Kazalci in računalniški spomin

1 Računalniški spomin

Spomin, pomnilnik ali angl. RAM (*Random access memory*) je ključna komponenta v računalniku. Med tekom programa so v njem vse spremenljivke, ki jih program uporablja. S spremenljivkami lahko naredimo tudi več, kot smo se do sedaj naučili, vendar pa moramo za to razumeti, kako je spomin zgrajen.

Osnovna enota za merjenje količine informacij je *bit*. En bit informacij ustreza odgovoru na eno vprašanje tipa da ali ne – če nam nekdo pove, da so vrata zaprta, nam je podal en bit informacij, ker so lahko vrata bodisi odprta bodisi zaprta. Če imamo v omari dva para hlač, dve majici in dve kapi, lahko opišemo, kako smo oblečeni, s tremi biti informacije – za vsak kos oblačila porabimo en bit.

V računalništvu bite najpogosteje označujemo z ničlami in enicami. Običajno ničla predstavlja odgovor "ne" na dano vprašanje, enica pa odgovor "da". Bit pa je zelo majhna količina informacije, zato pogosto govorimo v večjih enotah, kot so *bajti*, *kilobajti*, *megabajti* itd. En bajt ustreza osmim bitom, kilobajt je tisoč bajtov, megabajt je tisoč kilobajtov, gigabajt je tisoč megabajtov, in terabajt je tisoč gigabajtov.

Pogoste napake

Pogosto, a napačno mišljenje je, da kilobajt ustreza 1024 bajtom. Ta mit izvira iz zgodnjih časov računalništva, ko so za hitrejše računanje nekateri programi med spominskimi enotami pretvarjali s to napačno številko; 1024 je namreč potenca 2, s katerimi računalniki pogosto lahko hitreje računajo. Če za pretvorbo uporabljamo faktor 1024, moramo za enote podati *kibibajte* (KiB), *mebibajte* (MiB), *gibibajte* (GiB), ipd., namesto običajnih SI predpon.

Računalniški spomin je sestavljen iz spominskih celic, ki so dolge en bajt. Vsaka od teh celic ima svoj *naslov* – številko, s katero lahko to celico ločimo od ostalih. Naslovi so zaporedne številke od 0 do velikosti pomnilnika, ki ga imamo nameščenega v računalniku. Celice so naraščajoče urejene po svojih naslovih – celica številka 150337 je sosednja celicama s številkama 150336 in 150338.

Upravljanje z računalniškim spominom je ena od nalog operacijskega sistema. Naši programi operacijski sistem med izvajanjem prosijo za neko količino spomina, operacijski sistem pa določi, katere spominske celice bo program prejel. Te celice tedaj pripadajo programu, dokler se ta ne zaključi, ali dokler tega spomina ne vrne operacijskemu sistemu na drugačen način. Med izvajanjem našega programa praviloma noben drug program nima dostopa do tega dela spomina.

Kako pa program ve, koliko spomina bo potreboval? Da to izračuna, se zanaša na tipe. Vsaka spremenljivka ima tip, vsak tip pa ima fiksno dolžino, ki jo zavzame v spominu. Dolžine pogostih tipov so sledeče:

- `int`: 4 bajti
- `long long`: 8 bajtov
- `char`: 1 bajt
- `bool`: 1 bajt

Spremenljivke, ki v spominu zavzamejo več kot 1 bajt, moramo shraniti v več kot eno spominsko celico. Celice, v katere te vrednosti zapišemo, so v spominu zaporedne; če imamo spremenljivko tipa `int`, bo zavzela 4 zaporedne celice.

2 Kazalci

Pogoste napake

Kazalci so pomemben koncept v programiranju, in razumevanje kazalcev je ključno za razumevanje bolj zapletenih podatkovnih struktur ter nekaterih algoritmov. S kazalci pa se lahko zelo hitro zmotimo, in razhroščevanje kode z veliko kazalci je pogosto zelo zapleteno. Zaradi tega in drugih razlogov se kazalcem izogibamo v tekmovalnem programiranju, razen če jih res nujno potrebujemo.

Ker so spominski naslovi številke, jih lahko shranjujemo, kakor shranjujemo ostale številke; za to pa imamo v C++ na voljo poseben tip, ki mu rečemo *kazalec* (angl. *pointer*). Pravzaprav kazalec ni sam svoj tip, ampak razširitev nekega drugega tipa; pravimo, da kazalec *kaže na drug tip*. Da ustvarimo nov kazalec, zapišemo ime tipa, na katerega želimo kazati, nato pa pred ime spremenljivke damo zvezdico `*`. Kazalcu lahko nastavimo vrednost tako, da vanj shranimo naslov neke spremenljivke, ki smo že ustvarili. Do naslova dostopamo z operatorjem `&`. Da dostopamo do vrednosti, shranjene v celici, na katero kazalec kaže, uporabimo operator `*` (ki ima drugačen pomen, kot zvezdica v deklaraciji spremenljivke).

