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;
}
```

- C-hez hasonló szintaxis
- Objektumelvű programozás (is)
  - osztályok
  - névterek
  - templatek (generikus prog.)
- `std::` namespace, streamek, konténerek magyarázata később...

Nyelvi elemek I.

Deklaráció és inicializáció

```
int i = 1;
char c = '+';
bool b = 3; // van bool is
std::string s = "alma";

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;
```

Nyelvi elemek 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
```

Nyelvi elemek III.

Elágazások

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

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

Nyelvi elemek IV.

Függvények

```
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
```

- A C-ben megszerzett ismereteinket átültethetjük C++ba
- Vannak eltérések, az alábbi kód csak C-ben valid:

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

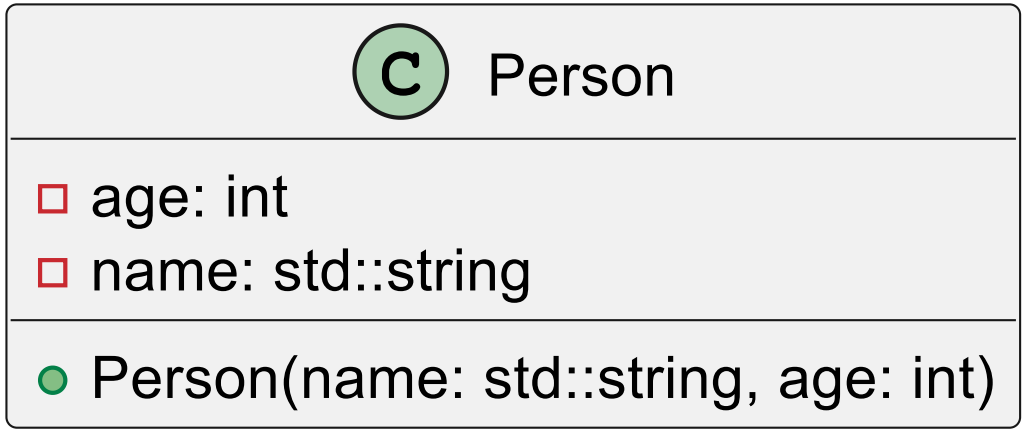
Class & Struct

Osztályok 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);
```



- Figyeljük meg, hogy nem írtunk üres/default konstruktort, így nem is használhatjuk

Osztályok II.

```
class MyClass
{
  // alapvetően minden adattag(member), metódus, konstruktor privát
public:
  // public kulcsszó után minden adattag, metódus, konstruktor
  // publikus lesz
private:
  // itt minden privát lesz
protected:
  // itt minden protected
public:
  // nem ajánlott, de lehet ismételni a "címkéket"
};
```

Osztályok III.

- Adattagok inicializálása konstruktorban:

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

  MyClass() : num(1), delta(1.f) { /* itt még lehet kód */ };
  MyClass(int num) : num(num), delta(num) { /* kód */ };
  MyClass(int num, float d) : num(num), delta(d)
  {
    // itt is lehet még kód
    int a = 2;
  };
};
```

Osztályok IV.

```
class MyClass
{
public:
  MyClass()
  {
    /* Az objektum létrehozásakor a megfelelő konstruktor fut le mindenek előtt.
        Ezután történhet csak az adattagok, metódusok hivatkozása. */
  }

  ~MyClass()
  {
    /* Destruktort is megadhatunk, ez az objektum élettartamának a végén fut le.
        Ha memóriát foglaltunk le, azt itt kell felszabadítanunk. */
  }
};
```

Struct I.

- C-ben már láttuk a `struct` -okat:

```
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.

- C++-ban a `struct` -ok teljesértékű osztályok: lehet metódusuk, konstruktoruk
- `struct` esetén alapvetően minden **publikus**, míg `class` esetén **privát**

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

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

```
MyStruct ms;
MyStructClass msc; // ilyenkor van default konstruktor

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

Include & Macros

Include I.

