Syntax (from C)

Hello World

```c
// helloworld.c
#include <stdio.h>

int main()
{
  printf("Hello World!\n");
  return 0;
}
```

```cpp
// helloworld.cpp
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

- Similar syntax to C
- OOP (also)
  - classes
  - namespaces
  - templates (generic prog.)
- `std::` namespace, streams, containers later...

Syntax I.

Declaration & initialization

```
int i = 1;
char c = '+';
bool b = 3; // we have bool too
std::string s = "apple";

int iarr[] = {1, 2, 3};
char carr[4] = {'c', 'p', 'p', '\0'};
float farr[10];

int *ip = &i;
float *fp = farr;
void *p = (void *)carr;
```

Syntax II.

for & while

```
for (int i = 0; i < 5; i++)
{
  std::cout << i << " ";
}
```

```
int i = 0;
while (i < 5)
{
  std::cout << i << ' ';
  i++;
}
```

```
0 1 2 3 4
```

Syntax III.

Branching

```
if (2 + 2 < 5)
{
  std::cout << "sometimes" << std::endl;
}
```

```
switch (2 + 2)
{
case 4:
  std::cout << "hello "; // fall through
case 5:
  std::cout << "world" << std::endl;
  break;
default:
  std::cout << "!" << std::endl;
} // hello world
```

Syntax IV.

Functions

```
float add(float a, float b); // forward declaration

int add(int a, int b) { return a + b; }

float add(float a, float b) { return a + b; }
```

```
std::cout << add(1, 2) << std::endl
          << add(1.f, 2.f) << std::endl;

add(1, 2.f); // error
```

```
3
3
```

- Our knowledge gained in C can be transferred to C++
- There are differences, the following code is valid only in C:

```
int i = 5;
char *p = &i;
```

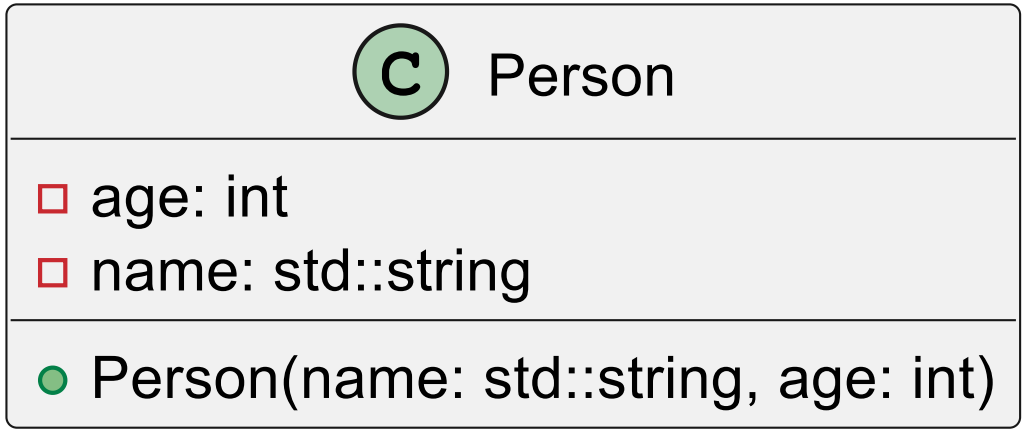
Class & Struct

Classes I.

```
class Person
{
private:
  int age;
  std::string name;

public:
  Person(std::string name, int age)
  {
    this->name = name;
    this->age = age;
  }
}; // ; is important
```

```
Person teacher; // error
Person student("Lajos", 19);
```



- Notice that we did not write an empty/default constructor, so we cannot use it.

Classes II.

```
class MyClass
{
  // By default all member variables, methods, and constructors are private
public:
  // After the public keyword, all member variables, methods, and constructors
  // will be public
private:
  // Everything here will be private
protected:
  // Everything here will be protected
public:
  // It is not recommended, but you can repeat "labels"
};
```

Classes III.

- Initializing member variables in the constructor:

```
class MyClass
{
  int num;
  float delta;

  MyClass() : num(1), delta(1.f) { /* some code */ };
  MyClass(int num) : num(num), delta(num) { /* some code */ };
  MyClass(int num, float d) : num(num), delta(d)
  {
    // some code
    int a = 2;
  };
};
```

Classes IV.

