C++ ismétlés

ADATSZERKEZETEK ÉS ALGORITMUSOK
1. GYAKORLAT

Fontos feladat

Kérjük, hogy az alábbi linken elérhető formot mindenki egyszer töltse ki!

- https://goo.gl/gnO1hH
- Azért van erre szükség, hogy a félév során kötelezően házi feladatok beadásához szükséges repository elkészüljön!

Kitöltési határidő:

- Szeptember 12, szombat 24.00

Repositoryk elkészülnek

- Szeptember 14, hétfő 24.00
  - Erről lesz tájékoztatás

Előszó – követelmények

Részvétel a gyakorlatokon kötelező
- Legfeljebb három hiányzás, amelyről nem szükséges semmilyen igazolás

Minden gyakorlat elején kis zárthelyi dolgozat
- Az értékelésük százalékosan történik
- A félév végén a három legrosszabbat nem számítva minimum 60%

Minden héten kis házi feladat
- A feladatok célja a tanultak gyakorlása és ennek bizonyítása, ezért:
  - 1 pont jól megoldott házi feladat
  - 0 pont, ha nem sikerült az újonnan tanultakat implementálni, de a feladattal érdemben foglalkoztál, és nem tartalmaz súlyos hibákat
  - -1 pont, ha valaki nem ad be megoldást, vagy a beadott megoldás értékelhetetlen
    - Segmentation fault
    - Nem fordul a megadott kapcsolókkal
      - Részletek a wiki oldalon
    - Súlyos elvi, vagy technikai hiba van a kódban
  - -2 pont, másolás esetén minden résztvevőnek
    - Visszaesőknek az aláírást megtagadjuk
- A félév végén minimum 3 pont elérendő.
  - A legrosszabb nem számít (a másolásért kapott büntetőpont mindenképpen beleszámít)
- Első zh-ig 0 pontot el kell érni legalább
  - Akinek nincsen meg, az nem jöhet megírni az első zh-t

Előszó – követelmények folytatása

Házi beadás módja és határideje
- A házi feladását követő hét kedd, délelőtt 11.00
- Kijavítás a beadást követő második hét végéig
- Beadás az SVN repositoryba

Féléves házi
- Másolás esetén 0% minden érintettnek
- Átlaguk beleszámít a féléves munkába
- Minimum szint 40%-ot kell elérni
- Az első zh után kell beadni – pontosítjuk a házi kiírásakor

Két zárthelyi dolgozat
- A őszi szünet utáni gyakorlaton egy papíros ZH (november 3., kedd)
- Gyakorlati ZH a szorgalmi időszakot követő héten (december 15., kedd)
- Minimum szint 50%, mindegyik esetén
- Pót ZH-k: december 18., péntek

A jegy számításának módja
- első ZH: 35%
- második ZH: 40%
- féléves házi: 20%
- féléves munka: 10% (óra eleji zárthelyik és a heti házifeladatok, órai aktivitás)
- végeredmény: 50% az elégséges

Előszó – követelmények folytatása

Jegyhatárok:

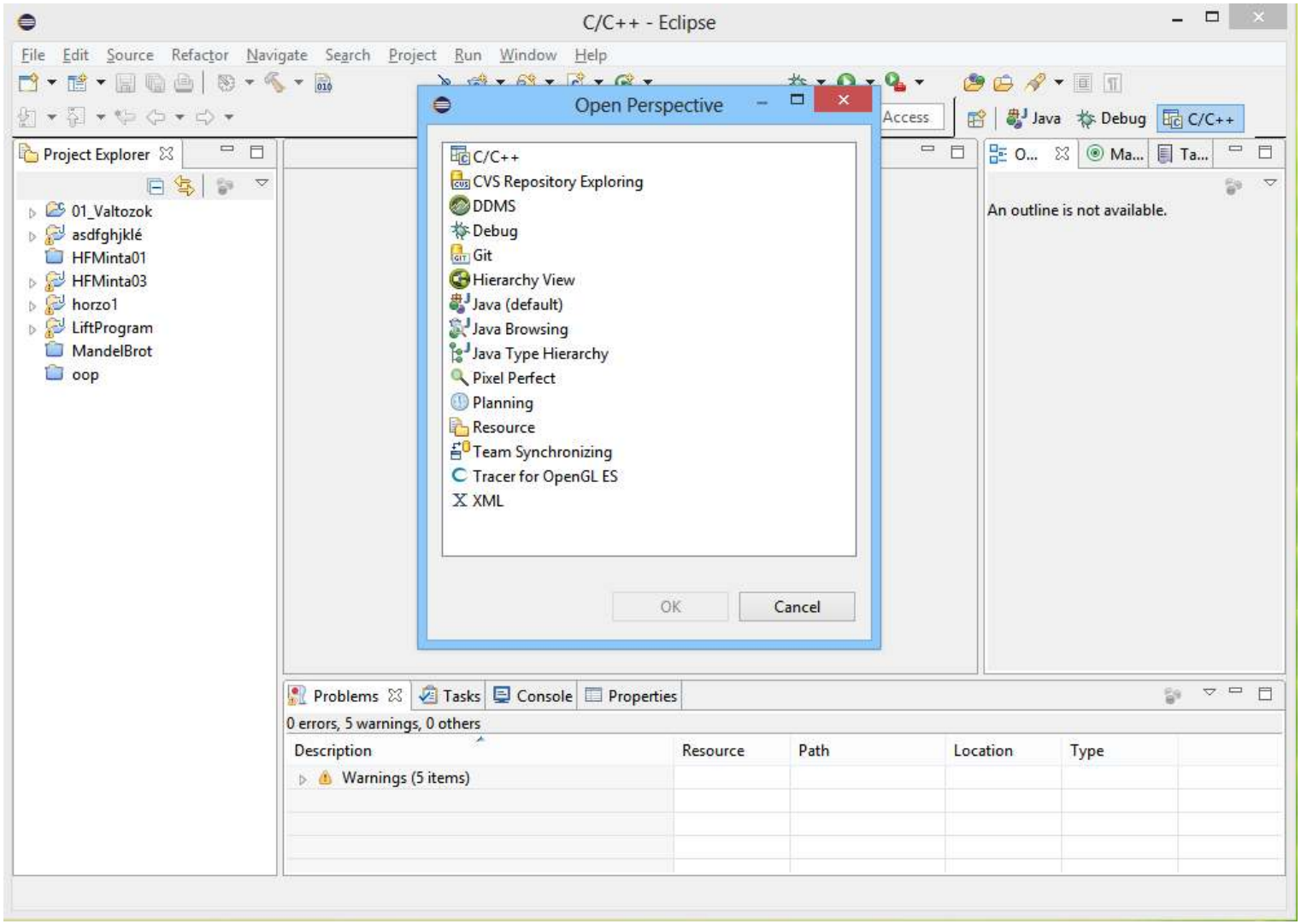
- 0% - 49%
- 50% - 65%
- 66% - 75%
- 76% - 85%
- 86% - 100%

Eclipse C++

Amit a félév során használni fogunk:

- Eclipse fejlesztőkörnyezet (Mars) 64 bit
  - Fontos, hogy 64 bites Java kell hozzá
  - Ha valakinek nem ez van otthon, akkor külön le kell töltenie a www.eclipse.org oldalról
    - https://www.eclipse.org/downloads/
    - Eclipse IDE for Java Developers
      - Emellé le kell tölteni a CDT-t, lásd lentebb
    - Eclipse IDE for C/C++ Developers
- CDT (C++ Development Tool)
  - Meglevő Eclipse mellé
    - Eclipseben telepíteni:
      - Help | Install new software
      - Available software sites – hozzá kell a következő repository-t:
        - http://download.eclipse.org/tools/cdt/releases/8.7
      - Ezt követően lehet telepíteni a CDT eszközeit
- MinGW (+ Code::Blocks)
  - http://sourceforge.net/projects/codeblocks/files/Binaries/13.12/Windows/codeblocks-13.12mingw-setup.exe/download

Eclipse C/C++ Perspective



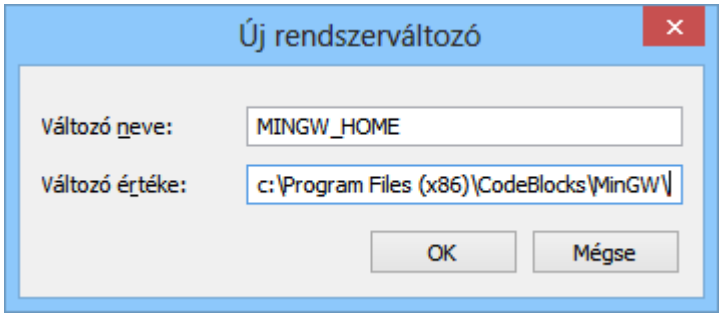
Windows beállítás

Mivel a géptermekben Windows van, itt ezt írjuk le

- Ha valakinek szüksége van egyéb OS-en segítségre, keressen meg minket

Meg kell adni a MinGW fordító útvonalát

- Vezérlőpult | Rendszer | Speciális rendszerbeállítások | Speciális | Környezeti változók
- Rendszerváltozók | Új…



- Ide természetesen az az útvonal kell, ahol a MinGW található
  - Ez elvileg a Code::Blocks-on belül, ami a Program Files-ban található.

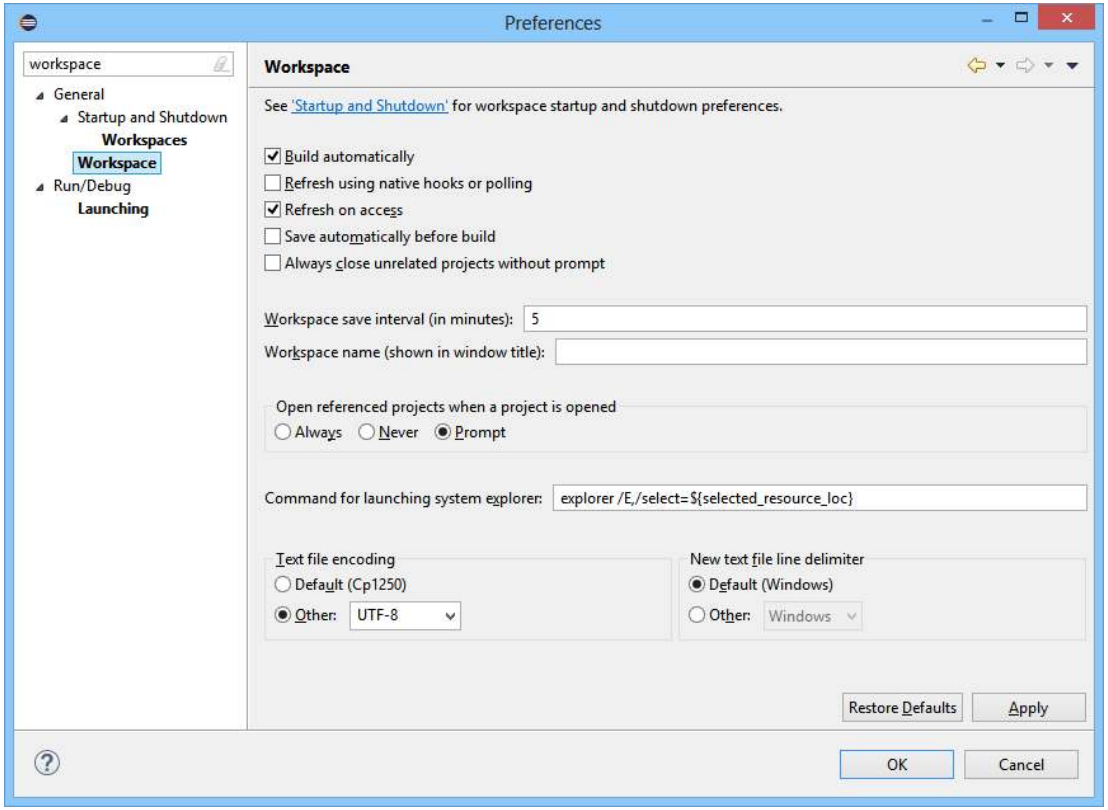
Workspace

Ide kerül minden projekt

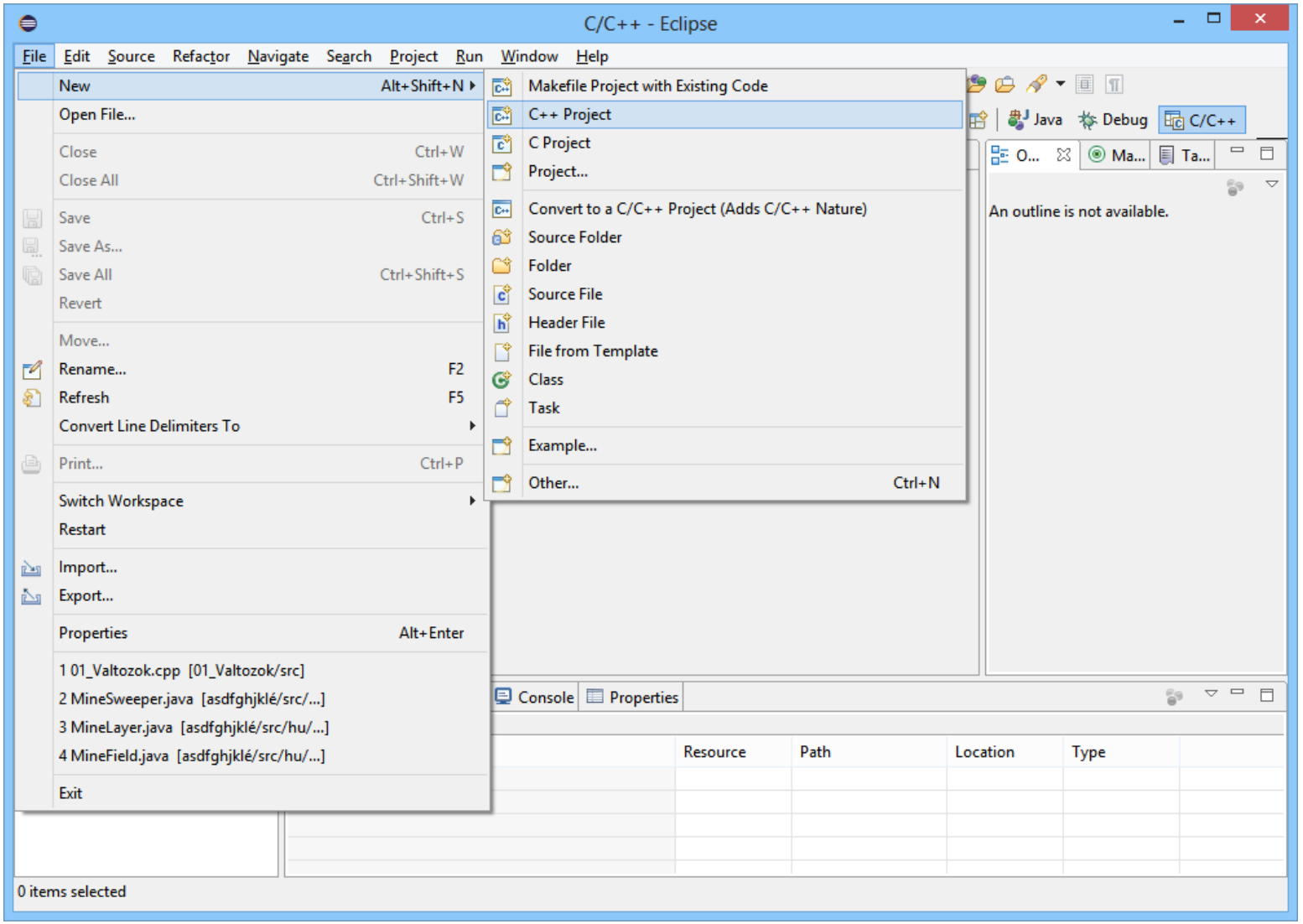
- Ezt a egyetemi gépen a C:\Users\<digitus>\workspace helyre érdemes tenni
  - Idéntől nincsen hálózati meghajtó
- Otthon tetszőleges helyre

Beállítások

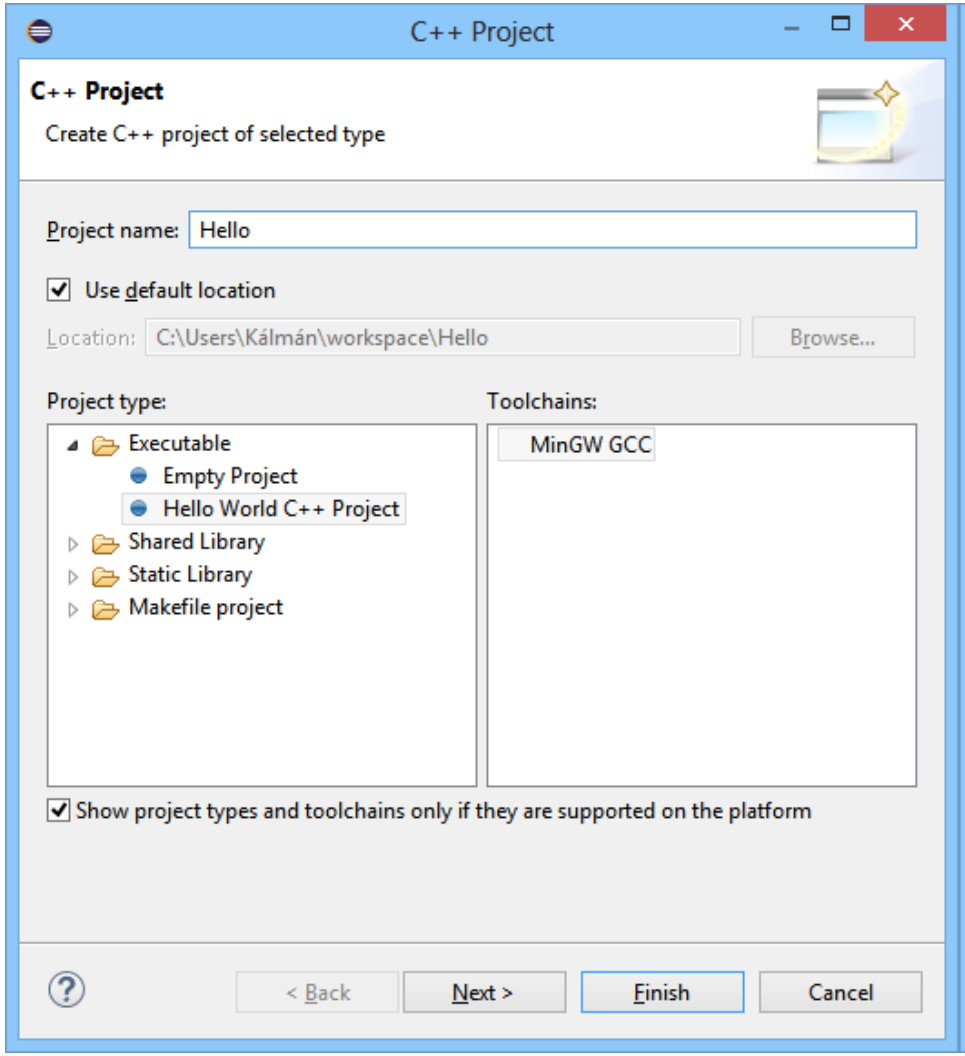
- UTF-8!

