## 11. Процедурно програмиране – основни информационни и алгоритмични структури на базата на C++)

Анотация: Изложението по въпроса трябва да включва следните по-съществени елементи:
1. Скаларни типове от данни. Логически тип. Числени типове цял и реален.
2. Съставни типове от данни. Структура от данни масив. Тип масив.
3. Тип указател – дефиниране, основни операции. Указателна аритметика.
4. Указатели и едномерни масиви. Указатели и двумерни масиви. Указатели и низове.
5. Функции. Дефиниране на функция. Обръщение към функция.
6. Предаване на параметрите по стойност, чрез указател и чрез псевдоним.
7. Функции. Масиви като формални параметри.

Типична задача. Да се състави функция, която въз основа на зададени като параметри масиви и/или матрици, чрез съответен анализ формира други такива.

**Процедурно програмиране от птичи поглед**
Процедурното програмиране е една от първите парадигми в програмирането, с която се сблъсква всеки начинаещ програмист. Тази парадигма използва линеен подход отгоре надолу и третира данните и процедурите като две различни единици. Въз основа на концепцията за извикване на процедура, процедурното програмиране разбива програмата на процедури, които също са известни като подпрограми или функции, просто съдържащи поредица от команди, които трябва да бъдат извършени.

По-просто казано, процедурното програмиране включва записване на списък с инструкции, за да се каже на компютъра стъпка по стъпка какво трябва да направи, за да завърши задачата.

**Основни характеристики на процедурното програмиране**
1. **Предварително определени функции** – инструкции, които се характеризират със сигнатура и тяло и сме написали сами или взимаме наготово от някаква библиотека;
2. **Локални променливи** – променливи, които са декларирани в основната структура на метода и са ограничени до локалния обхват. Може да се използват само в метода, в който са дефинирани;
3. **Глобални променливи** – променливи, които се декларират извън всяка друга функция, определена в кода и могат да се използват във всички функции, за разлика от локалните променливи;
4. **Модулност** – когато две различни системи имат две различни задачи под ръка, но са групирани заедно, за да приключат първо по-приоритетната задача. След това всяка група системи ще има свои собствени задачи, завършени една след друга, докато всички задачи не бъдат изпълнени.
5. **Предаване на параметри** – механизъм, използван за предаване на параметри към функции, подпрограми или процедури. Предаването на параметър може да се извърши чрез „предаване по стойност“, „предаване чрез указател“ и „предаване чрез псевдоним“.

**Предимства**
1. Потокът на програмата може лесно да се проследи;
2. Редуциране на използваната памет;
3. Кодът може да се използва повторно в различни части на програмата, без да е необходимо да се копира;
4. Изходният код е преносим, следователно може да се използва и за насочване към различни процесори.

**Недостатъци**
1. По-труднен за писане код, особено ако обема нараства, тъй като липсва йерархия;
2. Трудно се свързва с обекти от реалния свят;
3. Данните са изложени на цялата програма, което я прави не толкова защитена;

**Скаларни типове от данни**
Булеви, цели числа, реални числа, изброен тип (enum), указател, псевдоним. Всеки един от тези типове, се състои от една стойност, която наричаме скалар.

**Логически тип**
Тип, който представлява отговор на логически въпрос с два възможни отговора – истина или лъжа. Нарича се още булев тип и в C++ се представя с 1 байт – числото 0 за лъжа (false) и числото 1 за истина (true).

Примери: bool flag = true; bool is_set;

bool е част от декларацията на променливата, която задава нейния тип, true е стойността с която е инициализирана, а flag е името ѝ. Във втория пример, булевата променлива is_set не е с дефинирана стойност.

**Логически оператори**

1. Логическо „И“ ($x$ && $y$) – $x$ „И“ $y$, където $x$ и $y$ са логически изрази. Резултатът ще бъде истина, ако и x и y са истина едновременно и лъжа във всички останали случаи;
2. Логическо „ИЛИ“ ($x \parallel y$) – аналогично на логическото „И“, но тук резултатът ще бъде истина когато поне един от логическите изразите $x$ или $y$ е истина;
3. Логическо отрицание ($!x$) – за разлика от логическото „И“ и „ИЛИ“, които са бинарни оператори, логическото отрицание е унарен и резултатът от него ще се оцени на истина когато се приложи върху логически израз със стойност лъжа и дуално ще се оцени на лъжа, ако се приложи върху логически израз със стойност истина.

