Dynamická paměť

Automatická alokace a zásobníková proměnná

Proměnné, se kterými jsme dosud pracovali, byly vytvořeny na *zásobníkovém* segmentu paměti. Jakmile se vykonávání kódu dostane na začátek bloku (znak `{`), všechny „běžné“ proměnné v něm deklarované se postupně alokují za sebou od paměťového místa určeného registrem ESP (*Stack Pointer*). Po alokaci každé proměnné se tento ukazatel zvýší o velikost proměnné, po skončení bloku (znak `}`) se ukazatel přesune do místa, na které ukazoval před začátkem bloku (zapamatované v registru EBP, *Base Pointer*). Tím jsou proměnné v daném bloku *automaticky* uvolněny: proměnné následujícího bloku se budou zapisovat na místo uvolněných proměnných. Této alokaci se proto říká *automatická* a proměnným *zásobníkové*.

Statická alokace a statická proměnná

V případě řetězcových literálů jsme se setkali se *statickou* alokací paměti. Ta nepodléhá EBP a ESP, je vyhrazena ve vlastním segmentu na začátku programu a uvolněna na konci. Statické proměnné jsou také všechny globální proměnné (mimo jakýkoliv blok) a ty, které jsou deklarovány jako `static`. Statické proměnné jsou narozdíl od těch zásobníkových vynulovány, pokud nejsou explicitně deklarovány.

```
void rising() {
  static int i = 3; // statická alokace, pouze na začátku programu
  printf("%d ", i++);
}

int main() {
  for(int i=0; i<5; ++i) rising(); // 3 4 5 6 7
}
```

Lokální proměnné mají rozsah platnosti (*scope*) blok, ve kterém jsou definovány. Globální proměnné mají rozsah platnosti soubor[1].

Dynamická alokace a statická nebo zásobníková proměnná

Třetí typ alokace, *dynamická*, se alokuje na segmentu paměti typu halda (*heap*) v době volání funkce `malloc` a uvolňuje se v době volání `free` – tedy za běhu programu, proto *dynamická*. Správa paměti halda zajistí alokování souvislého bloku v aktuálně dostupné paměti, což je náročnější operace než prosté posunutí zásobníku. K této paměti se dostaneme pouze pomocí proměnné typu ukazatel, který může být statický nebo zásobníkový. V případě zásobníku hrozí ztráta ukazatele na dynamickou paměť, ke které pak do konce běhu programu není možné přistoupit, tedy ani ji uvolnit. Tomuto se říká *únik paměti* (*memory leak*), nezachytí to kompilátor a nedojde k běhové chybě, roste však potřebná paměť (často nad možnosti architektury). Následující funkce jsou z knihovny `stdlib.h`:

[1] Z jiného souboru lze na ně odkázat deklarací `extern`. To je užitečné zejména u statických členů tříd, o tom později.

```
void* test() {
  void* str = malloc(20); // str je zásobníkový ukazatel na dyn. paměť
  return str;
} // konec bloku, uvolňuje se str, ale ne odkazovaná paměť

void main() {
  void* a = test();
  free(a); // a odkazuje na uvolněnou paměť
  a = test(); // a opět ukazuje na alokovanou paměť
  a = test(); // únik 20B paměti, přepsán ukazatel!
}
```

Malloc

Funkce `malloc` očekává jako parametr počet bytů, které má vyhradit a vrací ukazatel, který nemá žádný typ (tj. ukazuje na místo v paměti a nesděluje, jakým způsobem se mají odkazovaný byty interpretovat). Tento ukazatel `void*` je možné přetypovat na cokoliv, ale pouze explicitně, viz příklady:

```
char* str = (char*)malloc(20); // dynamická alokace 20 znaků
int* nums = (int*)malloc(20*sizeof(int)); // dynamická alokace 20 čísel
```

To je vcelku zdlouhavý rutinní zápis, zde si lze usnadnit práci preprocesorem: definováním makra s parametry:

```
#define talloc(N,T) (T*)malloc((N)*sizeof(T))
int* nums = talloc(20,int);
```

Všimněte si závorky kolem N `(N)`. Preprocesor provede nahrazení textu bez jakékoliv analýzy. Bez závorek by se volání např. `talloc(10+x,int)` vyhradilo paměť pro `x` čísel a 10 bytů k tomu.

Free

`free` očekává argument v podobě ukazatele na dynamicky alokovanou paměť (libovolného typu, `void*`), kterou uvolní. Pokud argument ukazuje na jinou paměť (např. již uvolněnou), chování je nedefinované (tj. někde a někdy může fungovat, jindy způsobí pád aplikace nebo ještě horší věci).

Realloc

Tato funkce změní velikost dynamicky alokované paměti a pokud možno při tom použije předchozí paměť (garantováno v případě zmenšení).

```
int* nums = (int*)malloc(20*sizeof(int)); // alokována paměť pro 20 čísel
nums = (int*)realloc(30*sizeof(int)); // změna alokace na 30 čísel
```

Calloc

`malloc` ani `realloc` nově alokovanou paměť nijak neinicializují. `calloc` alokuje dynamickou paměť a vynuluje ji. Pozor, má dva parametry:

```
int* num = (int*)calloc(20, sizeof(int)); // alokováno a vynulováno
```

Memcpy

Tato funkce z knihovny `string.h` efektivně kopíruje blok paměti, např.

```
int src[] = {13, 21, 86, 13, 25, 22, 60};
int dst[10];
memcpy(dst+2, src+3, 4*sizeof(int));
for(int i=2; i<2+4; ++i) printf("%d ", dst[i]);
```

zkopíruje 4 čísla z pole src počínaje čtvrtým (`src+3`) od třetí pozice v dst (`dst+2`).