Dynamická alokácia

Náhradná Prednáška 7, PROG-2, 2020
Pavol Zajac

Poznámka: *Na prednáške zameranej na tému dynamická alokácia štandardne ukazujem dosť veľké množstvo vecí priamo v debuggeri a kreslím na tabuľu, pre objasnenie, ako to funguje. Plánoval som nahrať video, ale našiel som dve dobré videá (a v lepšej kvalite ako viem momentálne v home office skonštruovať), ktoré túto problematiku pokrývajú v približne tom rozsahu, ako to bežne prednášam:*

Používanie pamäte a koncepcia dynamickej pamäte:

https://www.youtube.com/watch?v=_8-ht2AKyH4

Využívanie knižničných funkcií jazyka C na prácu s dynamickou pamäťou:

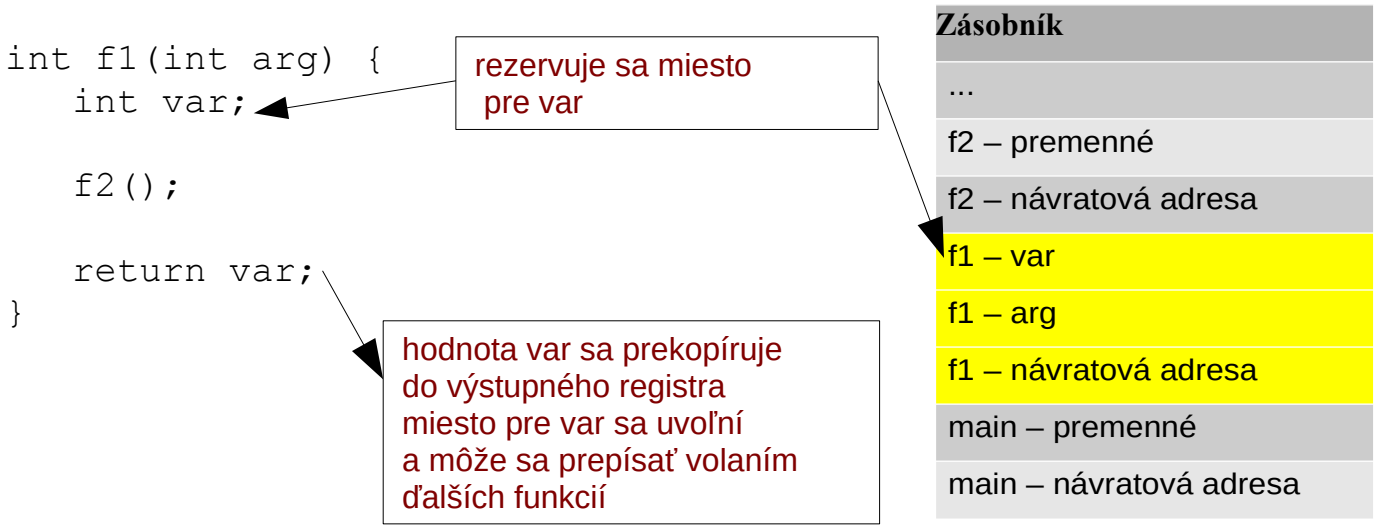
https://www.youtube.com/watch?v=xDVC3wKjS64

Je to v jednoducho zrozumiteľnej angličtine, ale je tam aj transkript a titulky, takže ak aj máte problém s angličinou, môžete si to dať preložiť googlu...

V každom prípade, sumarizujem dôležité informácie aj v textovej podobe. Na odskúšanie si rôznych typov alokácie reťazcov (platí aj pre iné typy polí) som pripravil súbor `p07-01.c`. Príklad v súbore `p07-02.c` potom umožňuje pozrieť si, ako sa dá využiť dynamická alokácia a `realloc` na vytváranie poľa, ktoré sa dá zväčšovať, podobne ako Python list.

1. Automatická alokácia

Každá lokálna premenná /vrátane polí/ má pridelenú pamäť automaticky na zásobníku ("*stack*"). Pamäť sa rezervuje pri volaní funkcie, a po skončení funkcie sa uvoľní na použitie ďalšími funkciami.



Automatické premenné sú po skončení funkcie neplatné. Pri novom volaní funkcie sa nanovo inicializujú (ak nemajú definovanú hodnotu, závisí na prekladači, ale štandardne sa použije to, čo je momentálne v pamäti, ktorá sa rezervuje).

Aj keď ako programátor môže na ne získať referenciu (adresu, ktorú si uloží do nejakého pointera), pri používaní týchto adries nie je garantované, že sa pamäť neprepíše (zväčša sa prepíše už pri prvom volaní inej funkcie).
Ak vraciame z funkcie smerník, mali by sme zaistiť, že neodkazuje na lokálne automatické premenné!

