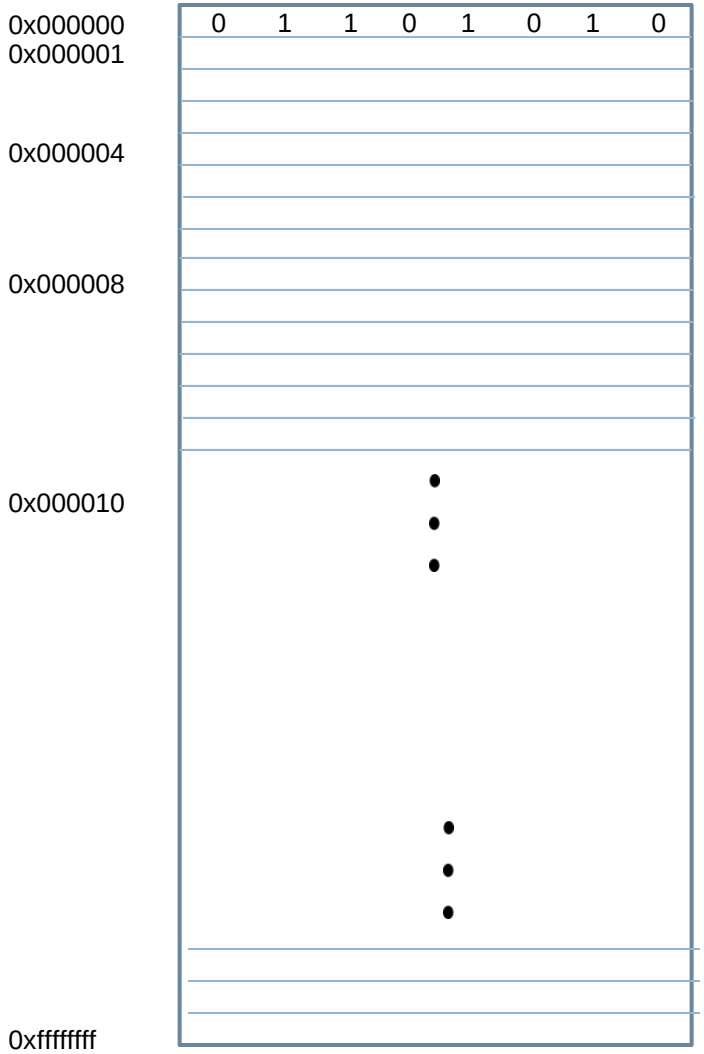
SPRÁVA PAMÄTE

Čo ak program potrebuje pamäť, ktorej veľkosť závisí od konkrétneho vstupu?

- a.) Programátor môže odhadnúť maximálnu možnú veľkosť a vyhradiť tento priestor v statických premenných (riešenie jazyka Fortran)
- b.) Program si môže vyžiadať (alokovať) pamäť v run-time.

Problém dynamickej pamäte

- Počítač/program má k dispozícii len lineárnu pamäť. T.j. postupnosť adries kam sa dá uložiť jeden byte. Počítač vie čítať a zapísať 1, 2, 4 a 8 po sebe nasledujúcich bytov.

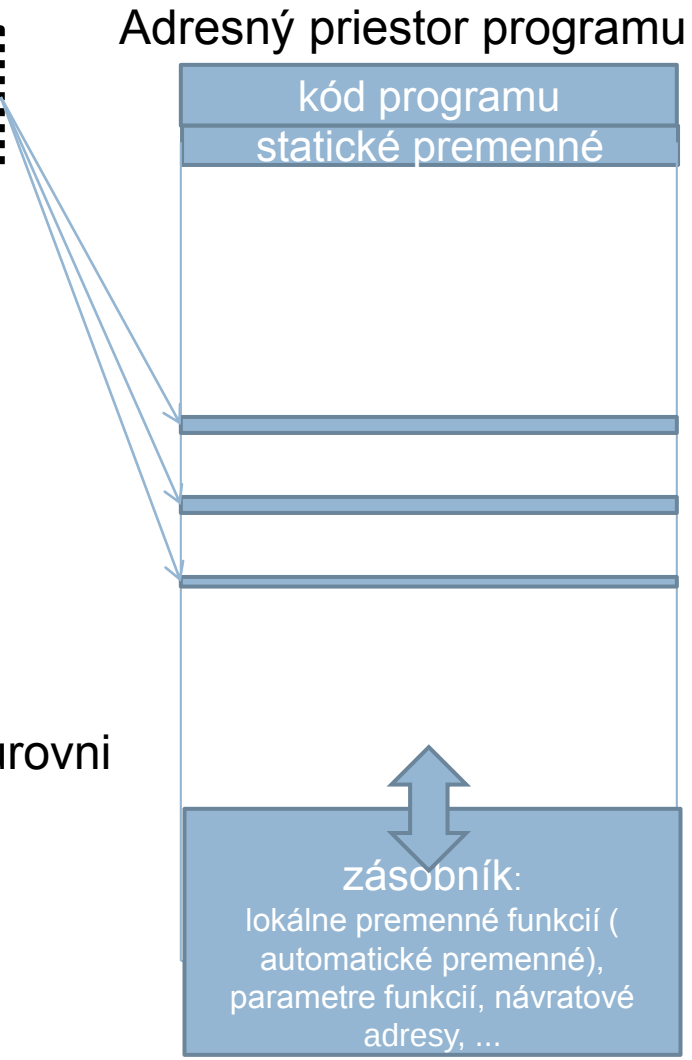


Správa pamäte

- Program počas behu zvyčajne požaduje kusy a kúsky pamäte rôznych veľkostí.
- Program požaduje stále novú a novú pamäť, pričom možno predtým alokované kúsky už nikdy nebude potrebovať.
- Ak by sa pamäť nejakým spôsobom "nerecyklovala", skôr alebo neskôr by došlo k zaplneniu všetkej fyzickej pamäte alebo vyčerpaniu celého adresného priestoru.
- T.j.: Správa pamäte je systém s minimálne dvomi operáciami: alokovanie a uvoľňovanie pamäte.