Primer

```
#include <stdio.h>

int main() {
    int a = 7;
    int b = 3;

    // Ustvarimo kazalec na int, ki kaže na spremenljivko a
    // To pomeni, da bo v kazalcu shranjen naslov prve od štirih
    // celic, ki jih zavzema a.
    int *kazalec = &a;

    // Izpišemo vrednost, na katero kaže kazalec
    printf("%d\n", *kazalec);  // 7

    // Če spremenimo vrednost, na katero kaže kazalec, se spremeni
    // vrednost v spominu; torej tudi spremenljivka, ki je tam shranjena
    *kazalec = 15;
    printf("%d\n", a);  // 15

    // Če spremenimo lokacijo, kamor kaže kazalec, nismo spremenili vrednosti,
    // na katero je kazal prej
    kazalec = &b;
    printf("%d\n", *kazalec);  // 3
    printf("%d\n", a);  // 15

    return 0;
}
```

Vse, kar smo sedaj delali s kazalci, je bilo možno (in lažje) narediti tudi z običajnimi spremenljivkami. Kazalci, ki kažejo na spremenljivke, ki tako in tako že obstajajo, so bolj ali manj neuporabni. Kako pa naredimo kazalec, ki kaže na del spomina, v katerem ni nobene spremenljivke?
Če želimo storiti kaj takega, moramo prevzeti odgovornost za upravljanje spomina v našem programu. Do sedaj je za to skrbel prevajalnik, ki je v naš program na pravilna mesta zapisal ukaze, ki si spomin sposojajo od operacijskega sistema, ter ga vračajo, ko ga ne potrebujemo več. Bolj natančno; ko smo

deklarirali spremenljivko, je prevajalnik poskrbel, da prosimo za natanko toliko spomina, kolikor ga za to spremenljivko potrebujemo (zato moramo za vsako spremenljivko zapisati tip), ter si njegov naslov zapomnil, ko pa spremenljivke nismo več potrebovali, je prevajalnik poskrbel, da ta del spomina vrnemo operacijskemu sistemu; temu pravimo *sprostitev* (angl. *deallocation*).
Za bolj sofisticirano uporabo spomina moramo ti vlogi prevzeti mi. Za to sta nam na voljo dve funkciji: `malloc` in `free`. Da ju uporabljamo, moramo vključiti `stdlib.h`. Oblika funkcij je naslednja:

```
void *malloc(size_t size);
void free(void *ptr);
```

V obliki je posebnost, ki je še nismo omenili; ni namreč nujno, da ima vsak kazalec tip. Lahko imamo kazalce, ki kažejo na del spomina, mi pa (še) ne vemo, kaj je v tistem delu spomina shranjeno. Za take kazalce pravimo, da kažejo na `void`, kar pa ne pomeni, da ne kažejo na nič; na lokaciji v spominu, kamor kažejo, je nekaj shranjeno; mi samo ne vemo, kako naj te podatke interpretiramo.
Funkcija `malloc` sprejme en argument, in vrne kazalec na `void`. Ta argument je tipa `size_t`, ki je za naše potrebe skoraj enak tipu `unsigned long long`; to je torej številka. Pove, koliko bajtov spomina si želimo sposoditi od operacijskega sistema. `malloc` nato vrne kazalec na prvi naslov znotraj bloka spomina, ki smo si ga ravno sposodili. Ker operacijski sistem ne ve, kaj bomo v ta spomin shranili, nam `malloc` vrne `void*`, mi pa ga moramo pretvoriti v pravi tip kazalca. To storimo tako, da tik pred klic funkcije v oklepaje zapišemo želeni tip kazalca.
Funkcija `free` je ravno nasprotna od `malloc`; sprejme kazalec, ki ga nam je dal `malloc`, ter sprosti del spomina, na katerega kazalec kaže. Kazalec bo po klicu `free` še vedno obstajal, in bo še vedno kazal na isto mesto. Edina sprememba je, da del spomina, na katerega kaže, ne pripada več našemu programu, in ga ne smemo uporabljati.

Primer

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *kazalec = (int*)malloc(sizeof(int));
    *kazalec = 3;
    printf("%d\n", *kazalec);  // izpiše 3
    return 0;
}
```

Operator `sizeof` lahko uporabljamo, da si pomagamo pri določevanju velikosti tipov.

3 Kako delujejo seznami

Nič nas ne omejuje, da od operacijskega sistema zahtevamo zelo velik blok spomina, tudi po več sto tisoč bajtov. Pa imamo lahko kakšen utemeljen razlog, da si toliko spomina izposodimo? Da, ravno to stori prevajalnik, ko ustvarimo seznam. Poglejmo si, kako ustvarimo seznam samo s kazalci.

Primer

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int dolzina_seznama = 100000;
    int *seznam = (int*)malloc(dolzina_seznama * sizeof(int));

    for (int i = 0; i < dolzina_seznama; i++) {
        scanf("%d", seznam+i);
    }

    // sedaj lahko spremenljivko ‘seznam‘ uporabljamo
    // kot katerikoli drugi seznam
    int vsota = 0;
    for (int i = 0; i < dolzina_seznama; i++) {
        vsota += seznam[i];
    }

    printf("%d\n", vsota);

    // ne smemo pozabiti sprostiti spomina
    free(seznam);

    return 0;
}
```

