Objektum orientált programozás C++ nyelven
3. Összetett adatszerkezetek

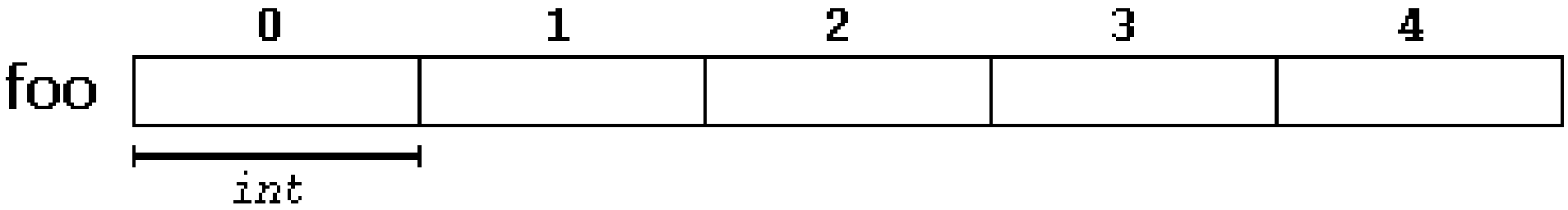
PEKÁRDY MILÁN – PANNON EGYETEM

PEKARDY@DCS.UNI-PANNON.HU

Tömbök

- Azonos típusú elemek sorozata, folytonos memória területen tárolva
- Az elemek közvetlenül elérhetők az indexük segítségével
- `type name [elements];`

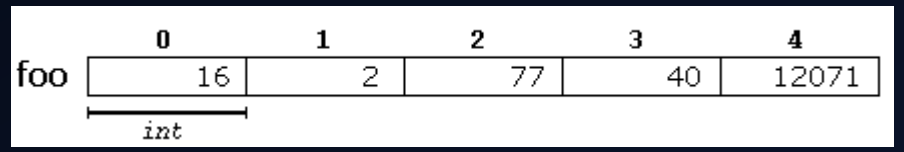
```
int foo [5];
```



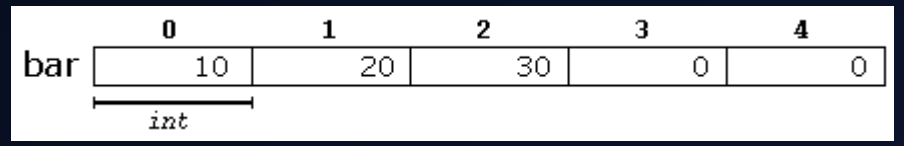
Tömbök - inicializáció

- Lokális hatókörben létrehozott tömb elemei nem inicializálódnak
- A tömb deklarálásakor az inicializációt is elvégezhetjük
- Stattikus tömbök és azok, amelyek közvetlenül névtérben (függvényen kívül) kerülnek deklarálásra mindig inicializálódnak a típusnak megfelelő alapértelmezett értékre

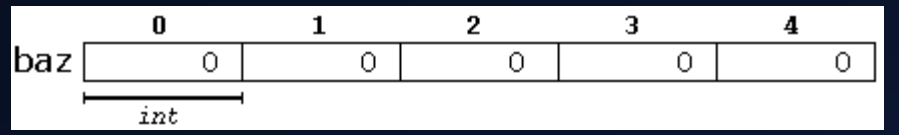
```
int foo [5] = { 16, 2, 77, 40, 12071 };
```




```
int bar [5] = { 10, 20, 30 };
```




```
int baz [5] = { };
```



- Méret megadása itt nem kötelező:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

- Universal initialization:

```
int foo[] { 10, 20, 30 };
```

Tömbök – elemek elérése

- A tömb elemeit a tömb nevének és az elem indexének a segítségével érhetjük el:
  - `name[index]`
  - Az indexelés 0-tól indul, méretnek nem megfelelő index megadása fordítási hibát nem okoz, azonban futás közbeni hibákat eredményezhet


```
foo [2] = 75;
x = foo[2];
```

- a []-nek két használati esetét különböztetjük meg:
  - Tömb deklarálása
  - Tömb elemének elérése

```
int foo[5];
foo[2] = 75;
```