```
class MyClass
{
public:
  MyClass()
  {
    /* When the object is created, the appropriate constructor is executed first.
        Only after that can member variables and methods be referenced. */
  }

  ~MyClass()
  {
    /* We can also define a destructor, which is executed at the end of the object's lifetime.
        If we have allocated memory, it should be freed here. */
  }
};
```

Struct I.

- In C, we have already seen `struct` -s:

```
struct MyStruct
{
  int num = 42;
  float f; // uninitialized
};
```

```
MyStruct ms;
MyStruct ms2 = {1, 2};
std::cout << ms.num << std::endl
          << ms2.f << std::endl;
```

```
42
2
```

Struct II.

- In C++, `struct`-s are "real" classes: they can have methods, constructors
- In a `struct` everything is **public** by default, while in a `class`, everything is **private**

```
struct MyStruct
{
  int num = 42;
  float f; // uninitialized
};
```

```
class MyStructClass
{
  int num = 42;
  float f; // uninitialized
};
```

```
MyStruct ms;
MyStructClass msc; // in this case we have default constructor

MyStruct ms2 = {1, 2};
MyStructClass msc2 = {1, 2}; // error
```

Include & Macros

Include I.

- The `#include` directive is already known from C
- It is similar to an import, but the preprocessor simply inserts the content of the file
- The file extension doesn't matter (usually `.h`, `.hpp`)

```
// funnyheader.hpp
std::cout << "Hello World!" << std::endl;
```

```
#include <iostream>

int main()
{
#include "funnyheader.hpp"
  return 0;
}
```

Include II.

- Some C++ header

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

- C headers are also available

```
// it will be globally accessible!
#include <stdio.h>
#include <time.h>

// it will be accessible in the std:: namespace
#include <cstdio>
#include <ctime>
```

Include III.

- Avoid multiple inclusion:

```
// funnyheader.hpp
std::cout << "Hello World!" << std::endl;
```

```
Hello World!
Hello World!
```

```
// funnyheader.hpp
#pragma once
std::cout << "Hello World!" << std::endl;
```

```
Hello World!
```

```
#include <iostream>

int main()
{
#include "funnyheader.hpp"
#include "funnyheader.hpp"
  return 0;
}
```

Include IV.

- It is recommended to write only declarations in header files, and to place function definitions in separate `.cpp` files
    - Since we don't include `.cpp` files, there's no need for `#pragma once` in them
- The `.cpp` files are compiled individually into object files (`.o`) and then linked
- Since we're using an IDE in the graphics course, we don't need to worry about the linker

Include IV.

```cpp
// myclass.h
#pragma once

class MyClass
{
  int value;

public:
  MyClass() : value(5) {}
  int getValue();
};
```

```cpp
// myclass.cpp
#include "myclass.h"

int MyClass::getValue()
{
  // you can access private members here
  return value;
}
```

```cpp
// main.cpp
#include <iostream>
#include "myclass.h"

int main()
{
  MyClass obj;
  std::cout << obj.getValue() << std::endl;
  return 0;
}
```

```
g++ main.cpp myclass.cpp -o main
```

```
5
```

Macro I.

- Also known preprocessor directive from C

```
#define MAX_STEP 5
#define VAR i

for (int i = 0; i < MAX_STEP; i++)
{
  std::cout << VAR << " ";
}
```

```
0 1 2 3 4
```

Macro II.

```
#define ADD(a, b) a + b

std::cout << ADD(1, 2);     // 3 == 1 + 2
std::cout << ADD(1, 2) * 2; // 5 == 1 + 2 * 2
```

```
35
```

```
#define DOUBLE(a) ((a) + (a))

int a = 1;
std::cout << DOUBLE(a);   // 2 == 1 + 1
std::cout << DOUBLE(++a); // 5 == 2 + 3
```

```
25
```

Macro III.

- Conditional compilation

```
#define DEBUG

#ifdef DEBUG
std::cout << "print something" << std::endl;
#endif
```

```
#define ALGORITHM 2

#if ALGORITHM == 0
  std::cout << "case 0" << std::endl;
#elif ALGORITHM < 3
  std::cout << "case <3" << std::endl;
#else
  std::cout << "case else" << std::endl;
#endif
```

Templates

Template I.

- It is similar to the generics known from C#

```
template <typename T>
class Adder
{
public:
  T add(T a, T b) { return a + b; }
};
```