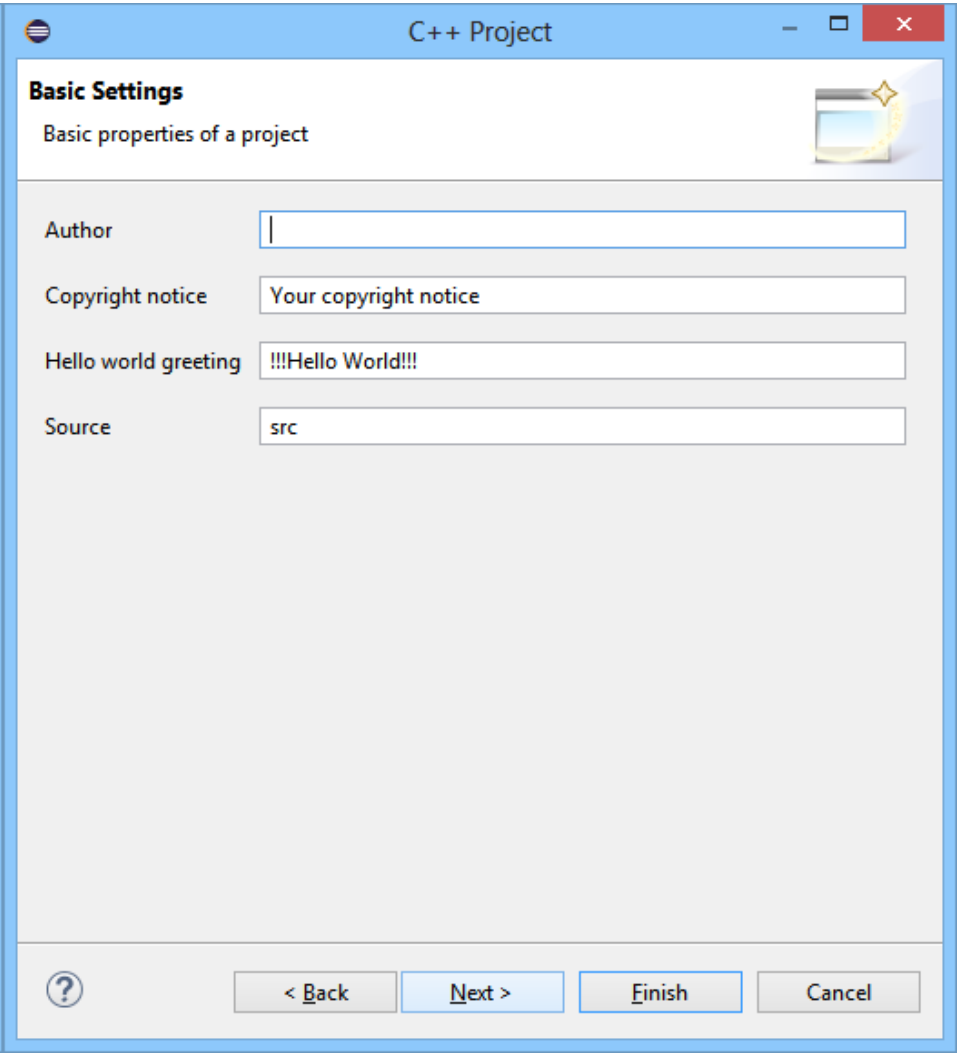


Új projekt

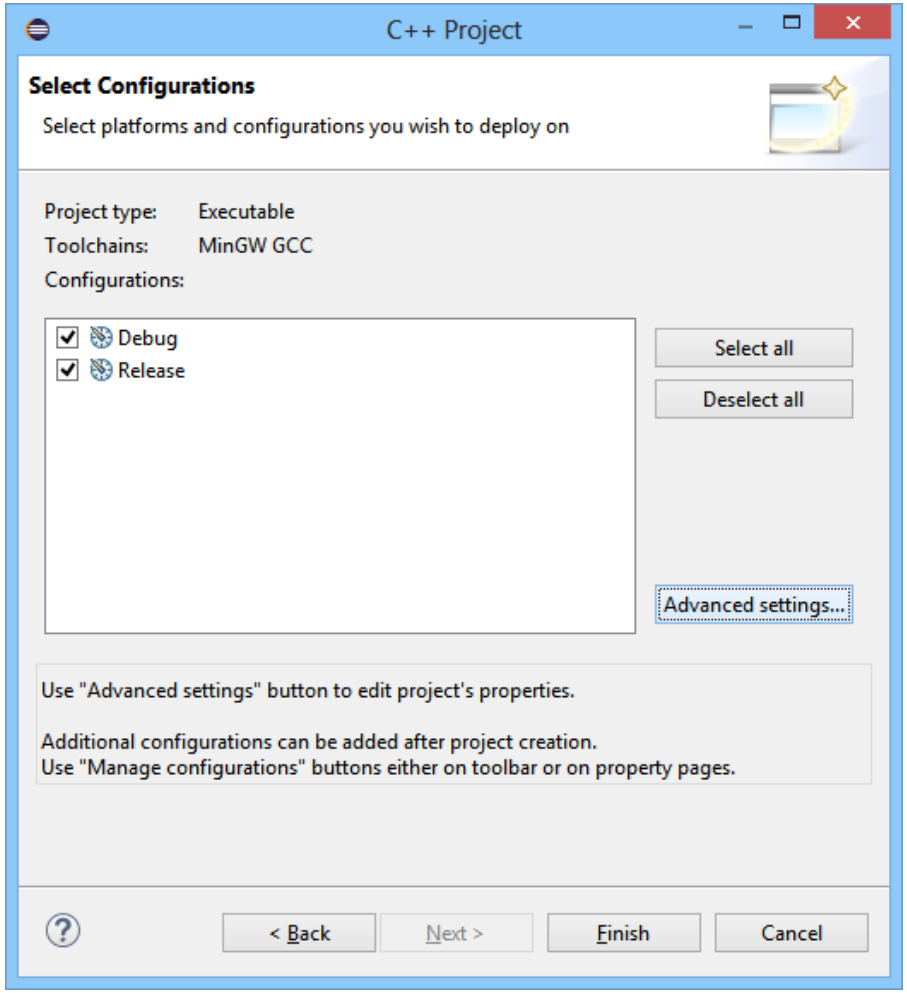

C/C++ - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
New Alt+Shift+N
Open File...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Properties Alt+Enter
1 01_Valtozok.cpp [01_Valtozok/src]
2 MineSweeper.java [asdfghjklé/src/...]
3 MineLayer.java [asdfghjklé/src/hu/...]
4 MineField.java [asdfghjklé/src/hu/...]
Exit
Makefile Project with Existing Code
C++ Project
C Project
Project...
Convert to a C/C++ Project (Adds C/C++ Nature)
Source Folder
Folder
Source File
Header File
File from Template
Class
Task
Example...
Other... Ctrl+N
Java
Debug
C/C++
An outline is not available.
Console
Properties
Resource
Path
Location
Type
0 items selected

C++ Projekt

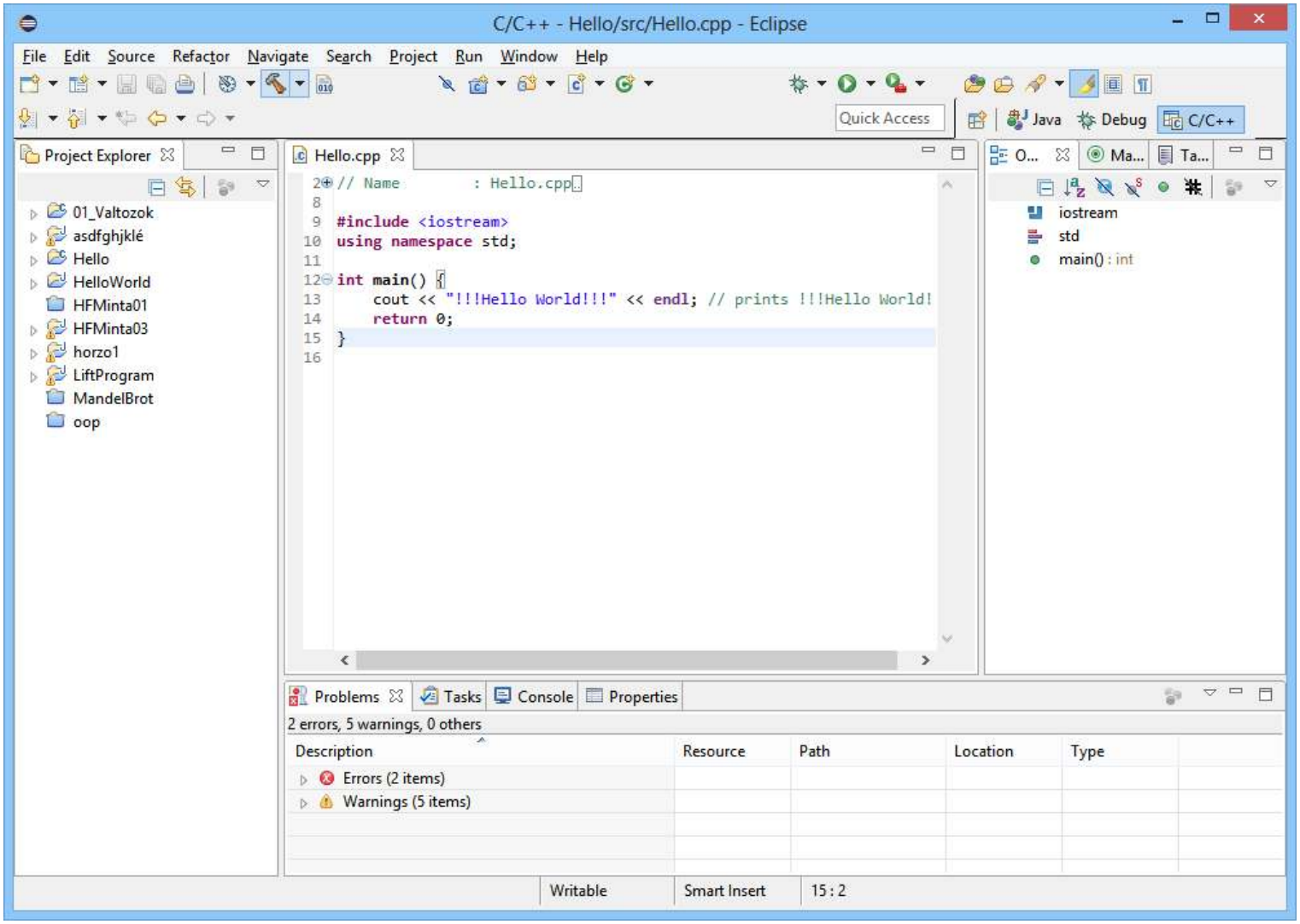

C++ Project
C++ Project
Create C++ project of selected type
Project name: Hello
Use default location
Location: C:\Users\Kálmán\workspace\Hello
Browse...
Project type:
Toolchains:
Executable
Empty Project
Hello World C++ Project
Shared Library
Static Library
Makefile project
MinGW GCC
Show project types and toolchains only if they are supported on the platform
< Back
Next >
Finish
Cancel


C++ Project
Basic Settings
Basic properties of a project
Author
Copyright notice
Your copyright notice
Hello world greeting
!!!Hello World!!!
Source
src
< Back
Next >
Finish
Cancel

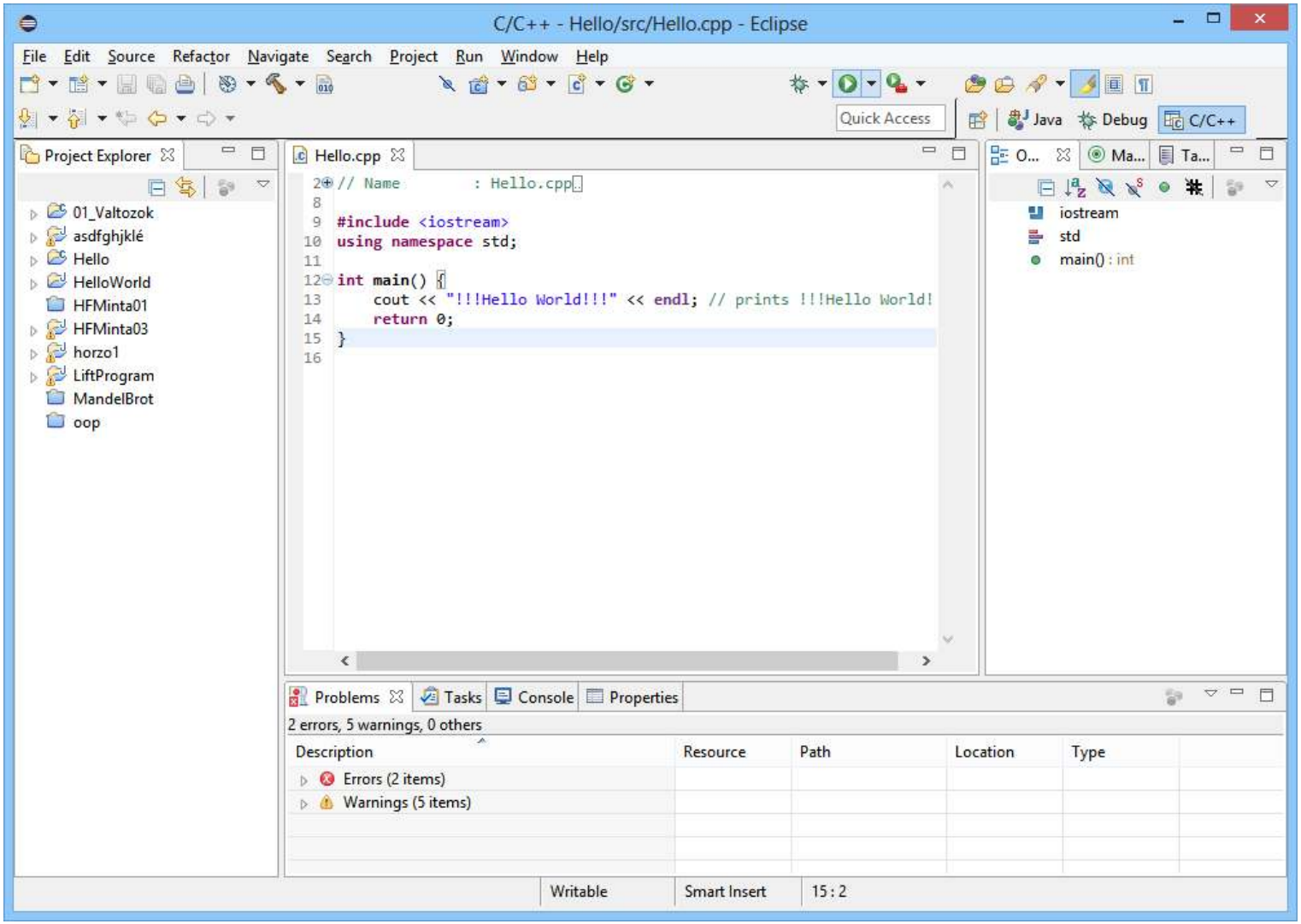
C++ projekt


C++ Project
Select Configurations
Select platforms and configurations you wish to deploy on
Project type: Executable
Toolchains: MinGW GCC
Configurations:
Debug
Release
Select all
Deselect all
Advanced settings...
Use "Advanced settings" button to edit project's properties.
Additional configurations can be added after project creation.
Use "Manage configurations" buttons either on toolbar or on property pages.
< Back
Next >
Finish
Cancel

