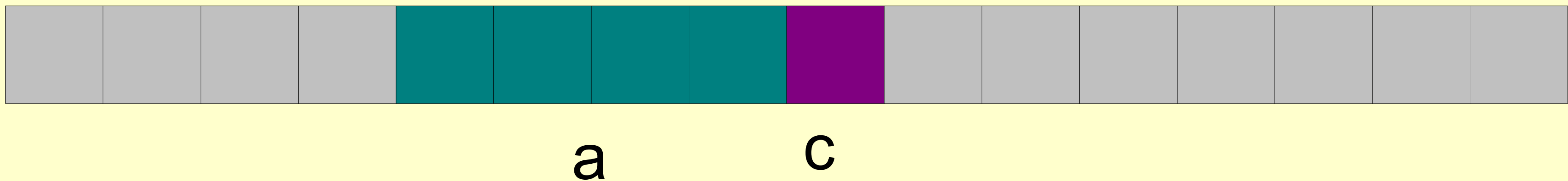
Bevezetés a programozásba 2

2. Előadás
Mutató, referencia,
dinamikus memóriakezelés

http://digitus.itk.ppke.hu/~flugi/

A memória

- Byte sorozat
- A változók valahol helyet kapnak




```
int main() {
  int a;
  char c;
}
```

Mutató

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

int main() {
   int a = 55;
   int *m = &a;
   cout <<"A változó értéke:" << a
       << endl;
   cout <<"A mutató értéke:" << m
       << endl;
   cout <<"A mutatott terület:" << *m
       << endl;
```

			a = 55	m					

Mutatók

- A mutatott terület tetszőleges használata olyan, mintha magával a változóval történne
- A mutató típusát a fordítóprogram számon tartja
- Egy változó címét & operátorral (referálás), egy mutató által mutatott értéket * operátorral (dereferálás) jelöljük
- Deklarációkban a * a névhez tartozik, tehát az `int *a, b` esetben b nem mutató, ahhoz `int *a, *b` kell

Dinamikus memóriakezelés

- **`new`** kulcsszó: egy adott típusú változót hozzunk létre a heap-en, és kérjük a címét
- **`delete`** kulcsszó: már nincs szükségünk a memóriára, a rendszer másra is felhasználhatja mostantól
- veszélyes: ha nem szabadítjuk fel, elfogyhat.

```
int main() {
  int *a = new int;
  int *t = new int[1000];
  ...
  delete a;
  delete[] t;
}
```

Memóriafolyás

```
int main() {
  while (sokáig) {
    …
    int *t = new int[1000];
    …
    // nincs delete[] t;
  }

}
```

Mutatók biztonságos használata

- Szorítkozunk a dinamikus memóriakezelésre
- Mindig van delete minden new-hoz
  - például csak konstruktorban van new, és csak destruktorban delete
- Referáláshoz használt mutatókat megkülönböztetjük a dinamikusan allokálttól, és nem szabadítjuk fel
  - „ownership”
  - ha két mutató egy helyre mutat, és egyiket felszabadítjuk, a másikra való hivatkozás fagy

Élettartam és a dinamikus memóriakezelés

- Különbség van a mutató típusú változó, és a segítségével dinamikusan allokált tartalom élettartamában
  - A mutató „hagyományos” lokális vagy globális változó
  - A dinamikusan foglalt tartalom élettartama a new pillanatától a delete pillanatáig tart
  - Ilyenkor fut a konstruktor és destruktor is
  - Könnyen előfordulhat, hogy az allokálásra használt mutatót túléli a dinamikus tartalom

Referencia: fordítóprogram által garantáltan biztonságos mutató

- Csak definiálni lehet, deklarálni nem, tehát a kezdeti értéke adott
- Kizárólag létező változóra lehet állítani
- Ha mező típusaként használjuk, kötelező konstruktor örökítéssel értéket adni neki
- Gyors, mert technikailag csak a mutató mozog paraméterátadáskor és visszatérési értékben
- Van veszély is: lokális változóra állított referencia, mint visszatérési érték

Rossz példa

```
int & fv() {
  int a;
  return a;
}

int main() {
  cout << fv() << endl;
}
```

warning:
reference to
local variable
'a' returned

Rossz példa

