Упражнение 7 - Указатели и динамични масиви

Понятие за указател

Указателят е едно от основните предимства, с които разполагат програмните езици като C и C++. Те позволяват директен достъп до елементи в паметта, както и динамична заделяне на памет. Основната идея при указателите е, че за достъп до данните, записани в паметта са нужни само мястото в паметта и типа на данните, които са записани там.

По своя смисъл указателят предсавлява число. Значението на това число е малко по-различно от до сега разгледаните числови стойности. Това число е 32 битово (съответно за 32 битов компилатор) и има значение на адрес в паметта.

Всяка клетка в паметта има свой собствен (пореден) номер. Този номер се нарича адрес.

За по-доброто разбиране на понятието указател, нека се разгледа следната асоциация: дадено е студентско общежитие. Всяка стая е номерирана с пореден номер. В една стая може да живее само един студент. Управител-домакинът може да разпределя студентите по стаите. Къде в този пример са указателите и какво всичко има общо с програмирането? Студентското общежитие, в случая, е паметта на един компютърна система. Номерацията на стаите са адресите в паметта. Самите стаи са клетките памет. Студентът притежава собствено име (да се приеме, че името на студента е уникално в рамките на общежитието) – това е името на промеливата, а самият човек, може да се приеме тогава, е стойността на самата променливата. Ролята на управител-домакина тук е ролята на операционната система – той настанява студентите по стаи и по зададено име може каже къде се намира даденият студент в общежитието(паметта). Променливата от тип указател е малко по-специална: това е стая, в която няма настанен студент, а има бележка „Студентът се намира в стая Номер...“.

Дефиниране на променлива от тип указател

Прилича по начин на дефиниране на променлива, но се използва знак *след избрания тип. Това означава, че променливата от тип оказател ще „сочи“ към определен тип данни.

```
int* a;
```

Операции с указател

Референциране

Операцията, която има за цел да вземе разположението на данните в паметта се нарича референциране. Операторът е &, който се поставя пред променливата и се чете „адресът на“.

```
int m = 6;
```

хон.ас. инж. Боян Петров, ***b_petrov@tu-sofia.bg***,
София, 2013

```
int* a = &m;
```

Дереференциране

Тази операция има за цел да вземе стойността, която е записана там, където сочи адресът на променлива от тип указетел.

```
int m = 6;
int* a = &m; // a равно на “адресът на m”
(*a)++; //Стойността записана там, където сочи a, се увеличава с 1 => m става равно на 7
```

Преразглеждане на понятието за масив

В досегашните упражнения беше разгледано понятието масив като начин за записване на множество данни от един тип под една обща променлива и достъпът до отделните елемененти се осъществяваше посредством индекс. Това обаче не е пълната идея на масивите.

Името на масива в същност представлява указател към първият елемент на масива. След това е известно, че в паметта се заемат толкова полета, колкото са зададени в []. Ролята на индекса също е малко по-различна. Индексът може да бъде заменен с понятието „отстояние от адресът на първият елемент“.

Нека е разгледан следният пример: Даден е едномерен масив с 10 елемента, както и променлива i-индекс за достъп до промеливите.

```
int array[10],i;
```

Следователно достъпът до елементите ще бъде:

```
array[i]
```

Ако се приеме, че array е указател, то следователно е число, от където следва, че сумата array+i също ще е число, но след като array има валидна адресна стойност, то новополученото число също ще е адрес.

Следователно може да бъде обработена стойността, която се намира в клетка с адрес array+i, а именно:

```
*(array+i)
```

Тъй като най-малката възможна единица данни, която може да бъде записана самостоятелно в единична клетка от паметта е байт, и се знае, че всяка клетка е номерирана, се поставя въпросът „Дали *(array+i) ще даде отстояние на данните на i байта от началната позиция, или ще даде стойността на поле с индекс i от началото на масива? Отговорът на този въпрос се крие в следната програма:

```
#include <stdio.h>

int main()
```

```
{
	int i,array[10] = {1,2,3,4,5,6,7,8,9,10};
	for(i = 0; i<10; i++)
	{
		printf("array[%d] = %d, *(array+%d) = %d\n",i,array[i],i,*(array+i));
	}
	return 0;
}
```

Както се вижда след изпълненението на програмата стойностите на array[i] и *(array+i) имат една и съща стойност за еднаква стойност на i, следователно може да се заключи, че отместването с *(array+i) не е на брой байтове, а на брой полета с големина, толкова байта, колкото е големината на един елемент от този масив, а именно в случая големината на int (в случая 4 байта).