От теоремата на Бул знаем, че тези логически оператори са достатъчни, за да се изразят всички останали съществуващи логически оператори.

Когато имаме „И“ лист ($x_1$ && $x_2$ && $\cdots$ && $x_n$ ) стойността на резултата се оценява на лъжа веднага след като се срещне първия израз $x_i$, $i = \overline{1, n}$ със стойност лъжа.
Когато имаме „ИЛИ“ лист ($x_1 \parallel x_2 \parallel \cdots \parallel x_n$) стойността на резултата се оценява на истина веднага след като се срещне първия израз $x_i$, $i = \overline{1, n}$ със стойност истина.

**Оператори за сравнение**
Изразите, които са сравними може да се оценят с истина или лъжа и по този начин може да създаваме предикати, които отново може да разглеждаме като булеви изрази. Операторите за сравнение са ==, !=, >, >=, <, <=.

Логически (булев) израз е всеки израз, който може да се оцени до истина или лъжа. Може да го дефинираме рекурсивно по следния начин:
- Булевите константи 1 и 0 (истина и лъжа) са булеви изрази;
- Булевите променливи са булеви изрази;
- Прилагането на логическите оператори или операторите за сравнение върху булеви изрази е булев израз.

Логическите оператори, които разгледахме по-горе могат да се прилагат и върху данни от целочислен тип. Както споменахме истина (true) и лъжа (false) в C++ се интерпретират като 1 и 0 и това води до тази свобода. Всяко число от цело§ислен тип, което има в двоичния си запис поне една единица се интерпретира като истина. В този смисъл, само числото 0 е лъжа.

**Числени типове цял и реален**

| Целочислен тип | домейн | памет в байтове |
|---|---|---|
| **short** или **short int** или **signed short** | $[-2^{15}, 2^{15}-1]$ | 2 |
| **int** или **signed** или **signed int** или **long** или **long int** или **signed long** | $[-2^{31}, 2^{31}-1]$ | 4 |
| **long long** или **long long int** или **signed long long** | $[-2^{63}, 2^{63}-1]$ | 8 |

Към всеки от типовете по-горе може да добавим първа ключова думиа unsigned и да получим неотрицателен целочислен тип като домейна му ще е от 0 до $2^{n+1}$, където $n$ е степенния показател на двойката от горната граница на същия тип без unsigned пред декларацията си.
Поредицата от битове за реалния тип число се разбива на: бит за знак, битове за експонента и битове за цяла част.

| Реален тип | памет в байтове | битове за цяла част | битове за експонента |
|---|---|---|---|
| **float** | 4 | 52 | 11 |
| **double** | 8 | 23 | 8 |

Т.е. double поддържа реални числа с до 15 цифри преди десетичната запетая, докато float поддържа реални числа само с до 7 цифри преди десетичната запетая.

Примери: double e = 2.71828; float pi = 3.1415f.

Когато сравняваме две числа от реален тип е необходимо да го правим с някаква точност $\varepsilon$. Например числата с плаваща точка $x$ и $y$ са равни, ако разликата им по модул е по-малка от $\varepsilon$. Това е така, тъй като реалните числа може да са с безкрайно много цифри след десетичната запетая, но паметта в която се съхраняват винаги е крайна и за това тези числа се апроксимират в паметта.

**Унарни оператори**

- $+x$ запазва знака на числовия аргумент $x$;
- $-x$ „обръща“ знака на числовия аргумент $x$. Ако е положително става отрицателно и обратно;
- ++ инкрементира числовия аргумент с единица. $++x$ първо инкрементира стойността на $x$ след което я оценява, а $x++$ първо оценява стойността на $x$, след което я инкрементира.
- – – дуално на ++, но вместо да инкрементира – декрементира с единица.
- $x-=n$ декрементира числовия аргумент с $n$
- +=дуално на –=, но инкрементира вместо да декрементира

**Бинарни оператори**

- + математическо събиране. $x + y$ събира числата $x$ и $y$;
- – математиеско изваждане. $x - y$ от числото $x$ изважда числото $y$;
- * математическо уножение. $x * y$ умножава числата $x$ и $y$;
- / целочислено деление. $x / y$ разделя числото $x$ на числото $y$ без остатък;
- % модулно деление. $x \% y$ разделя числото $x$ на числото $y$ и връща остатъка от делението.