Riešenie jazyka C

Pamäť alokovaná cez mmap



Alokovanie/uvoľňovanie na úrovni byte-ov:

```
void *malloc(int size)
void *realloc(void *p, int size)
void free(void *p)
```

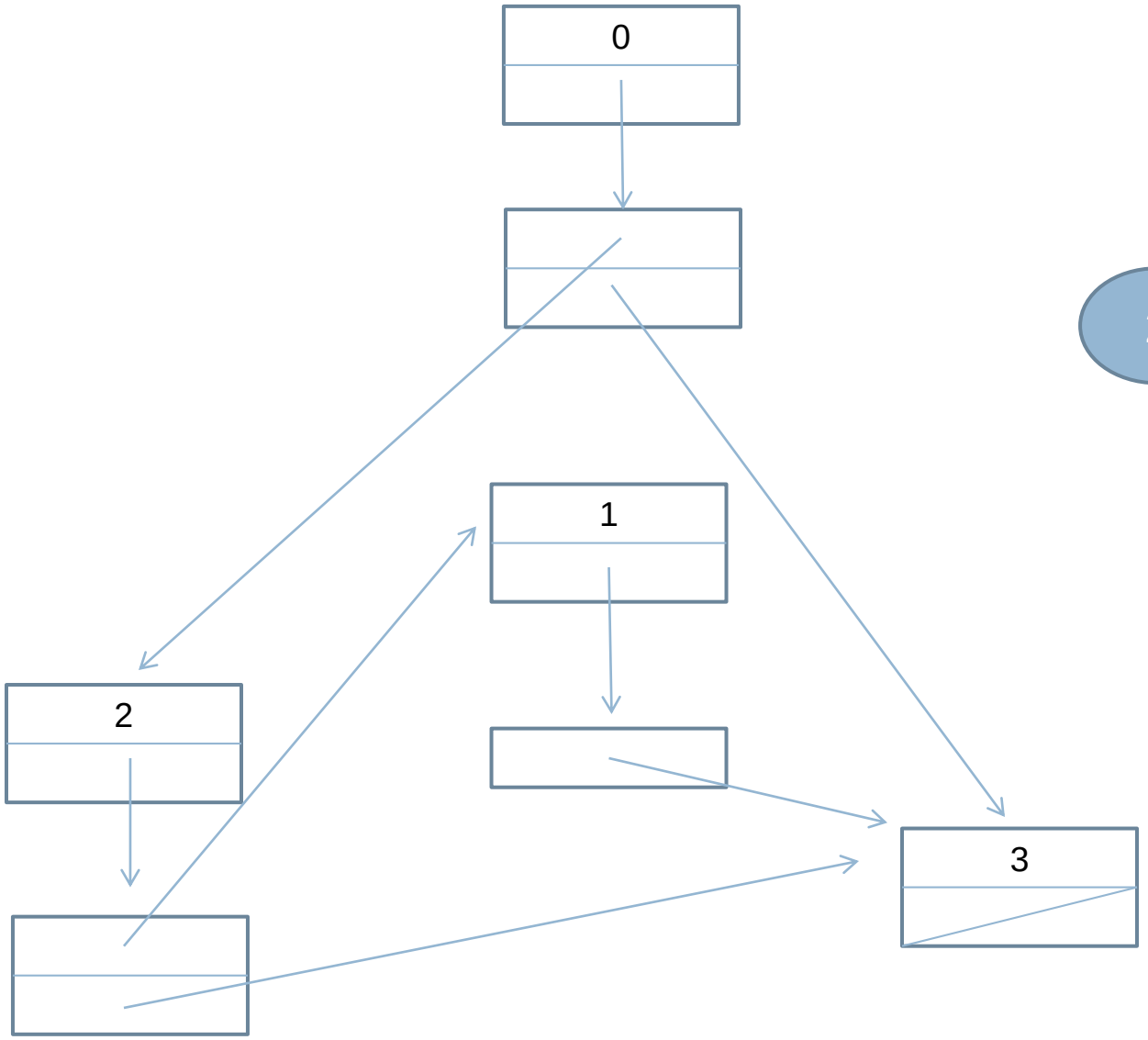
Alebo nízkoúrovňové alokovanie/uvoľňovanie na úrovni stránok pamäte (t.j. násobky 4KB alebo 4MB):

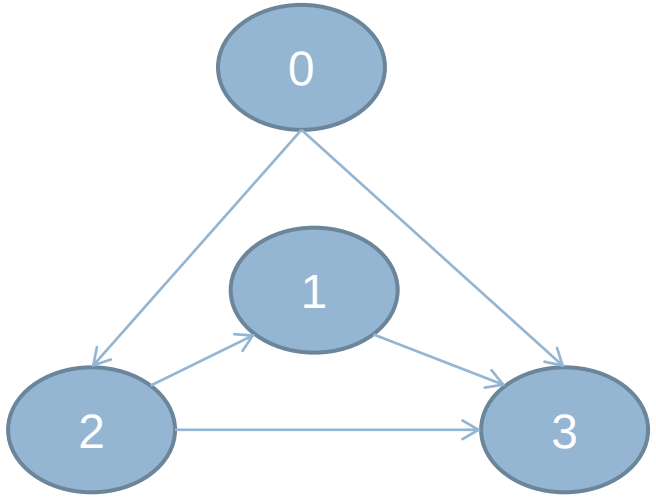
```
void *mmap(   parametre   )
void munmap(void *p, int size)
```