Príklad:

```
int* fn(){
    int pole[10];    //docasne pole
    ...
    return pole;
     //toto prekladac prelozi s Warningom,
     // ale logicky je to zavazna chyba!
}
//po skonceni fn je totiz adresa pole neplatna, odkazuje na pamat,
// ktora sa volanim hocijakej dalsej funkcie prepise
```

2. Statická alokácia

Globálne premenné a lokálne premenné označené ako `static` majú k dispozícii pamäť rezervovanú počas celého behu programu. V celom programe sa nachádza len 1 kópia globálnej/statickej premennej. Statická premenná si tak zachováva obsah medzi viacerými volaniami tej istej funkcie, a existuje aj medzi volaniami funkcie (dajú sa s ňou robiť operácie, ak na ňu máme odkaz cez nejaký smerník).

```
int pole[10];    //globalne pole
int* fn(){
    ...
    return pole;
}
//po skonceni fn vrati vzdy adresu toho isteho pola
// je to ale dost zbytocny smernik,
// kedze globalnu premennu vieme pouzit priamo

int* fn(){
    static int pole[10];    //staticke pole
    ...
    return pole;
}
//po skonceni fn vrati vzdy adresu toho isteho pola
// vdaka tomu smerniku moze vidiet volajuca funkcia obsah lokalnej
// statickej premennej z fn, a dokonca ho aj zmenit /ak nie je const/
```

3. Dynamická alokácia

Dynamickú alokáciu používame, keď nám nevyhovujú predošlé typy alokácie premenných, alebo keď chceme mať alokáciu dát viac pod kontrolou. Princípom je:

1. požiadame systém správy pamäte za behu programu o rezerváciu nejakého množstva pamäte (nemusíme vedieť koľko už pri preklade).

2. Systém pohľadá voľnú pamäť v špeciálnej pamäťovej oblasti na to určenej (*heap*, hromada), ak má pamäť k dispozícii, vráti adresu tejto pamäte.
3. Keď máme adresu pamäte, vieme si ju uložiť do vhodného smerníka, a ďalej ju používať pomocou operátorov `*` a indexovania `[]`.
4. Ak už pamäť na nejakej adrese nepotrebujeme, oznámime to systému na správu pamäte, a ten zruší rezerváciu pamäte. Táto sa teda bude dať znovu používať, keď budeme potrebovať rezervovať pamäť na iný účel.

Dynamická alokácia v jazyku C **nie je** podporovaná samotnou syntaxou jazyka C, ale je podporovaná knižničnými funkciami (deklarovanými v stdlib.h).

Knižničné funkcie umožňujú:
- rezervovať si pamäť, keď ju potrebujeme, a koľko potrebujeme (`malloc`, `calloc`, `realloc`)
- zmeniť veľkosť rezervovanej pamäte bez straty dát (`realloc`)
- uvoľniť pamäť, alebo jej časť, keď ju nepotrebujeme (`free`, `realloc`)

Poznámka: dynamickú alokáciu si môže programátor riešiť aj svojimi funkciami sám. Napr. si vytvoríte veľké statické pole, a z neho rezervujete položky, keď ich potrebujete. Knižničné funkcie sú pomerne rýchle, a spolupracujú s OS, ale niekedy dokážete pomocou vlastnej správy pamäte (alebo správneho používania knižničných funkcií) program značne urýchliť.

3.1. malloc

Funkcia `void* malloc(size_t bytes)` vyčlení pamäť pre požadovaný počet bajtov (parameter `bytes`) a vráti ich adresu. Pamäť nie je inicializovaná, a teda hodnoty sú nedefinované (toto zrýchľuje alokáciu).

Poznámky k použitiu:
- programátor musí určiť **počet bajtov**, ktoré potrebuje. Pomôcť nám môže operátor `sizeof`. Operátor `sizeof` sa dá použiť s premennou alebo dátovým typom, a určuje, koľko **bajtov** daná premenná, alebo údaj zadaného typu **zaberá v pamäti**. Napr. ak potrebujeme miesto, do ktorého sa zmestí 10 celých čísel, vypočítame ho výrazom `10*sizeof(int)`.
- `malloc` (a ďalšie funkcie) vracajú adresu typu `void*`. To je preto, že alokačný systém nevie, ako chcete ďalej pamäť používať. To znamená, že výstupná adresa sa nedá priamo použiť na smerníkovú aritmetiku, alebo sprístupnenie dát. Ak chcete pamäť použiť, je potrebné si adresu uložiť do smerníka vhodného typu, napr. ak chceme robiť s celými číslami, tak do `int*`. Syntakticky sa tiež používa explicitné pretypovanie výsledku `malloc` pred uložením do smerníkovej premennej (viď príklad).
- volanie funkcie malloc (a ďalších) môže zlyhať, ak nie je dosť použiteľnej pamäte. V tom prípade vráti adresu 0 (označuje sa aj makrom NULL), ktorá je v bežnom programe neplatná. Ak sa pokúsite sprístupniť túto adresu, program "spadne" (vyvolá sa systémová výnimka).