**Побитови оператори**
Побитовите оператори се прилагат върху числа в двоичен вид. Когато операторът е бинарен и дължините в двоичен вид на двете числа върху които се прилага са различни, тогава по-късото се запълва с водещи нули за да се изравнят дължините.

- & (И) – бинарна побитова операция AND. Връща 1 на дадена позиция, само когато и двата бита са 1;
- | (ИЛИ) – бинарна побитова операция OR. Връща 1 на дадена позиция, когато поне на един от сравняваните два бита е 1;
- ^ (XOR) – бинарна. Връща 1 на дадена позиция само когато двата сравнявани бита са различни;
- ~ (NOT) – побитово отрицание. Унарен оператор, който „обръща“ битовете. Ако битът е 0 ще стане 1 и обратно.

**Съставни типове от данни**
Тип данни, който се състои от множество (колекция) от скаларни типове или други съставни типове. Това са например колекции от реални числа, колекции от данни от логичекси тип, обекти (инстанции на дадена структура) и други. Тоест, съставните типове са редици от компоненти/скалари.

**Структура от данни масив**
Това е колекция от скаларни данни, които се съхраняват в непрекъснат сегмент от паметта. Това прави достъпването до елементите лесно, тъй като е необходимо да се знае само позицията на първия елемент в паметта и позицията на елемента, който търсим спрямо първоначалния (отместването). Елементите на масива са от един тип и това отместване винаги ще е с константна стъпка, съответваща на размера на скаларния тип на съставящите го данни умножен по отместването. Например, ако масивът съдържа цели числа от тип int, то тогава всеки скалар ще заема 4 байта и всяко отместване от даден скалар към съседен скалар ще се мени с 4 байта.

Масивите имат фиксирана дължин (размер), тъй като при създаването им е необходимо да знаем каква част от паметта ще заделим. Този размер може да се промени към по-голям, като се задели нова памет с по-голямия размер и се копират всички елементи на наличния масив. Но това е операция с линейна сложност, която може да е много скъпа при множество прилагания.

Един масив се дефинира като се посочи типът на данните, които той ще съхранява, размерът и името му. Ако типът на данните е T, константата SIZE е желаният размер, а arr е името му, то тогава дефиницията на масива ще е следната: T arr[SIZE]. Масив може да се инициализира и с предварително зададени стойности, като например:
T arr = {T1, T1, ..., Tn}, като в този случай размера му ще се зададе имплицитно по време на компилация.

Достъпът до елементите в масив става чрез индексиране. Индексирането в C++ започва от 0 (0-базирано) и всеки следващ елемент е на позиция с единица по-голяма от предходната. Индексирането се осъществява чрез предефинирания оператор [ ].

**Тип масив**
Масивът от скалари, който разгледахме по-горе е едномерен масив. Масив, чиито елементи са други масиви е двумерен масив. Масив, чиито елементи са масиви от масиви е тримерен масив и т.н. до $N$-мерен масив. Тъй като реалният свят е триизмерен, то много рядко може да ни се наложи да използваме масив от по-голяма размерност от 3.

$N$-мерен масив се дефинира, като се посочат размерите на отделните размерности. Ако те са например SIZE1, SIZE2, ..., SIZEn, то тогава T arr[SIZE1][SIZE2]...[SIZEn] е $N$-мерен масив.

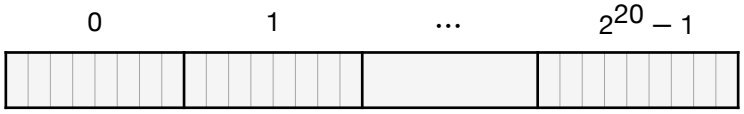
Пример: int m[2][3]; дефинира двумерен масив или матрица от два реда и три колони. Подобно на едномерния масив, може да използваме инициализиращи блокове по следния начин: int m[2][3] ={ {2,3,4}, {5,6,7} }. Достъпът до елементи на многомерния масив отново става с индексиране, като в този случай индексирането е по всички размерности. Например при двумерен масив, за да вземем елемент от $i$-тия ред и $j$-тата колона, трябва да го селектираме чрез arr[ $i$ ][ $j$ ]. Достъпваме елемента със стойност 3 от масива m чрез m[0][1].

