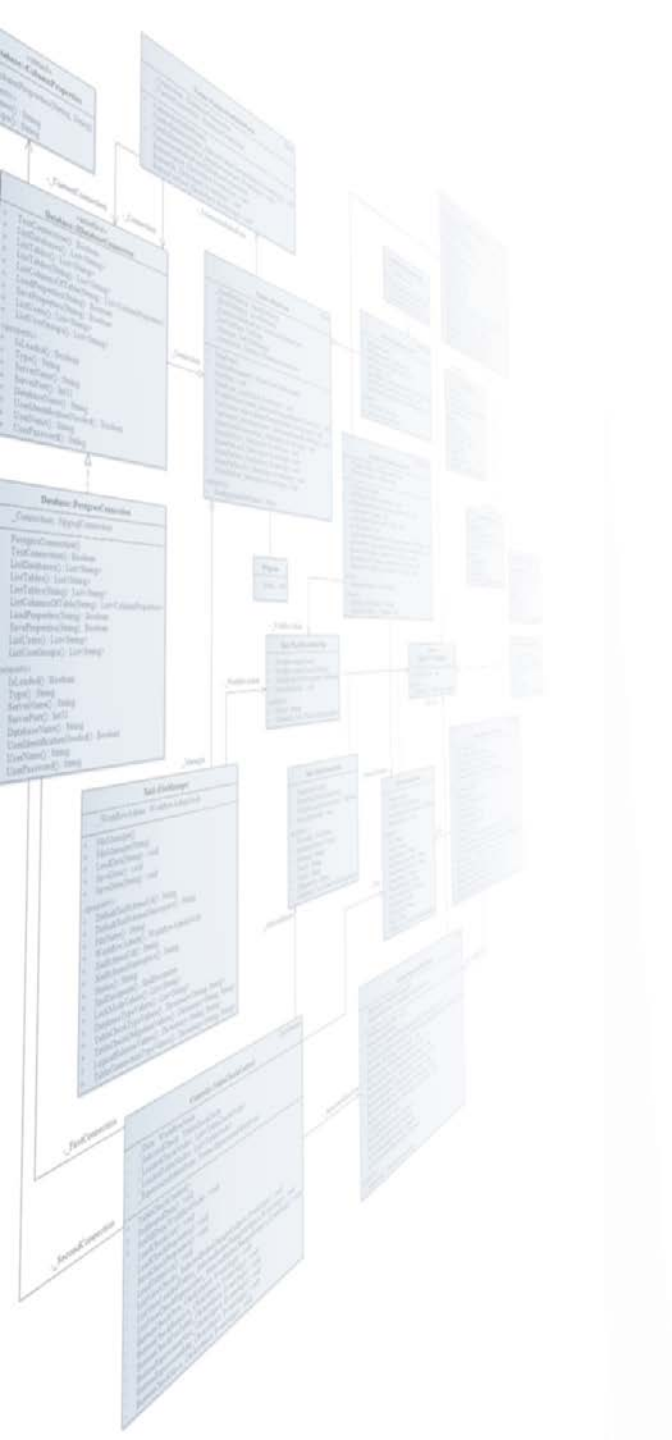


Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

Bevezetés a Programozásba II

10. előadás

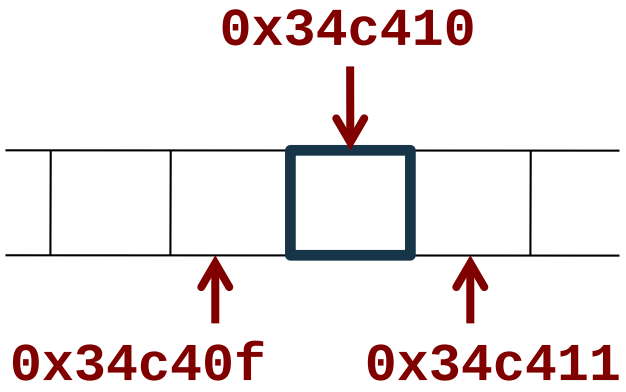
Dinamikus memóriakezelés

© 2014.04.28. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Dinamikus memóriakezelés

Memóriaszegmensek

- Az operációs rendszer minden futó program számára fenntart egy területet a memóriából, ezt nevezzük *memóriaszegmens*nek
  - minden program a saját szegmensében dolgozik, a szegmens mérete változhat futás közben
  - minden szegmensbeli memóriahely (bájt) rendelkezik egy sorszámmal, ez a *szegmensbeli memóriacíme*, amelyen keresztül elérhető a programban
- A memória(szegmens) tekinthető egy vektornak, a memóriacím pedig annak egy indexe (általában hexadecimálisan adjuk meg, pl. **`0x34c410`**)



Dinamikus memóriakezelés

Memóriacím lekérdezése

- Minden változó rendelkezik memóriacímmel
  - ezt C++-ban hasonlóan kezelhetjük, mint magát a változót
  - mivel a változó típusától függően több bájton is tárolódhat, mindig csak az első bájt címét kapjuk vissza
- Egy változó memóriacímét az **&** operátorral kérdezhetünk le, ez a *referenciaoperátor*, **&<*változónév*>** a változó első bájtjának memóriabeli címe
- Pl.:

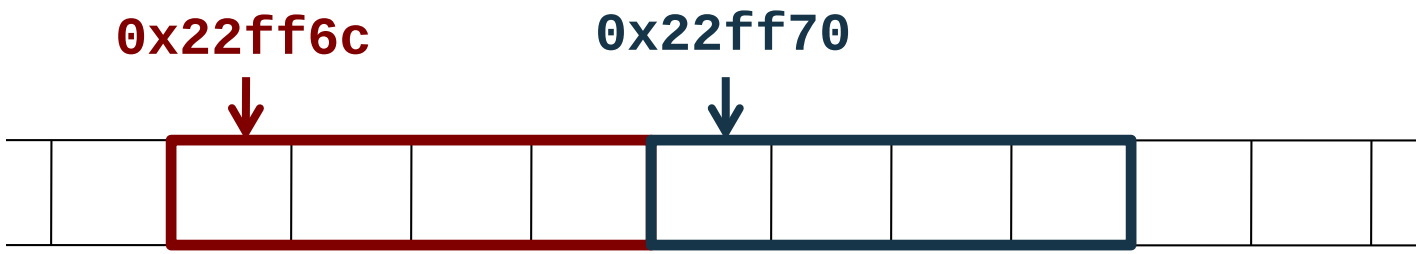
```
int i = 128;
cout << i << " " << &i;
// lehetséges eredmény: 128 0x22ff6c
```

Dinamikus memóriakezelés

Műveletek memóriacímekkel

- Lehetőségünk van a memóriában történő „ugrásra”
  - a memóriacímet számként kezelhetjük, növelhetjük, illetve csökkenthetjük (a **+**, **-**, **++**, **--** operátorokkal)
  - azonban egy egyszeri növelés esetén a címérték nem eggyel fog nőni, hanem a következő változó címét adja vissza
- Pl.:

```
cout << i << ” ” << &i << ” ” << &i+1;
// lehetséges eredmény: 128 0x22ff6c 0x22ff70
```