Забележка: Да се прави разлика между: *(array+i) и *array+i. В първия случай се използва стойността, намираща се на i полета от началото на масива, а във втория случай се взима първият елемент на масива и се домавя към стойността му i.

*Ако се тръгне от математическото тъждество A+B = B+A и се вземе в предвид уравнението, което беше експериментално доказано, че array[i] = *(array+i) => array[i] = *(array+i) = *(i+array) = i[array].*

От тук нататък може да се възприема операторът [] като сума между две числа, от които едното има значението на адрес, а другото на число за отстояние.

Динамични масиви

Динамичният масив представлява масив, чиято дължина не е известна преди началото на изпълнение на програмата и може големината да бъде променяна в хода на изпълнение на програмата. Динамичните масиви се използват в много случаи, когато се налага записването на предварително неизвестен брой елементи, както и когато е нужно намаляване на заеманата памет с цел оптимизация на заеманите системни ресурси.

За заделянето/презаделянето/освобождаването на динамични масиви се използват функции от стандартната библиотека stdlib.h

Фукциите имат следния вид:

```
void* malloc(size_t bytes);
void* calloc(size_t elements, size_t bytes_per_element);
void* realloc(void* prev_address, size_t new_size);
void free(void* array);
```

В дефиницията на функциите са използвани следните типове: void* - без типов указател това означава, че няма информация за типа на данните, които са разположени на даден адрес в паметта; size_t – предефиниран тип от unsigned long – неотрицателна стойност за задаване на размер.

malloc()

Функцията malloc() изисква следните параметри:

```
void * malloc(<брой_байтове>);
```

Функцията заделя определения брой байтове и връща указател към адреса на началото на тези заделени данни. „Заделени данни“ означава, че този ресурс памет е специално заделен за използване от програмата и няма да бъде достъпван от външни програми, както и няма промелниви, които да използват данните на заделополе данни. **Malloc НЕ изтрива данните, а само ги заделя**. Програмистът е длъжен да нулира данните предварително за дейности като броене и др.

calloc()

Функцията calloc() изисква следните параметри:

```
void* calloc(<брой_елементи>,<брой_байтове_на_един_елемент>);
```

Фукнцията заделя и нулира заделената памет. Тази функция е предназначена за заделяне на памет за масиви, тъй като приема два параметъра – броят на елементите в масива и големината в байтове на всеки един елемент. Резултатът, който връща е указател към първия елемент на масива.

realloc()

Функцията realloc() изисква следните параметри:

```
void* realloc(<стар_адрес>,<брой_байтове>);
```

Функцията има за цел да промени размера на текущо заделената памет за динамичния масив, като приема два параметъра: старият адрес, където са разположени данните и големината на масива след промяна на размера. Новият размер може да бъде по-голям или по-малък от стария. Realloc копира (ако е нужно) данните от стария масив в новия масив, до там докъдето новата големина. Функцията връща като резултат новото местоположение на данните.

Всички функции по-горе връщат като резултат NULL, ако не успеят да заделят памет с изисквания размер. Важно е да се проверява дали резултатът им е NULL, понеже може да се появи проблем по записването на данни на нерегламентирани места и да се получи конфликт на данните.

free()

Тази функция освобождава заетата от по-горните фукции памет, като приема само един параметър – адресът на началото на динамичния масива. Добър стил на писане е, ако в края на програмата програмистът освободи заделената памет. Добре е да се внимава, кога се освобождава памет, понеже двойно освободената памет може да предизвика конфликти, както и не навременното освобождаване да предизвика недостиг на памет. В други езици от по-висок клас има вградени подсистеми, които се грижат за автоматичното освобождаване на ненужно заета памет. Тъй като програмният един С е предвиден да е език от средно ниво, то тези функции са поверени на програмиста.