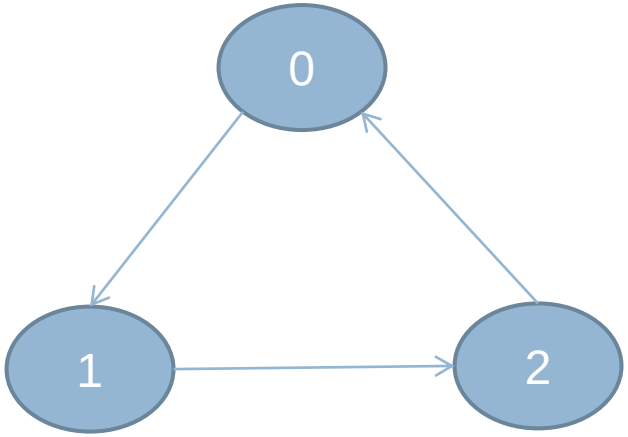
Príklad

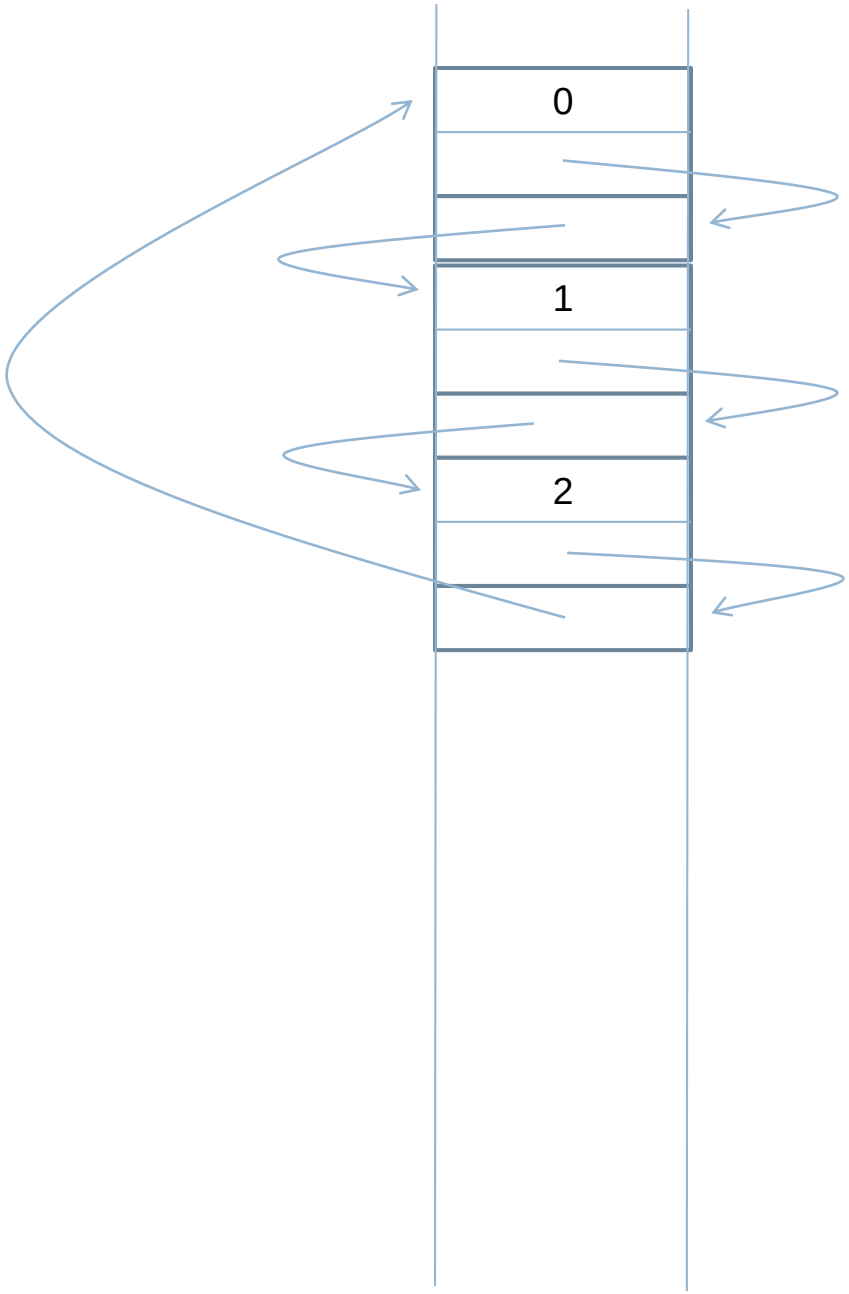
- Predstavme si program, ktorý implementuje dátovú štruktúru "graf", tak že každý vrchol obsahuje pole smerníkov na susedné vrcholy.

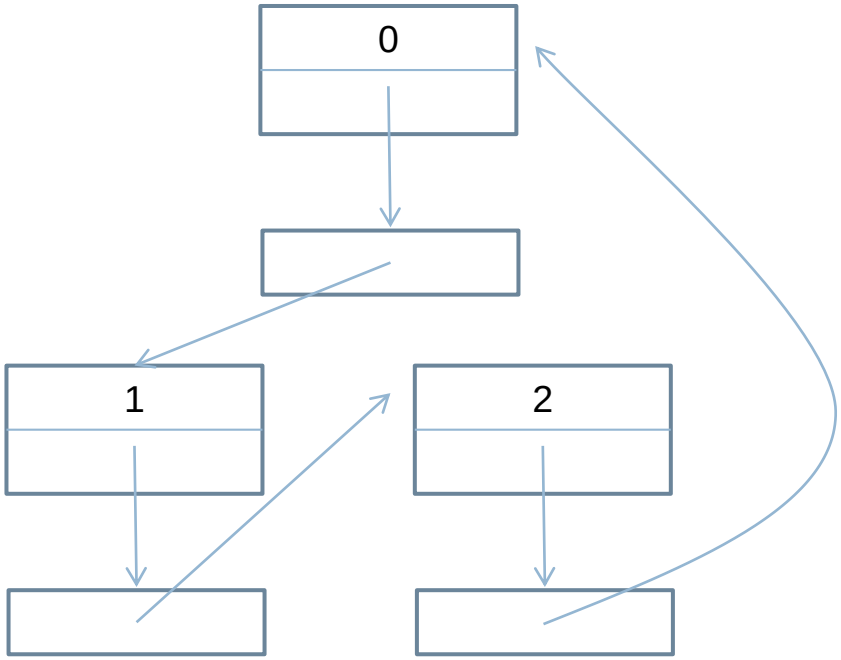
```
struct vrchol {
    int cislo;
    struct vrchol **susedia;
};
```