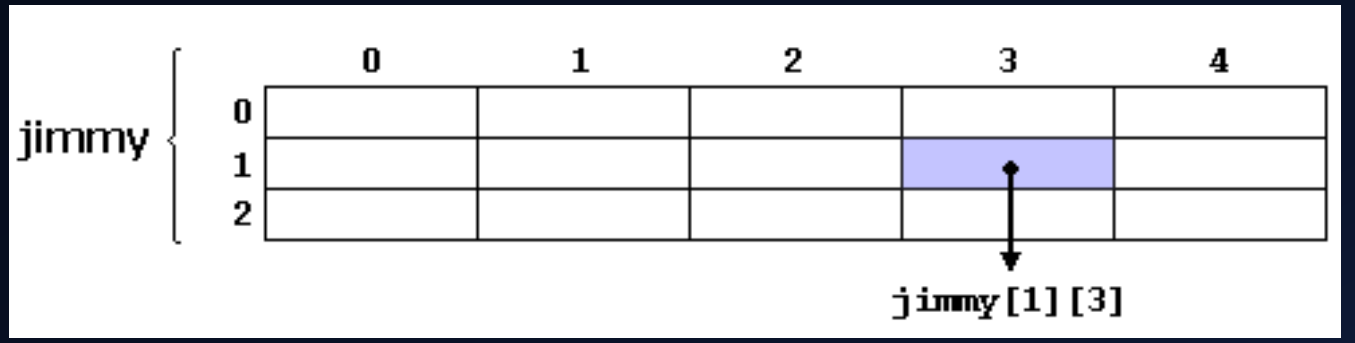
Tömbök – elemek elérése

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

```
foo[0] = a;
foo[a] = 75;
b = foo [a+2];
foo[foo[a]] = foo[2] + 5;
```

Tömbök – többdimenziós tömbök

- „tömbök tömbje” – „arrays of arrays”
- Pl.: két dimenziós tömb egy táblázatként képzelhető el
  - Deklaráció: int jimmy [3][5];
  - Elem elérése: jimmy[1][3]
- `char century [100][365][24][60][60]; //kb. 3GB memória`
- Többdimenziós tömb csak egy absztrakció




```
int jimmy [3][5];
int jimmy [15];
```

Tömbök – többdimenziós tömbök

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
}
```

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

Tömbök – tömb paraméter

- A teljes tömb átadása helyett csak a memória címét adjuk át (gyorsabb, hatékonyabb)
- Egydimenziós tömb esetén a méretet nem adjuk meg
- Több dimenzió esetén az első dimenzió méretén kívül a többi megadása szükséges a fordító számára
  - `base_type[][depth][depth]`

```
void procedure (int myarray[][3][4]);
```

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

Tömbök – tömb tároló

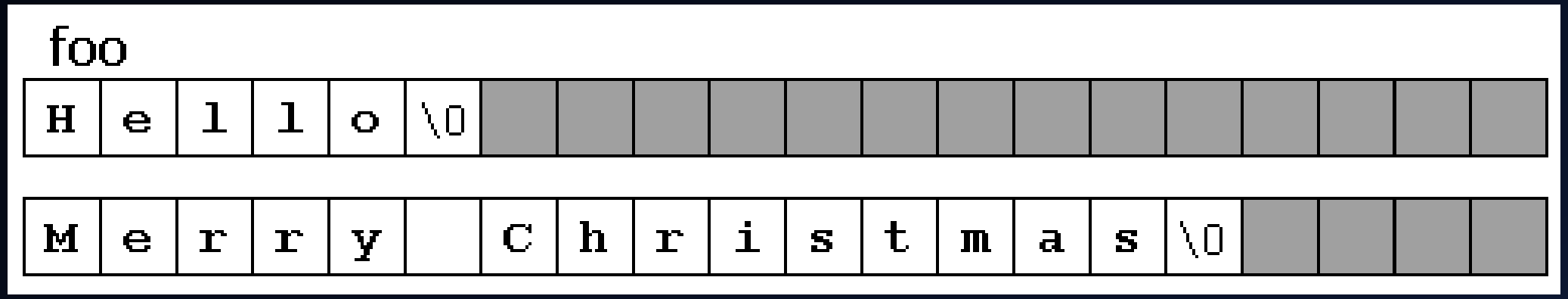
- A másolás lehetőségének korlátozása és a mutatók használatának nehézségei miatt a C++-ban létezik egy sablon tömb reprezentáció, továbbá a tömb mérete is elérhető így

```cpp
#include <iostream>

using namespace std;

int main()
{
  int myarray[3] = {10,20,30};

  for (int i=0; i<3; ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

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

int main()
{
  array<int,3> myarray {10,20,30};

  for (int i=0; i<myarray.size(); ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