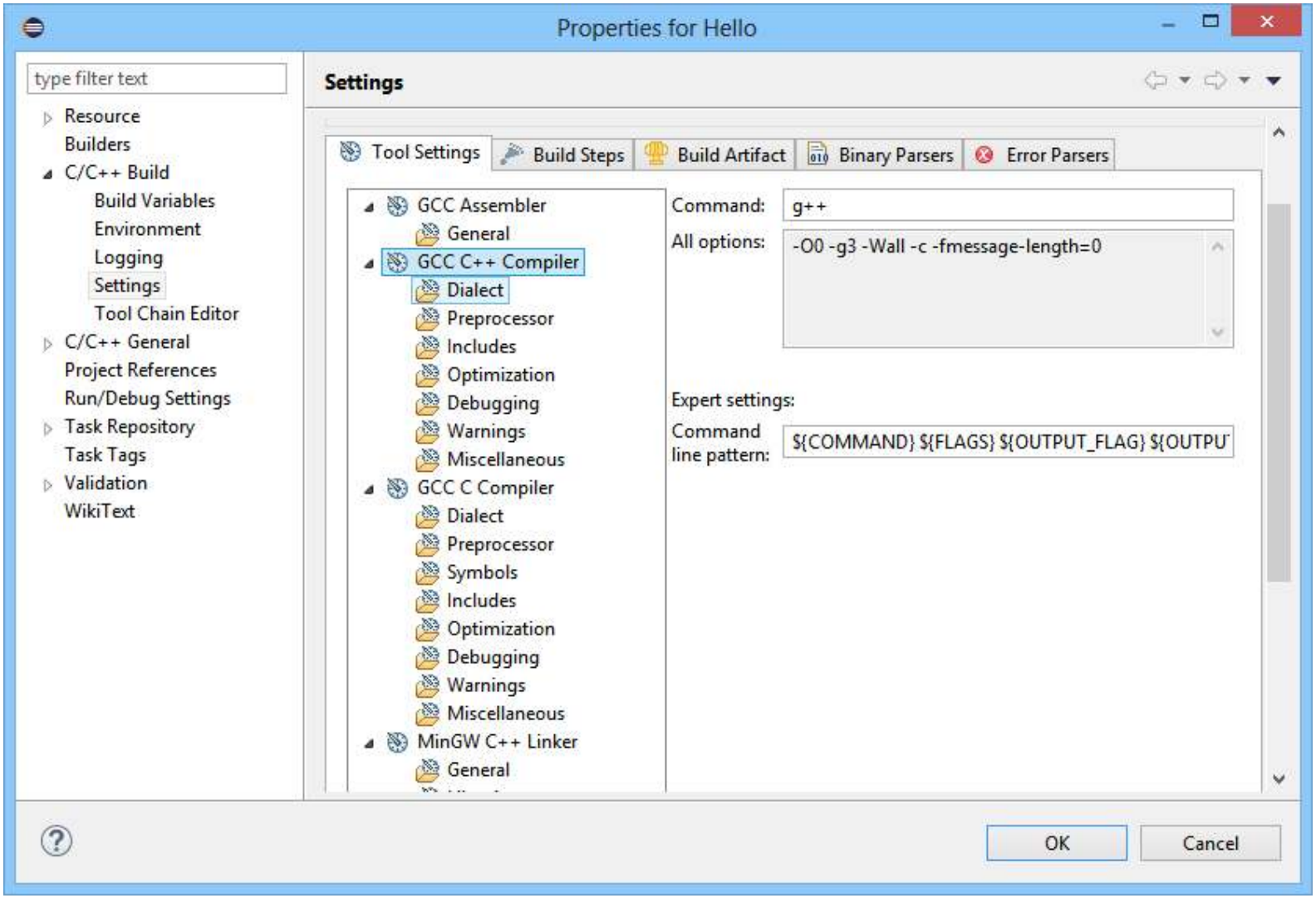
Fordítás


C/C++ - Hello/src/Hello.cpp - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
C/C++
Project Explorer
01_Valtozok
asdfghjklé
Hello
HelloWorld
HFMinta01
HFMinta03
horzo1
LiftProgram
MandelBrot
oop
Hello.cpp
// Name : Hello.cpp
#include <iostream>
using namespace std;
int main() {
cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!
return 0;
}
iostream
std
main() : int
Problems
Tasks
Console
Properties
2 errors, 5 warnings, 0 others
Description
Resource
Path
Location
Type
Errors (2 items)
Warnings (5 items)
Writable
Smart Insert
15 : 2

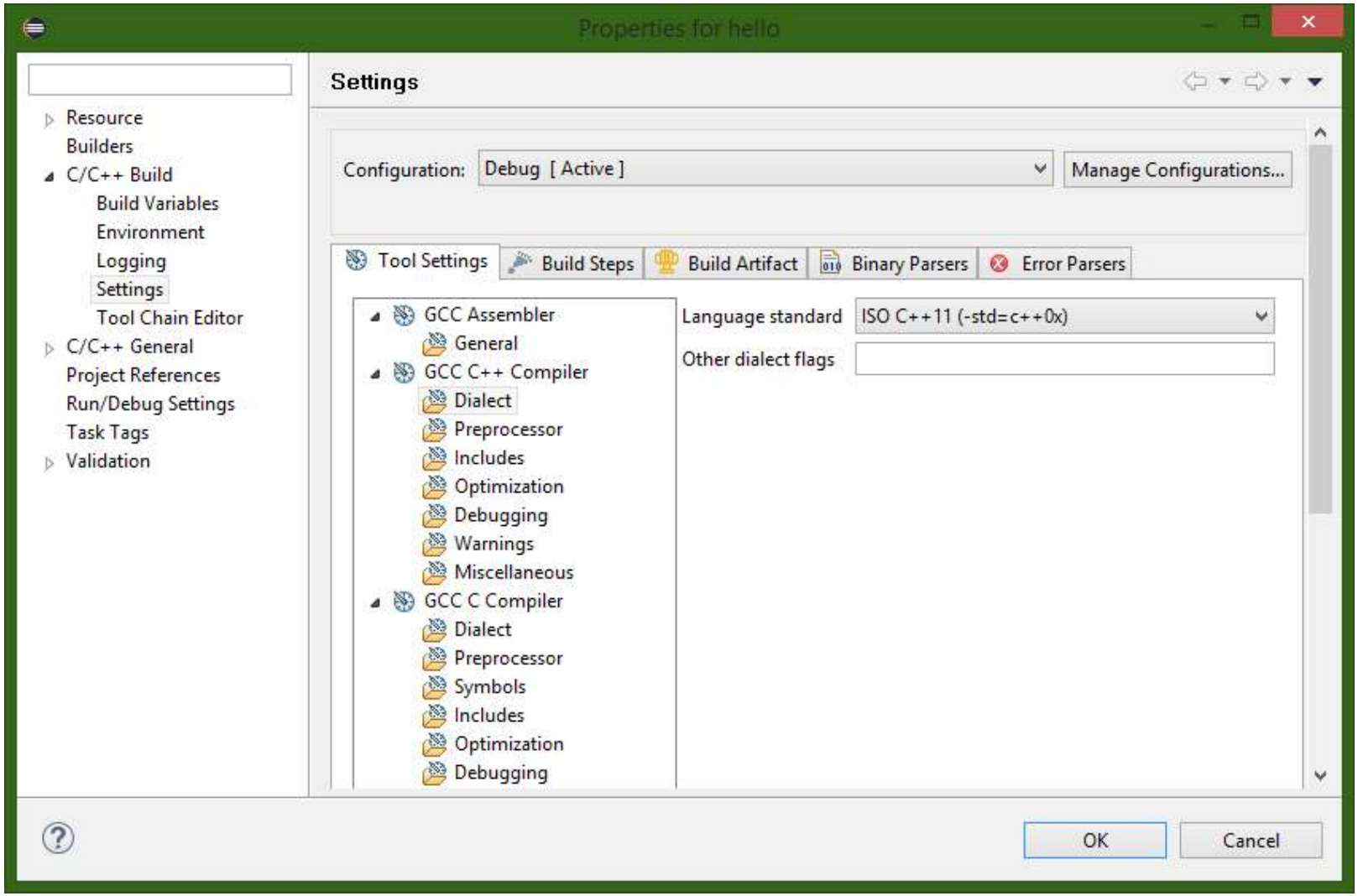
Futtatás


C/C++ - Hello/src/Hello.cpp - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
C/C++
Project Explorer
01_Valtozok
asdfghjklé
Hello
HelloWorld
HFMinta01
HFMinta03
horzo1
LiftProgram
MandelBrot
oop
Hello.cpp
// Name : Hello.cpp
#include <iostream>
using namespace std;
int main() {
cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!
return 0;
}
iostream
std
main() : int
Problems
Tasks
Console
Properties
2 errors, 5 warnings, 0 others
Description
Resource
Path
Location
Type
Errors (2 items)
Warnings (5 items)
Writable
Smart Insert
15 : 2

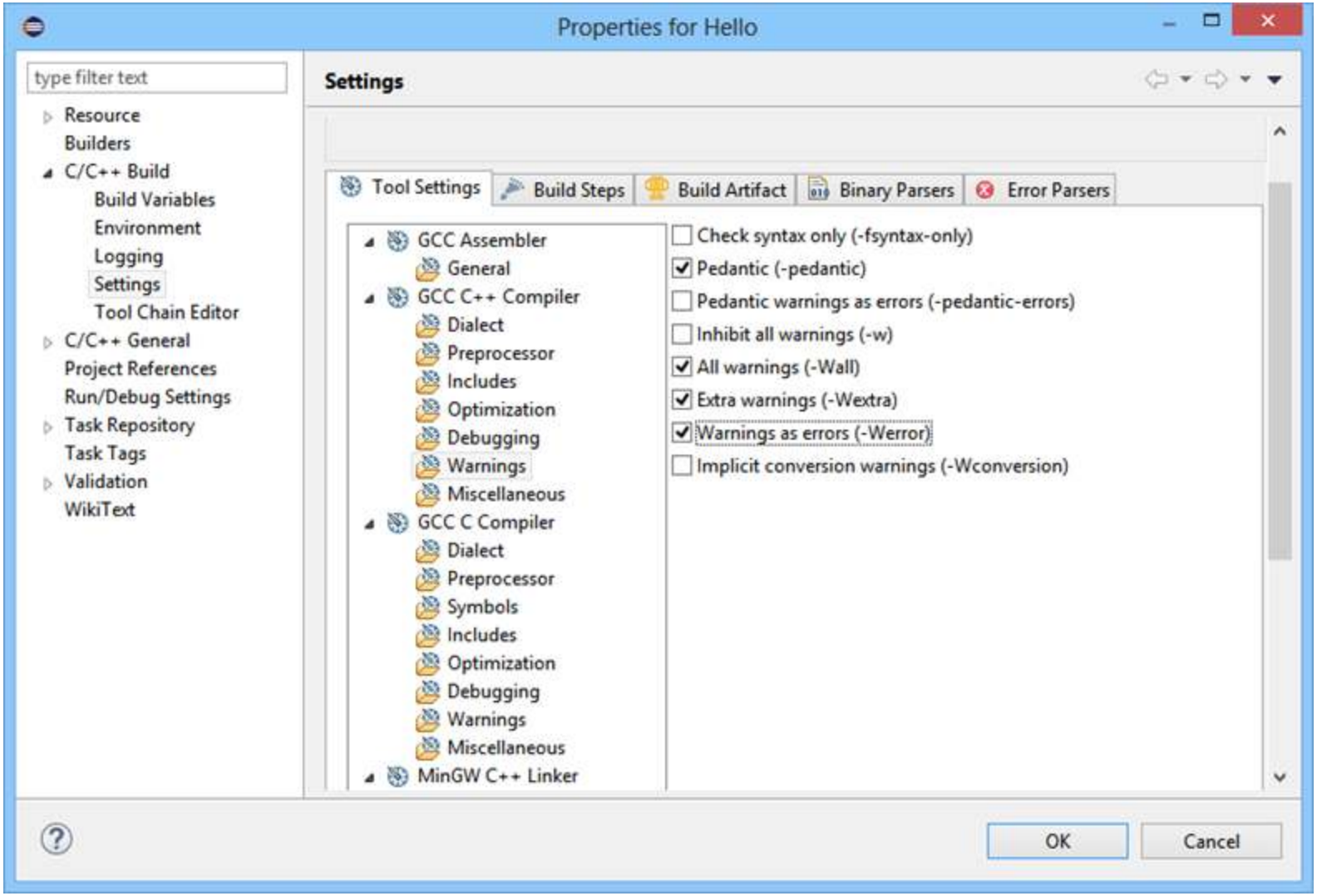
Project | Properties



Beállítások



Beállítások



C++

C++ alapok

Rendelkezésre álló eszköztár

- Változók
- Referenciák
- Pointerek
- Ezekből képezett tömbök
- Függvények
- Eljárások
- Paraméterek átadása
- Eredmény visszaadása
- Vezérlési szerkezetek
  - Szekvencia
  - Ciklus
  - Elágazás

Példák

- `int i;`
- `int & r = i;`
- `int * p = & i;`
- `int t[] = {1, 2, 3, 4};`
- `int fv() { return 1;}`
- `void elj() { };`
- `int square(int x) {`
- `    return x*x; }`

  - `{ int i = 1; int j = 2;}`
  - `while (feltétel) utasítás;`
  - `for( ; ; ) utasítás;`
  - `if (feltétel) utasítás;`
    `else utasítás;`

Változók

A változó

- Lefoglalt memóriaterület, amelyben értéket lehet tárolni
  - Egész, valós, logikai, memóriaterület címe, karakter, …
- A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
  - Ez a változónév

A változóhoz rendelt memóriaterület lefoglalása és címkézése egy lépésben történik:

- Deklaráció
  - Típus és név meghatározása
  - `int i;`
- Értékadás
  - `i = 4;`

Változók

A változó

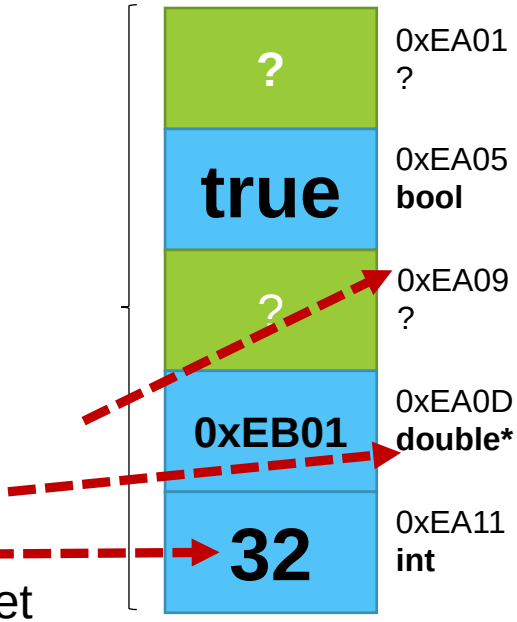
- Lefoglalt memóriaterület, amelyben értéket lehet tárolni
  - Egész, valós, logikai, memóriaterület címe, karakter, …
- A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
  - Ez a változónév

A változóhoz rendelt memóriaterület lefoglalása és címkézése egy lépésben történik:

- Deklaráció
  - Típus és név meghatározása
  - `int i;`
- Értékadás
  - `i = 4;`

Ebben a példában 32 bites rendszert, valamint a könnyebb érthetőség kedvéért 4 bájtos rekeszeket feltételezünk.

Memória – rekeszek sorozata
Minden rekesznek van címe (sorszáma)
Ismerjük a benne tárolt érték típusát
Ismerjük a benne tárolt értéket
Kék színnel jelezzük a lefoglalt rekeszeket



Változók

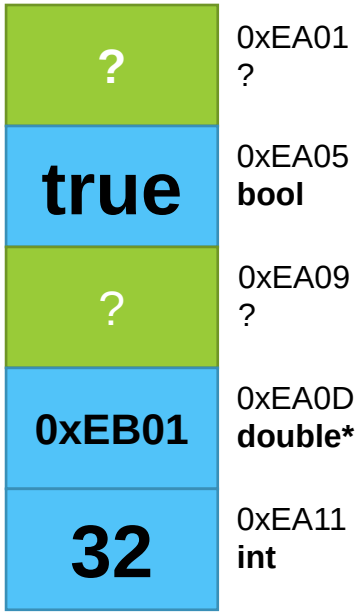
A változó

- Lefoglalt memóriaterület, amelyben értéket lehet tárolni
  - Egész, valós, logikai, memóriaterület címe, karakter, …
- A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
  - Ez a változónév

A változóhoz rendelt memóriaterület lefoglalása és címkézése egy lépésben történik:

- Deklaráció
  - Típus és név meghatározása
  - `int i;`
- Értékadás
  - `i = 4;`

Vegyük észre, hogy a memória bizonyos részein általunk nem ismert érték van.
Ha nem adunk értéket egy változónak, attól még a mögöttes memóriaterület fog tartalmazni egy értéket.



