Pointers

Náhradná Prednáška 5, PROG-2, 2020
Pavol Zajac

1. Skompilovaný kód a údaje sa všetky nachádzajú v pamäti počítača:
    - kód je postupnosť binárne kódovaných inštrukcií
    - inštrukcie sa zaradom vykonávajú, okrem podmienených a nepodmienených príkazov skokov (tak sa preloží vetvenie, cyklus) a okrem volaní funkcií
    - Funkcia v jazyku C sa zavolá menom: Prekladač si robí tabuľku mien, poznačí si adresu prvej inštrukcie, táto adresa sa použije pri volaní funkcie
2. Pri volaní funkcie:
    - Program pripraví na vyhradené miesto parametre
    - Vykoná sa inštrukcia CALL adresa funkcie
    - Program si počas CALL inštrukcie zapamätá adresu miesta v pamäti, kde by mal pokračovať
    - Na začiatku funkcie sa vyčlení dostatok pamäte na deklarované lokálne premenné a polia, každá premenná a pole má rezervované v pamäti nejaké adresy (aj globálne premenné, ale na inom mieste)
    - Počas behu fukcie sa môže volať iná funkcia. Údaje volajúcej funkcie zostanú v pamäti, pre parametre a premenné novej funkcie sa vyhradí nové miesto. Postupnosť volaní vytvára tzv. CALL STACK, ktorý môžete vidieť pri debug-ovaní programu. Vyskúšajte si to s nejakou väčšou postupnosťou funkcií, alebo rekurzívnou funkciou, napr.:

```c
#include <stdio.h>

void fun(int pole[], int i, int j)
{
    int tmp;
    if (i < j)
    {
        tmp     = pole[i];
        pole[i] = pole[j];
        pole[j] = tmp;
        fun(pole, i+1, j-1);
    }
}

int main()
{
     int test[] = {1,2,3,4,5,6,7,8,9};
     int i, n = sizeof(test)/sizeof(test[0]);

     fun(test, 0, n-1);

     for (i = 0; i < n; i++)
          printf("%i ", test[i]);

     return 0;
}
```

- Keď funkcia skončí, vyhradené miesto sa uvoľní, a inštrukciou RET sa vrátime na adresu, ktorá bola zapamätaná pri CALL. Výsledok funkcie /ak nie je void/ je zapísaný do špeciálneho registra, odkiaľ ho môže prečítať volajúca funkcia.

Situácia v pamäti počas behu (orientačná) pri druhom volaní fun:

100	**0x7045610**	návratová adresa z main	main
108	**0xdead**	i, nedefinovaná hodnota	
112	9	n	
116	9	test[0] - jedno fun sa už vykonalo...	
120	2	test[1] - druhé fun ešte len začína	
	...		
	1	test[9] - toto sa zmenilo prvým fun	
156	**0x7051204**	návratová adresa (z fun do main)	fun (prvé volanie)
160	**112**	adresa začiatku poľa	
164	0	i	
168	8	j	
172	1	tmp = pole[0] sa vykonalo	
176	**0x7051304**	návratová adresa (z fun do fun)	fun (druhé volanie)
	112	adresa začiatku poľa (stále rovnakého)	
	1	i	
	7	j	
	0xbeef	tmp zatiaľ nedefinované	

Boldom zvýraznené údaje sú adresy...

Adresa označuje pozíciu v pamäti (virtuálnej, pre každý program samostatne), je to akoby index v obrovskom poli bajtov. V tomto poli sa nachádza všetko, s čím kód pracuje, vrátane dát, inštrukcií, a v niektorých prípadoch aj I/O portov /zápis na adresu sa fyzicky realizuje poslaním údaju na nejaký hardvér, čítanie z I/O adresy číta stav nejakého hardvéru/.

Pri starých architektúrach adresa bol 32-bitový údaj, väčšina moderných systémov má 64-bitové adresy.