Karakterláncok

- Szöveget reprezentálhatunk karaktertömbként is
  - `char foo [20];`
  - Konvenció szerint a karakter szekvenciákat a ’\0’ karakter zárja le



Karakterláncok - inicializáció

- A tömböknél bemutatott módokon inicializálhatók a karakter szekvenciák is
- Közvetlenül sztring literálokkal is inicializálhatunk (ebben az esetben a null karakter automatikusan bekerül a végére)

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword[] = "Hello";
```

- A tömbök korlátozásai miatt deklarálás után, másik utasításban nem rendelhetünk a változóhoz értéket. Az alábbi utasítások érvénytelenek:

```
myword = "Bye";
myword[] = "Bye";
myword = { 'B', 'y', 'e', '\0' };
```

- A tömbökhöz hasonlóan az egyes elemek elérhetők az indexükkel:

```
myword[0] = 'B';
myword[1] = 'y';
myword[2] = 'e';
myword[3] = '\0';
```

Karakterláncok és sztringek

- C++-ban bevezetésre került a `string` típus
- A programokban mind a két megközelítés használható
- A karakter szekvenciák implicit átalakíthatók sztringre, illetve a sztringek is karakter szekvenciára

```
// strings and NTCS:
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  char question1[] = "What is your name? ";
  string question2 = "Where do you live? ";
  char answer1 [80];
  string answer2;
  cout << question1;
  cin >> answer1;
  cout << question2;
  cin >> answer2;
  cout << "Hello, " << answer1;
  cout << " from " << answer2 << "!\n";
  return 0;
}
```

```
char myntcs[] = "some text";
string mystring = myntcs;  // convert c-string to string
cout << mystring;          // printed as a library string
cout << mystring.c_str();  // printed as a c-string
```

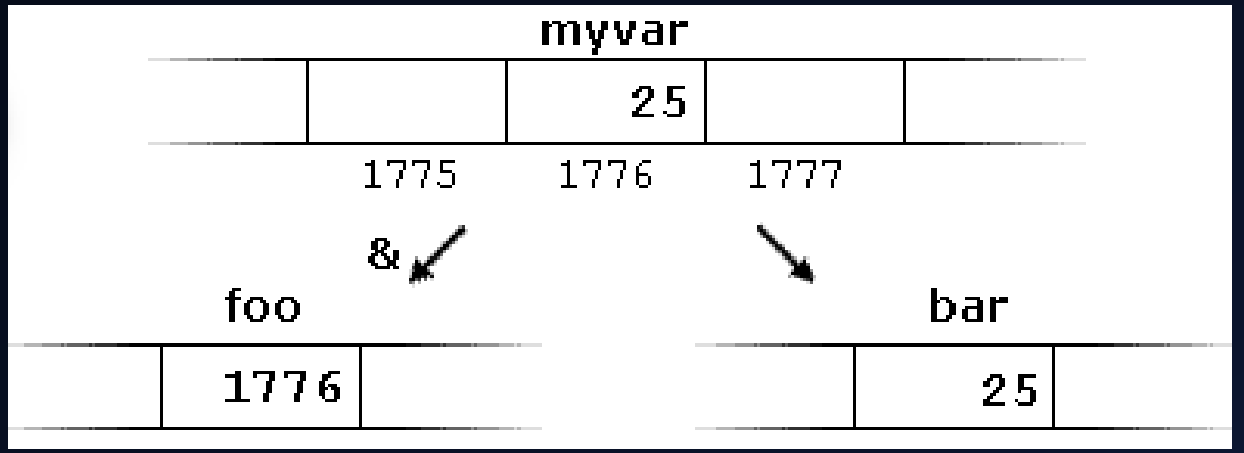
Mutatók (pointers)

- Az eddigi példákban a különböző változók olyan memória területeket jelentettek, amelyekre a nevükkel hivatkozhattunk
- A program számára a memória egymást követő memória cellákból épül fel, mindegyik 1 byte méretű és egyedi címmel rendelkezik
- Minden memória cella beazonosítható az egyedi címe alapján
- Amikor egy változót deklarálunk, akkor a változó számára szükséges memória lefoglalásra kerül egy meghatározott memória területen
- A memória-kezelés részletei a futtató környezetre vannak bízva (pl. az operációs rendszer dönti el, hogy pontosan melyik blokkokat foglalja le a program a változók számára)
- Futásidőben elérhetjük az egyes változók konkrét memóriacímét a mutatók segítségével

Mutatók – címképző operátor (address-of)

- Egy változó címét neve elé írt címképző operátor (&) segítségével kérhetjük le
- A változót, amiben a címet tároljuk mutatónak nevezzük

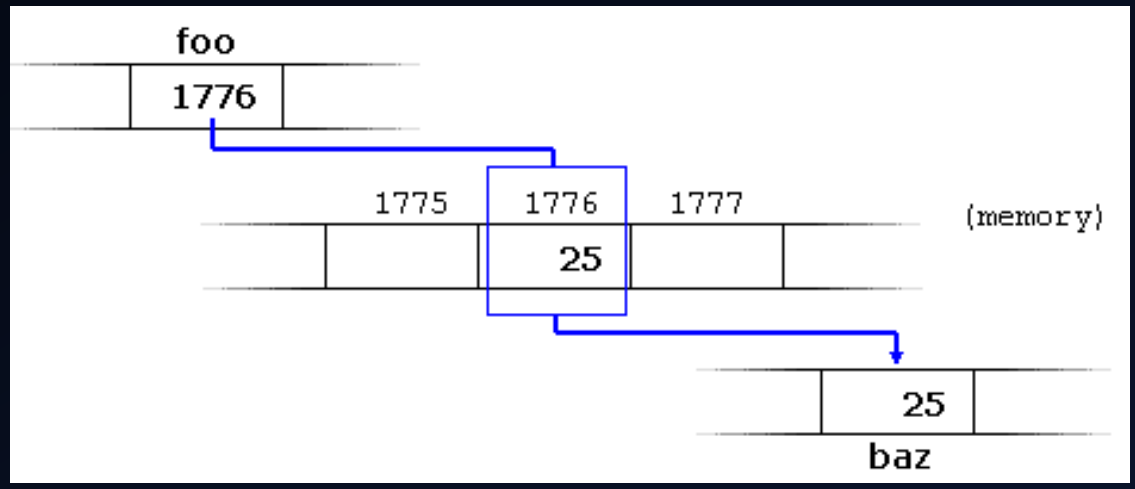
```
myvar = 25;
foo = &myvar;
bar = myvar;
```



Mutatók – dereferencia (dereference) operátor

- A mutató segítségével elérhető a mutatott változó értéke (value pointed to by) a * operátor segítségével

