15. Процедурно програмиране - указатели, масиви и рекурсия.

1. **Указатели**

Оператор за реферирание (достъпване на адрес на променлива) (&)

Операторът & е унарен, връща адресът на операнда си. Пример:

$int\ x\{5\};$
$std::cout \ll \&x \ll$ n;

Вторият ред ще принтира адресът в паметта на променливата x (например 0023FEA0).

Оператор за дереференция (*)

Операторът * е унарен, приема адрес и връща стойността, съхранявана в подадения адрес в паметта.
Пример:

$int\ x\{5\};$
$std::cout \ll *(\&x) \ll$ n;

Вторият ред ще изведе 5.

Указател

Указателят е обект, който съдържа като своя стойност адрес в паметта. Подобно на референкциите, които се дефинират с &, указателите се дефинират с *:

⟨тип⟩* ⟨име⟩ [= ⟨израз⟩];

Съхраняваният адрес представлява адресът на началото на стойността, а типът ни дава информация за това какъв размер има стойността, съхранявана на този адрес.

Стойностите от тип указател са с размер на машинна дума

- 32 бита (4 байта) за 32-битови архитектури
- 64 бита (8 байта) за 64-битови архитектури

Физическото представяне на указател е цяло число, указващо адреса на lvalue в паметта.

Пример:

$int\ x\{5\};$
$int*\ ptr\ \{\&x\};$

Ако не инициализираме променлива с тип указател, то тя ще съдържа случайна стойност - подобно на останалите променливи.

nullptr е специална стойност, която указва, че показателят не сочи към никакъв адрес.

Операции с указатели

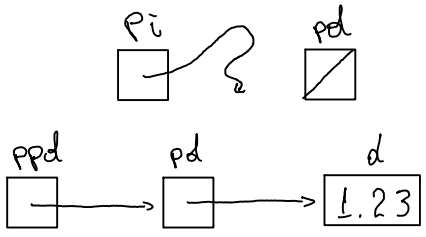
Операторът за реферирание връща като резултат указател.
Операторът за дереферирание приема като аргумент указател

- Сравнение (==, !=, <, >, <=, >=)
- Извеждане (<<)
- Не можем да въведем указател с оператора за въвеждане (>>)

Пример

$int\ *pi;$
$double\ *pd = nullptr;$

$double\ d = 1.23;$
$double\ *pd = \&d;$
$double\ **ppd = \&pd;$



2. Масиви

Масивът е съставен тип данни, представляващ крайна редица от елементи с един и същи тип, с фиксирана дължина. Позволява произволен достъп до всеки негов елемент по номер (индекс). Индексите са последователни цели числа, като първият елемент е с индекс 0.

Синтаксис:

⟨тип⟩ ⟨идентификатор⟩ [[⟨константа⟩]] $\left[= \left\{ \langle \text{израз} \rangle \{, \langle \text{израз} \rangle \} \right\} \right]$;

Пример:

$bool\ b[10]$; - масив с 10 елемента от тип bool. Тъй като не е инициализиран, ще съдържа произволни стойности

$double\ x[3] = \{ 0.5, 1.5, 2.5 \}$;

$int\ a[\,] = \{3 + 2, 2 * 4\}$; $\Leftrightarrow$ $int\ a[2] = \{5, 8\}$; - ако пропуснем дължината и инициализираме, компилаторът ще я подразбере от дължината на инициализиращия списък

$float\ f[4]\ \{ 2.3, 4.5 \}$; $\Leftrightarrow$ $float\ f[4] = \{ 2.3, 4.5, 0, 0 \}$; - директно инициализиране (с = е сору инициализиране), когато сме подали по-малко елементи, останалите клетки се попълват със стойността по подразбиране за типа

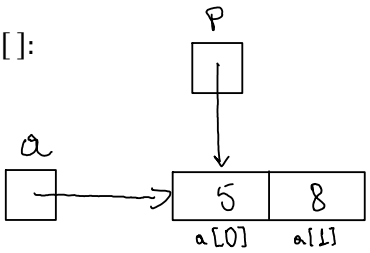
Физическо представяне:

Елементите са разположени последователно един след друг в паметта (съседни елементи имат съседни адреси). Името на масив е **константен указател** към първия му елемент.