V jazyku C štandardne adresy funkcií a premenných (vrátane polí) pozná prekladač. Ako programátor vieme zistiť adresu akejkoľvek premennej aj funkcie pomocou operátora &:

`&i` - adresa int premennej i

POZOR: pri lokálnej premennej, jej adresa je za behu vždy iná pri rôznych volaniach funkcie fun. Globálne a statické premenné majú konštantné adresy za behu (ale môžu sa líšiť pri rôznych spusteniach programu, závisí na OS). Prekladač vždy vráti adresu tej premennej, s ktorej by sme pracovali, keby sme napísali meno premennej.

`&fun` - adresa funkcie fun (kde sa nachádza v pamäti prvá inštrukcia tejto funkcie)

`&test[0]` - adresa prvého prvku poľa test
`&test[-1]` - adresa údaju, ktorý je v pamäti uložený pred prvým prvkom poľa

POZOR: podobne C neošetruje indexy, môže nám vrátiť aj adresy, s ktorými by sme normálne nemali pracovať

`&pole[0]` - adresa prvého prvku poľa, to isté ako `&test[0]`
`&test` - adresa začiatku poľa test vo funkcii main, to isté ako `&test[0]`
`&pole` - adresa premennej, v ktorej má funkcia fun uloženú adresu poľa test

POZOR: `&pole` vráti inú hodnotu ako `&test[0]`, a táto adresa sa mení pri každom volaní fun. `pole` samotné je totiž nová premenná - smerník.

Pozn.: v jazyku C namiesto `&test` a `&fun` stačí napísať `test` a `fun` (platí pre polia, lokálne aj globálne, a funkcie). Toto neplatí pre `&pole`, keďže sa jedná o smerník a nie o pole alebo funkciu!

Smerník je premenná, v ktorej si program ukladá nejakú pamäťovú adresu. V angličtine *pointer,* niekedy sa neprekladá, niekedy sa uvádza "slovenskejší" preklad *ukazovateľ*.

Pozn.: niekedy sa nesprávne miešajú pojmy adresa/smerník (žiaľ aj ja to občas robím). Smerník je premenná, adresa je obsahom tejto premennej, alebo výsledkom nejakého výrazu (zväčša s operátorom &, alebo vyhodnotením výrazu, ktorý osahuje smerníky a artimetiku s nimi).

Syntax smerníkov:

Na vytvorenie smerníka potrebujeme definovať jeho typ. Typ smerníka sa viaže na údaje, na ktoré adresa uložená v smerníku bude odkazovať *(zjednodušene hovoríme, na ktoré smerník odkazuje).* Pri deklarácii smerníku sa použije typ, a pred názov smerníku sa dá operátor `*`:

```
void *vp;   //tzv. beztypový smerník, viď ďalej
int *ip;    //smerník na int, adresa odkazuje na nejaké celé číslo
double *dp1, *dp2;    //dva smerníky na double, hviezdička musí byť pri každom názve

int (*fp)(int);     //smerník na funkciu, ktorej vstupom je int a výstupom je int, ozátvorkovanie *fp je nutné
```

Pozn.: Funkcie môžu vracať adresy. Potom je syntax podobná ako pri deklarácii smerníka:

```
int *f1(int);    //funkcia, ktorá má vstup int a vracia adresu int premennej
```

```
int *f2(int *p);    //funkcia, ktorá má vstup smerník na int s názvom p a vracia adresu int premennej

int* (*pf)(int*);    //smerník na funkciu, ktorej vstupom je smernik na int a výstupom je adresa int premennej
```

Pozn.: Smerník môže obsahovať adresu iného smerníka. Potom je jeho typ smerník na smerník na typ, syntakticky:

```
int **ipp;     //smerník na smerník na int
int ***ippp;     //smerník na smerník na smerník na int
```

Pozn.: aj keď hviezdička pri deklarácii smerníka patrí k názvu, je súčasťou typu. Preto niekedy býva `int* p;` *preferovaný zápis, problém je, ak chceme deklarovať viac smerníkov naraz.*

Pri konverzii typov sa používa celý typ smerníka aj s hviezdičkou v zátvorke:

```
dp = (double*) ip;
```

Po tomto zápise bude v `dp` uložená rovnaká adresa ako v `ip`, ale na dáta na tej adrese sa budeme pozerať ako na double. Vďaka tomu môžeme na tú istú pamäť pozerať rôznymi spôsobmi, napr. na int ako na postupnosť bajtov.