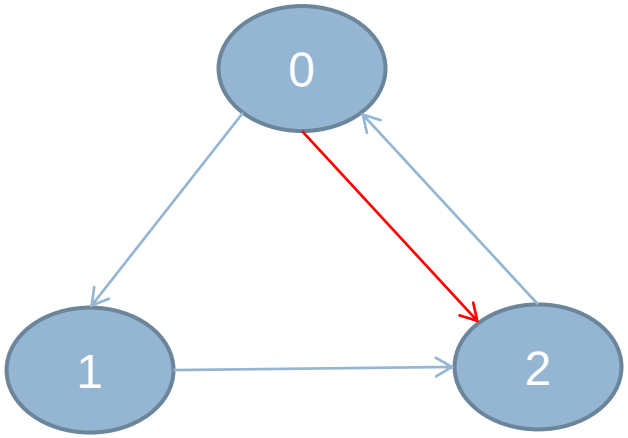

0
1
2
3

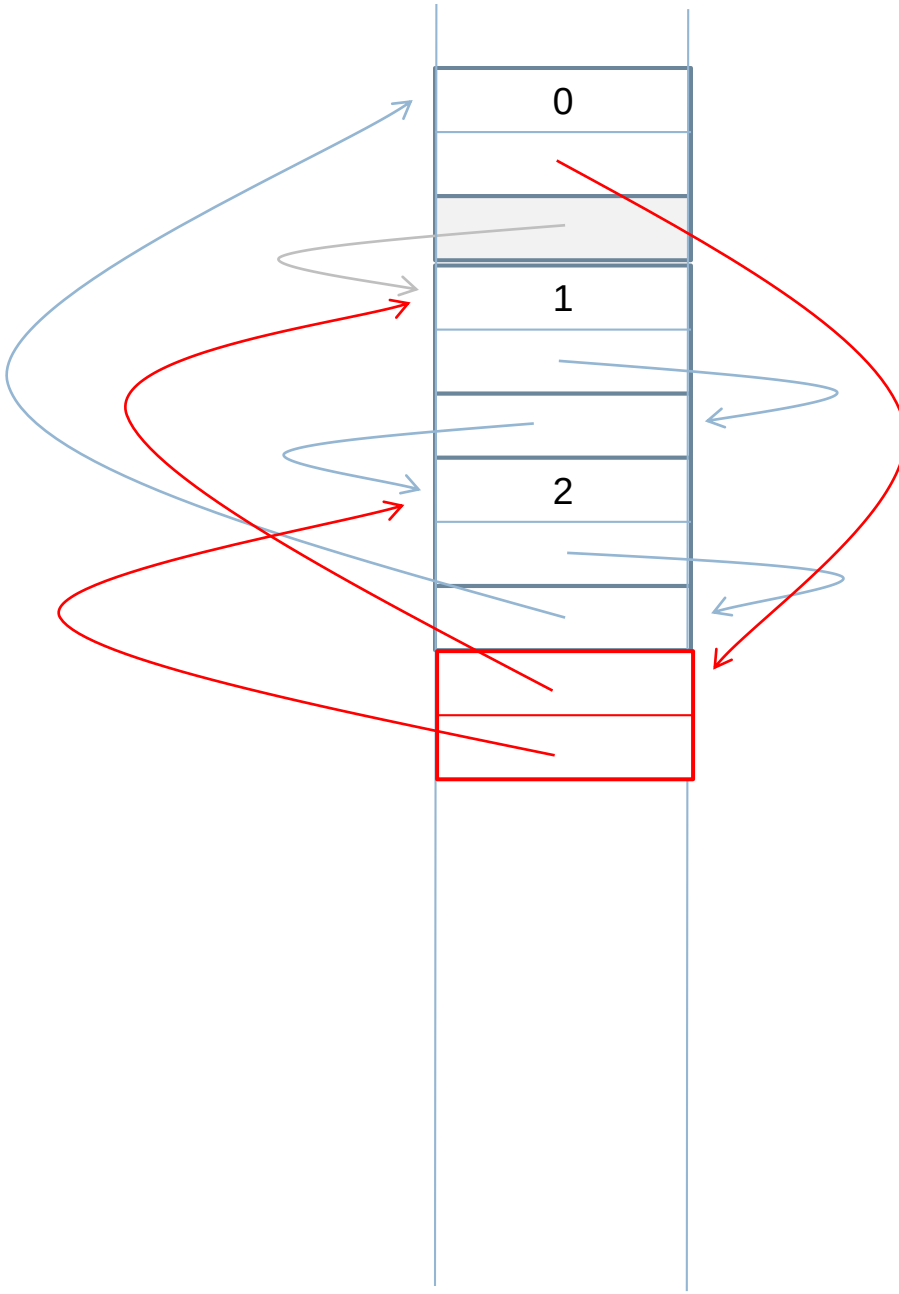

0
1
2
3

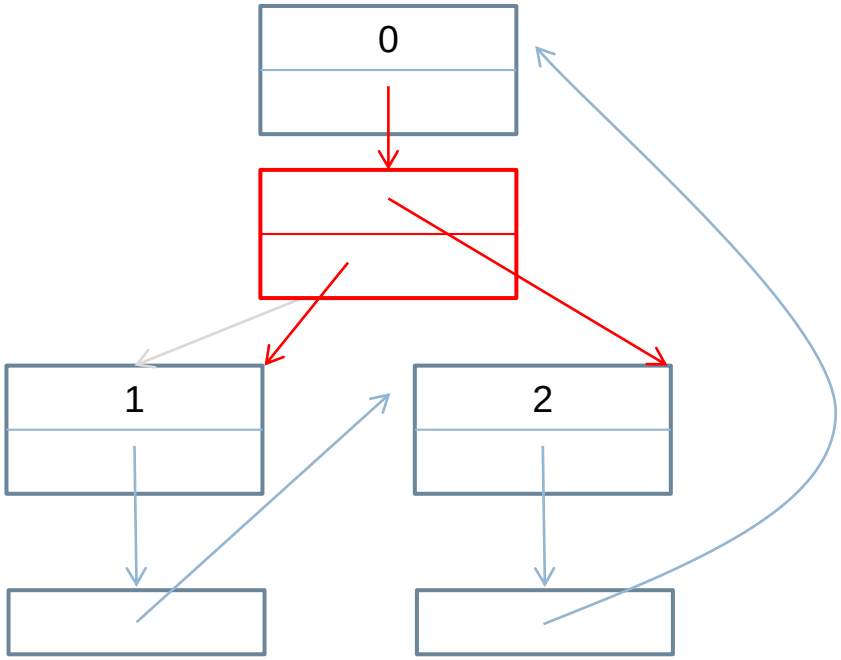

0
1
2

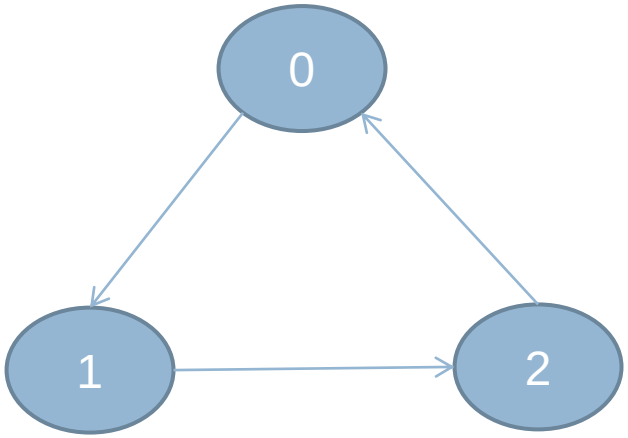

0
1
2

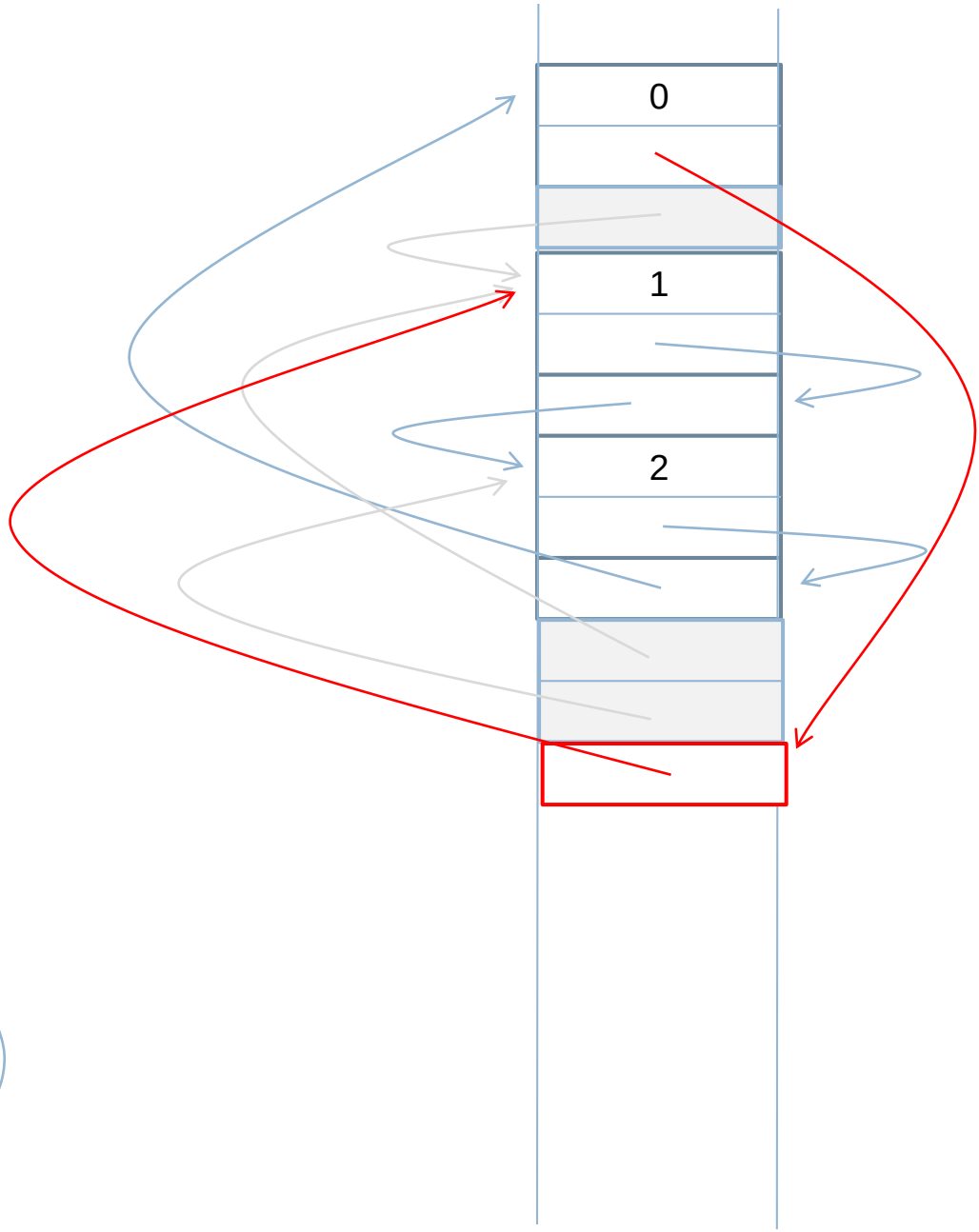

0
1
2

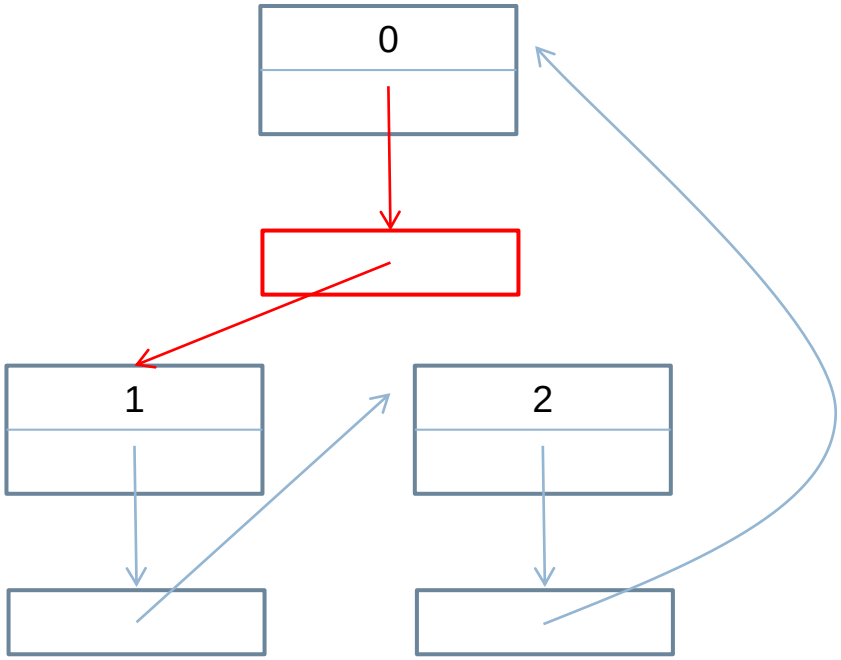

0
1
2

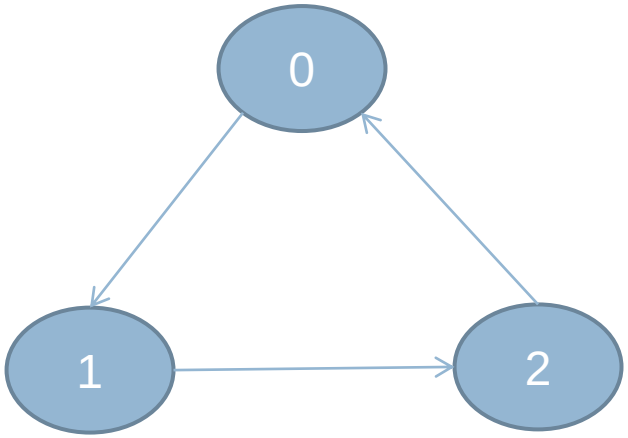

0
1
2

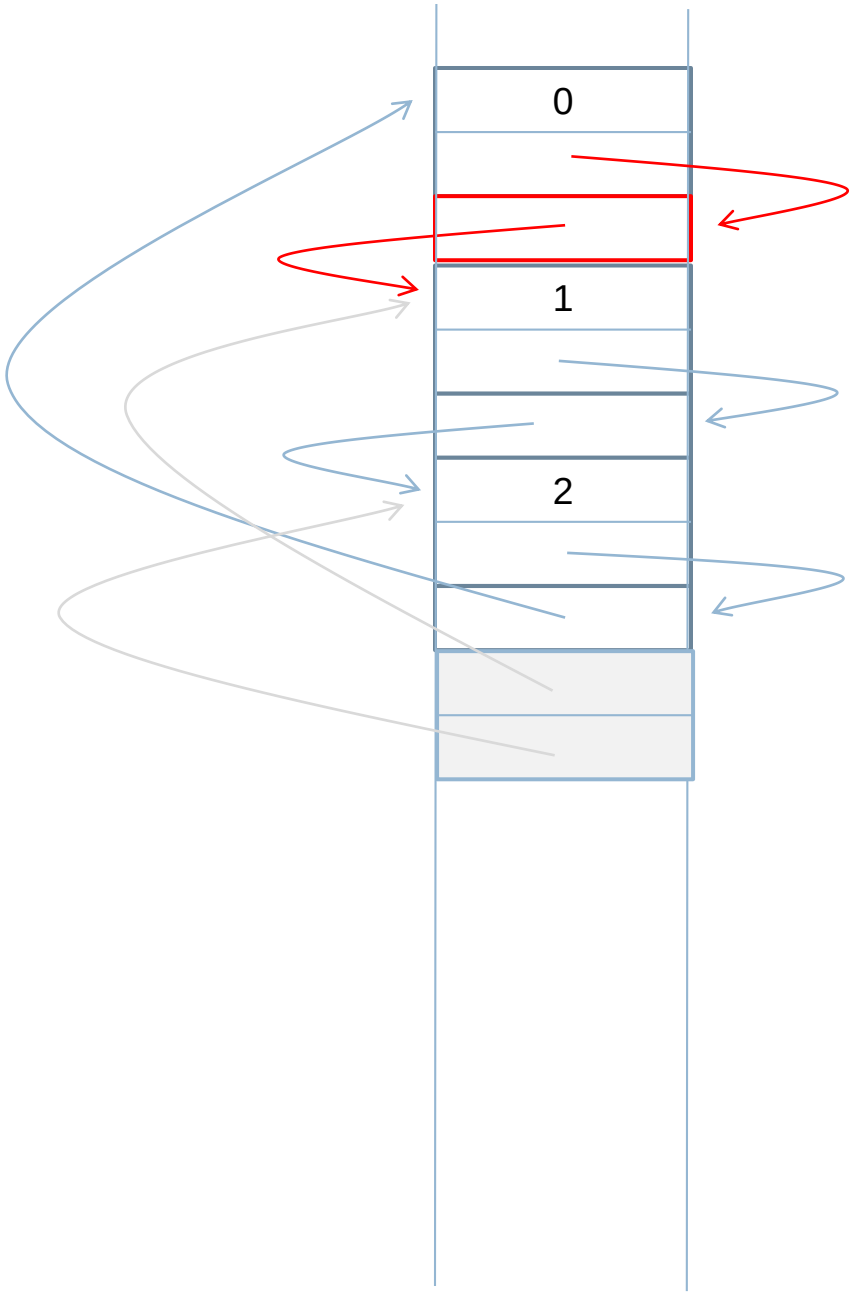

0
1
2

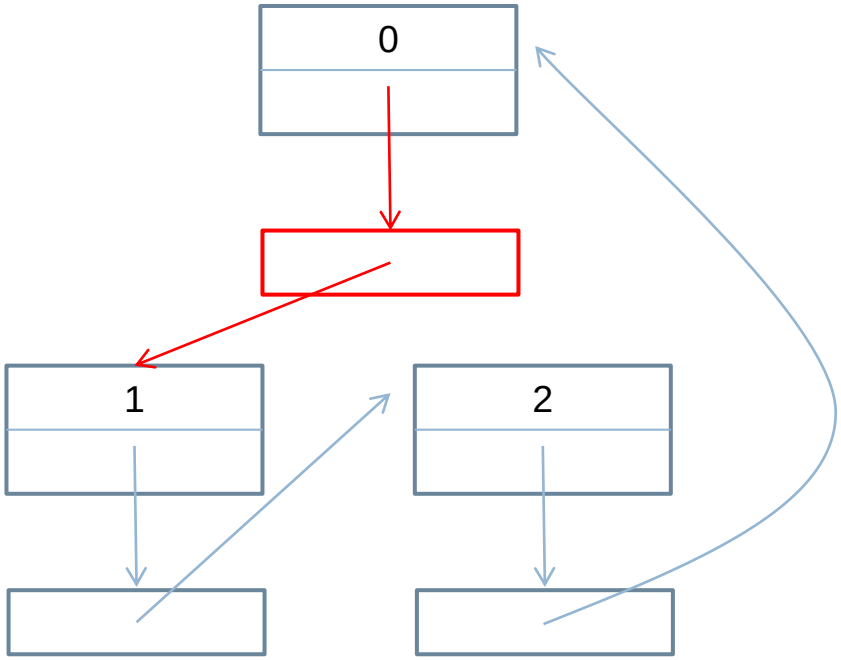

0
1
2


0
1
2


0
1
2


0
1
2


0
1
2


0
1
2

Riešenie jazyka C++

- adoptovanie funkcií malloc, realloc, free, mmap, munmap z C
- zavedenie nadstavby nad týmito operáciami vo forme operátorov `new` a `delete` a možnosť ich preťaženia (overloading of new and delete). Navyše možnosť špecifikovať initializáciu pri novo alokovanej pamäti a "deinitializáciu" pred jej uvoľnením (konštruktory, deštruktory).

Operátor new a delete

```
class A {
public:
    int x;
    A() { x = 0; }
};

int main() {
    A *a;
    a = new A();
    ...
    delete a;
}
```

Vyvolanie new spája dve funkcie:
- alokovanie pamäte
- vyvolanie konštruktora (ak existuje)

Vyvolanie delete spája dve funkcie:
- vyvolanie deštruktora (ak existuje)
- uvoľnenie pamäte

Konštruktory a deštruktory

```
class A {
  int x;

public:

  A() {
     printf("A() ");
  }
  A(int i) {