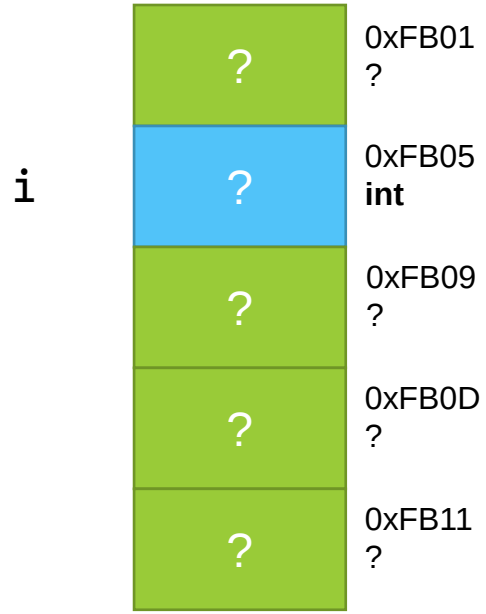
Változók

A változó

- Lefoglalt memóriaterület, amelyben értéket lehet tárolni
  - Egész, valós, logikai, memóriaterület címe, karakter, …
- A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
  - Ez a változónév

A változóhoz rendelt memóriaterület lefoglalása és címkézése egy lépésben történik:

- Deklaráció
  - Típus és név meghatározása
  - `int i;`
- Értékadás
  - `i = 4;`



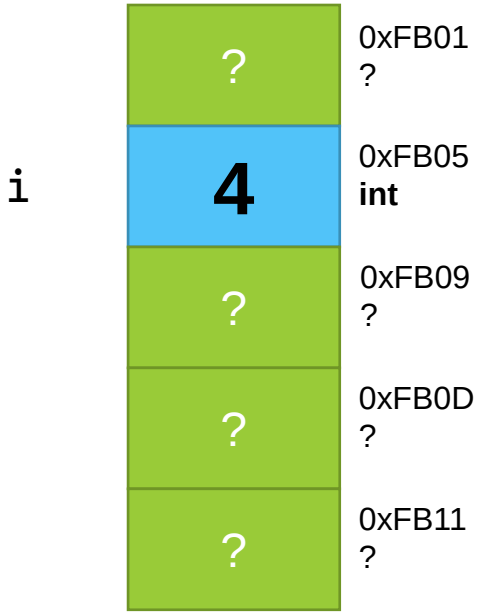
Változók

A változó

- Lefoglalt memóriaterület, amelyben értéket lehet tárolni
  - Egész, valós, logikai, memóriaterület címe, karakter, …
- A memóriaterülethez egy nevet rendelünk, amivel hivatkozhatunk rá
  - Ez a változónév

A változóhoz rendelt memóriaterület lefoglalása és címkézése egy lépésben történik:

- Deklaráció
  - Típus és név meghatározása
  - `int i;`
- Értékadás
  - `i = 4;`



Referencia

A referencia

- Egy már lefoglalt területre új címke létrehozása
  - Ezután két (vagy több) névvel hivatkozhatunk ugyanoda

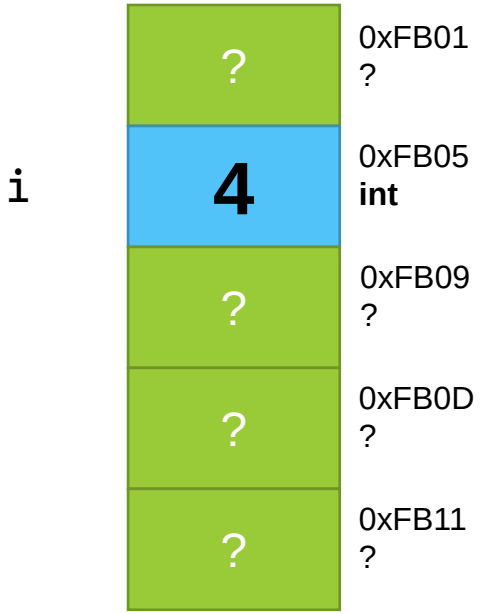
Referencia esetén csak címkét hozunk létre, ezért meg kell adni a már lefoglalt területet, amire az új címkét tesszük

- Deklaráció és inicializálás
  - Típus,név és létező változó megadása
  - `int & r = i;`
- Értékadás
  - `r = 5;`

Referencia

A referencia
- Egy már lefoglalt területre új címke létrehozása
  - Ezután két (vagy több) névvel hivatkozhatunk ugyanoda

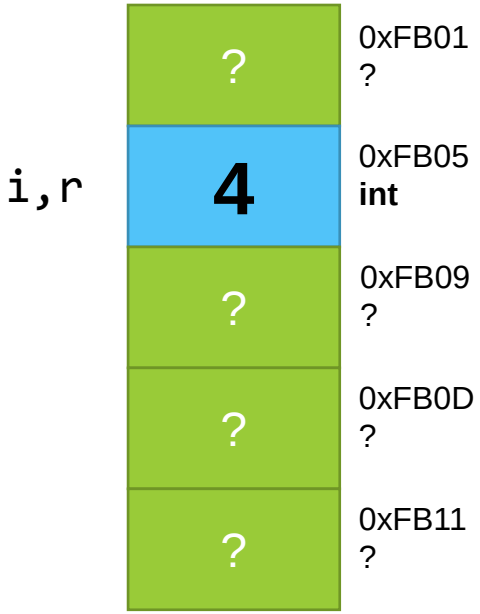
Referencia esetén csak címkét hozunk létre, ezért meg kell adni a már lefoglalt területet, amire az új címkét tesszük
- Deklaráció és inicializálás
  - Típus,név és létező változó megadása
  - `int & r = i;`
- Értékadás
  - `r = 5;`



Referencia

A referencia
- Egy már lefoglalt területre új címke létrehozása
  - Ezután két (vagy több) névvel hivatkozhatunk ugyanoda

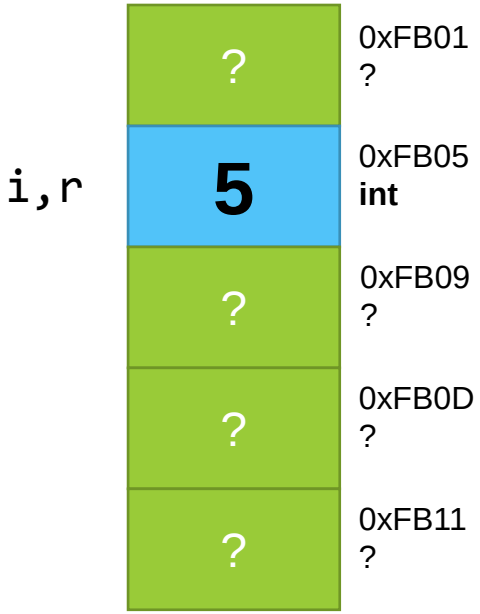
Referencia esetén csak címkét hozunk létre, ezért meg kell adni a már lefoglalt területet, amire az új címkét tesszük
- Deklaráció és inicializálás
  - Típus,név és létező változó megadása
  - `int & r = i;`
- Értékadás
  - `r = 5;`



Referencia

A referencia
- Egy már lefoglalt területre új címke létrehozása
  - Ezután két (vagy több) névvel hivatkozhatunk ugyanoda

Referencia esetén csak címkét hozunk létre, ezért meg kell adni a már lefoglalt területet, amire az új címkét tesszük
- Deklaráció és inicializálás
  - Típus,név és létező változó megadása
  - `int & r = i;`
- Értékadás
  - `r = 5;`



Mutató, pointer

A mutató

- Új memóriaterület kerül lefoglalása, mint közönséges változó esetén
  - Azonban a memóriaterületen memóriacímet tárolunk
- Természetesen a mutatónak is van neve
  - Sőt típusa is, amivel meghatározzuk, hogy milyen típusú tároló memóriára tud hivatkozni

A pointer létrehozása és használata a közönséges változóéhoz hasonló, a viselkedés a referenciához hasonló

- Deklaráció
  - Típus és név megadása
  - `int * p;`
- Értékadás – itt memóriacímet kell értékül adni
  - `p = & i;`
- A mutatott memóriaterület elérése (dereferencing)
  - `*p = 6;`

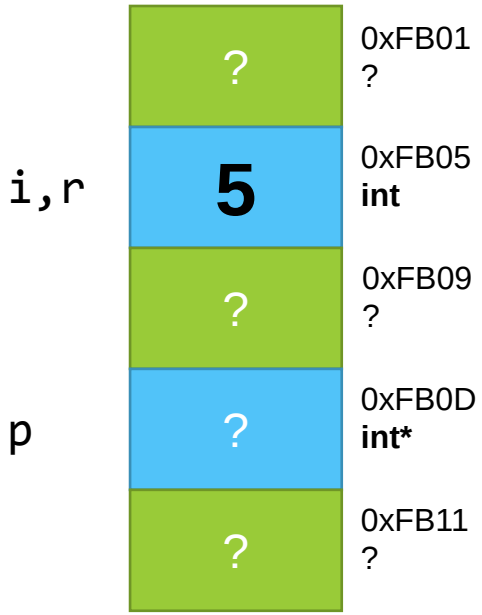
Mutató, pointer

A mutató

- Új memóriaterület kerül lefoglalása, mint közönséges változó esetén
  - Azonban a memóriaterületen memóriacímet tárolunk
- Természetesen a mutatónak is van neve
  - Sőt típusa is, amivel meghatározzuk, hogy milyen típusú tároló memóriára tud hivatkozni

A pointer létrehozása és használata a közönséges változóéhoz hasonló, a viselkedés a referenciához hasonló

- Deklaráció
  - Típus és név megadása
  - `int * p;`
- Értékadás – itt memóriacímet kell értékül adni
  - `p = & i;`
- A mutatott memóriaterület elérése (dereferencing)
  - `*p = 6;`



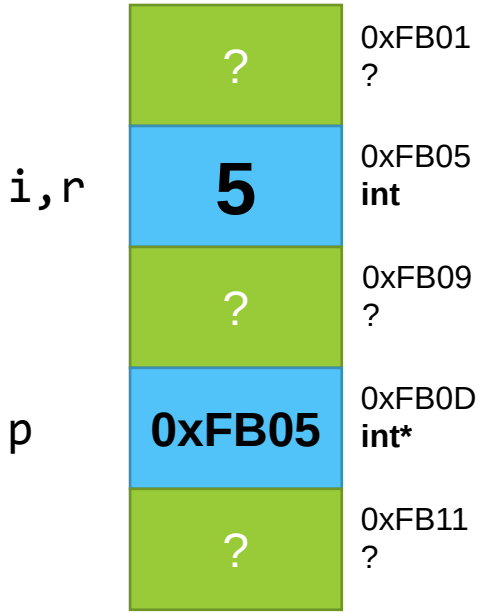
Mutató, pointer

A mutató

- Új memóriaterület kerül lefoglalása, mint közönséges változó esetén
  - Azonban a memóriaterületen memóriacímet tárolunk
- Természetesen a mutatónak is van neve
  - Sőt típusa is, amivel meghatározzuk, hogy milyen típusú tároló memóriára tud hivatkozni

A pointer létrehozása és használata a közönséges változóéhoz hasonló, a viselkedés a referenciához hasonló

- Deklaráció
  - Típus és név megadása
  - `int * p;`
- Értékadás – itt memóriacímet kell értékül adni
  - `p = & i;`
- A mutatott memóriaterület elérése (dereferencing)
  - `*p = 6;`



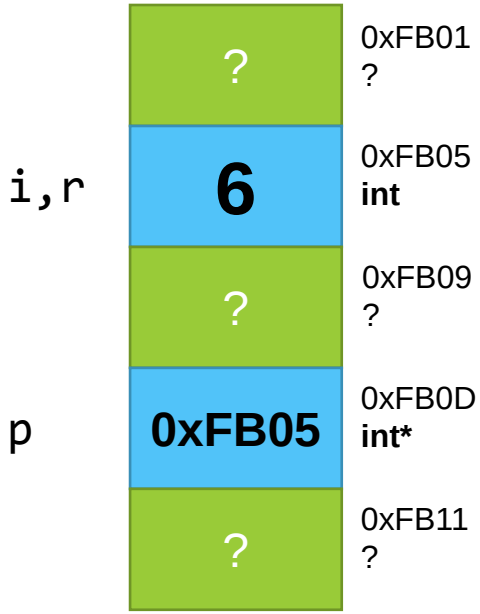
Mutató, pointer

A mutató

- Új memóriaterület kerül lefoglalása, mint közönséges változó esetén
  - Azonban a memóriaterületen memóriacímet tárolunk
- Természetesen a mutatónak is van neve
  - Sőt típusa is, amivel meghatározzuk, hogy milyen típusú tároló memóriára tud hivatkozni

A pointer létrehozása és használata a közönséges változóéhoz hasonló, a viselkedés a referenciához hasonló

- Deklaráció
  - Típus és név megadása
  - `int * p;`
- Értékadás – itt memóriacímet kell értékül adni
  - `p = & i;`
- A mutatott memóriaterület elérése (dereferencing)
  - `*p = 6;`



Változó, referencia, pointer

Példák

- `int    i  = 5;`
- `double d  = 4.0;`
- `int &  r  = i;`
- `r++;`
- `int *  p  = &i;`
- `(*p)++;`
- `int* & r2 = p;`
- `(*r2)++;`
- `int &  r3 = *p;`
- `const int & k = 6;`
- `int ** p2 = &p;`
- `(**p2)++;`

Jelentése

- Változó deklarálása
- Változó deklarálása
- Referencia létrehozása
- Hozzáférés a változóhoz a referencián keresztül (az `i` értéke 6)
- Új pointer, a változó címét tárolja
- Hozzáférés a pointeren keresztül (az `i` értéke 7)
- Másik referencia, a pointerre
- Hozzáférés a pointerhez referencián keresztül (az `i` értéke 8)
- Referencia a pointer által mutatott változóra (ez ugyanoda referál, ahova az `r`)
- Konstans referencia konstans értékre
- Pointer-re mutató pointer
- Hozzáférés a pointer pointerén keresztül (az `i` értéke 9)

Próbáljuk ki!

A 01_Valtozok projekt megnyitása után kipróbálhatod az eddigieket.

Egészítsük ki a tesztprogramot és próbáljuk ki a konstans referenciát és a pointerre mutató referenciát!

A & operátor mindig egy változó mögötti terület címét adja vissza.

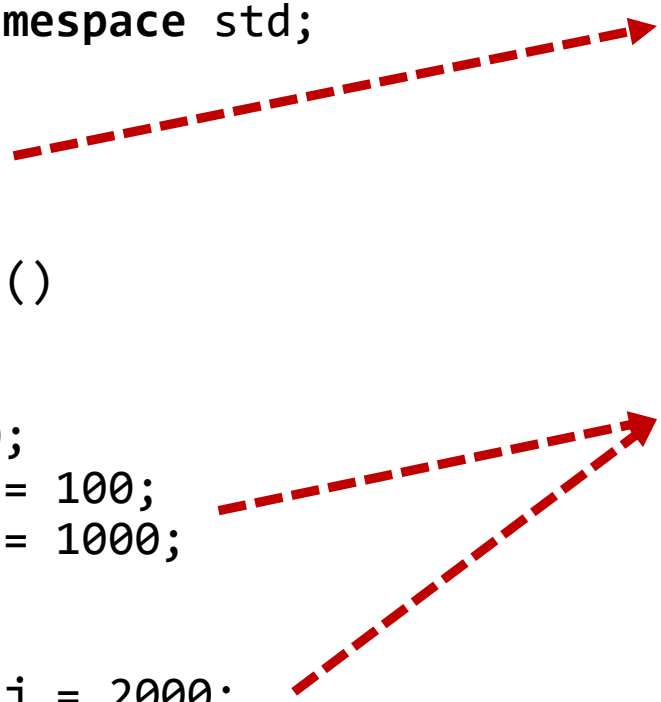
A * operátor mindig a memóriacímet követi és a mögöttes területet érjük el vele.

- `int i = 5;`
- `int * p = & i;`
- `&(*p) == p;`

Az utóbbi sornak mindig igaznak kell lennie, mivel a mutató által mutatott terület címe a mutató értéke kell, hogy legyen.

Globális, lokális változók

```
#include <iostream>
using namespace std;

int i;
int j;

int main()
{
  i = 1;
  j = 10;
  int k = 100;
  int j = 1000;
  {
    j++;
    int j = 2000;
    j++;
  }
}
```



Globális:

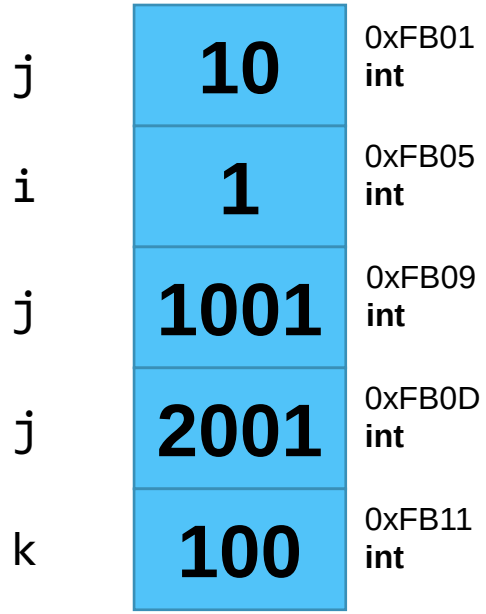
- Függvényeken kívüli deklaráció
- Mindenhol hivatkozhatunk rájuk (ha nincs elfedés)

Lokális:

- Blokkon belül deklarált
- Deklarációkor tárterület foglalódik le
- A blokk végén a tárterület felszabadul, a változó megszűnik.

