Указатели и динамична памет

Виктор Кетипов
Николай Димитров
Христо Стефанов
elsys.os.2014@gmail.com

[1]Технологическо училище "Електронни системи"
Технически университет, София

26 септември 2014 г.

Съдържание

Въведение

- Компютърна паметта представлява проста последователност от байтове.
- По никакъв начин не е обозначено къде започват едни данни и къде свършват.
- Няма означение и за техния тип, предназначение или име.

Въведение

- Когато една програма на C бъде компилирана, тя се трансформира от код на езика C в машинен код.
- Машинният код представлява поредица от инструкции, които процесорът разбира.
- В този машинен код за обръщение към данни от паметта се използват само и единствено адреси.

Адрес

- Адресът в паметта представлява отместване от началото ѝ.
- Тъй като рядко ни е необходим само един бит от паметта, е въведено по-голямото понятие байт.
- Отместването се измерва в байтове. Тоест адрес 100 означава, че данните се намира на 100 байта след началото на паметта.

Байт

- След като адресите са в байтове, то ние не можем да подадем инструкция към процесора, която да прочете само един бит
- Поради тази причина определението за байт е най-малката адресируема единица, т.е. най-малкото количество данни, с които процесорът може да работи.
- В почти всички случаи един байт е 8 бита, което е избрано, тъй като може да представи един символ.
- За работа с определени битове от даден байт трябва да използваме побитови операции на процесора, които променят данните бит по бит, но резултатът от тях винаги е байт.

Адреси в C

- В езикът C можем да видим адресът на данните, които използваме.
- Това се постига чрез оператора &
- Адресите обикновено се извеждат в шестнайсетична бройна система.

```
1  int a = 42;
2  printf("%p\n", (void *) &a); //0x7fff02bf4bac
```

Тип на указател

- Тъй като указателя е просто отместване, ние трябва да укажем колко байта искаме да прочетем
- Поради това указателите имат тип - int*, char*, long*, double* и други.
- В езикът C има и указател без тип - void*. За да прочетем нещо от указател void*, трябва да го превърнем към указател от някой друг тип.

```
1  int a = 42;
2  void *p = &a;
3  int b = *p; // Грешка
4  int c = *(int *)p; //42
```

Локални променливи

- В езикът C част от паметта се управлява автоматично - заделя се при влизане в дадена функция и се освобождава при излизането от нея.

```
1  void func1() {
2    int a = 5;
3    int b = 6;
4  }
5
6  void func2() {
7    int k;
8    func1();
9  }
10
11 int main() {
12   func2();
13 }
```

Динамична памет

- Възможно е и ние да управляваме ръчно паметта, което се налага в множество случаи в C
- За да използваме някакъв участък от паметта е необходимо да поискаме памет от операционната система.
- Това се налага, тъй като ОС се грижи да разпределя паметта между различните процеси.

malloc

- За да заделим памет се използва системната функция malloc, която приема като аргумент броя байтове, които искаме да заделим
- malloc връща указател към заделената памет
- Ако malloc върне NULL, то това означава грешка. В повечето случаи причината е липсата на памет.

```
1  char *p = malloc(1); //Заделя 1 байт
```

sizeof

- Важно е да се подава правилния брой байтове към malloc
- За целта се използва оператора sizeof
- sizeof връща колко байта са необходими за даден тип

```
1  struct point {
2    double x;
3    double y;
4  }
5
6  //Заделя памет за 1 int
7  int *p1 = malloc(sizeof(int));
8
9   //Заделя памет за 2 double
10 struct point *p2 = malloc(sizeof(struct point));
```

sizeof

- sizeof(char *) НЕ е равно на sizeof(char)
- sizeof(char) връща броя байтове, необходими за запомнянето на един char
- sizeof(char*) връща броя байтове, необходими за запомнянето на един указател
- Всички указатели са една и съща големина, определена от архитектурата на процесора (32 или 64 бита)
- sizeof(char*) = sizeof(int*) = sizeof(double*)

sizeof

- За да запомним един адрес винаги са ни необходими един и същи брой байтове, независимо от типа.
- Типът определя какво седи на този адрес, а не големината на самия адрес.
- Адресът просто показва къде започват данните в паметта и поради това големината му не зависи от това какво се намира на този адрес.

```
1  //Големината в байтове е примерна
2  char a = 'X'; //1 байт
3  char *pa = &a; //4 байта(при 32 битови процесори)
4
5  int b = 42; //4 байта
6  int *pb = &b; //4 байта
```

Масиви

- Заделянето на масив динамично е аналогично с заделянето на една променлива.
- Единствената разлика е, че е необходимо да заделим памет за всеки елемент.

```
1  //Големината в байтове е примерна
2
3  //Масив от 100 елемента, 400 байта
4  int *arr = malloc(100 * sizeof(int));
5
6  //Масив от 100 елемента, 100 байта
7  char *arr = malloc(100 * sizeof(char));
```

Адресна аритметика

- Възможно е да добавяме и изваждаме числа от указател.
- Така можем да достъпваме следващите елементи от масив
- Операторът arr[i] е просто по-удобен запис за *(arr + i);

```
1  //Масив от 100 елемента, 400 байта
2  int *arr = malloc(100 * sizeof(int));
3  arr[0] = 5; //Първият елемент
4  arr[1] = 6; //Вторият елемент
5
6  printf("%d\n", *(arr + 1)); //Извежда 6
7
8  arr++;
9  printf("%d\n", arr[0]); //Извежда 6
10 printf("%d\n", arr[-1]); //Извежда 5
```

Адресна аритметика

- Нека приемем, че имаме масив от int елемента, започващи от адрес 0x100
- Ако приемем, че за един int са необходими 4 байта, то втория елемент ще намира на адрес 0x104
- Нека приемем, че имаме масив от char елемента, започващи от адрес 0x500
- Ако приемем, че за един char е необходим 1 байт, то втория елемент ще намира на адрес 0x501

Адресна аритметика

- Операторите +, -, [] взимат предвид тези особености
- Очевидно не можем да инкрементираме void*, тъй като не знаем големината на елемента, към който сочи.

```
1  int *arr = malloc(100 * sizeof(int));
2  printf("%p\n", (void *) arr); //0x100
3  arr++;
4  printf("%p\n", (void *) arr); //0x104
5
6  char *carr = malloc(100 * sizeof(char));
7  printf("%p\n", (void *) carr); //0x500
8  carr++;
9  printf("%p\n", (void *) carr); //0x501
```

Освобождаване на паметта

- За да укажем на ОС, че вече не ни е необходима даден памет е необходимо да я освободим.
- Това става чрез функцията free.
- Освобождаването на памет трябва да става възможно най-скоро след като вече не ни е необходима, за да могат други програми да я използват.

```
1  int *arr = malloc(100 * sizeof(int));
2  ....
3  ....
4  free(arr);
```