     printf("A(%d) ", i);
  }
  ~A() {
     printf("~A() ");
  }
};
```

```
int main() {
   A *a = new A();
   delete a;
}
```

```
int main() {
  A *a = new A(1);
  delete a;
}
```

Užívateľom definované operátory new a delete

```
class A {
  int x;
public:

  void *operator new(size_t size) {
    printf("new(%d) ", size);
    return(malloc(size));
  }

  void operator delete(void *p) {
    printf("delete(%p) ", p);
    free(p);
  }

  A() { printf("A() "); }
  A(int i) { printf("A(%d) ", i); }
  ~A() {printf("~A() ");}
};
```

```
int main() {
   A *a = new A();
   delete a;
}
```

```
int main() {
  A *a = new A(1);
  delete a;
}
```

Užívateľom definované operátory new a delete

```
class A {
  int x;
public:

  void *operator new(size_t size) {
    printf("new(%d) ", size);
    return(malloc(size));
  }

  void operator delete(void *p) {
    printf("delete(%p) ", p);
    free(p);
  }

  A() { printf("A() "); }
  A(int i) { printf("A(%d) ", i); }
  ~A() {printf("~A() ");}
};
```

```
int main() {
    A *a = new A();
    delete a;
}
```

Vypíše:

new(4) A() ~A() delete(0x8557008)

```
int main() {
  A *a = new A(1);
  delete a;
}
```

Vypíše:

new(4) A(1) ~A() delete(0x9345008)

Oddelená alokácia a initializácia, príklad

```
class A {
  int x;
public:

  void *operator new(size_t size) {
    printf("new(%d) ", size);
    return(malloc(size));
  }

  void operator delete(void *p) {
    printf("delete(%p) ", p);
    free(p);
  }

  A() { printf("A() "); }
  A(int i) { printf("A(%d) ", i); }
  ~A() {printf("~A() ");}
};
```

```
int main() {
    A x;
    A *y = new A(1);
    delete y;
}
```

Vypíše:

A() new(4) A(1) ~A() delete(0x9d4b008)

Operátor delete pre polia

```
int main() {
    A *a;
    a = new A[10];
    ...
    delete [] a;
}
```

Vyvolanie new:
- alokovanie pamäte
- vyvolanie konštruktora (ak existuje) pre každý prvok pola

Vyvolanie delete:
- vyvolanie deštruktora (ak existuje) pre každý prvok poľa
- uvoľnenie pamäte

Užívateľom definované operátory new a delete pre polia

```
class A {

   void *operator new[](size_t size) {
      ...
   }

   void operator delete[](void *p) {
      ...
   }

};
```

Užívateľom definované operátory new a delete pre polia, príklad

```
class A {
   int x;
public:
   void *operator new(size_t size) {
     printf("new(%d) ", size);
     return(malloc(size));
   }
   void *operator new[](size_t size) {
     printf("new[](%d) ", size);
     return(malloc(size));
   }

   void operator delete(void *p) {
     printf("delete ");
   }
   void operator delete[](void *p) {