Операции за работа с масиви:

- Достъп до елемент по индекс, оператор [ ]:
  Синтаксис:
  ⟨масив⟩ [⟨цяло_число⟩]

  Пример: $int\ y = a[1]$; (y=8)
  $x[2] = 0$; *($x = \{0.5, 1.5, 0\}$)*
  $int\ *p = a$;



  Не се извършва проверка за валидността на индекса (дали не е твърде голям или отрицателен)
- Няма присвояване или поелементно сравнение

Многомерни масиви

Масивите, разгледани по-горе са едномерни. Масив, чиито елементи са едномерни масиви е двумерен масив. Тримерен масив е масив, чиито елементи са двумерни масиви е тримерен и т.н. до N-мерен масив. В реалния свят обикновено се ползват до 3-мерни масиви.

Синтаксис за дефиниране на N-мерен масив:

⟨тип⟩ ⟨идентификатор⟩[⟨размер_1⟩][⟨размер_2⟩] ... [⟨размер_N⟩];

Първата размерност може да бъде изпусната, ако е даден инициализиращ списък

Пример:

$int\ a[2][3] = \{\{1,2,3\},\{4,5,6\}\}$;

$int\ num[3][4] = \{$
$\{1, 2, 3, 4\},$
$\{5, 6, 7, 8\},$
$\{9, 10, 11, 12\}\}$;

	<— row 0 —>				<— row 1 —>				<— row 2 —>			
value	1	2	3	4	5	6	7	8	9	10	11	12
address	1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1022

3. Указателна аритметика

Позволява по дадена отправна точка в паметта (указател) да реферираме съседни на нея клетки. За целта трябва да укажем колко клетки напред или назад в паметта искаме да прескочим
Размерите на клетките зависят от големината на типа, който указателят реферира. Ако имаме $T\ *p$:

- $p + i$ прескача $i * sizeof(T)$ байта напред
- $p - i$ прескача $i + sizeof(T)$ байта назад

Пример: $int\ *p$, обикновено int е 4 байта, следователно $p + 2$ ще значи "прескочи 8 байта напред".с

Указателна аритметика за масиви

Нека a е масив, тогава:

$a[i] \Leftrightarrow *(a+i) \Leftrightarrow *(i+a) \Leftrightarrow i[a]$

Примери:

```
int a[5] = {1, 2, 3, 4, 5};
cout << *a;  // Принтира 1
*(a + 1) = 7; // a = {1, 7, 3, 4, 5}
--*(a + 4); // a = {1, 7, 3, 4, 4}
```

Указатели и константи

- Константен указател - константа е (не се променя накъде сочи).

  ⟨тип⟩ $*const$

  Пример:

```
int x, *p = &x;
int *const q = p;
*q = 5; // x=5
q = p + 2; // грешка, не може да променяме константа
```

- Указател към константа - сочи към константа

  $const$ ⟨тип⟩ $* \Leftrightarrow$ ⟨тип⟩ $const *$

  Пример:

```
int x,  *p = &x;
const int *q = &x;
q++; // сочи клетката след тази на x
p = q;  // не може на указател да бъде присвоена стойността на указател към константа (няма го обещанието, че няма да се променят елементите)
*q = 5; // грешка, не може да променяме елементите, когато указателят е към константа.
```

Ако x е константа, то $\&x$ е указател към константа

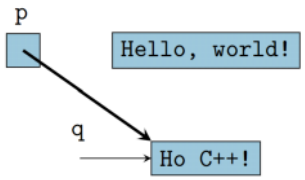
Указатели и низови константи

Името на низ е константен указател към първия му символ ($char *const$)

Низовите константи са указатели към константен символ ($char\ const *$)

Пример:

```
char const *p = "Hello world!";
char *q = "Hi"; // Не може да променяме константен символ
char q[] = "Hi C++!";
p = q;
q[1] = 'o';
```



4. Сортиране и търсене в едномерен масив - основни алгоритми

1. Selection sort - Разделя входния масив на две части - сортиран подсписък с елементи, който се изгражда от ляво надясно и останалите елементи. Намира най-малкия елемент от несортираната част и го размеря с най-левия елемент на несортираната част, така сортираната част се увеличава с един елемент.