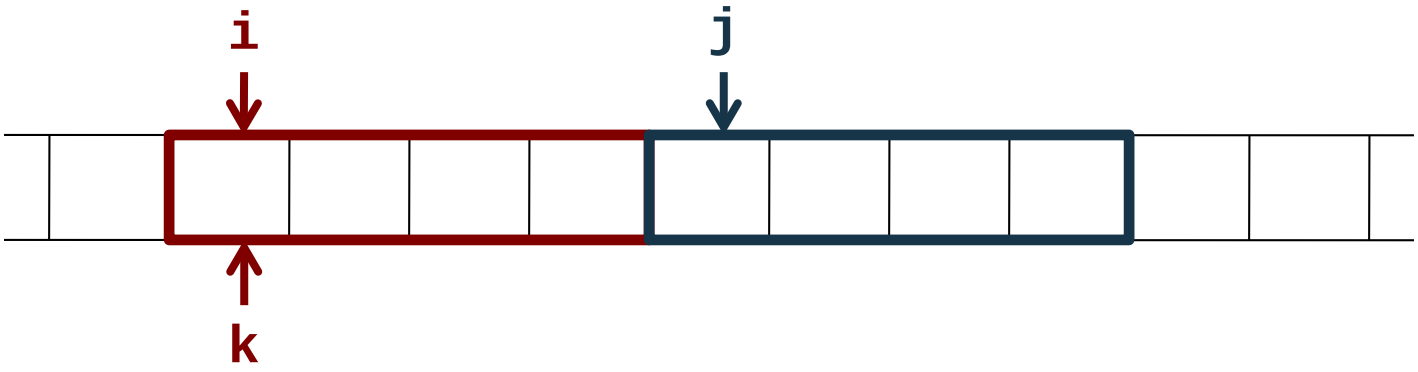
Dinamikus memóriakezelés

Referencia változók

- A *referencia változók* (vagy álnevek) olyan változók, amelyek nem a változók értékét, hanem memóriacíműket másolják át, így ugyanarra a területre mutatnak a memóriában

- Pl.:

```
int i = 128;
int j = i; // egyszerű változó
int& k = i; // referencia változó
```



Dinamikus memóriakezelés

Mutatók deklarálása

- Egy még speciálisabb változótípus a *mutató* (*pointer*), amely memóriacímet tárol értékként
  - mutató létrehozásával egy új adatot viszünk a memóriába, amely másik adat memóriacímét tartalmazza
  - általánosabb célú, mint a referencia
- A mutató létrehozásakor meg kell adnunk, milyen típusú változó címét fogja eltárolni, és ez onnantól nem változtatható
  - egy típushoz a hozzá tartozó mutató típus a **`<típusnév>*`**
  - mutató létrehozása: **`<típus> *<mutatónév>;`**
  - pl.: `int* ip; // egy int-re mutató pointer`

Dinamikus memóriakezelés

Mutatók használata

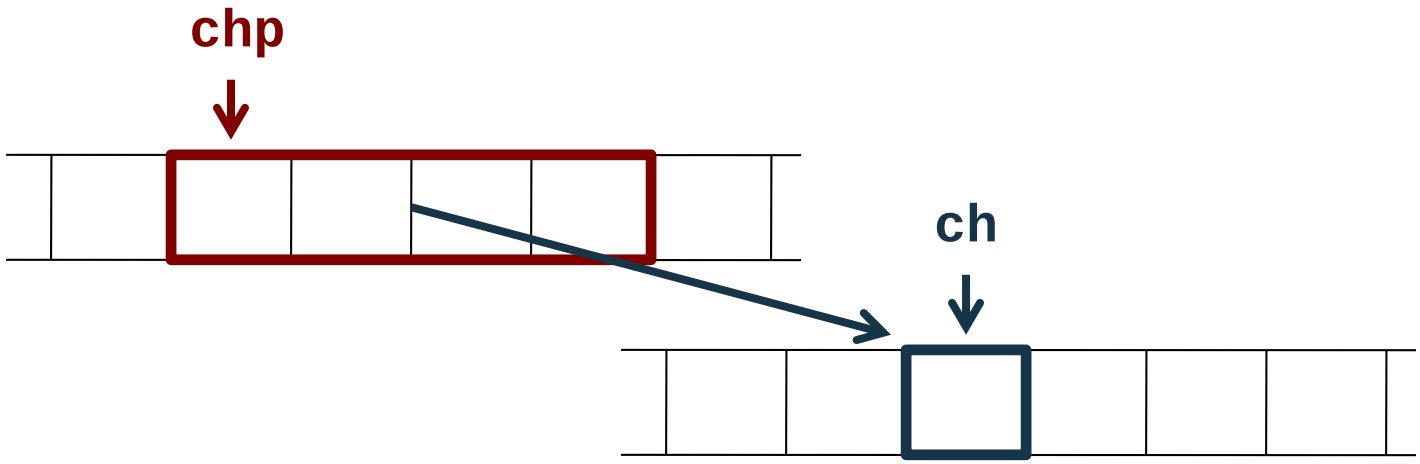
- A mutatók hasonlóan viselkednek, mint más változóink
  - értéket adhatunk nekik, élettartammal rendelkeznek
  - nem kell nekik adni kezdőértéket, ekkor egy véletlenszerű címet tartalmaznak kezdetben
  - az értéküket lehet növelni, csökkenteni (**+**, **-**, **++**, **--**), ekkor a megfelelő memóriacímbeli objektumra ugranak
  - mutatókat nem csak egyértékű változókra, hanem tömbökre, függvényekre, vagy bármilyen típusra állíthatunk
- A mutató mérete rögzített minden típusra, 32 bites architektúrában 4 byte (emiatt csak 4GB memória címezhető meg), 64 bites architektúrában 8 byte

Dinamikus memóriakezelés

Mutatók használata

- Mutató értékadására használhatjuk a referencia operátort, így ráállíthatjuk egy már létező változó memóriacímére, pl.:

```
char ch = 'a';
char *chp = &ch;
// lekérdezzük ch memóriacímét, amit értékül
// adunk chp mutatónak, innentől rámutat
```



Dinamikus memóriakezelés

Mutatók lekérdezése

- Amikor mutatók értékét kezeljük, akkor egy memóriacímet kapunk, ha az általa mutatott változó értékére vagyunk kíváncsiak, akkor ismét használnunk kell a * operátort, pl.:

```
char ch = 'a', *chp = &ch;
   // deklarálunk egy karakter változót és egy
   // mutatót
cout << chp;
   // lekérdezzük a chp tartalmát, azaz ch címét
   // tehát az eredmény a memóriacím
cout << *chp;
   // lekérdezzük a chp által mutatott változó
   // tartalmát, az eredmény 'a' lesz
cout << &chp;
   // lekérdezzük a chp mutató címét
```

Dinamikus memóriakezelés

Biztonságos használat

- Mutatók használata veszélyes lehet, ha olyan mutatóra hivatkozunk, amely nem mutat megfelelő helyre (pl. nem adtunk neki értéket, vagy amit adtunk, már megsemmisült), *szegmenshibá*t kapunk (futási időben)
- Célszerű a mutatónak létrehozás után a nulla (**`NULL`**, **`0`**) kezdőértéket adni, mert így utólag ellenőrizhető lesz
  - pl.:

```
int *ip = NULL; // vagy int *ip = 0;
…
if (ip){
   /* ez az ág akkor hajtódik végre, ha ip
      nullától különböző értéket tárol */
}
```