```
baz = *foo;
```



- A címképző és a dereferencia operátorok komplementer jelentéssel bírnak: a címképző operátorral lekért cím által mutatott érték a dereferencia operátorral olvasható ki

Mutatók – mutató deklarálása

- A mutató deklarálásakor meg kell adni a mutatott érték típusát
  - `type * name;`

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

Mutatók – mutatók deklarálása

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed to by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

Mutatók – mutatók és tömbök

- Egy tömb mindig implicit konvertálható egy megfelelő típusú mutatóra, ilyenkor a mutató a tömb első elemére mutat
- A mutatók beállíthatók másik címre, a tömbök viszont nem
- A mutatókkal is végezhetünk aritmetikai műveleteket

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

Mutatók - inicializáció

- A mutatók a definiálással egy időben inicializálhatók is

```
int myvar;
int * myptr = &myvar;
```

A mutatók inicializálhatók egy változó címére vagy közvetlenül egy másik mutató értékére

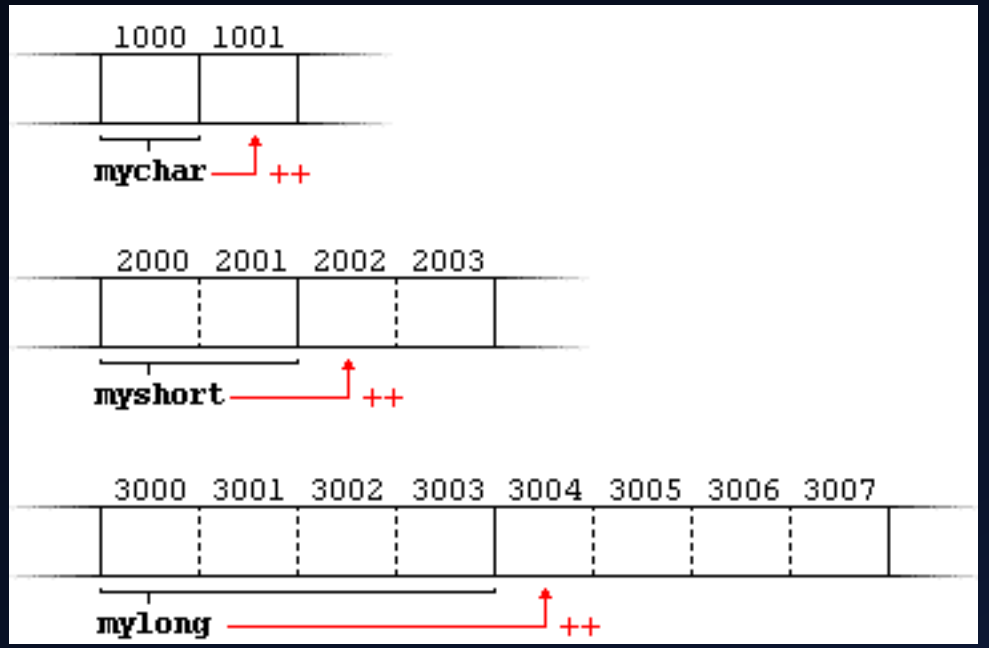
```
int myvar;
int *foo = &myvar;
int *bar = foo;
```

Mutatók – mutató aritmetika

- A mutatókkal csak összeadás és kivonás műveleteket végezhetünk, a többi nem értelmezett
- Mindkét művelet viselkedése a mutató által mutatott változó típusának méretétől függ, a mérettől függően tudjuk a címet növelni vagy csökkenteni (a következő vagy megelőző adott típusú elemre módosul a mutató)