Времевата сложност на алгоритъма е $O(n^2)$

```
void selectionSort(int *a, int n) {
    for (int i = 0, i < n - 1; ++i) {
        int jmin = i;
        for (int j = i + 1; j < n; ++j) {
            if (a[j] < a[jmin]) {
                jmin = j;
            }
        }

        if (jmin != i) {
            swap(&a[i], &a[jmin]);
        }
    }
}
```

2. bubbleSort - многократно преминава през входния списък елемент по елемент, сравнява текущия елемент с този след него и разменя стойностите им, ако е необходимо. Повтаря, докато не се извършат никакви размери по време на преминаване, което значи, че списъкът е сортиран.

Времевата сложност на алгоритъма е $O(n^2)$

```
void bubbleSort(int * a, int len) {
    bool swapped;
    for (int i = 0; i < n - 1; ++i) {
        swapped = false;
        for (int j = 0; j < n - 1 - i; ++j) {
            if (a[j] > a[j + 1]) {
                swap (&a[j], &a[j + 1])
                swapped = true;
            }
        }
        if (! swapped) break;
    }
}
```

3. insertionSort - на всяка итерация намира локацията на един елемент и го слага там. Повтаря докато няма повече елементи

Времевата сложност на алгоритъма е $O(n^2)$

```
void insertionSort(int * a, int len) {
    int key;
    for (int i = 1; i < n; ++i) {
        key = a[i];
        int j = i - 1;
        while (j ≥ 0 & key < a[j]) {
            a[j + 1] = a[j];
            --j;
        }
        a[j + 1] = key;
    }
}
```

4. quickSort - избира разделящ елемент и разделя другите елементи от масива на два подмасива, в зависимост дали са по-големи или по-малки от разделящия. Подмасивите рекурсивно се сортират. Има различни стратегии за избиране на разделящия елемент

Времевата сложност на алгоритъма е $O(n^2)$

```
void parititon(int * a, int l, int r) {
    int pivot = a[r];

    int i = l - 1;
    for (int j = l; j < r; ++j) {
        if(a[j] <= pivot) {
            ++i;
            swap(&a[i], &a[j]);
        }
    }

    swap(&a[i + 1], &a[r]);
    return i + 1;
}

void quickSort(int * a, int l, int r) {
    if (low > high) return;

    int pivot = partition(a, l, r);
    quicksort(a, l, pi - 1);
    quicksort(a, pi + 1, r);
}
```

5. Merge sort - разделя несортирания списък на n подсписъка, всеки съдържа по 1 елемент, обединява подсписъци като запазва наредбата, докато не остане само един списък, той е сортиран.

Времевата сложност на алгоритъма е $O(n \cdot \log n)$

```
void merge(int * a, int l, int m, int r, int * tmp) {
    int i = l, j = m + 1, k = 0;
    while (i <= m && j <= r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            tmp[k++] = a[j++];
        }
    }
    while (i <= m) tmp[k++] = a[i++];
    while(j <= m) tmp[k++] = a[j++];
    memcpy(a + l, tmp, k * sizeof(int));
}

void mergeSort(int * a, int l, int r, int * tmp) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(a, l, m);
        mergeSort(a, m + 1, r);
        merge(a, l, m, r, tmp);
    }
}
```

6. Двоично търсене - търсене на елемент в сортиран масив.

```
int binarySearch(int * a, int n, int x) {
    int l = 0, r = n - 1, m = l + (r - l)/2;
    while (l <= r) {
        m = l + (r - l)/2;

        if (a[m] == x)
            return m;

        if (a[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }

    return -1;
}
```

5. **Рекурсия - пряка и косвена рекурсия, линейна и разклонена рекурсия**

Рекурсивна функция наричаме функция, която извиква себе си пряко или косвено.
Функция е пряко рекурсивна, ако в тялото си се извиква. Функция е косвено рекурсивна, ако извиква друга функция и в изпълнението се извиква обратно първата ($F_1 \to F_2 \to \cdots \to F_n \to F_1$)

Линейна рекурсия е такава, при която функция прави само едно рекурсивно извикване на себе си в кода. Пример:

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Разклонената рекурсия е такава, при която функцията извиква себе си два или повече пъти, за да реши един проблем. Пример:

```
int fib(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return fib(n − 2) + fib(n − 1);
}
```

Рекурсията е удобна за работа с рекурсивни структури, обхождане на графи, търсене с връщане назад, алгоритми от тип "разделяй и владей" и т.н.

Недостатък е скритото използване на памет за стекови рамки.