СИНТЕЗ И АНАЛИЗ НА АЛГОРИТМИ

Лабораторно упражнение №5

Указатели. Операции с указатели. Структури. Връзка между структури и указатели

1. Указател – променлива, чиято стойност е адрес от паметта на компютъра се нарича указател. Използва се за осигуряване на косвен достъп до данните, съхранявани на тези адреси.

2. Деклариране на указатели.

*тип * име_на_указател;*

Базовият тип на указателя е типа на сочената от указателя променлива.

3. Инициализиране на указатели.

След дефиниране на указателя, той съдържа произволна стойност и може да се използва след като се инициализира, т.е. се зададе "валиден адрес от паметта" или NULL(специален адрес 0, не сочещ нищо).

4. Операции с указатели.

а) извличане на адрес – оператора & връща адреса на променливата стояща зад него;

& име_на_променлива;

б) извличане на стойност – оператора * връща стойността, съхранявана на адреса, сочен от указателя;

** име_на_променлива;*

Двете операции извличане на адрес и извличане на стойност са унарни и имат приоритет и асоциативност от дясно наляво.

Пример 1:

```
#include<iostream>
using namespace std;
int main()
{int q, *pq;
pq = &q; // pq - присвоява адреса на q
cout << "pq=" << pq << endl;
cout << "&q=" << &q << endl;// &q - връща адреса на q

//Стойностите на pq и &q са едни и същи, като на различните компютри ще се
//получат различни резултати.

 q = 10;
cout << "q=" << q << endl;
cout << "*pq=" << *pq << endl;  //*pq връща стойността, съхранявана на адреса,
//сочен от указателя, т.е. 10
*pq = 20;// стойност 20 се присвоява на q посредством указателя, който сочи
//адреса на тази променлива
cout << "q=" << q << endl;
cout << "*pq=" << *pq << endl;
cout << "&pq=" << &pq << endl;//извежда адреса на указателя
return 0; }
```

5. **Аритметични операции**: ++ -- + -

Компилаторът използва базовия тип, за да се определи колко байта, принадлежат на обекта, сочен от указателя.

Пример 2:

```
int q, *pq;
pq = &q; // pq - присвоява адреса на q
cout << "pq=" << pq << endl;
pq++;// адреса ще се увеличи
cout << "pq=" << pq << endl;

float   qf ,*pqf;
pqf = &qf;
cout << "pqf=" << pqf << endl;
pqf++; // адреса ще се увеличи
cout << "pqf=" << pqf << endl;

double   qd ,*pqd;
pqd = &qd;
cout << "pqd=" << pqd << endl;
pqd++; // адреса ще се увеличи
cout << "pqd=" << pqd << endl;
```

Примерен резултат:

```
pq=0x61fe04
pq=0x61fe08
pqf=0x61fe00
pqf=0x61fe04
pqd=0x61fdf8
pqd=0x61fe00
```

Подобни са примерите и с декрементиране.

Операторите за инкрементиране и декрементиране могат да бъдат прилагани и към обекта, сочен от указателя. В подобни случаи се налага използването на скоби от вида:

(*pq)++; // съдържанието на сочената от pq целочислена стойност се увеличава с 1.

Еквивалентни на тази операция са:

(*pq)++; ≡ *pq+=1; ≡ ++ *pq;

Пример 3:

```
int q, *pq;
pq = &q;
*pq = 20;
```

```
(*pq)++;
cout << "q=" << q << endl;
cout << "*pq=" << *pq << endl;
*pq += 1;
cout << "q=" << q << endl;
cout << "*pq=" << *pq << endl;
++*pq;
cout << "q=" << q << endl;
cout << "*pq=" << *pq << endl;
```

Резултатът е:

```
q=21
*pq=21
q=22
*pq=22
q=23
*pq=23
```

6. Връзка между масиви и указатели.