Dinamikus memóriakezelés

Mutató, mint bejáró

- A mutatókat ráállíthatjuk tömbre is (pontosabban az első elemére), illetve használhatjuk tömbök bejárására is, így nem csupán indexeléssel férhetünk hozzá az adatokhoz
  - pl.:

```
int array[10];
for (int* p = array; p != array + 10; p++)
    // mutató használata indexelés helyett,
    // ugyanúgy 10 lépést teszünk meg
    cin >> *p; // tömb eleminek feltöltése

// ugyanez rövidebben:
int array[10], *p = array;
while (p != array + 10) cin >> *p++;
```

Dinamikus memóriakezelés

A referencia, mint mutató

- A referencia tekinthető úgy is, mint egy korlátozott felhasználású mutató, amely mindig garantáltan biztonságos
- Ugyanakkor a referencia használata is kiválthat szegmenshibát, lokális változó értékének visszaadásakor, pl.:

```
int& BadFunction(){
   // cím szerint adja vissza a változót
   int value = 1;
   return value;
} // itt a lokális változó megsemmisül
…
int val = BadFunction();
cout << val << endl;
   // szegmenshiba, a változó már nem létezik
```

Dinamikus memóriakezelés

Konstans mutatók és referenciák

- Referencia, illetve mutató változók is lehetnek konstansok
  - referencia esetén az érték nem módosítható:
    `<típus> const &<név> = <változó>;`
  - mutató esetén kétféle módon is korlátozhatjuk a használatot
    - lehet a mutatott érték konstans, ekkor nem változtatható a hivatkozott változó értéke, de a mutatót átállítható:
      `<típus> const *<név>;`
    - lehet a mutató konstans, ekkor nem állítható át másik memóriacímre, de a mutatott érték változtatható:
      `<típus> * const <név> = <változó>;`
    - lehet a mutató és a mutatott érték is konstans:
      `<típus> const * const <név>;`

Dinamikus memóriakezelés

Konstans mutatók és referenciák

- Pl:

```
double d1 = 10, d2 = 50;
double const &d1r = d1; // konstans referencia
double const * d1p1 = &d1; // mutató konstansra
double * const d1p2 = &d1; // konstans mutató
double const * const d1p3 = &d1;
   // konstans mutató konstans értékre
d1r = 100; // HIBA, az érték nem módosítható
*d1p1 = 50; // HIBA, az érték nem módosítható
*d1p2 = 50; // az érték módosítható
*d1p3 = 50; // HIBA
d1p1 = &d2; // átállíthatjuk más memóriacímre
d1p2 = &d2; // HIBA, a mutató nem állítható át
d1p3 = &d2; // HIBA
```

Dinamikus memóriakezelés

Mutatóra állított mutatók és referenciák

- Mivel a mutatók is értékek, rájuk is lehet mutatót állítani
  - ekkor jeleznünk kell, hogy a mutató célja is mutató, azaz halmoznunk kell a `*` jelet
  - pl.:

```
int value = 0;
int *intp = &value;
int **intpp = &intp; // mutatóra állított mutató
cout << **intpp; // kiírja value értékét
```

  - hasonlóan referencia is állítható mutatóra, így a mutató is használható cím szerinti paraméterátadáskor, pl.:

```
int *&intpref = intp;
cout << *intpref; // kiírja value értékét
```

Dinamikus memóriakezelés

Memóriafoglalási lehetőségek

- Memóriahelyeket kétféleképpen foglalhatunk le:
  - *automatikusan*: változó létrehozásakor lefoglalódik hozzá egy memóriahely is, ezt nem befolyásolhatjuk
  - *manuálisan* (*dinamikusan*): lehetőségünk van explicit megadni a kódban, hogy lefoglalunk egy a memóriahelyet
    - ehhez a **`new`** operátort használjuk, és meg kell adnunk a típusát is, pl. **`new double;`**
    - a létrehozás visszaad egy memóriacímet, amelyet a helyfoglalás megkapott (illetve annak az első bájtját)
- A lefoglalással visszakapott memóriacímet megkaphatja egy mutató, pl.: **`int *ip = new int;`**

Dinamikus memóriakezelés

Memóriafoglalási lehetőségek

- szétválaszthatjuk a változó deklarációját a hozzá tartozó memóriaterület lefoglalásától, pl.:

```
int *ip; // ekkor i még csak egy mutató
ip = new int; // új memóriaterület a mutatónak
```

- két hely kerül lefoglalásra a memóriában, egy a mutatónak, egy az értéknek
- többször is lefoglalhatunk helyet egy mutatónak, pl.:

```
int *ip = new int;
ip = new int; ip = new int;
```

- új memóriaterület foglalásakor a régi memóriaterület is bent marad a szegmensben, de a mutatón keresztül már nem elérhető (de memóriaműveletekkel igen)

Dinamikus memóriakezelés

Memóriaterületek

- A programok a használat szempontjából három területet különböztetnek meg:
  - *globális terület* (*global*): konstansok és globális változók, amelyek a program futása során mindig jelen vannak
  - *verem* (*stack*): a lokális változók, amelyeket automatikusan hoztunk létre
    - működésében olyan, mint egy verem, mert mindig az utolsó blokkban létrehozott változó törlődik elsőként a blokk végeztével
  - *kupac* (*heap*): a manuálisan lefoglalható memóriaterület, általában a legnagyobb részét képezi a szegmensnek
    - a tömbök és szövegek is ide kerülnek

Dinamikus memóriakezelés

Memóriahely felszabadítás

- Ahogy lefoglalunk, úgy lehetőségünk van törölni is memóriahelyet programunkban
  - az automatikusan lefoglalt memória törlését a program magától végzi, ezt nem befolyásoljuk
  - a manuálisan létrehozott memóriahelyeket nekünk kell törölnünk, vagy a program végéig a memóriában maradnak
  - a törlésre a **`delete`** operátor szolgál
  - pl.:

```
float* flp = 0; // flp nem hivatkozik semmire
flp = new float;
   // manuálisan lefoglaljuk a helyet
delete flp; // töröljük a lefoglalt helyet
```

Dinamikus memóriakezelés

Biztonságos dinamikus helyfoglalás

- Minden **`new`** operátornak kell rendelkeznie egy **`delete`** párral, azaz a dinamikusan létrehozott változókat törölni is kell
- A nem törölt változók használatuk után is foglalják a memóriát, bár már nincs mutató rájuk állítva, az ilyen területeket nevezzük *memóriaszemét*nek, pl.:

```
int *ip = new int;
// dinamikusan lefoglaltuk a memóriaterületet
ip = new int;
// ekkor az előző terület memóriaszemét lesz
```

- Sosem a mutatót, csak a dinamikusan lefoglalt területet töröljük, így ha több mutató hivatkozik ugyanarra a területre, elég egyszer elvégeznünk a törlést

Dinamikus memóriakezelés

Többszörös dinamikus foglalás

- Egyszerre több memóriahelyet is lefoglalhatunk azonos típusból a **[]** operátorral, ekkor azok egymás után helyezkednek el a memóriában, pl.:

```
int *ip = new int[5];
// öt memóriahely lefoglalása
```

