alokace paměti (no)

- typově orientovaná (dynamická) práce s pamětí
- klíčová slova, operátory: **new** a **delete**
- jednotlivé proměnné nebo pole proměnných
- alternativa k **xxxalloc** resp. **xxxfree** v jazyce C, zůstává možnost jejich použití (nevhodné)
- new resp. delete implicitně volají konstruktory resp. destruktory (což malloc a free nedělají !!!)
- lze ve třídách přetížit (volání ”globálních” ::new, ::delete)

```
char*  pch = (char*) new char;

delete pch;
```

Dynamická alokace pro jednu proměnnou (objekt). Lze předepsat

```
void* :: operator new    (size_t)
void  :: operator delete (void *) noexcept

char*  pch = (char*) new char;
// alokace paměti pro jeden prvek typu char
// konverze void* na/z jiného typu je povolena
delete pch; // vrácení paměti pro jeden char

Komplex *kk;
kk = new Komplex(10,20); // alokace paměti
// pro jeden prvek Komplex s inicializací
// zde předepsán konstruktor se dvěm parametry
```

dynamická alokace pro pole hodnot – implicitní konstruktory

```
void* :: operator new[ ] (size_t)
void  :: delete[]        (void *) noexcept

Komplex *pck = (Komplex*) new Komplex [5*i];
// alokace pole objektů typu Komplex, volá se
// implicitní konstruktor pro každý prvek pole
```

Vysvětlete rozdíl mezi **`delete`** a **`delete[]`**

Obě dvě verze zajistí odalokování paměti.
První verze volá destruktor pouze na jeden (první) prvek.
Druhá verze zavolá destruktory na každý prvek v poli.
Každý z destruktorů se tedy chová odlišně. Zavoláme-li obyčejné delete na pole, nemusí dojít k volání destruktorů na prvky (a odalokování jejich interní paměti, vrácení zdrojů…). V případě zavolání „polního“ delete na jeden prvek může dojít k neočekávaným výsledkům (například může očekávat před polem hlavičku s informacemi o počtu prvků pole. Pokud zde hlavička nebude, může dojít k chybné interpretaci dat, které zde budou).

```
delete[]  pck; // vrácení paměti pole
// destruktory na všechny prvky
delete pck;//destruktor pouze na jeden prvek!!
```

zjednodušený zápis/výraz new X se skládá z volání funkčního operátoru new pro získání paměti, volání konstruktoru, výsledkem je ukazatel na alokovaný objekt (či pole objektů)

využívané funkční operátory
new T = new(sizeof(T)) = T::operator new (size_t)
new T[u] = new(sizeof(T)*u+hlavička) = T::operator new[ ](size_t)
new (2) T = new(sizeof(T),2) - první parametr void*
new T(v) + volání uvedeného konstruktoru – zde s jedním parametrem typu jako má v

```
void T::operator delete(void *ptr)
 { // přetížený operátor delete pro třídu T
   // ošetření ukončení ”života” proměnné
  if (změna) Save(”xxx”);
  if (ptr!=nullptr)
   ::delete ptr; //”globální” delete,
// jinak možné zacyklení
…}
```

konstruktory při nenaalokování paměti podle zadaných požadavků používají systém výjimek, konkrétně std::bad_alloc z knihovny <new>.

“původního” vracení nullptr (ve starších překladačích NULL) je možné docílit ve verzi s potlačením výjimek pomocí volby nothrow (opět nutno přidat knihovnu - #include <new>). Nepoužívat není-li to vyloženě nutné

```
char *uk = (char *)new(std::nothrow) char[10];

try{ // alokace paměti generuje výjimky při neúspěchu
     // musíme řešit výjimku bad_alloc
   mat = new TType*[aY]; // vlastní alokace s výjimkou
   for(y = 0; y < aY; ++y)
      mat[y] = new TType[aX]; // další alokace s výjimkou
}
catch(std::bad_alloc)
{
   if(mat) // pokud došlo v minulém bloku k výjimce
      deallocate(mat, y - 1); // odalokujem co se naalokovalo
   throw(ENoMemory); // pošlem výjimku dále, změníme její typ
}
```