```
char *mychar;
short *myshort;
long *mylong;
```

```
++mychar;
++myshort;
++mylong;
```



Mutatók – mutató aritmetika

- A műveletek során figyelni kell az operátor precedenciára
- A postfix (pl.: increment, decrement) operátoroknak nagyobb a precendenciájuk mint a prefix operátoroknak (pl. dereferencia)
- Javasolt a megfelelő zárójelezésekkel egyértelműsíteni a kifejezéseket

```
*p++   // same as *(p++): increment pointer, and dereference unincremented address
*++p   // same as *(++p): increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

Mutatók – mutatók és konstansok

- A mutatón keresztül elérhető és módosítható a mutatott érték
- Előfordulhat olyan helyzet, amikor csak olvasni engedlyük az értéket, de módosítani nem, ilyenkor használhatjuk a `const` kulcsszót

```
int x;
int y = 10;
const int * p = &y;
x = *p;          // ok: reading p
*p = x;          // error: modifying p, which is const-qualified
```

Mutatók – mutatók és konstansok

- Konstans elemekre mutató mutatók jól használhatók függvény paraméterek esetében:
  - nem konstans esetben a függvény módosíthatja az értéket
  - konstans esetben viszont nem, így elkerülhetünk nem kívánt mellékhatásokat
  - (persze van olyan helyzet, amikor pont azt szeretnénk, hogy módosítson a függvény…)
- A mutatók ebben az esetben továbbra is módosíthatók

```
// pointers as arguments:
#include <iostream>
using namespace std;

void increment_all (int* start, int* stop)
{
  int * current = start;
  while (current != stop) {
    ++(*current);  // increment value pointed
    ++current;     // increment pointer
  }
}

void print_all (const int* start, const int* stop)
{
  const int * current = start;
  while (current != stop) {
    cout << *current << '\n';
    ++current;     // increment pointer
  }
}

int main ()
{
  int numbers[] = {10,20,30};
  increment_all (numbers,numbers+3);
  print_all (numbers,numbers+3);
  return 0;
}
```

Mutatók – mutatók és konstansok

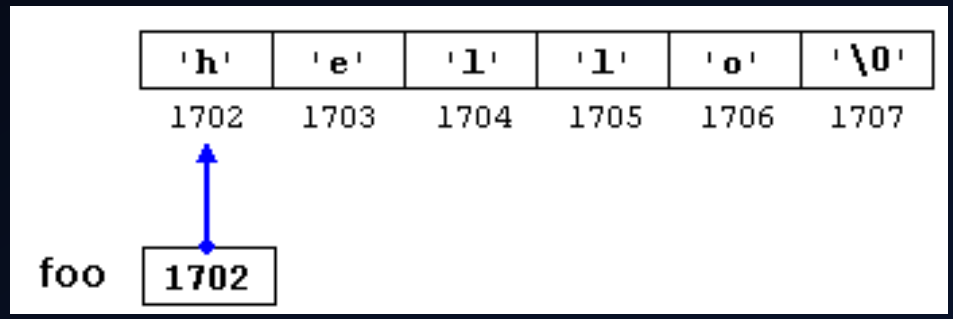
- Konstansok segítségével elérhetjük azt is, hogy se a mutatót se a mutatott értéket ne tudjuk módosítani

```
int x;
      int *       p1 = &x;  // non-const pointer to non-const int
const int *       p2 = &x;  // non-const pointer to const int
      int * const p3 = &x;  // const pointer to non-const int
const int * const p4 = &x;  // const pointer to const int
```