  - a törléshez **delete** operátornak jelölnünk kell, hogy több helyről van szó, szintén a **[]** operátorral, pl.:

```
delete[] ip;
```

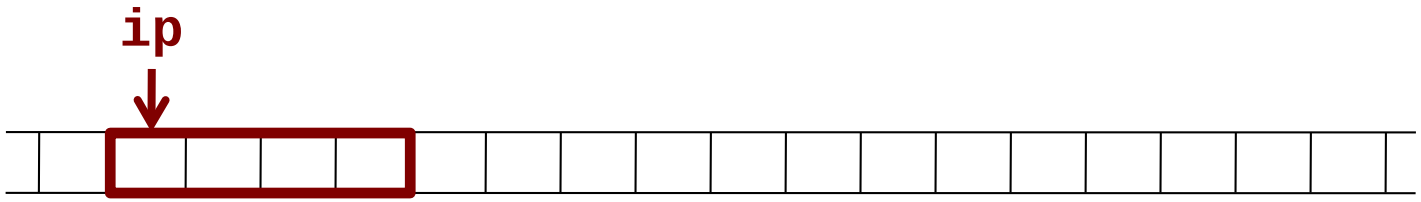
  - ha törlésnél véletlenül lefelejtjük a tömb jelölést, akkor csak az első érték törlődik, a többi a memóriában marad
  - a törlés után a mutató továbbra is használható, de az értékek elvesznek

Dinamikus memóriakezelés

Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
```

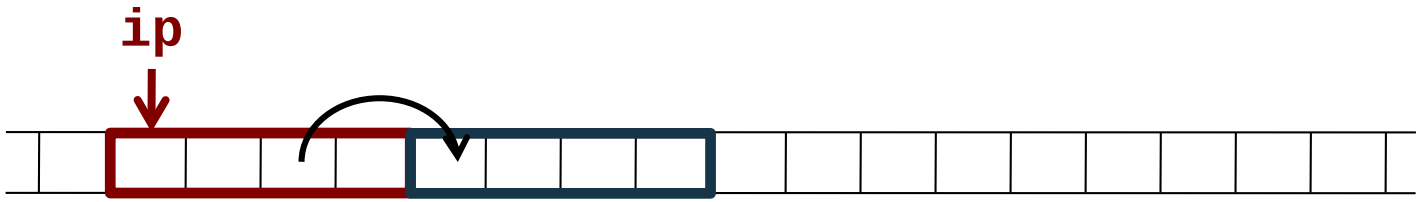


Dinamikus memóriakezelés

Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
```

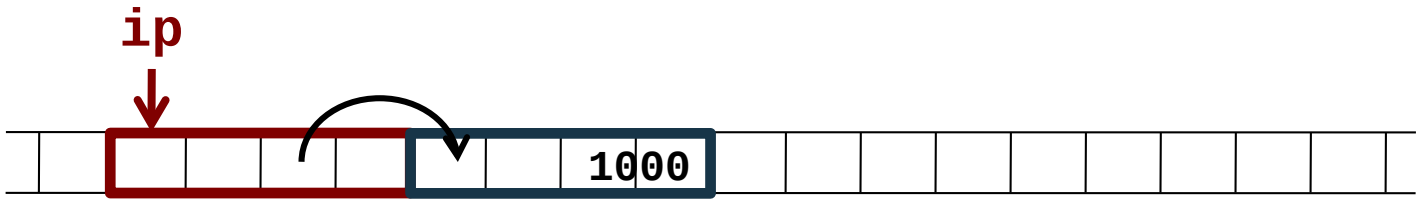


Dinamikus memóriakezelés

Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // új érték
```

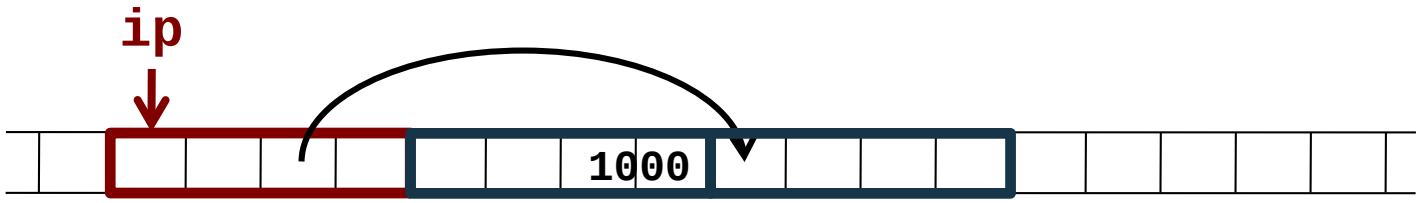


Dinamikus memóriakezelés

Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // új érték
ip = new int;
   // új érték, az előzőből memóriaszemét lesz
```

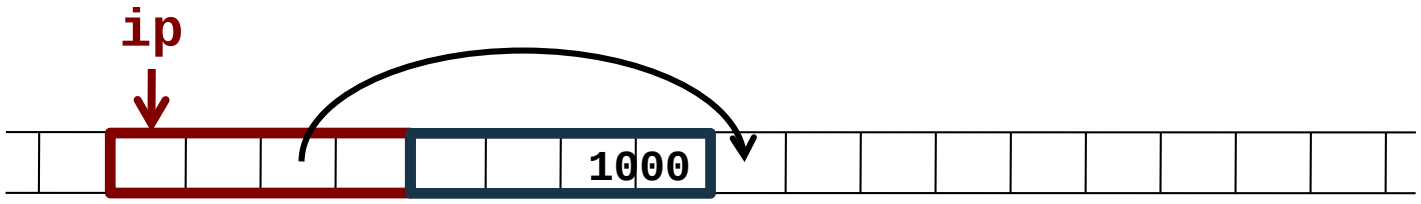


Dinamikus memóriakezelés

Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // új érték
ip = new int;
   // új érték, az előzőből memóriaszemét lesz
delete ip;
   // memóriahely törlése, ip-ben megmarad a cím
```

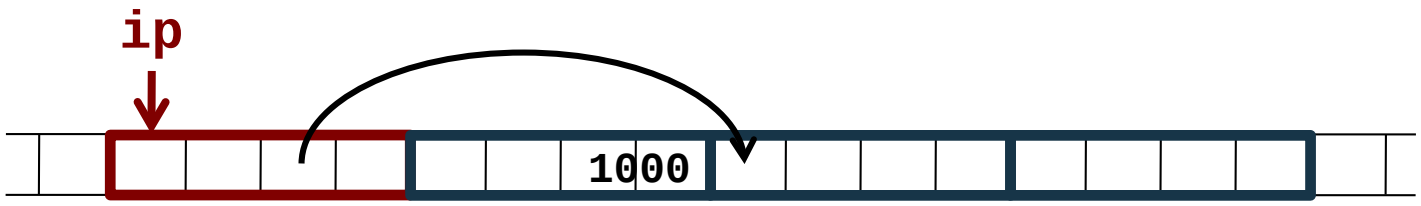


Dinamikus memóriakezelés

Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // új érték
ip = new int;
   // új érték, az előzőből memóriaszemét lesz
delete ip;
   // memóriahely törlése, ip-ben megmarad a cím
ip = new int[2]; // 2 memóriahely foglalása
```

