << square_area(dimension) << endl;
    cout << "Input the radius of a circle: ";
    cin >> dimension;
    cout << "Perimeter: " << circle_perimeter(dimension) << endl
        << "Area:      " << circle_area(dimension) << endl;
}
```

Each module providing services to other modules in its public interface needs a header file that describes all the services included

geometry.hh

```
#ifndef GEOMETRY_HH
#define GEOMETRY_HH
double square_perimeter(double side);
double square_area(double side);
double circle_perimeter(double radius);
double circle_area(double side);
```

geometry.cpp

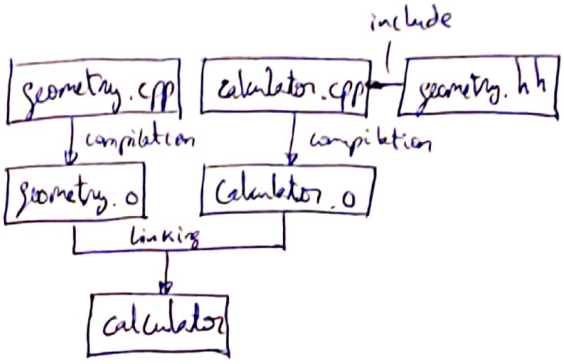
```
const double PI = 3.141593;
double square_perimeter(double side){
    return 4*side;
}
double square_area(double side){
    return side*side;
}
double circle_perimeter(double radius){
    return 2*PI*radius;
}
double circle_area(double radius){
    return PI*radius*radius;
}
```

Compilation: #.pro is not included

```
g++ calculator.cpp geometry.cpp
```

If you list all the files after the command, you can see a.out in the directory

```
./a.out
```



1° `g++ -c geometry.cpp`

2° `g++ -c calculator.cpp`

3° `g++ calculator.o geometry.o`

If you want to name the binary something other than a.out

4° `g++ -o calculator calculator.o geometry.o`

Module's public interface (hh file)

The services an module provides for other modules are written in the header file

It can contain:
- declarations of functions
- definitions of constants
- definitions of new data types (also classes)
- or any combination of the above

It should NOT contain:
- definitions of the variables
- definitions of functions or class methods

Each source code file that needs a service from the public interface of another module has to contain #include "module_proving_services.hh"
You must NOT use #include for including .cpp files

```
#include "my_module_1.hh"
...
#include "my_module_n.hh"
#include <standard_library_1>
...
#include <standard_library_m>
```

Module's private interface (.cpp file)

It defines all the functions and methods that were declared in the module's public interface in the header file

```
namespace { // declaration part
    void private_function_of_the_module();
    ...
}
...
the function definitions of the public interface here
...
namespace { // definition part
    void private_function_of_the_module() {
        ...
    }
...
}
```

```
namespace my_space {
    void my_function() {
        ...
    }
}
```

```
my_space::my_function();
```

```
using namespace my_space;
...
my_function
```

~~How to design modules~~ → Benefits of modularity

Mostly the same as classes because they both stem from the use of interfaces:

- The implementation of a module (that is, the .cpp i.e. the private interface) can be modified while the public interface stays intact
- The parts of the program that logically belong together can be combined in the same package, which simplifies the package
- The modules can be developed in the project side by side after agreeing on the public interface
- Modularity is a good tool in managing large programming projects
- You can reuse whole or partial modules
- It is possible to compile modules seperately. This speeds up the developing and uses less resources, since after completing the changes, you only need to re-compile the modules that were modified.

Abstract data types

They are defined by its possible values and operations handling these values
The type can be used by means of the provided operations

Information hiding and encapsulation

Information hiding means hiding the details of the implementation or a limited access to them. The public interface reveals the names of the operation with a comment describing its purpose, the parameters and the return value of the operation. A user is able to use a library function without knowing its implementation

Encapsulation means that the data and the operations processing the data are bound or packed or wrapped together. E.g. with the stack implementation we have a stack class and the public methods of the stack class. A class (like a module) enables encapsulation

Programming style

- The object or module that have allocated memory, is principally responsible to deallocate memory too.
- If a delete command is targeted to an assignable pointer variable, it will be assigned to the value nullptr immediately after the delete command
- A destructor must deallocate all the resources that have been allocated by the object in question
- Unnecessary copy destructor and assignment operator must be disable by introducing them in the public part of the class and using the word delete. ~~as explained~~