     printf("delete[] ");
   }
   A() {printf("A() ");}
   ~A() {printf("~A() ");}
};
```

```
int main() {
    A *a = new A[5];
    delete [] a;
}
```

Užívateľom definované operátory new a delete pre polia, príklad

```
class A {
   int x;
public:
   void *operator new(size_t size) {
     printf("new(%d) ", size);
     return(malloc(size));
   }
   void *operator new[](size_t size) {
     printf("new[](%d) ", size);
     return(malloc(size));
   }

   void operator delete(void *p) {
     printf("delete ");
   }
   void operator delete[](void *p) {
     printf("delete[] ");
   }
   A() {printf("A() ");}
   ~A() {printf("~A() ");}
};
```

```
int main() {
    A *a = new A[5];
    delete [] a;
}
```

Vypíše:

new[](24) A() A() A() A() A()
~A() ~A() ~A() ~A() ~A() delete[]

Preťaženie (overloading) operátora new

```
class A {
public:
  void *operator new(size_t size) {
       printf("new(int) ");
       return(malloc(size));
  }
  void *operator new(size_t size, int x) {
       printf("new(int, int) ");
       return(super_malloc(size));
  }
  A(int y) {printf("A(int) "); }
  A(int x, int y) {printf("A(int,int) "); }
};

int main() {
   A *a;
   a = new A(1);
   a = new(1) A(2);
}
```

Vypíše:
new(int) A(int) new(int, int) A(int)

Preťaženie operátora delete?