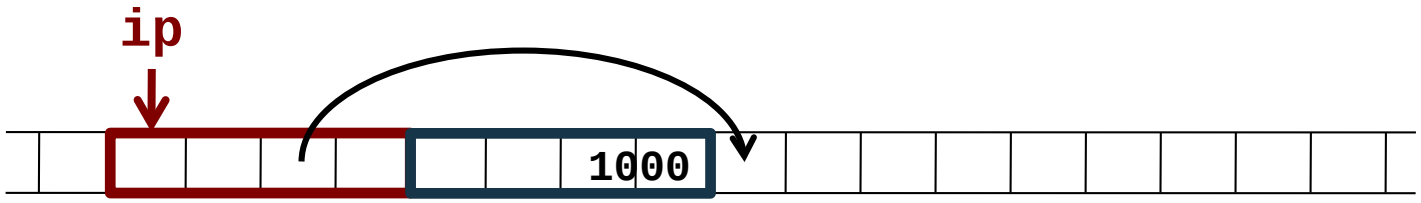


Dinamikus memóriakezelés

Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // új érték
ip = new int;
   // új érték, az előzőből memóriaszemét lesz
delete ip;
   // memóriahely törlése, ip-ben megmarad a cím
ip = new int[2]; // 2 memóriahely foglalása
delete[] ip; // törlés
```

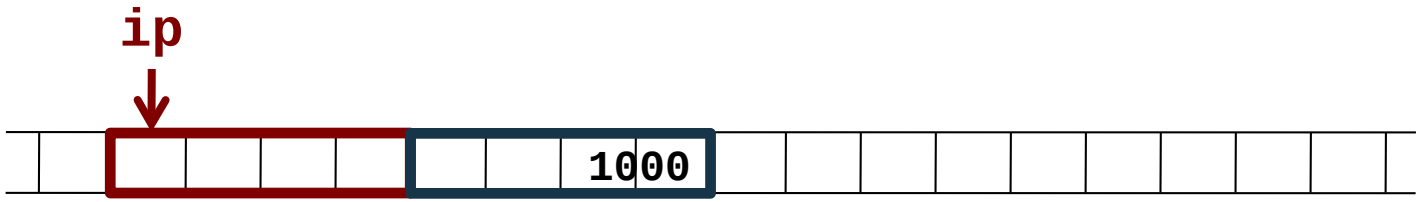


Dinamikus memóriakezelés

Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // új érték
ip = new int;
   // új érték, az előzőből memóriaszemét lesz
delete ip;
   // memóriahely törlése, ip-ben megmarad a cím
ip = new int[2]; // 2 memóriahely foglalása
delete[] ip; // törlés
ip = NULL; // újbóli kinullázás
```



Dinamikus memóriakezelés

A primitív dinamikus tömb

- A többszöri memóriafoglalással lényegében egy tömböt hozhatunk létre, amely a primitív tömb dinamikus megfelelője
  - az elemei elérhetőek a `[ ]` operátorral, 0-tól indexelve
  - működése lényegében megegyezik a statikus dinamikus tömbével, de paraméterben megadható változó is méretnek
  - pl.:

```
int size; cin >> size;
int* array = new int[size]; // tömb lefoglalása
for (int i = 0; i < size; i++)
   cin >> array[i]; // elemek bekérése
…
delete[] array; // tömb törlése
```

Dinamikus memóriakezelés

Tömbelem címzés

- A tömbelem indexelés igazából a memóriában való címelérés, mivel a memóriacímek a **+** operátorral is elérhetőek
  - azaz **a[i]** leírható ***(a+i)** formában is, hiszen a tömb címével a változó mennyiséggel arrébb lévő memóriacím értékét akarjuk kiolvasni (ezért indexelünk 0-tól)
  - mivel az összeadás kommutatív, ezért az index és a tömbnév fel is cserélhető a kifejezésben
  - pl.:

```
float* a = new float[10];
cin >> a[5];
// ugyanez: cin >> *(a+5);
// ugyanez: cin >> 5[a];
```

Dinamikus memóriakezelés

Többdimenziós tömbök

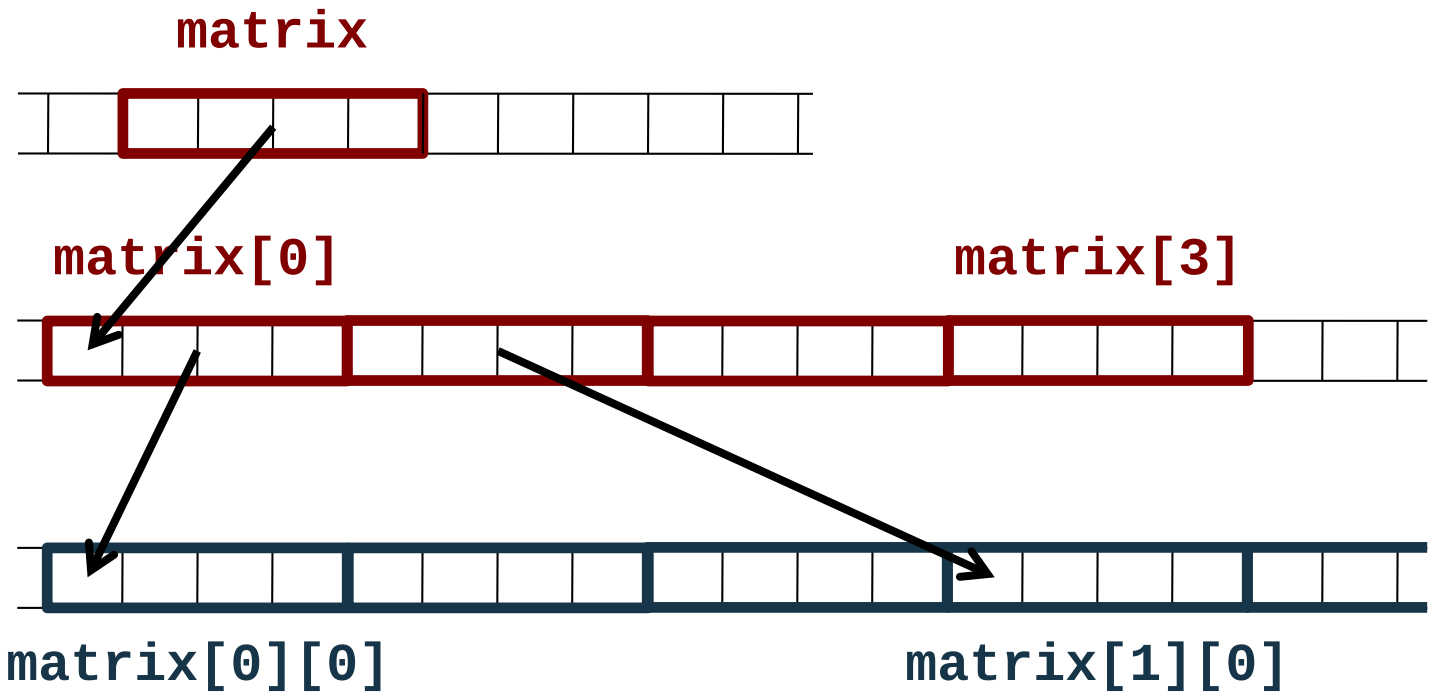
- Lehetőségünk van többdimenziós tömbök létrehozására is a tömbök tömbje módszerrel
  - azaz mutatókra mutatót állítunk, így a külső tömbünk fogja tartalmazni a mutatókat, amelyek a mátrix soraira hivatkoznak
  - létrehozzuk a mutatókat tároló tömböt, majd utána mindegyikre felfűzzük az értékeket tároló tömböt, tehát egy ciklusra van szükségünk, pl.:

```
float** matrix = new float*[4];
// 4 sora lesz a mátrixnak
for (int i = 0; i < 4; i++)
   matrix[i] = new float[3];
   // 3 oszlopa lesz a mátrixnak
```