Použitie smerníkov:
Hlavný účel smerníkov je *nepriame adresovanie*: do premennej si poznačíme nejakú adresu, s ktorou chceme pracovať (čítať z nej údaje, zapisovať do nej).

POZOR: Ak napíšeme vo výraze iba názov smerníka, pracujeme vždy iba s adresami.

Na nepriame adresovanie sa používa dereferenčný operátor `*`:

```
int i = 0;  //standardna int premenna s hodnotou 0
int *ip;     //smerník na int, vnutri je zatial nedefinovana hodnota, odkazuje na potenciálne ľubovoľné miesto v pamäti (nebezpečné!)

ip = &i;     //naľavo je smerník, napravo výraz, ktorý sa vyhodnotí ako adresa premennej i, a táto adresa sa uloží do smerníka

*ip = 10;      //naľavo je smerník na int spojený s operátorom *, napravo výraz typu int
// vysledok int vyrazu sa potom uloží na adresu, ktorá sa prečíta z ip, t.j. adresu i, ktorú sme tam predtým uložili

i = *ip * *ip;     //* je teraz v dvoch významoch, ako dereferencia ip, aj ako násobenie. *ip znamená, že sa prečíta z ip tam uložená adresa, potom sa následne z tejto adresy prečíta hodnota (10). Na záver sa výsledok 100 zapíše do i. Tento zápis sa síce tiež deje cez adresu i, ale priamo, túto adresu určuje prekladač.
```

Pozn.: Beztypový smerník `void*` *je vhodný na uloženie adresy premennej akéhokoľvek typu (typové smerníky by mali rešpektovať typ dát). Keďže je smerník beztypový, operátor * nemá zmysel a nedá sa s beztypovým smerníkom použiť! Ak chceme pracovať s dátami na adrese, ktorú máme v beztypovom smerníku, musíme najprv tento pretypovať.*

```
int i = 0;
void p = &i;
*(int*)p = 2;    //moze byt nebezpecne, ak nedame spravny typ
```

Zložitejší príklad, funkcia swap:

```
//vstupom funkcie su dve adresy, ulozene do smernikov p1 a p2
void swap(int *p1, int *p2)
{
     int tmp;

      //zo smernika p1 sa precita adresa, a z hodnota z tej adresy
(operator *) sa ulozi do tmp
    tmp = *p1;

      //zo smernika p2 sa precita adresa, a z hodnota z tej adresy
(operator *) sa ulozi na adresu, ktora sa precita z p1
    *p1 = *p2;

     //zo smernika p2 sa precita adresa, na ktoru sa ulozi hodnota z
premennej tmp
    *p2 = tmp;

        //vysledkom funkcie je, ze sa vymenili hodnoty na zadanych
adresach
       // kedze sme pracovali priamo s pamatou, na ktoru sme dostali
odkaz, modifikovali sme nieco vo volajucej, alebo vyssej funkcii
      // a nemusime uz nic vracat

}

void fun(int pole[], int i, int j)
{
    if (i < j)
    {
     //volanie swap, vstupom musia byt dve adresy intov
      // tieto adresy nemusia odkazovat na nase udaje, mozu to byt ine
adresy, ktore mame dostupne
     swap(&pole[i], &pole[j]);
     fun(pole, i+1, j-1);
    }
}
```

Smerníky a polia.