- Neexistuje
- Allokátor musí uložiť do pamäte všetku informáciu potrebnú na uvoľnenie.

Nedostatok pamäte

- Ak operátor new vráti NULL (t.j. 0), tak konštruktor sa nevyvolá.

- **`set_new_handler`** (void (*handler)()) umožňuje definovať funkciu, ktorá sa vyvolá ak alokácia zlyhala. Je to pozostatok z čias, keď v C++ neboli výnimky.

Symetria/asymetria new, delete a konštruktorov, deštruktorov

- Konštruktory môžu byť preťažené, operátor new tiež
- Deštruktory nemôžu byť preťažené, operátor delete tiež nie.
- Konštruktory nemôžu byť virtuálne, operátor new tiež nie.
- Avšak! Deštruktory môžu byť virtuálne. Operátor delete nemôže byť deklarovaný ako virtual, ale vyvolá sa operátor delete z triedy, z ktorej sa vyvolal deštruktor (de-facto deštruktor vyvolá "svoj" delete).

Dynamická alokácia pamäte v zásobníku

```
void fun(int x) {
        int a[x];
        ...
}
```

Obe tieto konštrukcie sú povolené v C aj v C++

```
void fun(int x) {
        int *a;
        a = malloca(x * sizeof(int));
        ...
}
```

Diskusia

- Pri programovaní často nevieme či už môžme nejaký kus pamäte uvoľniť alebo nie. Zvyčajne pri komplikovaných zdielaných dátových štruktúrach nie je jasné či nejaká časť štruktúry je ešte niekde použitá alebo nie.
  - Ak miesto uvoľníme (free, delete), a potom ho použijeme cez nejaký zabudnutý smerník ...
  - Ak ho neuvoľníme možno spôsobíme nekontrolovanú spotrebu pamäte (memory leak).

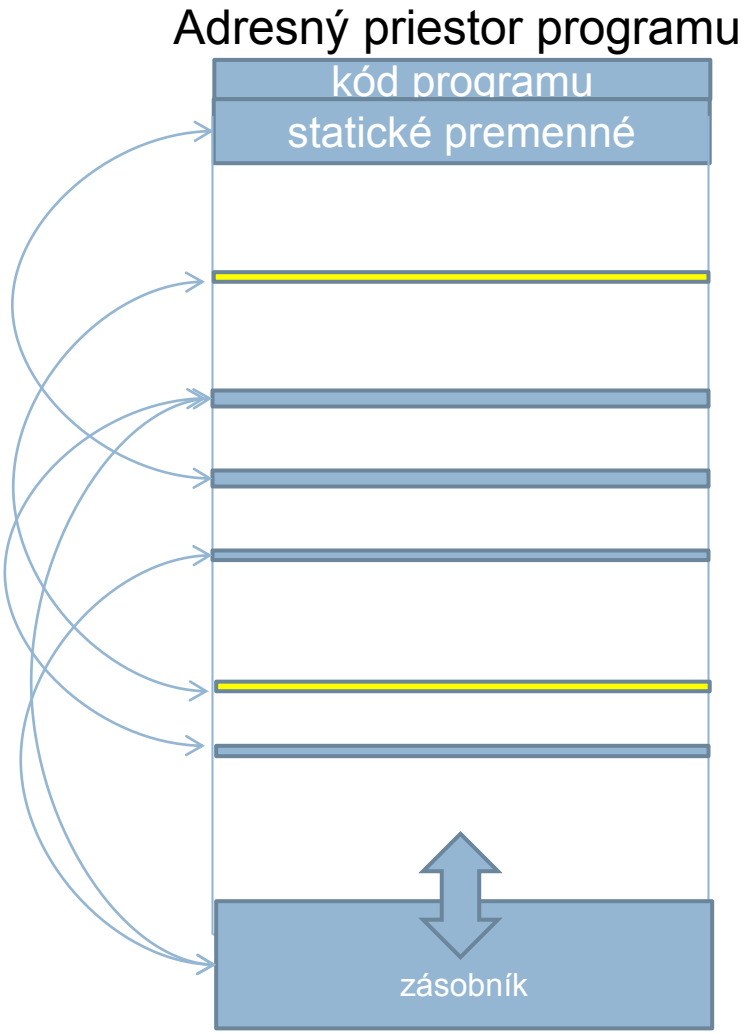
Správa pamäte v jazykoch Java, C#

- Built-in operácia **`new`** a úplne automatizované uvoľňovanie nepoužitej pamäte (Automatic garbage collection).
- Implikácie pre jazyk:

1) Žiadna smerníková aritmetika
2) Zakázanie netypových parametrov (printf, ...)
3) Kontrola indexov polí

--> pointre strácajú zmysel a sú skryté

GC obrázok


Adresný priestor programu
kód programu
statické premenné
zásobník

Je možné automatické uvoľňovanie pamäte v C/C++ ?

- Problém: ako identifikovať, kde v pamäti sú uložené smerníky (pointre) a kde iné dátové typy?

Boehmov konzervatívny GC pre C/C++

- Predpokladá, že pointre nevznikajú ako výsledok aritmetických operácií
- Funguje nasledovne:
  - prehľadá statické dáta a zásobník a každú štvoricu byte-ov považuje za smerník. Ak tento "smerník" ukazuje na alokovanú pamäť usúdi, že pamäť je ešte dostupná.
  - Následne prehľadáva všetky dostupné alokované časti a podobne interpretuje každú štvoricu bytov v nich ako smerníky. Ak ukazujú na alokovanú pamäť pridáva ju k dostupnej pamäte.
  - Ak prehľadal všetky dostupné kusy pamäte, zvyšnú pamäť považuje za nedostupnú a uvoľní ju.