```
Adder<float> floatAdder;
std::cout << floatAdder.add(1.f, 2.f) << std::endl
          << floatAdder.add(1, 2) << std::endl;
```

```
3
3
```

Template II.

- There are template functions too:

```
template <typename T>
void print(T s)
{
  std::cout << s << std::endl;
}

// explicit instantiation
template void print<double>(double);
template void print(float); // type deduction
```

```
print<int>(1); // implicit instantiation
print(2.f);
```

```
1
2
```

Standard Library

Vector I.

- Dynamic array in C++:

```
#include <vector>
```

```
std::vector<int> ivec;
ivec.push_back(0);
ivec.push_back(1);
ivec.push_back(2);
ivec.push_back(3);

for (int i = 0; i < 4; i++)
{
  std::cout << ivec[i] << " ";
}
```

```
0 1 2 3
```

Vector II.

- STL iterator:

```
// std::vector<int>::iterator
for (auto i = ivec.begin(); i != ivec.end(); i++)
{
  std::cout << *i << " ";
}
```

- for each loop:

```
for (const int i : ivec)
{
  std::cout << i << " ";
}
```

Vector III.

- Initialization

```
std::vector<float> fvec;                  // empty
std::cout << fvec.empty() << std::endl; // 1 == true

std::vector<double> dvec(4);              // 4 elements
std::cout << dvec.size() << std::endl; // 4

std::vector<int> ivec(3, 2);              // 3 elements initialized to 2
std::cout << ivec.front() << ivec.at(1) << ivec.back() << std::endl; // 222
```

```
ivec.pop_back();
std::cout << ivec.size() << std::endl; // 2

ivec.clear();
std::cout << ivec.size() << std::endl; // 0
```

Vector IV.

```
std::vector<int> ivec = {1, 2, 3};
std::vector<int> ivec_copy(ivec); // deep copy

ivec[1] = 0;
ivec.clear();

std::cout << ivec_copy[1] << std::endl; // 2
```

- The most used data structure in computer graphics course...

std::

- set
- map
- stack
- queue
- list
- *string*
- **algorithm**

Algorithm I.

```
#include <algorithm>
```

- Sorting:

```
std::vector<int> ivec = {8, 3, 4, 2, 1, 5, 6, 7, 9};

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

for (const int i : ivec)
{
  std::cout << i << " ";
}
```

```
1 2 3 4 5 6 7 8 9
```

Algorithm II.

- Searching:

```
std::vector<int> ivec = {8, 3, 4, 2, 1, 5, 6, 7, 9};

// std::vector<int>::iterator
auto it = std::find(ivec.begin(), ivec.end(), 2);

while (it != ivec.end())
{
  std::cout << *(it++) << " ";
}
```

```
2 1 5 6 7 9
```

Stack & Heap, Lifetime

Stack vs Heap I.

- The following all allocate on the stack:

```
int i;
float f = 4.f;
double d[10];
MyStruct s;
MyClass c;
```

- The following allocate on the heap:

```
int *i = new int[10];
MyStruct *s = new MyStruct;
MyClass *c = new MyClass;
```

Stack vs Heap II.

In the latter case, a pointer is stored on the stack, which points to the data stored on the heap.

`std::vector` stores its data on the heap, so we do not allocate them on the heap separately.

It is important to remember that data stored on the stack is automatically freed when its lifetime ends, while memory allocated on the heap must be manually freed by us.

Lifetime I.

```
class MyClass
{
public:
  ~MyClass()
  {
    std::cout << "destructor ";
  }
};
```

```
{
  std::cout << "start ";
  MyClass c;
  std::cout << "middle ";
} // here c is freed (from the stack)
std::cout << "end" << std::endl; //start middle destructor end
```

Lifetime II.

```
{
  std::cout << "start ";
  MyClass *c = new MyClass;
  std::cout << "middle ";
}
std::cout << "end" << std::endl; // start middle end
```

```
{
  std::cout << "start ";
  MyClass *c = new MyClass;
  std::cout << "middle ";
  delete c;
}
std::cout << "end" << std::endl; // start middle destructor end
```

Lifetime III.

- If we allocated an array:

```
float *farr = new float[5]{1, 2, 3, 4, 5};
delete[] farr;
```

Pointers & references

Pointer I.

- Pointers are already familiar from C:

```
int i = 5;
int *ip = &i;
std::cout << ip << std::endl
          << *ip << std::endl;
void *p;
std::cout << p << std::endl;
```

```
0x61ff04
5
0x61ff50
```

Pointer II.

- Changing a variable's value through a pointer:

```
void swap(int *a, int *b)
{
  int a_value = *a;
  *a = *b;
  *b = a_value;
}
```

```
int a = 2, b = 4;

swap(&a, &b);

std::cout << a << b << std::endl; // 42
```

Reference I.