V zgornjem primeru uporabljamo dva nova operatorja na kazalcih; seštevanje in oglate oklepaje. Če kazalcu `seznam` prištejemo število `i`, dobimo nov kazalec, ki kaže na mesto `seznam + (velikost tipa) * i`, torej na idealno mesto, kamor zapišemo `i`-ti element seznama, če jih zapisujemo enega za drugega.
Drug novi operator so oglati oklepaji – ti se obnašajo popolnoma enako kot v seznamih. Oglati oklepaj `seznam[i]` je pravzaprav krajšava za zapis `*(seznam+i)`, torej za dostop do `i`-tega elementa v bloku spomina.
Tudi seznami, kakor smo jih spoznali prej, so dejansko kazalci na blok spomina, le da s tem spominom upravlja prevajalnik. Trik s prištevanjem števila k kazalcu deluje tudi za prištevanje števila k seznamu.

4 Podajanje po referenci

Opazimo, da smo operator `&` že srečali, in sicer čisto na začetku. Pri branju številk iz vhoda moramo v `scanf` zapisati ta operator pred imenom spremenljivke. Sedaj razumemo, zakaj je temu tako; `scanf` sprejme kazalce na spremenljivke, ki jih želimo prebrati, ter popravi vrednosti, na katere kažejo kazalci, s prebranimi vrednostmi.

Primer

`scanf` je tudi funkcija, le da je nismo zapisali mi. Zmožna pa je nečesa, česar naše funkcije niso sposobne; spremeniti vrednosti spremenljivk zunaj nje. Spodnji program se ne bo niti prevedel:

```
#include <stdio.h>

void f(int x) {
    y = 2 * x;
}

int main() {
    int y;
    int x = 3;
    f(x);
    printf("%d\n", y);
    return 0;
}
```

Naslednji program pa se bo prevedel, vendar bo morda izhod v nasprotju s pričakovanji:

```
#include <stdio.h>

void f(int x) {
    x = 2 * x;
}

int main() {
    int x = 3;
    f(x);
    printf("%d\n", x);
    return 0;
}
```

Primer vhoda in izhoda

```
3
```

Spremenljivke, ki jih deklariramo v funkciji, to je znotraj telesa funkcije, ali pa v seznamu argumentov, so lokalne na to funkcijo – zunaj nje sploh ne obstajajo. Če želimo, da funkcija popravi neko vrednost, ki jo uporabljamo tudi zunaj funkcije, smo do sedaj lahko to naredili samo tako, da smo spremenljivko naredili globalno – torej dostopno vsem funkcijam (tudi `main` je funkcija). Kaj pa, če želimo neko spremenljivko na tak način deliti samo med dvema funkcijama?

Da odgovorimo na to vprašanje, moramo razumeti, kako se argumenti podajajo v funkcije. Ko neko funkcijo pokličemo, se argumenti, ki jih funkciji podamo, *prekopirajo* v poseben del spomina, ki ji pripada. Ko smo znotraj ene funkcije, ne poznamo imena spremenljivk v drugih funkcijah; prav tako ne vemo, kje so te spremenljivke shranjene. Nič pa nam ne preprečuje, da spreminjamo spomin, ki našemu programu pripada, pa četudi je zunaj funkcije; razen tega, da ne vemo, kateri del spomina je naš, in kateri ni. Lahko si predstavljamo, da smo zabredli v spominsko džunglo, v kateri ne prepoznamo prave poti do spremenljivk, ki jih želimo popraviti. Če pa s seboj prinesemo zemljevid, bomo nenadoma to lahko naredili. Ta zemljevid je kazalec.

Funkcija lahko brez težav sprejme kazalec kot argument. Če to storimo, pravimo, da smo spremenljivki vrednost podali *po referenci* (angl. *pass by reference*), namesto da bi argument podali običajno, čemur pravimo *podajanje po vrednosti* (angl. *pass by value*). Kazalec, ki smo ga podali, se bo še vedno prekopiral v del spomina, ki pripada funkciji; vrednost, na katerega kazalec kaže, pa bo ostala tam, kjer je. Tako lahko skozi kazalec spremenimo vrednosti spremenljivk zunaj funkcije.

Primer

Primera od zgoraj, narejena tako, da delujeta.

```c
#include <stdio.h>

void f(int vrednost, int *kazalec) {
    *kazalec = 2 * vrednost;
}

int main() {
    int x = 3;
    int y;
    f(x, &y);
    printf("%d\n", y);
    return 0;
}
```

Kot vidimo, mora funkcija vedno sprejeti tudi argument, ki ga spremeni.

```c
#include <stdio.h>

void f(int *kazalec) {
    *kazalec = 2 * (*kazalec);
}

int main() {
    int x = 3;
    f(&x);
    printf("%d\n", *x);
    return 0;
}
```