- C-ből már ismert a `#include` direktíva
- Hasonlít az importhoz, de csak a preprocesszor bemásolja a fájl tartalmát
- Nem számít a fájl kiterjesztése (többnyire `.h`, `.hpp`)

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

```
#include <iostream>

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

Include II.

- Néhány C++ header

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

- C-s headerek is elérhetőek

```
// globálisan elérhető lesz!
#include <stdio.h>
#include <time.h>

// std:: namespace-ben lesz elérhető
#include <cstdio>
#include <ctime>
```

Include III.

- Figyeljünk a többszörös include-ok elkerülésére

```
// 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.

- Header fájlokba ajánlott csak deklarációkat írni, a függvénydefiníciókat írjuk külön `.cpp` fájlokba
  - `.cpp` fájlokat nem include-olunk, ezért ezekben nem szükséges `#pragma once`
- A `.cpp` fájlok egyessével kerülnek fordításra (`.o` fájlok), majd linkelésre
- Mivel a grafika tárgyon IDE-t használunk, nem kell foglalkoznunk a linkerrel

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()
{
  // itt eléred az osztály privát tagjait
  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.

- Már szintén ismert preprocesszor direktíva C-ből

```
#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.

- Feltételes fordítás

```
#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.

- Hasonlít a C#-ból már ismert generikusokhoz

```
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.

- Template függvények is lehetnek:

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

// explicit példányosítás
template void print<double>(double);
template void print(float); // típus dedukció
```

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

```
1
2
```

Standard Library

Vector I.

- Dinamikus tömb C++-ban:

```
#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 iterátor:

```
// 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.

- Inicializálás

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

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

std::vector<int> ivec(3, 2);              // 3 elem 2-re inicializálva
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.

```cpp
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
```

- Grafika tárgyon legtöbbet használt...

std::

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

Algorithm I.

```
#include <algorithm>
```

- Rendezés:

```
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.

- Keresés:

```
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.

- Az alábbiak mind a stacken allokálnak:

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

- Az alábbiak pedig a heapen:

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

Stack vs Heap II.

Meg kell jegyezni, hogy az utóbbi esetben a stacken tárolódik egy pointer, ami a heapen tárolt adatokra mutat.

A `std::vector` alapvetően a heapen tárolja az adatokat, így nem allokáljuk külön a heapen.

Fontos megjegyezni, hogy a stacken tárolt adatok az élettartam végeztével automatikusan felszabadulnak, míg a heapen lefoglalt memóriát nekünk kell felszabadítani manuálisan.

Lifetime I.

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

```
{
  std::cout << "start ";
  MyClass c;
  std::cout << "middle ";
} // itt c felszabadítódik (a stackről)
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.

- Ha tömböt foglaltunk le:

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

Pointers & references

Pointer I.

- C-ből már ismertek a pointerek:

```
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.

- Változó értékének módosítása pointeren keresztül:

```
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
```

- Nincs `nullptr` -nek megfelelő null referencia
- Inicializálás után nem tud másra mutatni

```
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
```

Kerüljük az explicit kasztolást, ha egy függvény nem változtat egy paraméter értékén, jelöljük `const` -nak, így gond nélkül átadhatjuk:

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

const int a = 2;
int b = 4;

constfn(a);
constfn(b); // ez még mindig működik
constfn(6); // ez is
```

Pointer Arithmetics I.

C-ből már ismerhetjük. Tekintsünk két pointert, tegyük fel, hogy mindkettő pointer ugyanazon tömb 2 tetszőleges elemére mutatnak, valamint tegyük fel, hogy a pointerek típusa megegyezik a tömbben tárolt elemek típusával.

Ekkor értelemszerűen értelmezhetünk műveleteket...

Pointer Arithmetics II.

- Lépés előre/hátra a tömbben (pointer típusából kikövetkeztethető, hol van a következő elem memóriacíme):

```
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.

- Két elem távolsága (memóriacímek távolsága / egy elem mérete)

```
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.

- Két pointer összehasonlítása (memóriacímből eldönthető, ugyanarra az elemre mutatnak-e, melyik mutat előrébb)

```
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

- Pszeudo véletlen számokat generál
- Szükséges seedelni: ugyanazzal a seeddel, ugyanazokat a számokat generálja (determinisztikus)
- `std::random_device` *képes* valódi véletlent generálni, de lassú, ezért PRNG seedeléshez jó
- Elég választani egyet, pl.: Mersenne Twister, Lineáris Kongruencia Generátor (gyenge), 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
    }
}
```