```
int i = 5;
int &ri = i;

i++;
std::cout << ri << std::endl; // 6

ri++;
std::cout << i << std::endl; // 7
```

- There is no null reference equivalent to `nullptr`
- After initialization, it cannot point to anything else

```
std::cout << &i << std::endl   // 0x61ff08
          << &ri << std::endl; // 0x61ff08
```

Reference II.

```
void swap(int &a, int &b)
{
  int a_value = a;
  a = b;
  b = a_value;
}
```

```
int a = 2, b = 4;

swap(a, b);

std::cout << a << b << std::endl; // 42
```

Reference III.

```
const int a = 2, b = 4;

swap(a, b); // error
```

Avoid explicit casting, if a function does not modify a parameter's value, mark it as `const`, so it can be passed without issues:

```
void constfn(const int &a) {};

const int a = 2;
int b = 4;

constfn(a);
constfn(b); // it also works
constfn(6); // fine too
```

Pointer Arithmetics I.

We might already be familiar with this from C. Consider two pointers, assume both pointers point to any two elements of the same array, and also assume that the type of the pointers matches the type of the elements stored in the array.

In this case, we can define operations...

Pointer Arithmetics II.

- Step forward/backward in the array (the pointer type determines where the next element's memory address is):

```
int a[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

int *p = a;
p++;

std::cout << (*p);                  // 2
std::cout << (*(++p));              // 4
std::cout << (*(--p));              // 2
std::cout << *(p + 3) << std::endl; // 8
std::cout << p << " " << p + 1 << " " << sizeof(int) << std::endl;
//0x61fee4 0x61fee8 4
```

Pointer Arithmetics III.

- Distance between two elements (distance between memory addresses / size of one element)

```
int a[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

int *p = a;
int *s = &a[6];

std::cout << s - p << std::endl; // 6
```

Pointer Arithmetics IV.

- Comparing two pointers (it can be determined from the memory addresses, whether they point to the same element or which one points further ahead)

```
int a[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

int *p = a;
int *s = &a[4];

std::cout << (s < p) << " "
          << (s == p) << " "
          << (s > p) << std::endl; // 0 0 1
std::cout << (p == p) << " "
          << (s != p + 4) << " "
          << (p == nullptr) << std::endl; // 1 0 0
```

Random numbers

PRNG

- Pseudorandom Number Generator
- Needs to be seeded: with the same seed, it generates the same numbers (deterministic)
- `std::random_device` *can* generate true random numbers, but it's slow, good for seeding a PRNG
- It's enough to choose one, e.g.: Mersenne Twister, Linear Congruential Generator (weak), RANLUX

Random integer generation

```cpp
#include <iostream>
#include <random>

int main()
{
    // Uniform distribution in range [0, 9]
    std::uniform_int_distribution<int> dist(0, 9);

    std::random_device rd; // Class to get a seed number
    std::mt19937 gen(rd()); // PRNG

    for (int i = 0; i < 10; i++)
    {
        std::cout << dist(gen) << " "; // 2 8 1 8 8 4 3 9 0 8
    }
}
```

Random float generation

```cpp
#include <iostream>
#include <random>

int main()
{
    // Uniform distribution in range [0, 4)
    std::uniform_real_distribution<> dist(0.0, 4.0);

    std::random_device rd; // Class to get a seed number
    std::mt19937 gen(rd()); // PRNG

    for (int i = 0; i < 4; i++)
    {
        std::cout << dist(gen) << " "; // 1.36674 3.01534 2.17733 3.98481
    }
}
```

Normal distribution

```cpp
#include <iostream>
#include <random>

int main()
{
    // normal distribution - mean=5, stddev=2
    std::normal_distribution<float> dist(5, 2);

    std::random_device rd; // Class to get a seed number
    std::mt19937 gen(rd()); // PRNG

    for (int i = 0; i < 10; i++)
    {
        std::cout << (int)dist(gen) << " "; // 6 5 3 4 4 4 5 5 3 4
    }
}
```