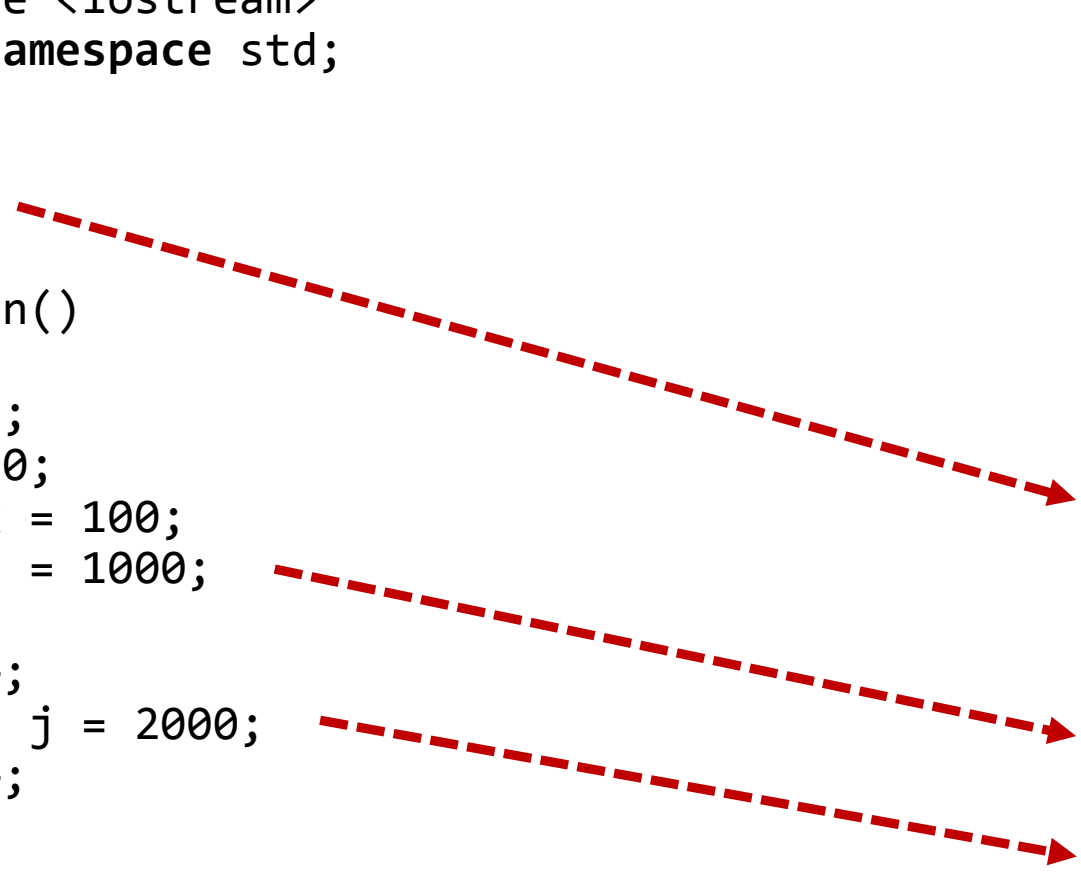
Globális, lokális változók

```
#include <iostream>
using namespace std;

int i;
int j;

int main()
{
  i = 1;
  j = 10;
  int k = 100;
  int j = 1000;
  {
    j++;
    int j = 2000;
    j++;
  }
}
```



Globális, lokális változók

```
#include <iostream>
using namespace std;

int i;
int j;

int main()
{
  i = 1;
  j = 10;
  int k = 100;
  int j = 1000;
  {
    j++;
    int j = 2000;
    j++;
  }
}
```



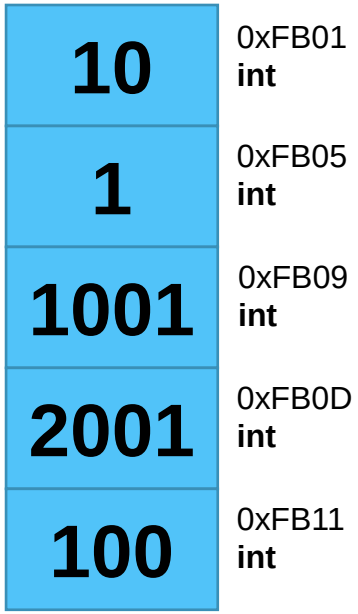
j

i

j

j

k



Eljárások és függvények

FÜGGVÉNY

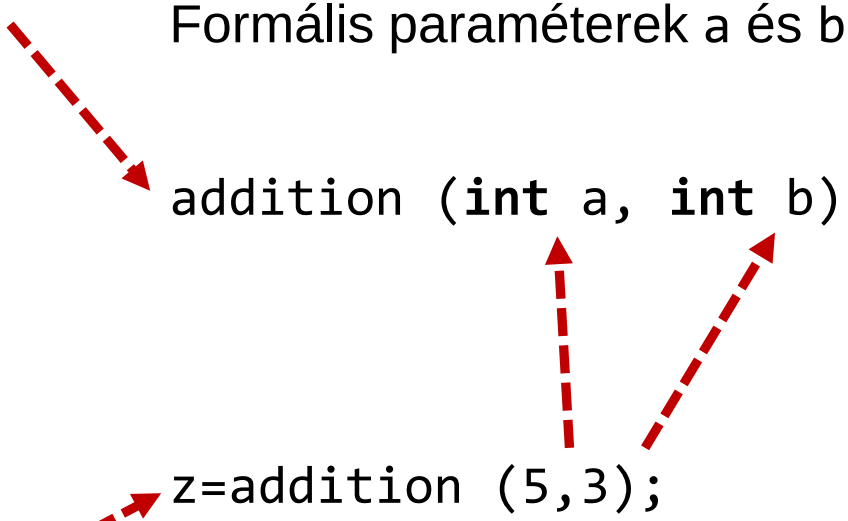
```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```

ELJÁRÁS

```
#include <iostream>
using namespace std;
void kiir ()
{
   cout << ”Ezt írom ki!”;
}

int main ()
{
   kiir ();
   return 0;
}
```

Paraméterátadás

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```



Paraméterátadás és visszatérési érték

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```

Formális paraméterek a és b.

```
addition (int a, int b)
```

```
z=addition (5,3);
```

Aktuális paraméterek értékei 5 és 3.

Paraméterátadás szabályai

C++-ban a **közönséges változó és pointer** esetén **érték szerinti paraméterátadás** történik.

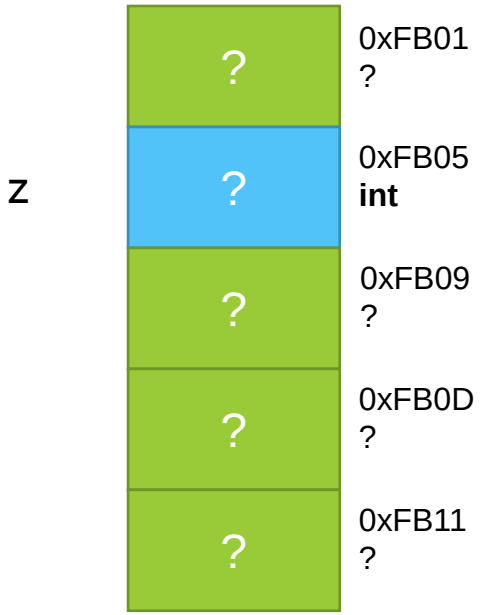
- Függvényhíváskor a formális paraméterek lokális változóként deklarálásra kerülnek
- Ezt követően inicializálódnak az aktuális paraméterek értékével
  - Ez közönséges változó esetén az az érték, amit beletettünk
  - Ez pointer esetén a memóriacím
- Ennek megfelelően az eredeti változó tartalmáról egy másolat készül egy másik memóriaterületre
- Összefoglalva a formális paraméterek másik memóriaterületre hivatkoznak, mint az aktuális paraméterek!

C++-ban a **referencia** formális paraméterek **esetén az aktuális paraméterre** (változóra) **egy** (új) **referencia kerül deklarálásra.**

- Ennek megfelelően az eredetileg lefoglalt memóriaterületet címkézzük fel újra
- Minden a formális paraméteren (lokális változón) keresztüli változtatás az eredeti memóriaterületet változtatja meg

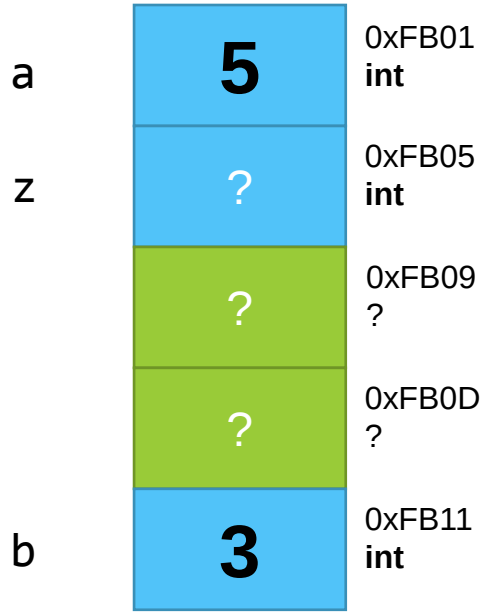
Paraméterátadás bemutatása

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```



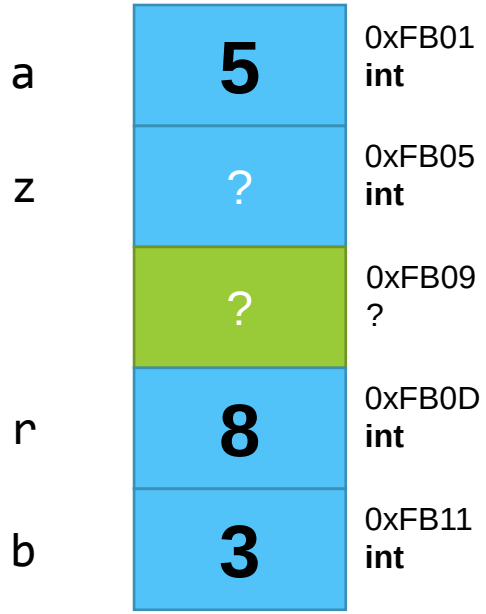
Paraméterátadás bemutatása

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```



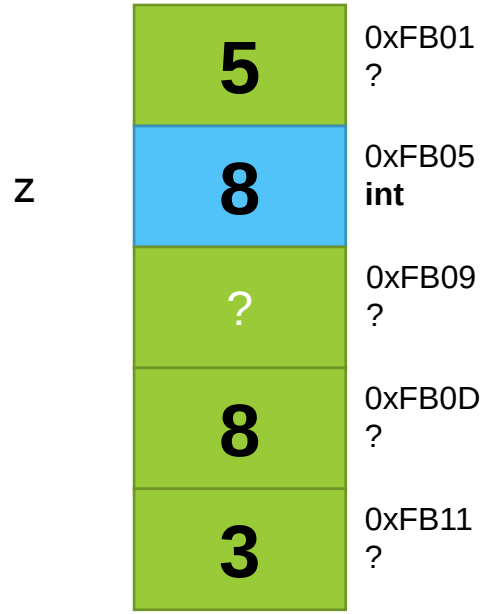
Paraméterátadás bemutatása

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```



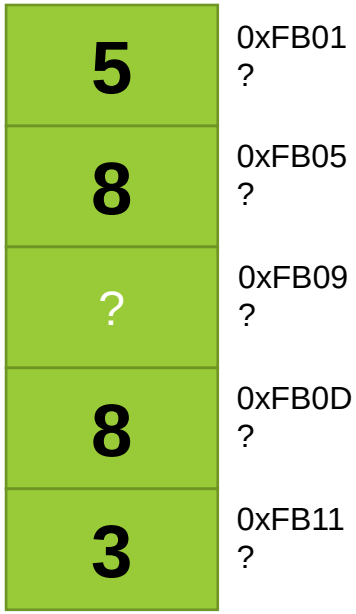
Visszatérési érték bemutatása

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```



Paraméterátadás és visszatérési érték

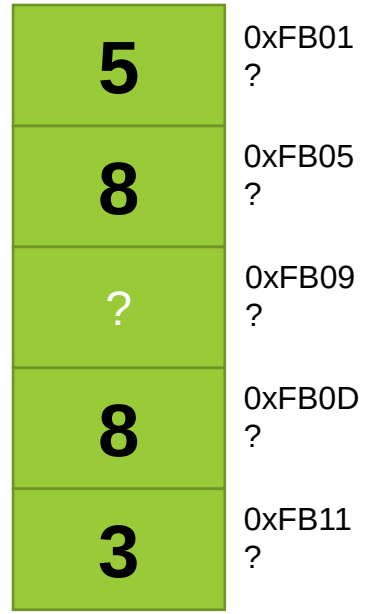
```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```



Paraméterátadás és visszatérési érték

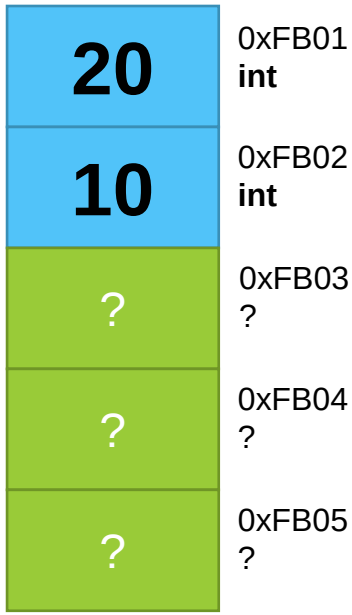
```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
   int r;
   r = a + b;
   return r;
}


int main ()
{
   int z;
   z = addition (5,3);
   cout << ”Eredmeny ” << z;
   return 0;
}
```

Vegyük észre, hogy a memóriában a korábban tárolt értékek továbbra is ott vannak, csupán egyikre sincs érvényes deklarált változó! Később azonban felülíródhatnak!



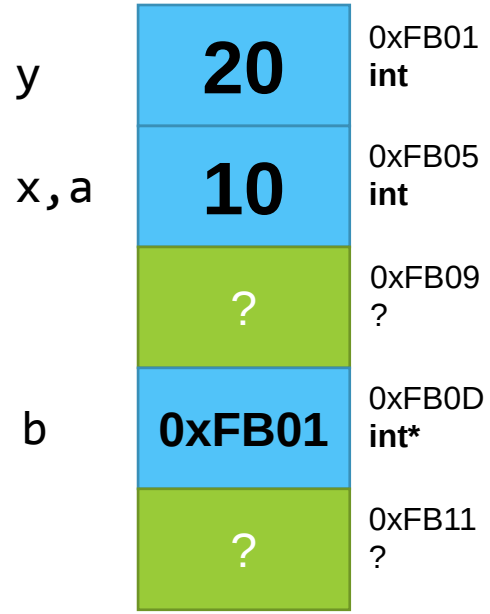
Ugyanez referenciával és pointerrel

```
#include <iostream>
using namespace std;
void doSomething(int & a, int * b)
{
   a++;
   (*b)++;
}


int main ()                                          y
{
   int x = 10;                                       x
   int y = 20;
   doSomething (x, &y);
   return 0;
}
```



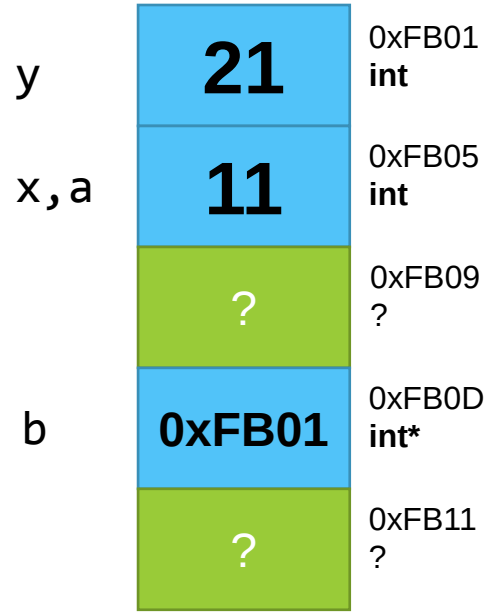
Ugyanez referenciával és pointerrel

```
#include <iostream>
using namespace std;
void doSomething(int & a, int * b)
{
   a++;
   (*b)++;
}

int main ()
{
   int x = 10;
   int y = 20;
   doSomething (x, &y);
   return 0;
}
```



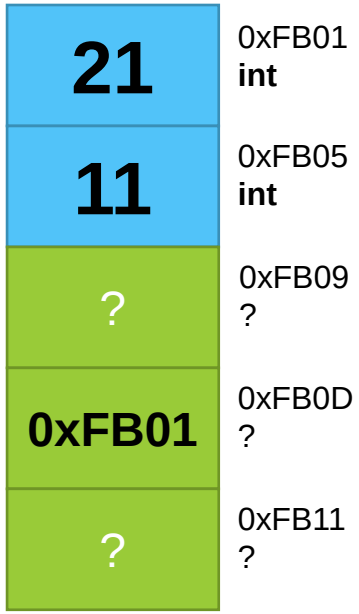
Ugyanez referenciával és pointerrel

```
#include <iostream>
using namespace std;
void doSomething(int & a, int * b)
{
   a++;
   (*b)++;
}


int main ()
{
   int x = 10;
   int y = 20;
   doSomething (x, &y);
   return 0;
}
```



Ugyanez referenciával és pointerrel

```
#include <iostream>
using namespace std;
void doSomething(int & a, int * b)
{
   a++;
   (*b)++;
}


int main ()                                          y
{
   int x = 10;                                       x
   int y = 20;
   doSomething (x, &y);
   return 0;
}
```



Konstans paraméterek

A (referencia) paraméterátadás során garantálható, hogy a hívott függvény az eredeti értéket ne tudja megváltoztatni

- Ekkor a paraméter konstans
- `const int & i`
- A módszer lényege, hogy az esetlegesen nagyméretű paraméter (nagy memóriaterületet elfoglaló) nincs lemásolva (a referencia paraméterátadás miatt), emellett nem megváltoztatható, mintha érték szerinti paraméterátadás lenne

A konstans paraméterátadás pointerekkel is működik

- `void f(int * const p)`
  - Ebben az esetben a cím konstans, mivel a pointer értéke a cím
- `void f(const int * p)`
  - Itt a cím által mutatott memóriaterület nem megváltoztatható
  - Ezzel ekvivalens a következő írásmód is `void f(int const * p)`
- `void f(int const * const p)`
  - Itt a pointer értéke (cím) és a mutatott memóriaterület egyaránt konstans

Konstans paraméterek?