Името на масива изпълнява функция и на указател и неговата стойност е базовият (начален) адрес. Началният (базов) адрес е адресът на първия му елемент (елемент с индекс 0).

```
int a[30], *pa;
```

то pa = a; ≡ pa =&a[0];

също pa = a+1; ≡ pa =&a[1];

Пример 4:

```
int *pa, a[3] = { 10,20,30 };
pa = a;
cout << *pa << "\t" << *(pa + 1) << "\t" << *(pa + 2) << endl;
cout << a[0] << "\t" << a[1] << "\t" << a[2] << endl;
```

Резултатът ще е:

```
10      20      30
10      20      30
```

К*опирайте предпоследния ред и махнете скобите.*

```
cout << *pa << "\t" << *pa + 1 << "\t" << *pa + 2 << endl;
```

Вижте разликите.

```
10      11      12
```

7. Структури – дефиниране и инициализация.

Използването на структури предоставя възможността да се групират данни и да се работи с тях като едно цяло. Всяка *структура* представлява колекция от един или няколко типа променливи. *Всеки елемент* в дадена структура може да бъде от *различен тип* за разлика от елементите на масива, които са от един и същи тип.

Типичен пример за структура са данните за студент: всеки студент се описва с набор от атрибути като име, специалност, адрес, факултетен номер и така нататък. Някои от тези компоненти (използва се и термина членове) също могат да бъдат структури, например адресът, който съдържа от своя страна също няколко компонента: град, улица и номер. Структурите могат да се присвояват една на друга, да се предават на функции и да се връщат от функции.

7.1. Дефиниране на структури.

Ключовата дума struct означава декларация на структура, която представлява списък от декларации, заграден от фигурни скоби. Променливите, именувани в структурата, се наричат *членове* (елементи, компоненти) на структурата. Декларацията struct определя потребителски дефиниран тип. Дясната фигурна скоба означава края на списъка с членове и след нея може да се постави списък от променливи, както при всеки друг тип. Променливата myaddress, масивът addressbook[20] и указателят *ptrstr в следващия пример са дефинирани от тип структура и се заделя памет за тях.

Пример:

```
struct /* Адрес */
{ char town[20]; /* Град */
char street[20]; /* Улица */
int number; /* Номер */
} myaddress, addressbook[20], *ptrstr;
```

Всяка структура може да бъде именувана, но това не е задължително. Името се поставя след думата struct и може да се използва впоследствие като кратък запис за декларациите във фигурните скоби.

Пример:

```
struct address /* Адрес */
{ char town[20]; /* Град */
char street[20]; /* Улица */
int number; /* Номер */
};
```

Декларация на структура, след която няма списък от променливи, не заделя място в паметта (тя описва шаблона или формата на структурата). Ако структурата е именувана, то впоследствие името може да се използва при дефиниране на обекти от тип структура:

```
struct address myaddress, addressbook[20], *ptrstr;
```

7.2. Вложени структури.

Съществува възможност да се влага една дефиниция на структура в друга. Веднъж декларирана дадена структура, тя се превръща в нов тип данни в програмата и може да бъде използвана навсякъде, където може да присъства друг тип данни (като int, char, float и т.н.). В типа структура student по-горе, се съдържа елемент addr от тип struct address.

7.3. Инициализиране на елементите на структура.

Инициализирането на елементите на дадена структура може да се осъществи по **два** начина: могат да се инициализират, когато се **декларира структурата** като след декларацията ѝ се постави **списък от инициализатори**, съответстващи на елементите на структурата, и всеки инициализатор представлява постоянен израз или по-късно в

програмата посредством присвояване или при извикване на функция, която връща структура от даден тип:

Пример:

```
/* Деклариране и инициализиране на структура едновременно */
struct address /* Адрес */
{ char town[20]; /* Град */
char street[20]; /* Улица */
int number; /* Номер */
}myaddress = {"Варна","Свобода", 9};
.....
/* Деклариране и инициализиране на променлива от тип структура student */
student stud={0};/* Ако в списъка има по-малко инициализатори, отколкото е броят на членовете в структурата, последните членове се инициализират с 0. */
```

7.4. Достъп до елементите на структура.

Използването на *оператора точка* . е един от начините за третиране на всеки елемент на структурата. Пред оператора точка винаги трябва да има име на променлива от тип структурата, а след него винаги трябва да се поставя име на елемент на структурата.

Например:
myaddress.town Достъп до елемент **town** на променливата **myaddress** от тип структура **address.**
stud.name Достъп до елемент **name** на променливата **stud** от тип структура **student**.
stud.addr.town Достъп до елемент **addr.town** на вложената структура **address** на променливата **stud** от тип структура **student**.