```
int * fv() {
  int a;
  return &a;
}

int main() {
  cout << fv() << endl;
}
```

warning: address of local variable ‘a’ returned

Mutatók és rekordok

```
struct A {
  int a;
  int *m;
};
int main() {
  A *a = new A;
  cin >> (*a).a;
  (*a).m = new int;
  cin >> *(*a).m;
}
```

Mutatók és rekordok

```
struct A {
  int a;
  int *m;
};
int main() {
  A *a = new A;
  cin >> a->a;
  (*a).m = new int;
  cin >> *a->m;
}
```

Mutatók biztonságos használata 2

- Honnan tudom, hogy a mutató által mutatott területre hivatkozni biztonságos?
  - Az értékéből sehogy.
- Nullmutató: a 0 értéket adjuk a mutatónak, azaz a 0-s memóriacímre hivatkozunk, ami biztosan nem a mienk, tehát használható arra, hogy „érvénytelen”, vagy hogy „még nem kapott értéket”, esetleg „felszabadítottuk, ne hivatkozz rá”

Nullmutató

```
int main() {
  int *m=0;
  …
  if (rand()%2) {
    m = new int;
  }
  …
  if (m) {
    cout << *m;
  }
}
```

Típuskonstrukció

- Mutatóra is lehet mutatót állítani:

```
int main() {
  int a = 55;
  int *m = &al
  int **mm = &m;
  int ***mmm = &mm;
  cout <<*m << **mm << ***mmm;
}
```

Főbb memóriaterületek

- Verem
  - lokális változók: deklaráláskor létrejönnek, blokk végével megszűnnek
- Heap
  - a „nagy” memória, tömbök, stringek tartalma itt van
  - kérni kell memóriát, és szólni kell, ha felszabadítjuk
- Globál
  - Neumann elvű gépekben lehetséges még a kód közé adatot tenni, konstansok, globálváltozók

Primitív tömbök és mutatók

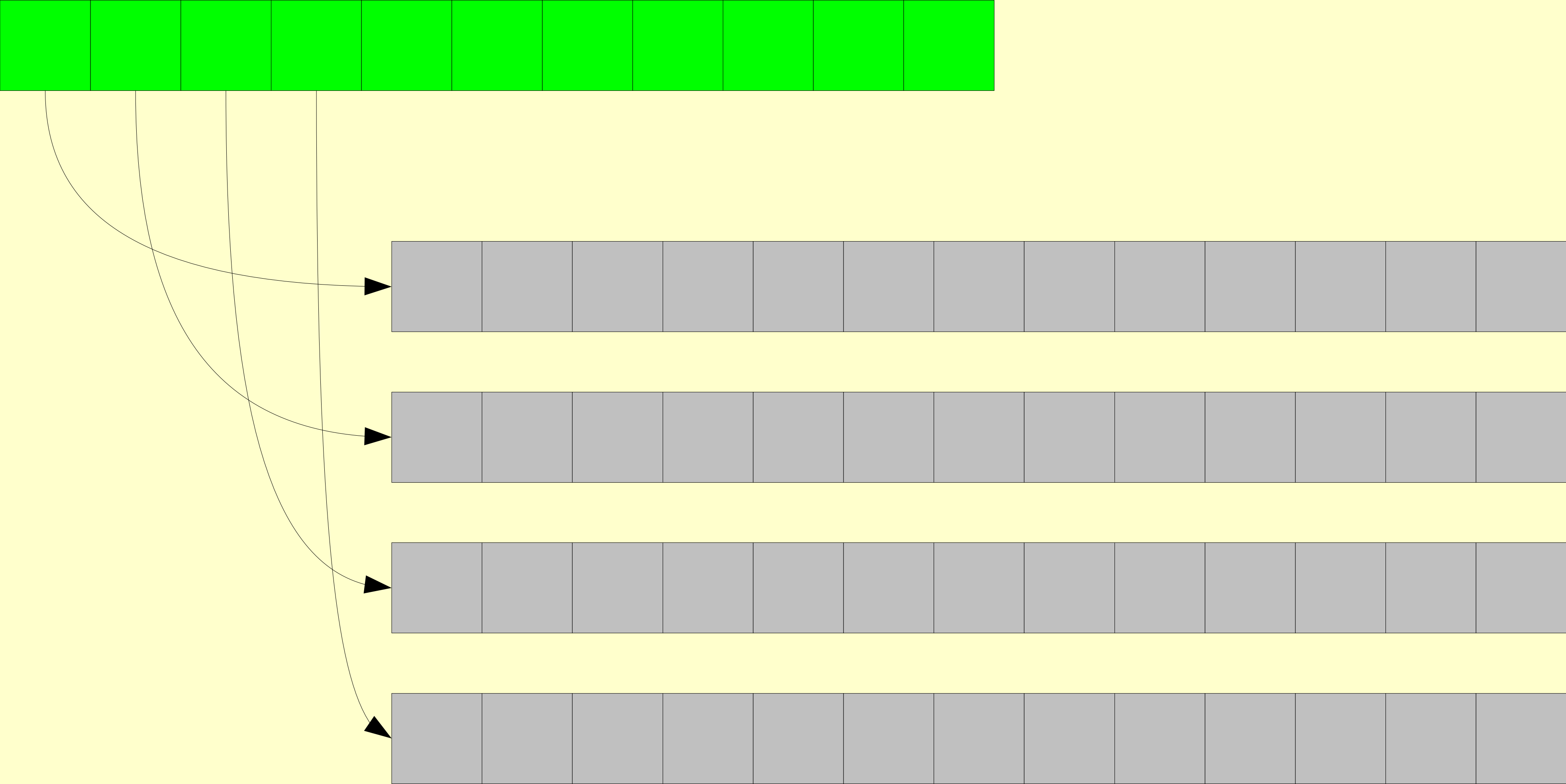
- A primitív tömb egy speciális mutató, amiről a fordítóprogram megjegyzi, hogy mennyi memóriát foglal le
  - ezért kell ismerni fordítási időben a méretet
- A mutató az első elemre mutat