A cím konstans – **`void f(int * const p)`**

- A függvényben nem tehetem meg a következőt
  - `p = new int;`
  - `p = q;`

A címzett terület konstans – **`void f(const int * p)`**

- A függvényben nem tehetem meg a következőt
  - `*p = 5;`
  - `*p = *q;`

Pointer paraméterek és visszatérési értékek

Tekintsük a következő két függvényt:

- `void f(int i);`
- `void f(int* i);`

Mi történik a következő függvényhívások esetén?

- `f(0); f(NULL);`
- Mivel a 0 és a NULL is int, ezért mindkét esetben az elsőt hívja meg.

Hogyan hívhatjuk meg a másodikat?

- `f((int*) 0);`
- C++11 óta a **nullptr** értéket is használhatjuk: `f(nullptr);`

Pointert paraméterként átadva a referenciához hasonló viselkedést kapunk

- Fontos ellenőrizni, hogy a pointerben érvényes címet kaptunk-e
- Ha **`nullptr`**, `0` vagy `NULL` az érték (az egyiket elég vizsgálni), akkor az természetesen probléma

Visszatérési érték gyanánt:

- Leggyakrabban valaminek a legyártására szoktuk használni
- Borzasztó fontos, hogy a lefoglalt memóriaterületeket mindig fel kell szabadítani
  - Nemsokára lesz erről még szó

Próbáljuk ki!

A következő programot hozd létre:

- Legyen egy eljárás, ami paraméterként vár egy valós számokból álló vektort
- Az eljárás a vektorban tárolt számok átlagát számolja ki.
- A paraméterátadás során ne másold le a vektort!

Ügyelj arra, hogy csak olyan memóriaterületet tudjon megváltoztatni az eljárás, amely feltétlen szükséges a feladat kiírás szerinti megvalósításához.

Gondolkodjunk

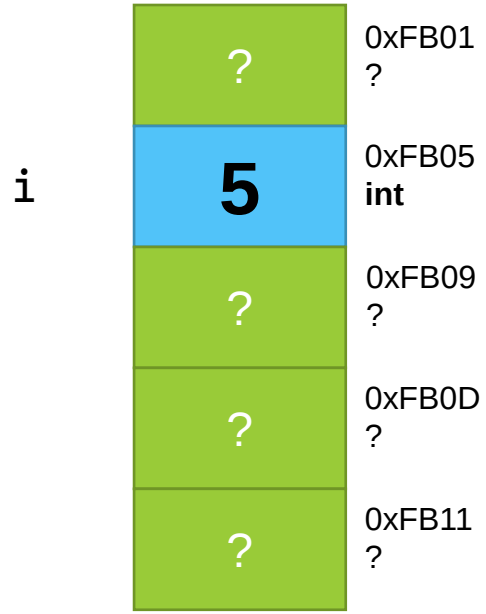
A következő programot gondoljuk át figyelmesen – mi a probléma vele?

```
#include <iostream>
using namespace std;
int & create()
{
   int i = 5;
   return i;
}


int main ()
{
   int & x = create();
   x++;
   return 0;
}
```

Gondolkodjunk

A következő programot gondoljuk át figyelmesen – nézzük meg

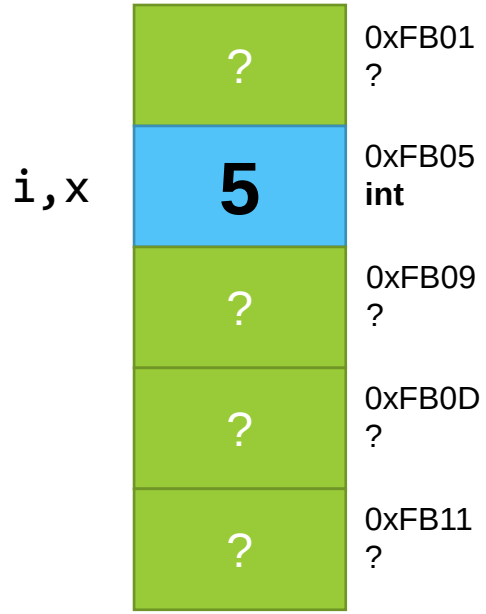
```
#include <iostream>
using namespace std;
int & create()
{
   int i = 5;
   return i;
}


int main ()
{
   int & x = create();
   x++;
   return 0;
}
```



Gondolkodjunk

A következő programot gondoljuk át figyelmesen – nézzük meg

```
#include <iostream>
using namespace std;
int & create()
{
   int i = 5;
   return i;
}


int main ()
{
   int & x = create();
   x++;
   return 0;
}
```

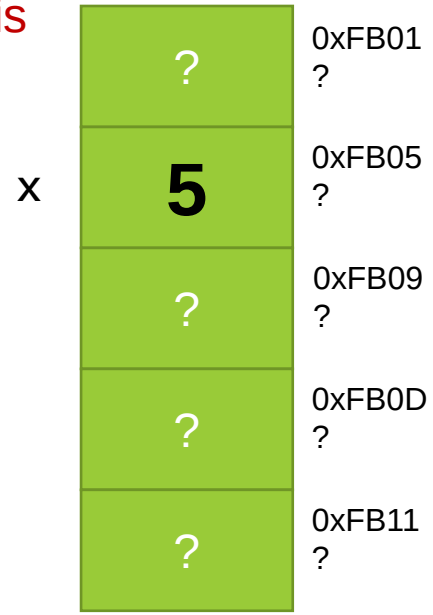


Gondolkodjunk

A következő programot gondoljuk át figyelmesen:

```
#include <iostream>
using namespace std;
int & create()
{
   int i = 5;
   return i;
}


int main ()
{
   int & x = create();
   x++;
   return 0;
}
```

A referencia egy lokális változó memória-területére hivatkozik, azonban a lokális változóhoz tartozó memóriaterület a függvény befejeztével felszabadításra kerül. A felszabadult memóriához történő hozzáférés helytelen!



Tömbök

A tömb:

- Azonos típusú elemek sorozata
- A memóriában folytonosan helyezkedik el
  - Indexelhető
    - Az index 0-val kezdődik
    - Ennek megfelelően egy *n* méretű tömb esetén az érvényes indexek halmaza [0..*n*-1]
- A mérete fordítási időben ismert érték kell, hogy legyen
- Deklarációja:
  - `int tomb[5];`
- Használata:
  - `tomb[4] = 5;`
- A tömbre hivatkozó változó tömb első elemének a memóriacímét tartalmazza
  - De ez nem azt jelenti, hogy az egy pointer
  - A változót felhasználva pointer-aritmetikával lehetséges a tömb elemeinek manipulálása
  - A [] operátor használatával a pointerhez kapcsolódó dereferencing is megtörténik

Figyeljük meg a következő példakódot a pointer-aritmetikára

Tömbök és pointer kapcsolata

Nézzük meg a következő kódrészletet

```
int main ()
{
   int tomb[5];
   int* p;
   p = tomb;     *p = 10;
   p++;              *p = 20;
   p = &tomb [2];   *p = 30;
   p = tomb + 3;    *p = 40;
   p = tomb;     *(p+4) = 50;
   return 0;
}
```

Tömbök és pointer kapcsolata

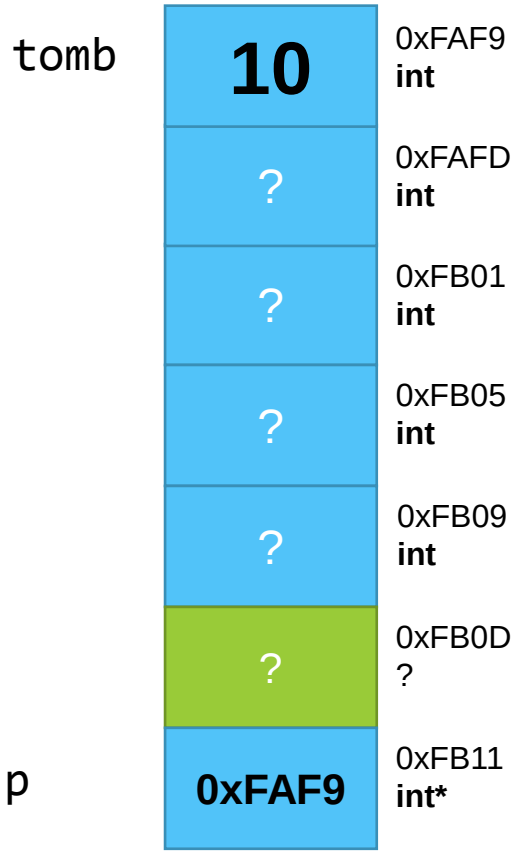
Nézzük meg a következő kódrészletet

```
int main ()
{
   int tomb[5];
   int* p;
   p = tomb;     *p = 10;
   p++;              *p = 20;
   p = &tomb [2];   *p = 30;
   p = tomb + 3;    *p = 40;
   p = tomb;     *(p+4) = 50;
   return 0;
}
```

Tömbök és pointer kapcsolata

Nézzük meg a következő kódrészletet

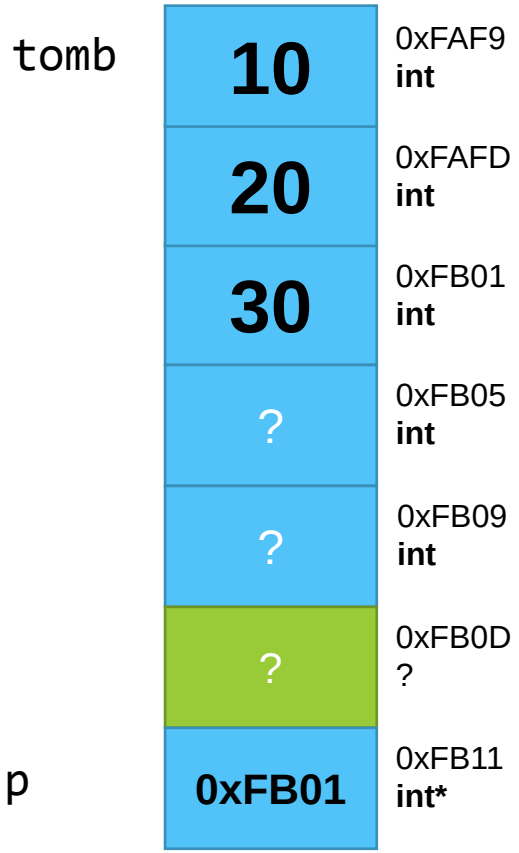
```
int main ()
{
   int tomb[5];
   int* p;
   p = tomb;     *p = 10;
   p++;              *p = 20;
   p = &tomb [2];   *p = 30;
   p = tomb + 3;    *p = 40;
   p = tomb;     *(p+4) = 50;
   return 0;
}
```



Tömbök és pointer kapcsolata

Nézzük meg a következő kódrészletet

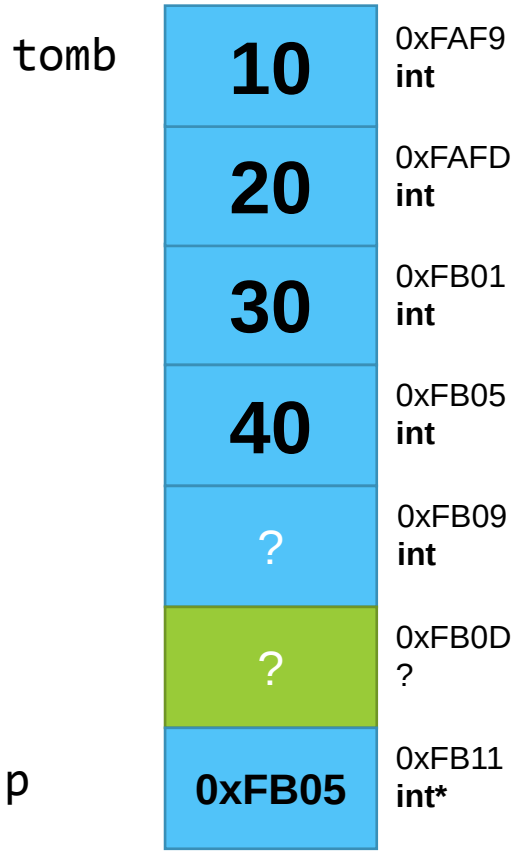
```
int main ()
{
   int tomb[5];
   int* p;
   p = tomb;     *p = 10;
   p++;              *p = 20;
   p = &tomb [2];   *p = 30;
   p = tomb + 3;    *p = 40;
   p = tomb;     *(p+4) = 50;
   return 0;
}
```



Tömbök és pointer kapcsolata

Nézzük meg a következő kódrészletet

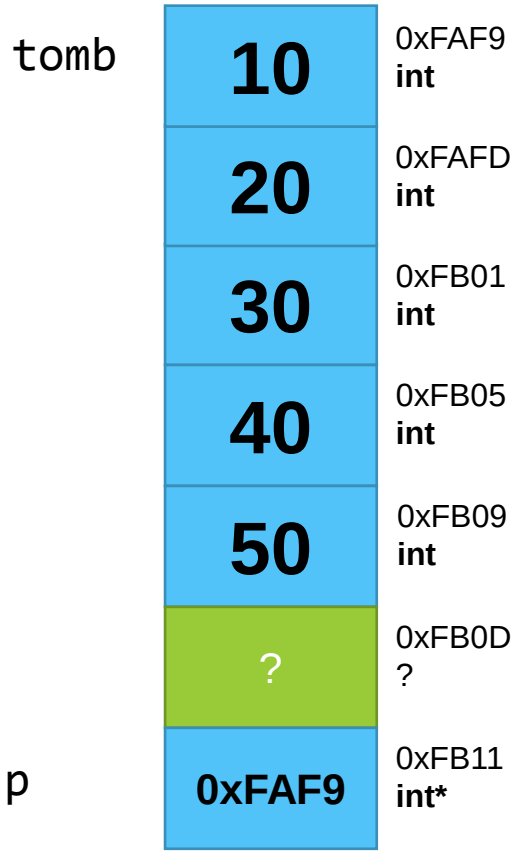
```
int main ()
{
   int tomb[5];
   int* p;
   p = tomb;     *p = 10;
   p++;              *p = 20;
   p = &tomb [2];   *p = 30;
   p = tomb + 3;    *p = 40;
   p = tomb;     *(p+4) = 50;
   return 0;
}
```



Tömbök és pointer kapcsolata

Nézzük meg a következő kódrészletet

```
int main ()
{
   int tomb[5];
   int* p;
   p = tomb;     *p = 10;
   p++;              *p = 20;
   p = &tomb [2];   *p = 30;
   p = tomb + 3;    *p = 40;
   p = tomb;     *(p+4) = 50;
   return 0;
}
```



Tömb-pointer – következmények

A tömbhöz tartozó változó nagyjából megfeleltethető a tömb első elemének a címével

- De nem pointer az első elemre!
- A tömbhöz tartozó változó – jelen esetünkben – `int[]` típusú
- De láttuk, hogy kezelhetjük úgy is, mintha egy mutató lenne.

Ennek következményei a következők:

- Paraméterátadáskor a pointerekkel egyező mechanizmus működik

```
void modifyArray(int t[], int length)
{
  if (length > 0) t[0]++;
}
```

  Ennek a függvénynek a hívásakor az átadott tömb címe másolódik és az eredeti tömbön kerül végrehajtásra bármilyen módosítás
- A tömb méretét paraméterátadáskor át kell adni
  - A méret futási időben ugyan nem mindig határozható meg, illetve csak pointer aritmetikával számolható ki.
- Semmilyen ellenőrzés nincsen a túlindexelést illetően
  - Nem csak pointer aritmetika használata esetén
- Sőt, a következő két utasítás fordul és fut, de ilyent nem teszünk!
  - `int* p; p[0] = 10;`

Próbáljuk ki!