8. Връзка между структури и указатели.

Достъпът до елементите на структура, сочена от указател е възможен чрез използване на оператор стрелка -> във вида:

указател-> име_на_елемент_на_структура;

Операторът стрелка се формира от знак минус, следван от знак за по-голям.

9. Оператор new.

За да се заяви динамична памет, съществува оператора new. Общата форма на new е:

указател = new тип ; // Използва се за заделянето на памет, съдържаща един елемент от съответния тип.

Или

указател = new тип [елементи] ; // Използва се за заделянето на блок (масив) от елементи от съответния тип

Пример:

```
int *p;
p = new int[5];
```

В този случай операционната система заделя място за 5 елемента от тип int в heap (динамичната памет) и връща указател към нейното начало, който се присвоява на **p**, т.е. **p** сочи към валиден блок от памет с размер от 5 елемента. Обикновено динамичната памет се управлява от операционната система, като при многозадачните системи, тя може да се споделя от няколко приложения, при което е възможно в даден момент тя да се изчерпа. При това положение операционната система не може да задели паметта, която сме поискали с оператора new и връща NULL указател.

10. Оператор delete

Динамичната памет обикновено е необходима само за дадени моменти от програмата. След като вече не е нужда, тя трябва да бъде освободена, за да може да бъде използвана от други заявки. За тази цел съществува оператора **delete**, чиято форма е:

delete указател; // Използва се за освобождаването на памет, която е била заделена за 1 елемент

Или

delete [ ] указател; // Използва се за освобождаването на памет, която е била заделена за множество елементи (масиви).

Задача: Сито на Ератостен – намиране на прости числа

Пример 5. Примерна реализация на сито на Ератостен с масив

```
#include <iostream>
using namespace std;
#define n 100
int main()
{
     int i, j, a[n];
     for (i=2; i<n; i++) a[i]=1;
     for (i=2; i<n; i++)
        if (a[i])
          for (j=i; i*j<n; j++)  a[i*j]=0;
     for (i=2; i<n; i++)
          if (a[i]) cout<<i<<' ';
     return 0;
}
```

Пример 6. Примерна реализация на сито на Ератостен с динамично заделяне на памет

```
#include <iostream>
using namespace std;
int main()
{
     int n, i, j;
     char c;
     do
```

```
    {
         cout<<"Input n:";
         cin>>n;
         int *a=new int [n];
         if (a==NULL)
              cout<<"Insufficient memory!";
         else
             {
              for (i=2; i<n; i++) *(a+i)=1;
               for (i=2; i<n; i++)
                   if (*(a+i))
                     for (j=i; i*j<n; j++)
                        *(a+i*j)=0;
              for (i=2; i<n; i++)
              if (*(a+i))
                cout<<i<<' ';
             }
         delete []a;
         cout<<"\nAnother n? (Y/N) "; cin>>c;
    }
    while (c=='Y'||c=='y');
return 0;
}
```

Допълнителни задачи с динамично заделяне на памет:

Задача 1. Да се напише функция ***shift_l(arr, n, k)***, която измества циклично масив с дължина ***n*** наляво с ***k*** позиции.

Задача 2. Да се напише функция ***shift_r(arr, n, k)***, която измества циклично масив с дължина ***n*** надясно с ***k*** позиции. Елементите, излизащи извън размера на масива, се премахват. Липсващите елементи вляво се заместват с нули.

Задача 3. Да се напише функция, която формира матрица ***X(n,m)*** с различни по стойност цели числа. Да се намерят най-малкия елемент за всеки ред и най-големия от тях.

Задача 4. Да се напише програма, която формира масив ***C*** и извежда тези негови елементи с индексите им, сумата от цифрите на които е равна на ***M***.

Задача 5. Да се напише функция, която в множеството точки с координати x[i], y[i], разположени в една плоскост, да се намира двете най-близки.

Задача 6. Да се състави функция, която за дадено естествено число $n>13$ намира всички двойки прости числа по-малки от n, разликата между които е равна на 4.

Задача 7. Да се напише програма, определяща колко пъти през XXI век 1 януари се пада в понеделник.

Задача 8. Да се състави програма, която сформира два масива - ***x[n]*** и ***y[m]*** и намира ***MIN*** елемент от всички в ***x***, които не се съдържат в ***y***.