Dinamikus memóriakezelés

Többdimenziós tömbök

- a mátrix megjelenése a memóriában:



Dinamikus memóriakezelés

Többdimenziós tömbök

- a létrehozást követően az indexelés és a sorok hozzáférése a megszokott módon történik, pl.:

```
cin >> matrix[3][2]; // elem bekérése
cout << **matrix; // mátrix 1. sorának 1. eleme
float* row = matrix[3];
   // sor átadása egy mutatónak
```

- törléskor külön kell törölnünk minden sort, majd a mutatókat tartalmazó tömböt, pl.:

```
for (int i = 0; i < 4; i++)
   delete[] matrix[i]; // sorok törlése
delete[] matrix; // mutatók törlése
```

- ugyanez megvalósítható magasabb dimenziókban is, pl.:

```
float*** m3d = new float**[4];
```

Dinamikus memóriakezelés

Objektumorientált környezetben

- Objektumorientált környezetben is számos helyen előfordul dinamikus memóriakezelés
  - hivatkozásokat (aggregációkat) mutatók segítségével valósíthatunk meg, hogy ne másolódjanak az objektumaink
  - polimorfizmus esetén mutatók segítségével tudjuk biztosítani a dinamikus típus futás során történő megadását
  - egy objektum bármilyen mezője lehet mutató, és a hozzá tartozó értéket lefoglalhatjuk dinamikusan (de ekkor gondoskodnunk kell annak törléséről is)
- Egy objektum mindig tisztában van saját memóriacímével (*objektumhivatkozás*), amelyet lekérhetünk

Dinamikus memóriakezelés

Objektumhivatkozások

- Egy típuspéldányban elérhető annak valamennyi mezője és metódusa, akár rejtett, akár nem
- Ugyanakkor lehetőség van magát a teljes objektumot (pontosabban a memóriacímét) elérni saját magán belül a **`this`** kulcsszó használatával
  - tehát ez egy mutatót ad vissza az aktuális példányra
  - ugyanúgy használható, mint bármely más mutató, amelyet az objektumra állítottunk, azaz lekérdezhető általa az összes tag: **`this->`**<***`tagnév`***>
  - ha a konkrét objektumra van szükségünk, akkor a **`*this`**-t használunk

Dinamikus memóriakezelés

Objektumhivatkozások

- Pl.:

```
class MyClass {
private:
   int _value;
public:
   …
   void SetValue(int value){
      _value = value;
      // ugyanez: this->_value = value;
   }
   MyClass* ReturnMyPointer(){
      return this;
      // visszaadja a mutatót a példányra
   }
```

Dinamikus memóriakezelés

Objektumhivatkozások

- Pl.:

```
    MyClass ReturnCopyOfMyself(){
        return *this;
        // visszaadja az objektum másolatát
    }
    …
};

MyType t;
MyType* tP = t.ReturnMyPointer(); // hivatkozás
tP->SetValue(10);
    // másként: t.SetValue(10);
MyType tCopy = t. ReturnCopyOfMyself(); // másolat
tCopy.SetValue(20); // nem befolyásolja t értékét
```

Dinamikus memóriakezelés

Dinamikusan foglalt mezők

- Természetesen típusok mezői is lehetnek mutatók, és allokálhatunk nekik dinamikusan memóriaterületet
  - ezt általában a konstruktorban végezzük
  - de az így létrehozott értékeket manuálisan kell törölni, különben a példány megsemmisülése után is megmarad
  - a törlést akkor kell elvégezni, amikor a példány törlődik
- A típuspéldány megsemmisítéséért felelős műveletet nevezzük *destruktor*nak
  - a destruktor automatikusan lefut, amikor törlődik a változó (lokális változó esetén a blokk végére érünk, dinamikus létrehozás esetén meghívjuk a **`delete`** műveletet)

Dinamikus memóriakezelés

Destruktor

- A destruktorban olyan utasításokat tárolunk benne, amelyek „kitakarítják” az általunk használt memóriaterületet
  - a dinamikusan foglalt mezőket töröljük, a többit nem kell
  - ha nincs dinamikusan foglalt mező, akkor nem kell
- A destruktor a **~<típusnév>** nevet kapja, mindig publikus, nincs típusa, nincs paramétere, ezért nem túlterhelhető:

```
class <típus> {
public:
   <típusnév>() { … } // konstruktor
   ~<típusnév>() { … } // destruktor
   …
};
```

Dinamikus memóriakezelés

Destruktor

- Pl.:

```
class MyClass {
public:
   MyClass(){ cout << "Hello! "; } // konstruktor
   ~MyClass(){ cout << "Byebye!"; } // destruktor
};

int main(){
   MyClass mc; // itt fut le a konstruktor
   return 0;
} // itt fut le a destruktor

// eredménye: Hello! Byebye!
```

Dinamikus memóriakezelés

Destruktor

- Pl.:

```
class MyClass {
private:
   int* t; // dinamikusan lefoglalt tömb
public:
   MyClass(){ t = new int[10]; } // konstruktor
   // nincs destruktor
};

int someFunction(){
   MyClass mc; // létrejön a t tömb
   …
} // a t tömb a memóriában marad (memóriaszemét)
```

Dinamikus memóriakezelés

Destruktor

- Pl.:

```
class MyClass {
private:
   int* t; // dinamikusan lefoglalt tömb
public:
   MyClass(){ t = new int[10]; } // konstruktor
   ~MyClass(){ delete[] t; } // destruktor
};

int someFunction(){
   MyClass mc; // létrejön a t tömb
   …
} // törlődik a t tömb (nincs memóriaszemét)
```

Dinamikus memóriakezelés

Destruktor öröklődése

- A destruktor (a konstruktorhoz hasonlóan) automatikusan öröklődik és minden ős destruktor megívódik a leszármazott destruktor meghívásakor
  - elsőként a leszármazotté, majd a sorrendben az ősök destruktorai felfelé az öröklődési fán (vagyis pont fordított sorrendben, mint a konstruktor esetén)
- Pl.:

```
class MyBaseClass {
public:
   MyBaseClass() { cout << "1 Hello! "; }
   ~MyBaseClass() { cout << "1 Byebye! "; }
};
```

Dinamikus memóriakezelés

Destruktor öröklődése

```
class MyChildClass : public MyBaseClass {
public:
   MyChildClass() { cout << "2 Hello! "; }
   ~MyChildClass() { cout << "2 Byebye! "; }
};

int main(){
   MyChildClass mcc; // konstruktor hívás
   return 0;
} // destruktor hívás

// 1 Hello! 2 Hello! 2 Byebye! 1 Byebye!
```