V jazyku C, názov alokovaného poľa sa vyhodnotí ako jeho adresa (adresa prvého prvku). Túto je možné uložiť do smerníka príslušného typu:

```
int pole[10];  //miesto na 10 intov
int *p;        //miesto na jednu adresu int-u

p = pole;     //zoberie sa adresa pola, zapise sa do premennej p
p = &pole[0]; //v jazyku C to iste, co v predoslom pripade, ale zapisane explicitne: do p sa ulozi adresa prveho prvku pola

*p = 10;      //na adresu, ktoru obsahuje p sa zapise int 10. Teda sa zapise do pole[0]
```

Keď teda voláme funkcie s poľom ako parametrom, v skutočnosti do funkcie vstupuje len 1 adresa. Príslušný parameter funkcie je smerník! Mohli by sme písať (aj to niekde býva použité):

```
void fun(int *pole, int i, int j);
```

Zápis `void fun(int pole[], int i, int j);` je výhodnejší z hľadiska prehľadnosti kódu: programátor vie, že funkcia bude s adresou narábať ako s poľom, a nie teda iba s jedným údajom ako napr. swap. Pre prekladač sú však oba zápisy ekvivalentné.

Otázka je, prečo potom funguje použitie poľa, keď máme vo funkcii len adresu jeho začiatku. Odpoveď je smerníková aritmetika a spôsob, ako v jazyku C funguje operátor indexovania `[]`.

Typové smerníky v jazyku C podporujú **smerníkovú aritmetiku**: operátor sčítania (s celým číslom), operátor odčítania (dvoch smerníkov) a porovnania (dvoch smerníkov navzájom).

Smerníky sa porovnávajú tak, že adresy sa chápu ako čísla, dva smerníky sú zhodné, ak odkazujú na tú istú pamäť. Príklad:

```
int i = 0, j = 0;
int *p = &i, *q = &j;
int cmp1 = (p == q);      //bude mať hodnotu 0, p a q obsahujú iné adresy
int cmp2 = (*p == *q);  //bude mať hodnotu 1, p a q obsahujú síce iné adresy, ale hodnoty na nich (získané cez operátor *) sa zhodujú
```

Keďže adresy sú čísla, môžeme s nimi počítať. Jednou možnosťou je pretypovať adresu na int (resp. long long int, keď je 64-bitová) a používať ju v hocijakom aritmetickom výraze. Opačne, celé číslo je možné skonvertovať na adresu:

```
int *p = (int*) 0xdeadbeef;
```

K adrese však vieme pripočítať a odpočítať celé číslo `n` priamo. Jedná sa o posun adresy, o `n` miest dopredu alebo dozadu. Veľkosť miesta je definovaná typom adresy. Napr. keď sa jedná o adresu int-u, prekladač pripočítanie o 1 interpretuje zväčšením adresy o 4 (bajty), t.j. sizeof(int). Príklad:

```
int pole[10];  //miesto na 10 intov
int *p;        //miesto na jednu adresu int-u

p = &pole[0];     //zoberie sa adresa pola, zapise sa do premennej p

*p = 10;      //na adresu, ktoru obsahuje p sa zapise int 10. Teda sa
zapise do pole[0]

p++;          //posunieme sa v pamäti o 1 int ďalej
*p = 5;       //keďže v pamäti za pole[0] nasleduje hneď pole[1], bude
p obsahovať &pole[1], teda do pole[1] sa zapíše číslo 5

*(p+3) = 7;   //p odkazuje na pole[1], prečítame túto adresu, zvýšime
o 3 pozície. A teda dostaneme &pole[4], a na pole[4] zapíšeme číslo 7

*p  =  *(p-1);     //p odkazuje na pole[1]. Najprv vykonávame pravú
stranu: prečítame adresu z p, znížime ju o 1 (získame &pole[0]),
použijeme operátor * a prečítame číslo 10. Toto zapíšeme na adresu
uloženú v p, t.j. do pole[1]

*p = *p-1;    //keď vynechám zátvorky, * sa deje skôr ako -. V tomto
príklade z p prečítam adresu &pole[1], zoberiem z tade hodnotu 10,
odčítam 1 a výsledok 9 zapíšem do pole[1] (cez adresu prečítanú z p)
```

Zložitejší príklad. Nahraďte v main volanie funkcie fun nasledovne:

```
fun(test+2, 3, 5);
//do funkcie teraz pošleme adresu nie prvého prvku poľa, ale adresu o
dva vyššiu, t.j. &test[2] : to iste ako fun(&test[2], 3, 5);
```

Aký bude výsledok?