Teszteljük egy programmal a tömbök és pointerek kapcsolatát

- Próbáld ki a tömb értékeinek feltöltését a korábban ismertetett pointer-aritmetikai módszerekkel
  - Találj minél több, különböző elvű módot
- Írj egy programot, amiben egy eljárás tömböt vesz át paraméterként
- Próbáld ki, hogy mi történik, ha az eljárásban megváltozik a tömb értéke
- Nézd meg, hogy mi történik, ha túlindexelsz egy tömböt
  - Lehetséges-e megváltoztatni a tömb értékét?
- Nézd meg, hogy mi történik, ha egy nem tömbre vonatkozó pointert tömbként kezelsz

Dinamikus memóriakezelés

Eddig amikor egy változó bevezetésre került (deklaráltuk), akkor az ahhoz tartozó memóriaterület automatikus lefoglalásra és a blokk végén felszabadításra került

- Így történt ez a tömbök esetén is

Azonban gyakran nem szeretnénk erre az automatizmusra bízni a memória és a változók kezelését

- A programnak egy futásidőben kapott érték szerinti memóriamennyiségre van szükség (felhasználói inputtól függő tömbméret)
- A függvény végén ne kerüljön felszabadításra a lefoglalt memóriaterület
  - Például mert a memóriában tárolt értékre továbbra is szükség van

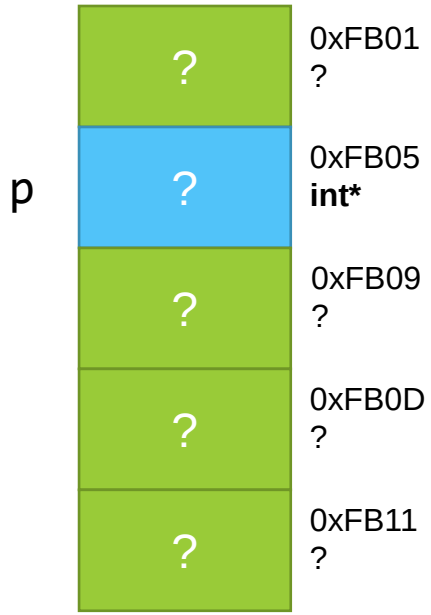
A dinamikus memóriakezelést a pointerek segítségével lehet megtenni

- Két művelet:
  - **`new`** – lefoglal egy memóriaterületet és a terület címét adja vissza
  - **`delete`** – egy lefoglalt memóriaterületet szabadít fel
- Mivel a memóriakezelést kivesszük az automatizmus kezéből ezért kritikus feladat a memória felszabadítás elvégzése

Dinamikus memóriakezelés – hibaforrások

Dinamikus változó, memóriaterület lefoglalása és felszabadítása:

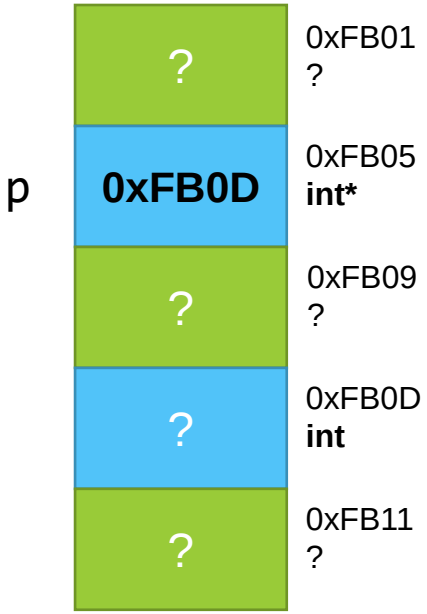
```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 *p = 20;



}
```



Dinamikus memóriakezelés – hibaforrások

Dinamikus változó, memóriaterület lefoglalása és felszabadítása:

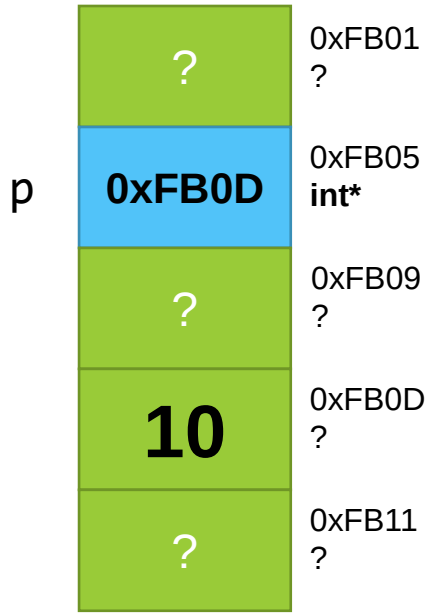
```
{
  int* p;
  p = new int;
  *p = 10;
  delete p;
  *p = 20;



}
```



Dinamikus memóriakezelés – hibaforrások

Dinamikus változó, memóriaterület lefoglalása és felszabadítása:

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 *p = 20;



}
```



Dinamikus memóriakezelés – hibaforrások

Dinamikus változó, memóriaterület lefoglalása és felszabadítása:

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 *p = 20;



}
```

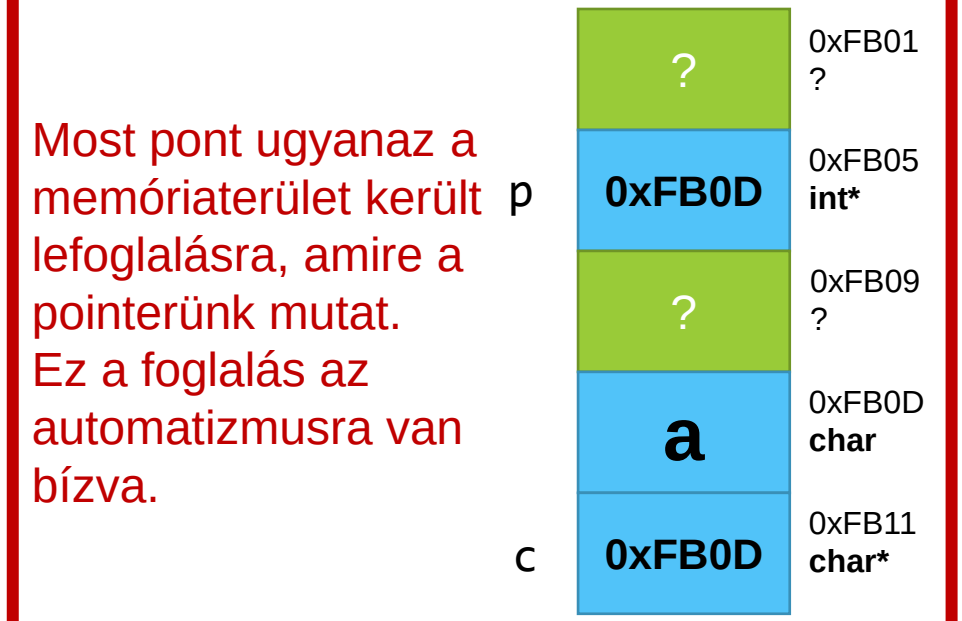
Vegyük észre, hogy a pointer létezik, az előzőleg lefoglalt és már **felszabadított** p memóriaterületre hivatkozik.
A memóriaterület *nem biztos*, hogy azonnal kerül újrahasznosításra, azaz le fog futni a teljes kódrészlet.

Dinamikus memóriakezelés – hibaforrások

Nézzük tovább, a hibás értékadó utasítás kihagyásával!

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 char* c = new char;
 *c='a';


}
```

Most pont ugyanaz a memóriaterület került lefoglalásra, amire a pointerünk mutat.
Ez a foglalás az automatizmusra van bízva.

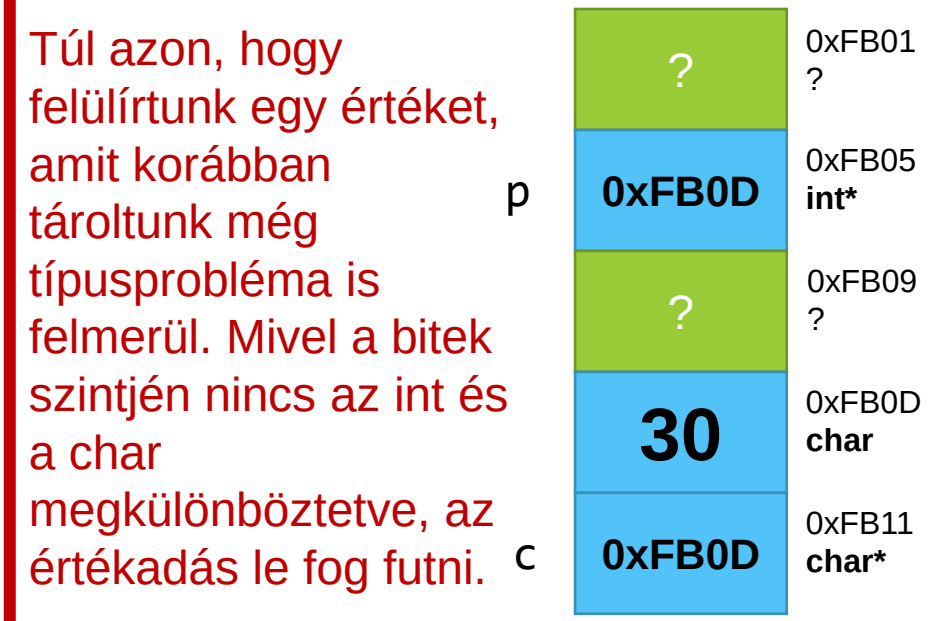


Dinamikus memóriakezelés – hibaforrások

Nézzük tovább!

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 char* c = new char;
 *c='a';
 *p = 30;

}
```

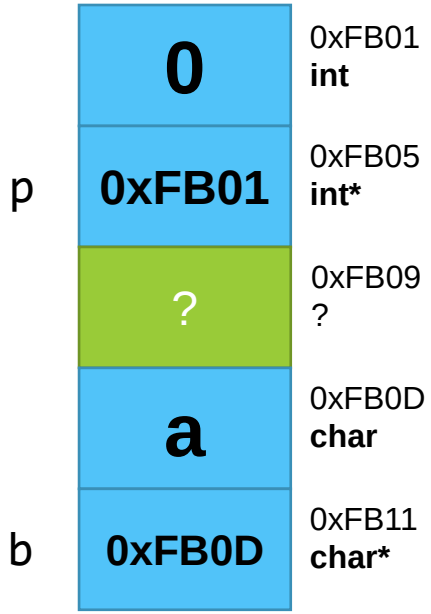
Túl azon, hogy felülírtunk egy értéket, amit korábban tároltunk még típusprobléma is felmerül. Mivel a bitek szintjén nincs az int és a char megkülönböztetve, az értékadás le fog futni.



Dinamikus memóriakezelés – hibaforrások

Folytassuk az előző, hibás értékadást figyelmen kívül hagyva!

```
{
 int* p;
 p = new int;
 *p = 10;
 delete p;
 char* c = new char;
 *c='a';
 p = new int;
 *p = 0;
}
```

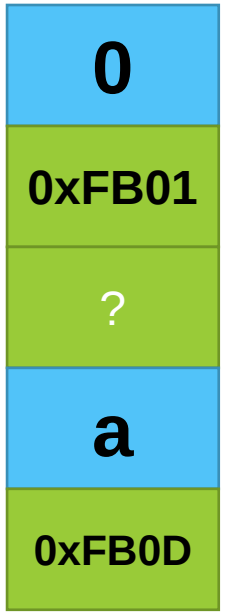


Dinamikus memóriakezelés – hibaforrások

Folytassuk!

```
{
  int* p;
  p = new int;
  *p = 10;
  delete p;
  char* c = new char;
  *c='a';
  p = new int;
  *p = 0;
}
```

Befejeződött a blokk. Az automatizmusra bízott változókhoz kapcsolódó memóriaterületek felszabadításra kerültek. Vegyük észre, hogy az 0xFB01 címen levő lefoglalt terület nincs felszabadítva és nincs érvényes pointer hozzá!



Dinamikus memóriakezelés – hibaforrások

Tanulságok

- Egy már felszabadított memóriaterülethez nem szabad a továbbiakban hozzáférnünk – hiszen felszabadítottuk
- A felszabadított memóriaterülethez a pointeren keresztül továbbra is hozzá lehet férni – mint minden egyéb memóriaterülethez, ezt semmi nem ellenőrzi
  - Legfeljebb az operációs rendszer szól közbe, illegális művelet miatt
  - Ennek ellenére meglehetősen rossz ötlet ilyen kódot írni, mivel ugyanazt a memóriaterületet így fel tudja használni a kód egy más része is
- Ha egy lefoglalt memóriaterületet nem szabadítunk fel és az egyetlen érvényes hivatkozást elveszítjük akkor
  - A memóriában szemetet hagyunk
  - Ezt a memóriaterületet az automatizmus nem találja meg, tehát a program lefutásának végéig nem szabadul fel és hozzáférni sem tudunk

Ökölszabály

- Amit lefoglalunk azt fel kell szabadítanunk: praktikusan a **`new`** és **`delete`** párban álljon valamilyen módon
- **`delete`** után tilos hozzáférni a memóriaterülethez, amit felszabadítottunk

Megoldás:

- A **`delete`** `p;` hívás után a `p` értékét `NULL`-ra állítjuk: `p=NULL;`
- *smart pointers – ezzel azonban nem foglalkozunk egyelőre.*

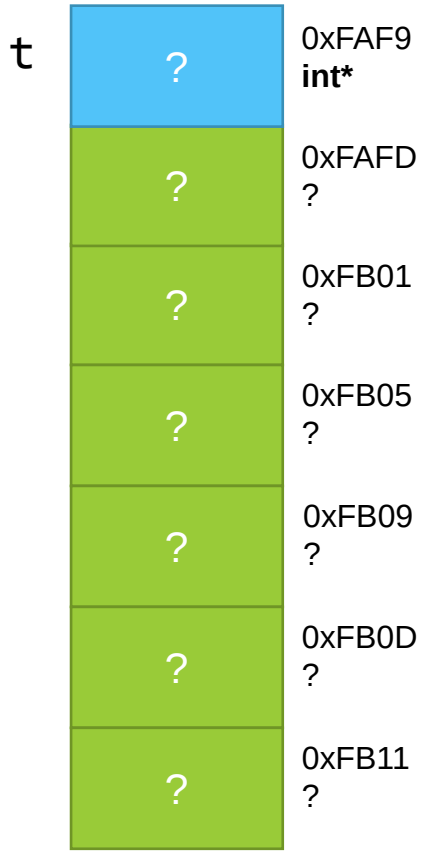
Dinamikus méretű tömbök

Láttuk, hogy

- a tömb változó ~ a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
- a dinamikus memóriakezelés pointerek segítségével történik

A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
t = new int[size];

delete[] t;
```



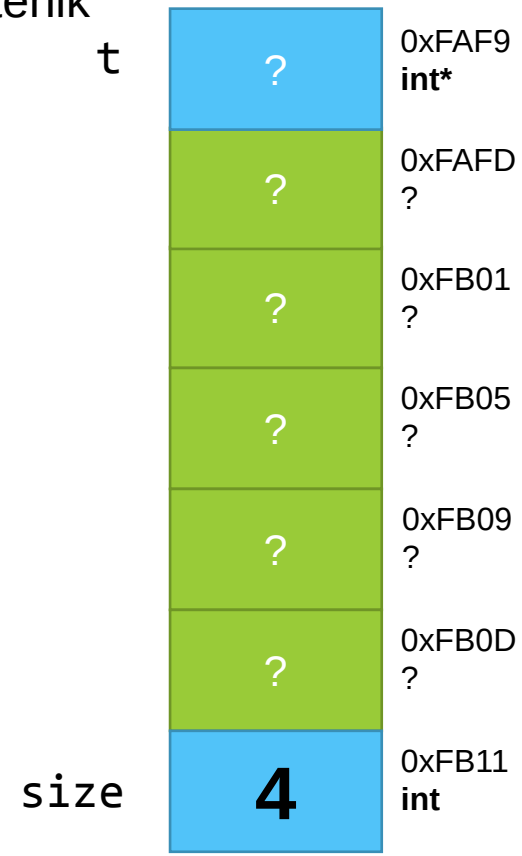
Dinamikus méretű tömbök

Láttuk, hogy

- a tömb változó a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
- a dinamikus memóriakezelés pointerek segítségével történik

A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
t = new int[size];

delete[] t;
```



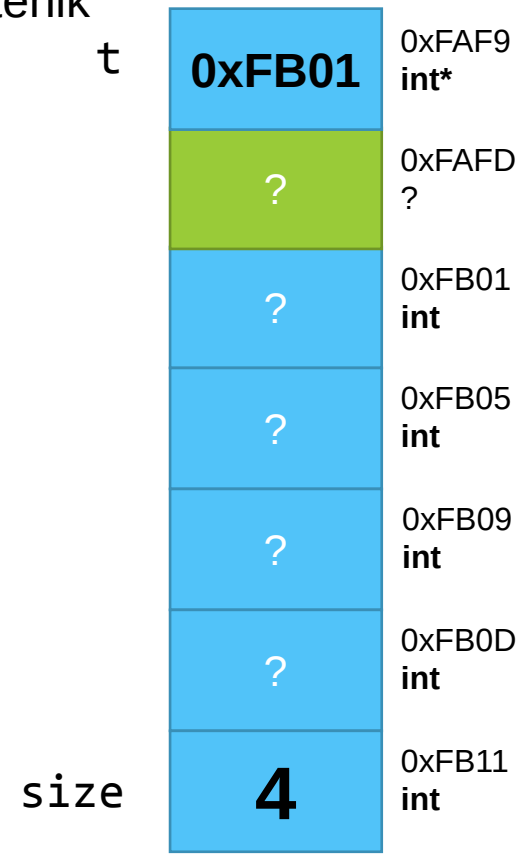
Dinamikus méretű tömbök

Láttuk, hogy

- a tömb változó a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
- a dinamikus memóriakezelés pointerek segítségével történik

A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
t = new int[size];

delete[] t;
```



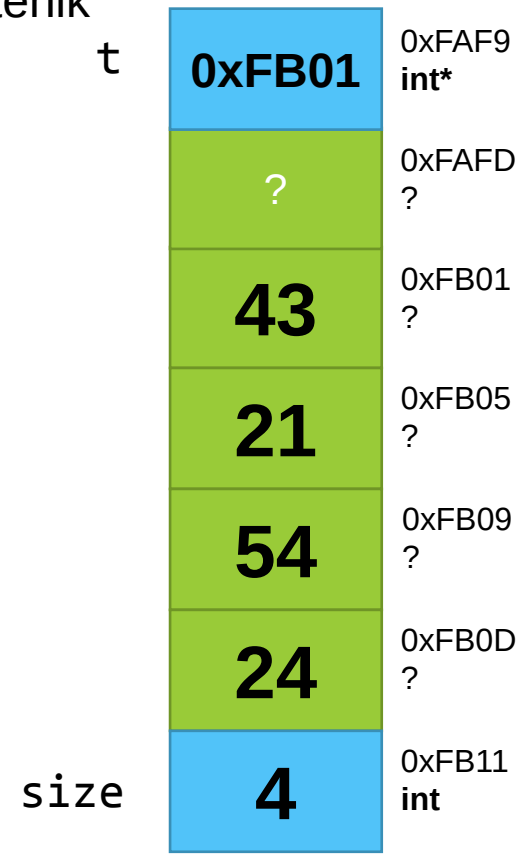
Dinamikus méretű tömbök

Láttuk, hogy

- a tömb változó a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
- a dinamikus memóriakezelés pointerek segítségével történik

A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
t = new int[size];

delete[] t;
```



Dinamikus méretű tömbök

Láttuk, hogy

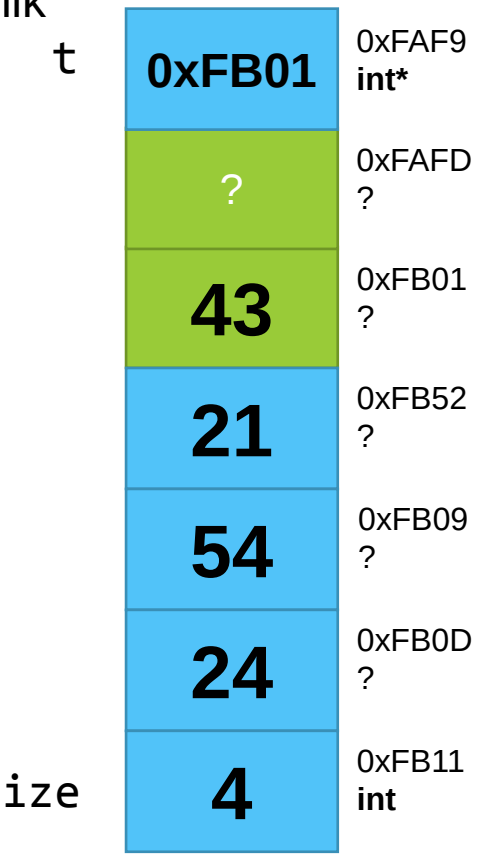
- a tömb változó a tömbként lefoglalt memóriaterület első pozíciójára mutató pointer
- a dinamikus memóriakezelés pointerek segítségével történik

A kettőt kapcsoljuk össze:

```
int* t;
int size;
cin >> size;
t = new int[size];
```

```
delete   t;
```

Változtassuk meg ezt a sort!
Ha (tévedésből) kihagyjuk a `[]`-t a **`delete`** kulcsszó mögül, akkor nem a tömb által elfoglalt memóriaterület kerül felszabadításra, csak az első elemhez tartozó memóriaterület.
Ez szintén memóriaszivárgáshoz vezet!