Príklad:
```
int n = 5;
int *ptr;   //v pointeri ptr je zatial nedefinovana adresa
// rezervujeme miesto na n int-ov, a zapiseme adresu do ptr
ptr = (int*) malloc( n * sizeof(int) );
*ptr = 0;    //do rezervovanej pamate mozeme pristupovat cez *
ptr[1] = 1;  //ale aj cez indexy
```

Poznámka: Keď si rezervujeme súvislú pamäť vhodnej veľkosti na uchovanie *n* prvkov nejakého typu, a jej adresu uložíme do smerníka, môžeme vďaka smerníkovej aritmetike pristupovať ku rezervovaným *n* pamäťovým miestam ako keby to bolo bežné pole (odovzdané funkcii). Hovoríme, že sme vytvorili dynamické pole.

Podobne ako keď odovzdáme pole do funkcie, dynamické pole je reprezentované len začiatočnou adresou (v príklade je uložená v `ptr`). Je preto potrebné si okrem tejto adresy pamätať, aj koľko prvkov sa zmestí do rezervovanej pamäte (v príklade je uložená v premennej `n`). Dynamické pole môžeme poslať aj do ďalších volaných funkcií, tak ako by to bolo obyčajné pole. Keďže dynamická pamäť sa neuvoľňuje automaticky, môžeme jej adresu vrátiť z funkcie, a používať dynamické pole aj vo volajúcej funkcii (a inde v programe, kam si pošleme adresu začiatku dynamického poľa). Viď príklady v `p07-01.c`.

Príklad: funkcia vytvorí dynamickú kópiu poľa

```
int* copy(int pole[], int n){
    int *nove;

    //najprv si rezervujeme novu pamat cez malloc
    nove = (int*) malloc(n * sizeof(int));

    //moze sa stat, ze nie je dost pamate
    // v opacnom pripade tam prekopirujeme povodne pole
    // na kopirovanie sme mohli pouzit cyklus, alebo
    // rychlu kniznicnu funkciu memcpy
    if (nove != NULL)
        memcpy(nove, pole, n*sizeof(int));

    return nove;
}
```

3.2. calloc

Funkcia `void* calloc(int pocet, size_t velkost_jedneho)` vyčlení pamäť požadovaný počet údajov v pamäti zadanej veľkosti a vráti ich adresu. Hodnoty sa nastavia na 0 (pomalšie volanie ako malloc, ale nemusíme zvlášť inicializovať hodnoty).

Príklad:

```
int n = 5;
int *pole;
pole = (int*) calloc( n,  sizeof(int) );
```

3.3. free

Funkcia `void free(void *ptr)` uvoľní dynamicky alokovanú pamäť (funkciami `malloc`, `calloc` alebo `realloc`). Funkcii ako vstup je potrebné zadať presne tú adresu, ktorú vrátila jedna z

alokačných funkcií. Funkcia `free` nepotrebuje veľkosť, keďže systém na správu pamäte si alokovanú veľkosť poznačí, keď pamäť rezervuje.

POZOR: po zavolaní funkcie `free` zostanú pôvodné adresy uložené v smerníkoch nezmenené! Takéto adresy však ukazujú na už neplatnú pamäť (naša rezervácia bola zrušená, a systém na správu pamäte si tam poznačil pomocné údaje).

Príklad:

```
pole = (int*) malloc( n * sizeof(int) );
pole[0] = 0;
free(pole);
pole[0] = 0; //prelozi sa, ale pocas behu nastane CHYBA!
             // nespravne sa pouzije neplatna pamat
```

Častá chyba je aj viacnásobné volanie `free` s tou istou adresou (*double free*). To spôsobí poškodenie pamäťových štruktúr a potenciálne bezpečnostné zraniteľnosti kódu. Aj vďaka takejto chybe bolo napr. možné spustiť malvér na iOS vložením vhodných dát do animovaného obrázku na webe.

3.4. realloc

Funkcia **`void *realloc(void *ptr, size_t size)`** umožní zmeniť množstvo alokovanej pamäte. Parameter `ptr` označuje adresu doteraz alokovanej pamäte /obsah smerníka sa nezmení/, `size` je nová veľkosť. Funkcia vráti novú adresu, kde bude alokovaná pamäť dostatočnej veľkosti. Dáta, ktoré boli v pôvodne alokovanej pamäti (na adrese `ptr`) sa nestratia (ak sa zmestia do novo alokovaného priestoru).

```
pole = (int*) realloc( pole, n2 * sizeof(int) );
```

Poznámky:
- ak `ptr` je `NULL`, `realloc` funguje ako `malloc`, alokuje novú pamäť zadanej veľkosti
- ak `size` je `0`, `realloc` funguje ako `free`, a vráti `NULL` adresu
- návratová adresa môže byť niekedy totožná s `ptr`, ale nie vždy (nie je miesto, ...)
- ak size je menšia ako pôvodná, pole sa skráti. Časť bajtov sa zachová, zvyšné bajty sa stratia.
- ak sa pole predlžuje, doterajší obsah sa vždy zachová.

Príklad použitia `realloc` je v súbore `p07-02.c`.