Mutatók – mutatók és sztring literálok

- A sztring literálok null-terminált karakter szekvenciák
- A sztring literál egy megfelelő típusú tömb, ami a karaktereket tartalmazza, mindegyik elem típusa `const char`, mivel a literálok nem módosíthatók

```
const char * foo = "hello";
```



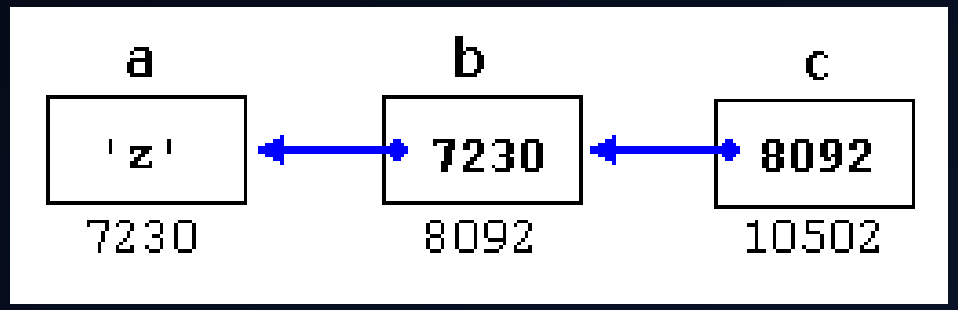
- A mutató ilyenkor karakterek szekvenciájára mutat és a segítségével elérhetők a karakterek hasonlóan, mint a karaktertömbök esetén

```
*(foo+4);
foo[4];
```

Mutatók – mutatók mutatókra

- Lehetőség van olyan mutatókat is kezelni, amik másik mutatókra mutatnak, ilyenkor a mutató deklarációban kell növelni a *-ok számát

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



- A példában a c változó három szinten használható:
  - c – típusa char**, értéke 8092, ami a b változó címe
  - *c – típusa char*, értéke 7230, ami a b változó értéke, ami az a változó címe
  - **c – típusa char, értéke ’z’, ami az a változó értéke

Mutatók – `void` mutatók

- void* típusú mutató speciális tulajdonságokkal rendelkezik:
  - void a típus hiányát jelenti
  - void* olyan értékre mutat, aminek nincs típusa (nem meghatározott a mérete és a dereferencia tulajdonságai sem)
- az ilyen mutatók tetszőleges típusú változókra mutathatnak
- hátránya, hogy mivel nem ismert a típus így közvetlenül nem működik a dereferencia, explicit konvertálni kell a megfelelő típusra a mutatót

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

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

Mutatók – érvénytelen és null mutatók

- a gyakorlatban a mutatók tetszőleges memória címre mutathatnak, olyanokra is, amik esetleg nem tartalmaznak értelmes értéket
- ilyenek lehetnek a nem inicializált mutatók, illetve egy tömb nem létező elemére mutató mutatók (pl. a tömb méretein kívül eső)

```
int * p;                    // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;   // element out of bounds
```

- az ilyen mutatók fordítási hibát nem okoznak, azonban futás közben nem várt viselkedés, illetve hibák jöhetnek elő (főleg, amikor dereferencia operátort alkalmaznánk)
- előfordul olyan eset, amikor olyan mutatóra van szükségünk, ami explicit nem mutat sehova, ilyenkor alkalmazhatjuk a null mutatókat (null pointer)

```
int * p = 0;
int * q = nullptr;
int * r = NULL;
```