Odčítanie smerníkov:

Odčítaním dvoch smerníkov získame počet pozícií medzi nimi. Napr.

```
int pole[10];
int *p, *q;

p = &pole[0];
q = &pole[9];

pole[0] = q-p;   //zapíše 9 do pole[0];
```

Smerníková aritmetika a operátor indexovania. V jazyku C sa po preklade indexovanie realizuje cez adresy a smerníkovú aritmetiku. Platí jednoduchý vzťah:

```
pole[i] == *(pole + i)
```

Ekvivalentne to tiež znamená:

```
*pole == pole[0]

pole + i == &pole[i]

pole[i] == *(pole + i) == *(i + pole) == i[pole]
//ale to posledne neodporucam kvoli citatelnosti kodu ;)
```

Kvôli čitateľnosti operátory &, * je vhodné používať tam, kde logicky robíte s individuálnymi adresami, a indexovanie cez [], kde logika algoritmu hovorí o práci s poľami.

Vzhľadom na to, že názov poľa sa interpretuje ako adresa prvého prvku, a do funkcie vstupujú namiesto polí adresy, dajú sa polia používať cez ekvivalenciu indexovania a smerníkových operácií rovnako vo volanej aj volajúcej funkcii.

Príklad:

```
void print(int data[], int n)
{
     for (int i = 0; i < n; i++)
          printf("%i ", data[i]);
}

int main()
{
     int test[] = {1,2,3,4,5,6,7,8,9};
     int n = sizeof(test)/sizeof(test[0]);

     print(test, n);     //vypise vsetky prvky
     print(test, 5);     //vypise prvych 5 prvkov
     print(test + 5, n - 5); //vypise vsetky, okrem prvych 5 prvkov

     return 0;
}
```

Poznámky:
- vo funkcii main je test pole, nie smerník:
  * Do premennej test priamo nevieme nič zapísať, ale vieme vyhodnotiť jej adresu, ako keby to bol smerník (pri volaní funkcie print sa adresa využije priamo, alebo aj so smerníkovou aritmetikou)
  * Operátor sizeof nám správne vráti veľkosť celého alokovaného poľa

- vo funkcii print je data smerník, nie pole:
  * Do premennej data môžeme zapísať ľubovoľnú adresu, robiť s ňou smerníkovú aritmetiku. Zmenou premennej data bez použitia dereferencie (* alebo indexov) nezmeníme nič v maine (ale môžeme stratiť prístup)
  * Operátor sizeof nám vráti miesto potrebné na uloženie **adresy** a nie veľkosť poľa. Preto je dodatočný parameter s veľkosťou poľa nutný.

Alternatívne riešenie pre výpis prvkov z pamäte zo smerníkového intervalu využitím smerníkovej aritmetiky:

```
//pouzijeme smernikovu syntax, lebo to nepouzivame ako pole,
ale ako premenne s adresami zaciatku a konca useku pamate
void printb(int *start, int *end)
{
     //porovname, ci start je pred end, *end sa nevypisuje
     while (start < end)
     {
          //pokym ano, vypisujeme obsah na adrese start
          printf("%i ", *start);
          //a posuvame adresu vzdy o 1 int
          start++;
     }
}

int main()
{
     int test[] = {1,2,3,4,5,6,7,8,9};
     int n = sizeof(test)/sizeof(test[0]);

     printb(test, test+n);    //vypise vsetky prvky
     printb(test, test+5);    //vypise prvych 5 prvkov
     printb(test + 5, test+n); //vypise vsetky, okrem prvych 5 prvkov

     return 0;
}
```