```
int main() {
  int t[10];
  t[0] = 55;
  cout << *t;
}
```

Többdimenziós primitív tömbök heap-en

```
int main() {
  int X=10,Y=20;
  int **m = new int*[X];
  for (int i=0;i<X;i++)
    m[i]=new int[Y];
  for (int i=0;i<X;i++) {
    for(int j=0;j<Y;j++) {
      cout << m[i][j] <<" ";
    }
    cout << endl;
  }
}
```

Többdimenziós primitív tömbök heap-en



A **this**

- Tagfüggvényekben használható kulcsszó, az aktuálisan használt objektum címét jelenti
- Más nyelvekben „self” is lehet
- Sokszor hasznos, pl. amikor objektumok egymás címét tárolják

```
struct A {
  void fv() {
    cout << this;
  }
};
int main() {
  A *a=new A;
  cout << a << endl <<
    a->fv() << endl;
}
```

Mutatóaritmetika

- A mutató értéke egy memóriacella címe, egy egész szám
- Ha változtatjuk, máshová mutat
- A változtatás egysége annak a típusnak a mérete, amire mutat.
- A a[b] jelentése: *(a+b)

```
int main() {
   int *t = new int[10];
   t[0] = 55;
   t[1] = 66;
   cout << *t; //55
   t=t+1;
   cout << *t; //66
}
```

Mutatóaritmetika

- Gyakori használat: bejáró mutató, léptetéssel

```
int main() {
  int a[10];
  for (int i=0;i<10;i++) a[i]=i+1;
  int *b = a;
  for (int i=0;i<10;i++)
    cout << *b++;

}
```

Mutatóaritmetika

- A mutatókra megfogalmazható feltételek elég rugalmasak

```
int main() {
  int a[10];
  for (int i=0;i<10;i++) a[i]=i+1;
  int *b = a;
  while (b!=a+10)
    cout << *b++;

}
```

Egy kis érdekesség

- A tömb is mutató
- Az a[b] jelentése *(a+b)
- Az összeadás kommutatív
- A következmény:

```
int main() {
  int a[10];
  a[5]=55;
  cout << 5[a];
}
```