Dinamikus memóriakezelés

Virtuális destruktor

- Polimorfizmus esetén a program az objektum megsemmisítésénél a statikus típust veszi figyelembe, ezért annak destruktorát hívja meg
  - pl.:

```
… ~MyBaseClass() …

int main(){
   MyBaseClass* mbc = new MyChildClass();
   delete mbc;
   return 0;
}

// 1 Hello! 2 Hello! 1 Byebye!
```

Dinamikus memóriakezelés

Virtuális destruktor

- ha a dinamikus típus szerint akarjuk elvégezni a törlést, akkor a statikus típusban a destruktort virtuálisnak kell megadni
- pl.:

```
… virtual ~MyBaseClass() …

int main(){
   MyBaseClass* mbc = new MyChildClass();
   delete mbc;
   return 0;
}

// 1 Hello! 2 Hello! 2 Byebye! 1 Byebye!
```

Dinamikus memóriakezelés

Példa

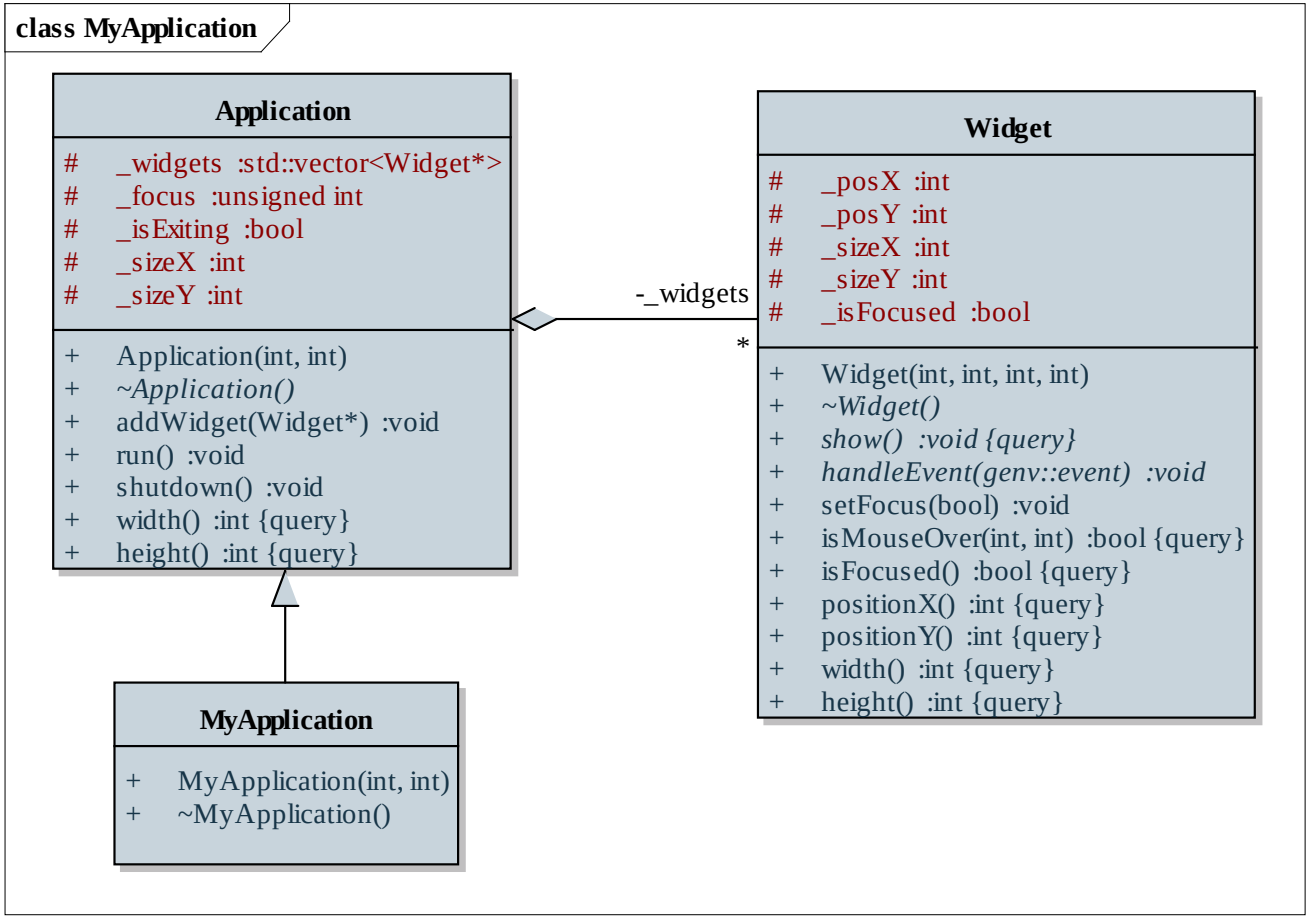
Feladat: Készítsünk egy teljesen objektumorientált grafikus felületű alkalmazást, amelyben egy gombbal tudjuk módosítani egy címke feliratát, egy másikkal pedig ki tudunk lépni.

- a programunk akkor lesz teljesen objektumorientált, ha a főprogram csupán elindít egy objektumot, más tevékenységet nem végez
- ehhez megvalósítunk egy saját alkalmazás osztályt (**`MyApplication`**), amely az alkalmazás leszármazottja lesz
  - konstruktorában létrehozza a vezérlőket, így azokat nem kell a főprogramban létrehozni
  - mivel dinamikusan hoztuk őket létre, törölni is kell, így készítünk egy destruktort is

Dinamikus memóriakezelés

Példa

Tervezés:



Dinamikus memóriakezelés

Példa

Megoldás (`myapplication.cpp`*):*

```
MyApplication::MyApplication(int sx, int sy)
   : Application(sx, sy) // átadjuk a méretet
{
   Label* l = new Label(50, 50, "HUHU");
   …
   // hozzáadjuk a vezérlőket
   addWidget(l);
   …
}
MyApplication::~MyApplication() {
   for (int i = 0; i < _widgets.size(); i++)
      // töröljük a létrehozott vezérlőket
      delete _widgets[i];
}
```

Dinamikus memóriakezelés

Függvénymutatók

- Mutatók nem csupán adatokra, de tevékenységekre, azaz alprogramokra, függvényekre is állíthatóak, ezeket nevezzünk *függvénymutató*knak (*function pointer*)
  - célja, hogy a végrehajtott függvény cserélhető legyen, azaz futás közben cserélhető legyen a konkrét végrehajtandó függvény
  - a függvény nevét helyezzük mutatóba, és megadjuk a teljes szintaxist:

    `<típus> (*<mutató név>)(<paraméterek>)`
  - a mutatott függvény szintaxisának meg kell egyeznie a mutató szintaxisával, a paramétereknek csak a típusát adjuk meg

Dinamikus memóriakezelés

Függvénymutatók

- Pl.:

```
// létrehozunk szabványos függvényeket
int add(int a, int b){ return a + b; }
int mul(int a, int b){ return a * b; }

int main() {
   // létrehozunk egy függvénymutatót
   int (*op)(int, int);
      // a mutató neve op, két egész számot fogad,
      // és egész számot ad vissza

   op = NULL; // ugyanúgy nullázható, mint bármely
              // más mutató
```

```
    // beállíthatjuk azonos szintaxisú függvényre:
    op = add; // beállítjuk az add-ra

    cout << (*op)(2, 3) << endl;
       // meghívjuk a mutató mögötti
       // függvényt, eredménye: 5

    op = mul; // beállítjuk az mul-ra

    cout << (*op)(2, 3) << endl;
       // meghívjuk a mutató mögötti
       // függvényt, eredménye: 6
    …
}
```