Mutatók – függvény mutatók

- lehetőség van függvényekre mutató mutatók létrehozására
- ez akkor hasznos ha egy másik függvénynek paraméterként szeretnénk átadni egy függvényt
- a deklaráció hasonló a sima függvényekhez, annyi különbséggel, hogy ebben az esetben a név zárójelek között van és megelőzi egy *

```
// pointer to functions
#include <iostream>
using namespace std;

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

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

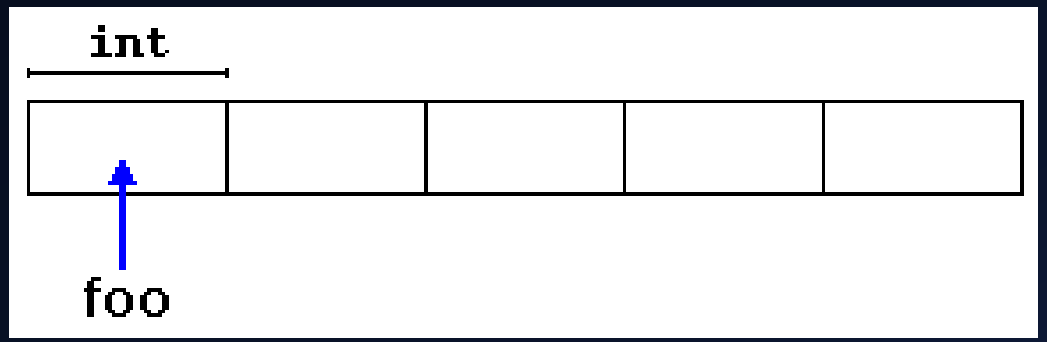
Dinamikus memória

- eddig memóriát egyszerű változókon keresztül használtunk, ami fordítási időben ismert volt
- egyéb esetekben szükségünk van dinamikus memória foglalási megoldásokra is, hogy futásidőben tudjuk a szükséges memóriát lefoglalni és használni
- pl. a dinamikusan lefoglalandó memória mérete egy a felhasználó által bevitt értéktől függ
- a C++-ban elsősorban a `new` és `delete` kulcsszavak segítségével kezeljük a dinamikus memóriát

Dinamikus memória – new, new[ ]

- dinamikus memória lefoglalása a new kulcsszó segítségével lehetséges, meg kell adni utána a típust és ha elemek sorozata számára foglalunk (tömb) memóriát akkor a []-t is
  - `pointer = new type`
  - `pointer = new type [number_of_elements]`
- az utasítás a lefoglalt memória kezdetére mutató mutatóval tér vissza

```
int * foo;
foo = new int [5];
```



Dinamikus memória – new, new[ ]

- a dinamikus tömb foglalás esetén a méretnek nem szükséges konstans kifejezésnek lenni, a program futása közben határozhatjuk meg dinamikusan, hogy mekkora legyen a mérete
- további különbség, hogy dinamikus esetben a memória a program számára elérhető memória kupacról (memory heap) foglalódik, a memória véges erőforrás, előfordulhat, hogy a foglalás nem sikerül:
  - kivételt dob a program, amit kezelhetünk (ez az ajánlott módszer, a kivételekről később lesz szó),
  - nothrow kulcsszó használatával nem dob kivételt a program, hanem null mutatóval tér vissza a new

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

Dinamikus memória – `delete`, `delete[]`

- a dinamikusan lefoglalt memória felszabadítására használhatjuk a `delete` kulcsszót
  - `delete pointer;` - egy a new-val lefoglalt elemet szabadít fel
  - `delete[] pointer;` - egy a new-val és `[]`-el foglalt elemek tömbjét szabadítja fel
- a `delete` paramétere vagy egy mutató, amit egy new utasítás adott vissza vagy egy null mutató

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

Adat struktúrák (data structures)

- Egy struktúra különböző adat elemek csoportja, amit egy névvel jelölünk
- a struktúra elemei (members) lehetnek különböző típusúak és méretűek
  - type_name: a struktúra típus neve
  - member_type: adattag típusa
  - member_name: adattag neve
  - object_name: struktúra típusú változók nevei (opcionális)

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

```
struct product {
  int weight;
  double price;
} ;

product apple;
product banana, melon;
```

```
struct product {
  int weight;
  double price;
} apple, banana, melon;
```

Adat struktúrák

- egy struktúra típushoz létrehozhatunk több objektumot is
- a létrehozott adott struktúra típusú változók (objektumok) segítségével elérhetők az egyes adattagok

```
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
  string mystr;

  mine.title = "2001 A Space Odyssey";
  mine.year = 1968;

  cout << "Enter title: ";
  getline (cin,yours.title);
  cout << "Enter year: ";
  getline (cin,mystr);
  stringstream(mystr) >> yours.year;

  cout << "My favorite movie is:\n ";
  printmovie (mine);
  cout << "And yours is:\n ";
  printmovie (yours);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

```
// array of structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} films [3];

void printmovie (movies_t movie);

int main ()
{
  string mystr;
  int n;

  for (n=0; n<3; n++)
  {
    cout << "Enter title: ";
    getline (cin,films[n].title);
    cout << "Enter year: ";
    getline (cin,mystr);
    stringstream(mystr) >> films[n].year;
  }

  cout << "\nYou have entered these movies:\n";
  for (n=0; n<3; n++)
    printmovie (films[n]);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

Adat struktúrák – mutatók struktúrákra

- a struktúra típusokra is hozhatók létre mutatók a megszokott módon
- struktúra típusú változókra mutató mutatókkal a . helyett a -> operátort használhatjuk az adattagok elérésére

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed to by a	(*a).b
*a.b	Value pointed to by member b of object a	*(a.b)

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";

  return 0;
}
```