Próbáljuk ki! G01F01

Írjunk egy példaprogramot, amely

- Bekér egy számot a felhasználótól és ennek megfelelően létrehoz egy tömböt
- A tömbbe feltölti azokat a számokat, amelyeket a felhasználó megad
- Ezt követően a tömböt paraméterként át kell adni egy függvénynek, amely
  - Az átvett tömb értékeit lemásolja egy új tömbbe
  - A másolatba minden érték kétszerese kerüljön bele, kivéve, ha a kétszeres érték osztható 10-zel
  - A függvény gondoskodjon arról, hogy ne hagyjon maga után memóriaszemetet
- A visszakapott új tömb egy másik függvény inputja legyen, amely
  - Meghatározza a tömbben található számok átlagát
  - Az átlagértékkel térjen vissza a függvény
- A program ne hagyjon maga után memóriaszemetet

Memóriaszivárgás

Példát láttunk memóriaszivárgásra

- Amikor egy blokkból hiányzik a **`new`** és **`delete`** párosból a **`delete`**, tehát nem kerül felszabadításra a memória
- Amikor a **`delete`** `[]` helyett **`delete`** utasítás kerül kiadásra a tömb által elfoglalt memóriaterületből csak az első elem kerül felszabadításra

Ennél bonyolultabb esetek is előfordulhatnak

- Függvényhívások között is lehetséges, hogy elveszítjük az összes tárolt memóriacímet, amivel felszabadíthatjuk a lefoglalt memóriaterületet

Figyeljük meg a következő kódot …

Memóriaszivárgás

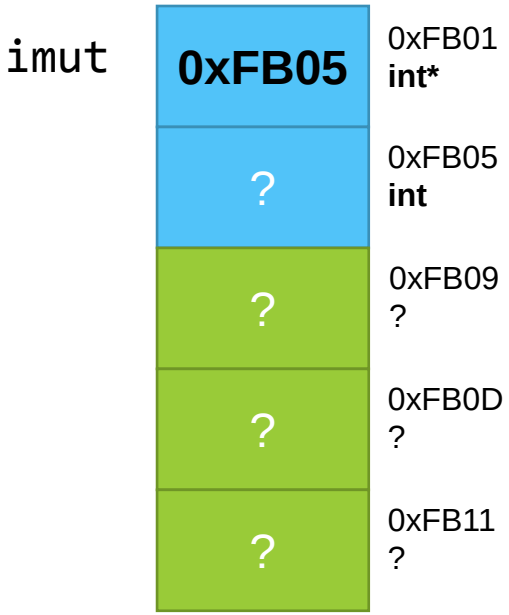
Tekintsük az alábbi kódot

```
int createInt()
{
   int* i = new int(5);
   return *i;
}

int* createInt2()
{
   return new int(3);
}

int main()
{
   int* imut = new int;
   *imut = createInt();
   imut = createInt2();
   delete imut;
   return 0;
}
```

Memóriaszivárgás

Tekintsük az alábbi kódot

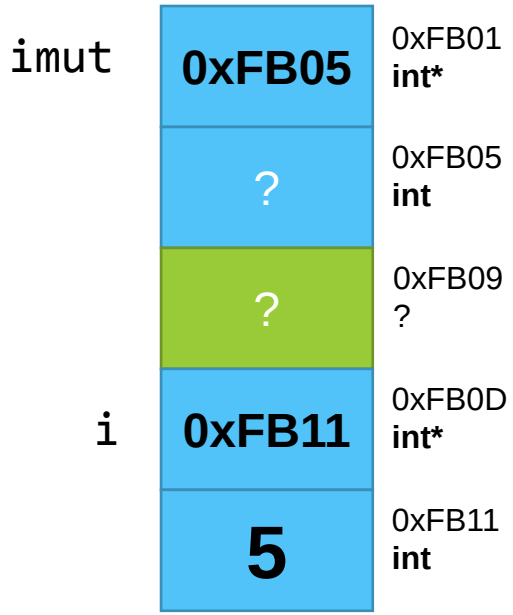
```
int createInt()
{
   int* i = new int(5);
   return *i;
}

int* createInt2()
{
   return new int(3);
}

int main()
{
   int* imut = new int;
   *imut = createInt();
   imut = createInt2();
   delete imut;
   return 0;
}
```



Memóriaszivárgás

Tekintsük az alábbi kódot

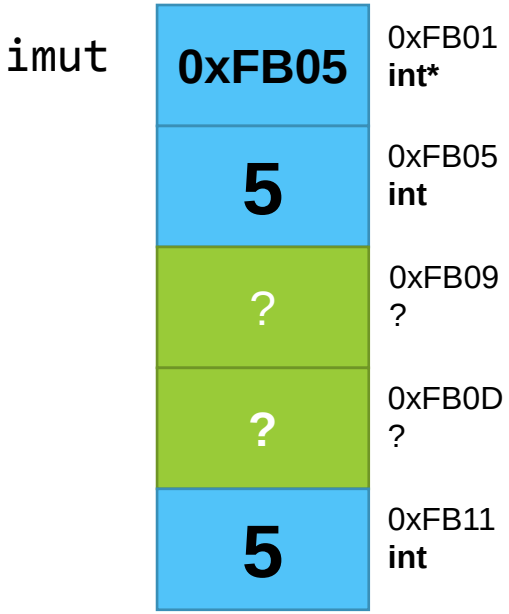
```
int createInt()
{
   int* i = new int(5);
   return *i;
}

int* createInt2()
{
   return new int(3);
}

int main()
{
   int* imut = new int;
   *imut = createInt();
   imut = createInt2();
   delete imut;
   return 0;
}
```



Memóriaszivárgás

Tekintsük az alábbi kódot

```
int createInt()
{
   int* i = new int(5);
   return *i;
}

int* createInt2()
{
   return new int(3);
}

int main()
{
   int* imut = new int;
   *imut = createInt();
   imut = createInt2();
   delete imut;
   return 0;
}
```

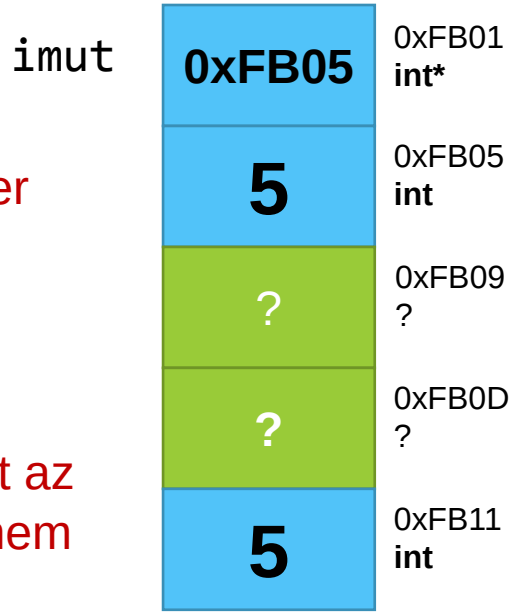


Memóriaszivárgás

Tekintsük az alábbi kódot

```
int createInt()
{
   int* i = new int(5);
   return *i;
}

int* createInt2()
{
   return new int(3);
}

int main()
{
   int* imut = new int;
   *imut = createInt();
   imut = createInt2();
   delete imut;
   return 0;
}
```

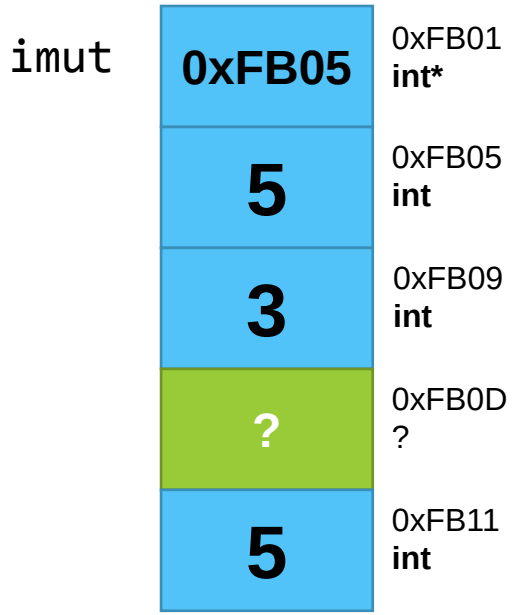
Befejeződött a függvény. Lefut a függvény és a létrehozott dinamikus változóhoz tartozó pointer automatikusan felszabadításra kerül. Vegyük észre, hogy a visszatérési típus nem **`int`***, hanem **`int`**. Tehát az értéket adjuk vissza és nem a memóriacímet.



Memóriaszivárgás

Tekintsük az alábbi kódot

```
int createInt()
{
   int* i = new int(5);
   return *i;
}

int* createInt2()
{
   return new int(3);
}

int main()
{
   int* imut = new int;
   *imut = createInt();
   imut = createInt2();
   delete imut;
   return 0;
}
```

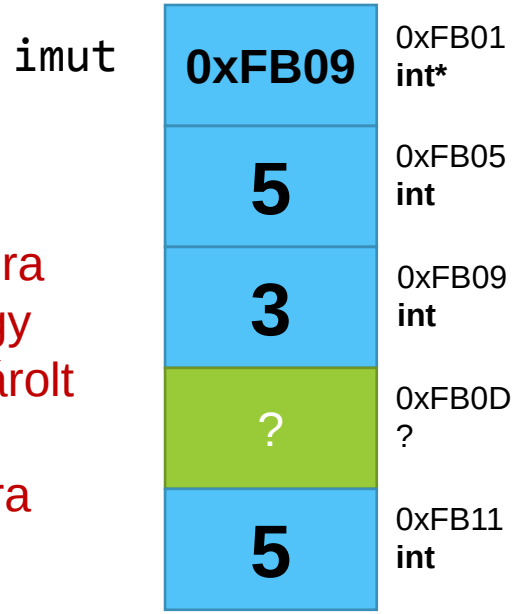


Memóriaszivárgás

Tekintsük az alábbi kódot

```
int createInt()
{
   int* i = new int(5);
   return *i;
}

int* createInt2()
{
   return new int(3);
}

int main()
{
   int* imut = new int;
   *imut = createInt();
   imut = createInt2();
   delete imut;
   return 0;
}
```

Befejeződött a második függvény.
Lefut a függvény és a létrehozott dinamikus változó címe visszaadásra kerül. Vegyük észre, hogy ez felülírja a korábban tárolt címet, azaz a `main()` elején lefoglalt memóriára vonatkozó cím elveszik.



Memóriaszivárgás

Tekintsük az alábbi kódot

```
int createInt()
{
   int* i = new int(5);
   return *i;
}

int* createInt2()
{
   return new int(3);
}

int main()
{
   int* imut = new int;
   *imut = createInt();
   imut = createInt2();
   delete imut;
   return 0;
}
```

A teljes lefutás után két helyen memóriacella sem került felszabadításra.

Mivel ez egy viszonylag egyszerű kód, három függvénnyel, ezért különösen fontos ügyelni a megfelelő visszatérési értékekre és a pointerek újrahasznosítására!

Próbáljuk ki!

Próbáljuk ki, hogy valójában mi történik:

- Ha egy inicializálatlan pointer mögötti memóriaterületet piszkálunk meg
- Ha egy törölt pointer mögötti memóriaterületet piszkálgatunk
- Ha egy (dinamikus) tömböt túlindexelünk
- Ha egy nagy (dinamikus) tömböt indexelünk túl
- Ha konstans értéket próbálunk megváltozatni
- Konstans érték címe létezik-e?
- Ha az `int &i=5;` utasítást adjuk ki.

Gyakorló feladat G01F03

Hozz létre egy n hosszú tömböt – dinamikus memóriafoglalással

- Töltsd bele az első n prímszámot

Másold le a tömböt (tényleges másolat)

- Emeld négyzetre a másolat elemeit, kivéve ha a szám háromjegyű
- Ha háromjegyű, akkor legyen a tömbben az új érték nulla

A másolatot ki kell írni a képernyőre.

Ezt követően írd ki a Pascal háromszög n-edik sorát

A program a futás során kezelje a memóriát helyesen!

Gyakorló feladat G01F04

Készíts egy olyan programot, amely

- Négyzetek és téglalapok adatainak tárolására alkalmas
  - Ehhez két tömböt kell használni, ahol a négyszögek két oldalának hosszát tároljuk el
- A két tömb méretét a program indulásakor a felhasználó adja meg, amelyet követően azokat dinamikusan kell létrehozni
- A tömbök értékekkel történő feltöltése után meg kell keresni:
  - A legnagyobb területű téglalapot
  - A legkisebb kerületű négyszöget
  - A megtalált síkidomok oldalainak hosszát és méretét ki kell írni
  - A két keresésre két függvényt kell írni, amelyek paraméterben kapják meg a tömböket
- Ügyelni kell arra, hogy ne legyen memóriaszivárgás és ne legyen felesleges memóriahasználat
- Legyen még egy függvény, amely paraméterként átveszi a tömböt és meghatározza a területértékek átlagát
  - Ezt ki is kell írni

Gyakorló feladat G01F05

Készíts egy programot, ami

- Képes tárolni egy összefüggő irányítatlan gráfot
- A gráf leírását tömbök tömbjével kell megoldani, dinamikus memóriakezeléssel
  - Az i-edik tömb j-edik értéke 1 vagy 0.
  - Ha 1, akkor az i és j pont között van él, ha 0, akkor nincsen
  - Ha az i-edik tömb j-edik értéke 1, akkor a j-edik tömb i-edik értéke is 1.
- A program indításkor kérdezze meg, hogy mennyi csúcsot szeretne tárolni a felhasználó
  - Ennek megfelelően hozza létre a tömböket és töltse fel 0 értékekkel
- Ezt követően legyen lehetőség élek felvételére
  - Természetesen ügyeljen a program, hogy érvényesek legyenek a bevitt értékek
- A program legyen képes eldönteni, hogy két tetszőleges csomópont között húzódik-e pontosan kettő hosszú út!

Házi feladat beadási konvenciók

Projektek elnevezése:

- <digitus>_hf_<hf dia sorszáma>

Kapcsolók legyenek beállítva!

Beadás SVN repositoryba:

- https://wsn.itk.ppke.hu/adatszerk_repo/<digitus_név>

Mindegyik házit a megfelelő mappába kell tenni

Házi feladat G01F02

Írj egy olyan programot, amely egy mátrixot tárol

- A mátrix méretét és az elemeket előre nem ismerjük
- Az adatokat egy „in.txt” fájlból kell beolvasni. A fájlban az első sorban a sorok száma (n) van, a második sorban az oszlopok száma (m), a fájl többi sorában (összesen még n×m sor van) pedig a beolvasandó értékek, sorfolytonosan.
- A mátrixot egy tömbben tároljuk el oszlopfolytonosan
  - Figyelem! Megváltozik a sorrend, az indexekre gondolni kell
- Írd meg az összeadás és szorzás műveleteket!
- Írj egy transzponálást végrehajtó függvényt!
- Írj egy kiíró műveletet, ami az elemeket áttekinthető formában megjeleníti!
- A főprogramon belül készíts menüt!
  - Egy ciklusban kérdezd meg a felhasználótól, hogy akar-e még műveletet végezni!
  - Ha igen, kérdezd meg, hogy hogy milyen műveletet!
  - Kérj be két mátrixot elemenként, majd végezd el a kért műveletet!
  - Az eredményt jelenítsd meg a képernyőn a kiíró műveletével!