Dinamikus memóriakezelés

Függvénymutatók

- A függvénymutatókat leginkább paraméterátadásnál használjuk, ezáltal jobban személyre szabhatjuk a függvény viselkedését

- Pl.:

```
int runOperation(vector<int> values, int start,
                 (int)(*op)(int, int))
   // a harmadik paraméter egy függvénymutató
{
   int result = start;
   for (int i = 0; i < values.size(); i++)
      result = (*op)(result, values[i]);
      // végrehajtjuk a tevékenységet
   return result;
}
```

Dinamikus memóriakezelés

Függvénymutatók

```
int main()
{
   vector<int> values(10);
   … // elemek megadása

   cout << runOperation(values, 0, add) << endl;
      // összeadjuk az elemeket
   cout << runOperation(values, 1, mul) << endl;
      // összeszorozzuk az elemeket
}
```

- A működés hasonló a virtuális függvényéhez (futás közben cserélhetjük a viselkedést), de jóval rugalmasabb annál, és nem szükséges öröklődés hozzá

Dinamikus memóriakezelés

Függvénymutatók objektumorientált környezetben

- Függvénymutatót metódusokra is létesíthetünk, ám ebben az esetben a hívónak mindig tisztában kell lennie, mely objektum metódusát kezeljük
  - deklarációkor jelölnünk kell a függvény osztályát:
    <*típus*> (<osztálynév>::*<*mutató név*>)(<…>)
  - átadáskor is jelölnünk kell az osztályt:
    <*mutató név*> = &<osztálynév>::<*függvény név*>;
  - végrehajtáskor az objektumot kell megadnunk:
    (<*objektum*>.*<*mutató név*>)(<*paraméterek*>)
    - megadhatjuk ugyanazon osztály egy metódusát is:
      (this->*<*mutató név*>)(<*paraméterek*>)

Dinamikus memóriakezelés

Függvénymutatók objektumorientált környezetben

- Pl.:

```
class MyClass {
public:
   int add(int a, int b){ return a + b; }
   int mul(int a, int b){ return a * b; }

   void setOperation(
      (int)(MyClass::*op)(int, int));
      // jelöljük a metódus osztályát is
   void RunOperation();
   …
private:
   int (MyClass::*op) (int, int) _operation;
      // mezőként is jelöljük az osztályt
```

Dinamikus memóriakezelés

Függvénymutatók objektumorientált környezetben

```
void MyClass::setOperation(
      (int)(MyClass::*op)(int, int) op)
{
   _operation = op;
}
void MyClass::runOperation()
{
   … (this->*_operation)(…) …
   // hívás az objektum jelölésével
}
…
MyClass mc;
mc.setOperation(&MyClass::add);
   // megadás az osztály jelölésével
```

Dinamikus memóriakezelés

Példa

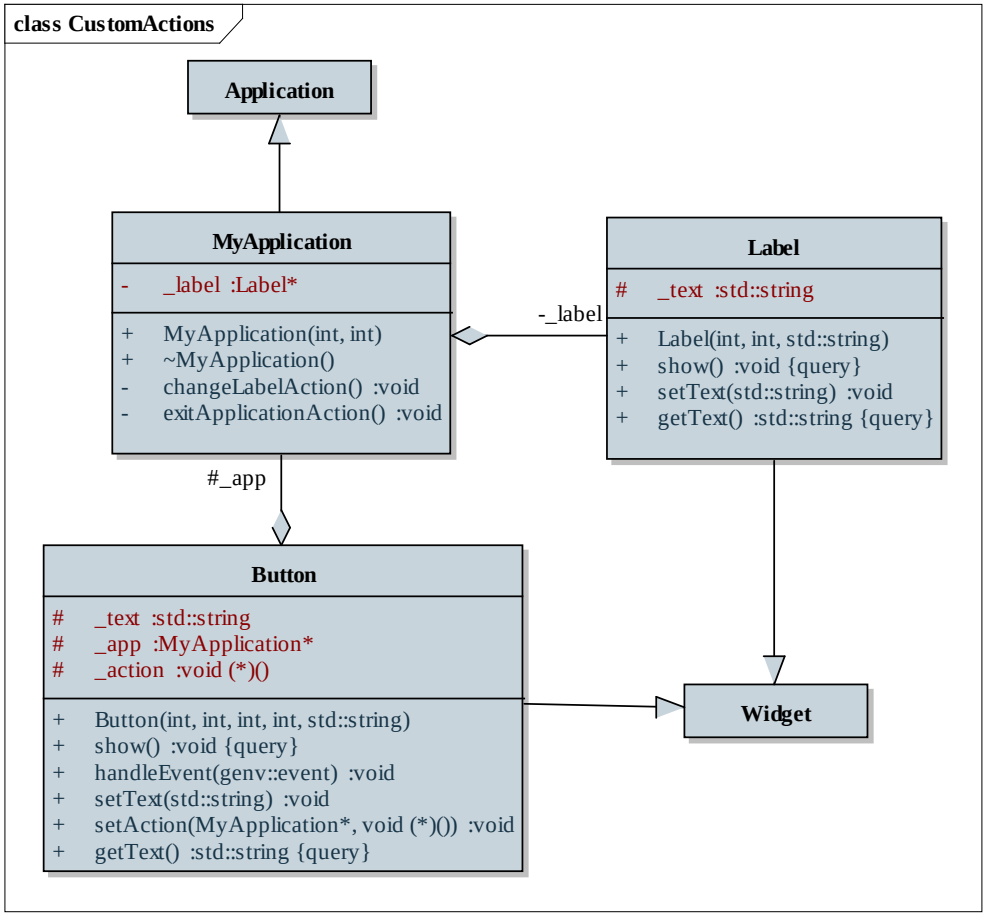
Feladat: Készítsünk egy teljesen objektumorientált grafikus felületű alkalmazást, amelyben egy gombbal tudjuk módosítani egy címke feliratát, egy másikkal pedig ki tudunk lépni.

- alakítsuk át a gombok működését úgy, hogy egyetlen gomb típussal megvalósíthassunk tetszőleges tevékenységet
- ehhez a gomb működését átírjuk úgy, hogy függvénymutatóban (**`void (*action)()`**) kapja meg a végrehajtandó tevékenységet
- a konkrét tevékenységeket (címke átírása, kilépés az alkalmazásból) a **`MyApplication`** osztályban írjuk meg metódusként

Dinamikus memóriakezelés

Példa

Tervezés:



Dinamikus memóriakezelés

Példa

Megoldás (`myapplication.cpp`*):*

```
MyApplication::MyApplication(int sx, int sy)
   : Application(sx, sy) // átadjuk a méretet
{
   _label = new Label(50, 50, "HUHU");

   Button* lmb = new Button(50, 150, 100, 25,
                            "Modify");
   lmb->setAction(this,
      &MyApplication::changeLabelAction);
      // beállítjuk a végrehajtandó akciót
      // (függvénymutató segítségével)
   …
}
```