Adat struktúrák – struktúrák egymásba ágyazása

- struktúrák tartalmazhatnak más struktúrákat is
- a beágyazott struktúrák adattagjainak az elérése az előbbi szintaktika láncolásával oldható meg

```
struct movies_t {
  string title;
  int year;
};

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
} charlie, maria;

friends_t * pfriends = &charlie;

charlie.name;
maria.favorite_movie.title;
charlie.favorite_movie.year;
pfriends->favorite_movie.year;
```

Egyéb adat típusok – típus aliaszok (type aliases)

- egy típusnak adhatunk egy másik nevet, amivel beazonosítható
- régebbi C szintaktika:

```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

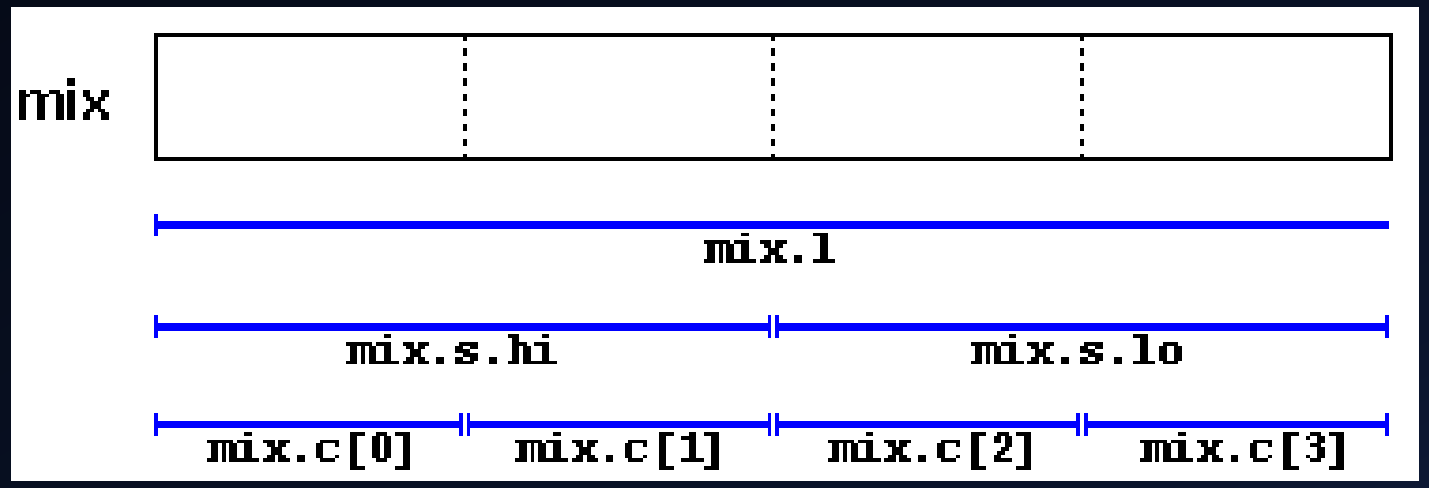
- újabb C++ szintaktika:

```
using C = char;
using WORD = unsigned int;
using pChar = char *;
using field = char [50];
```

Egyéb adat típusok - union

- az union típus lehetővé teszi, hogy egy memória területet különböző adat típusokon keresztül érjünk el, a deklaráció a struktúrákhoz hasonló, de funkcionálisan teljesen különböznek
- egy union mérete mindig a legnagyobb méretű adattagjának a méretével egyezik meg

```
union mix_t {
  int l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```



Egyéb adat típusok - enum

- enum típus (enumerated type) egyedi azonosítókat definiál, mint lehetséges értékek (enumerators), az ilyen típusokból létrejött változók ezeket az értékeket vehetik csak fel
- a lehetséges értékek implict konvertálhatók int típusra
- ha nincs másképp megadva, akkor az alapértelmezett 0 értékről indul, de explicit megadhatunk más értékeket is

```
enum type_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

```
mycolor = blue;
if (mycolor == green) mycolor = red;
```

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```