В паметта, многомерните масиви имат същото представяне като едномерните – линеен блок от клетки, в който се запаметяват скаларите. Чрез аритметика използваща отместването по всяка от димeнсиите се изчислява позицията на селектирания елемент. Масивът m[2][3] ={ {2,3,4}, {5,6,7} } се представя в чрез следния линеен блок:

| 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|

По този начин достъпът до втория елемент на втория ред се изчислява чрез формулата $2 \times i + j$, където $i = 0$, а $j = 1$, тъй като е 0-базирано индексиране.

**Тип указател – дефиниране, основни операции**
Паметта на един компютър се представя като поредица от байтове. Тези байтове може да ги индексираме с числата от 0 до последното число, необходимо за цялостното индексиране. Този индекс наричаме адрес. Един мегабайт от паметта графично може да се представи със следната „лента“ от байтове:




Скалар, който съхранява адрес се нарича указател. Указателят е с фиксиран размер според машината. Ако машината е 32-битова, указателя заема 4 байта, а ако машината е 64 битова – указателя заема 8 байта от паметта.

Указател се дефинира с тип на данната, която ще реферира, звездичка, за да укажем, че това е референтен тип и името на указателя. Например T* ptr е указател от тип T, който е именуван с ptr.

Извличането на адреса на дадена променлива става чрез унарния оператор &, който се прилага към нея. Например, ако променливата int num = 489 се намира на адрес $2^{12}$, то int**&** num = 4096. Унарният оператор & може да се прилага на lvalue данни (такива, които имат адрес) и не може да се прилага към rvalue данни (такива, които нямат адрес). Сега вече може да дефинираме как се инициализира указател: int***** ptr = **&**num.

Типът на указателя съответства на типа на данната, чийто адрес съхранява самият указател. Съществува тип указател, който няма съответстващ тип на данна. Това е nullptr, с който указваме, че даден адрес не сочи все още никъде. Ползваме го за стойност по подразбиране на неинициализиран указател. Пример: int***** ptr = nullptr;

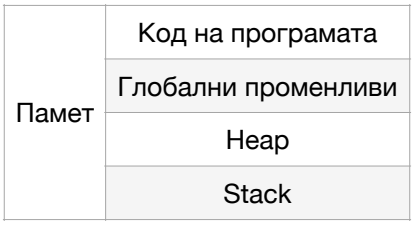
Ако искаме да вземем стойността към която сочи даден указател е необходимо да диференцираме като приложим унарния операто * към указателя. Например:
double pi = 3.14; double***** pi_ptr = **&**pi  е указател към pi, а *pi_ptr отново е стойността 3.14.

**Указателна аритметика**
Указателите могат да участват като операнди в аритметични и логически операции като например +, –, ++, – –, ==, !=, >, >=, <, <=. Изпълнението на аритметични операции върху указатели е свързано с понятието мащабиране. Да разгледаме пример. Нека разполагаме с два указателя int* p и double* q. Ако приложим операцията събиране или инкрементиране с 1 към двата указателя ще наблюдаваме две различни поведения. Стойността на p се увеличава с 4, а тази на q а 8, тъй като данните към които сочат тези указатели са съответно от 4 и 8 байта и за да траверсира указателя към следващата данна е необходимо да прескочи тези байтове. Формално, ако p е указател от тип T, то инкрементирането на T* с $i$ е еквивалентно на p + $i$ * sizeof T, където sizeof T връща броя байтове, необходими за съхраняване на данна от тип T.

При прилагането на оператори за сравнение към указатели се сравняват стойностите на адресите. По този начин два указателя ще са равни т.с.т.к. сочат към един и същ адрес (т.е. към една и съща променлива).

**Указатели и едномерни масиви**
В C++ името на масива е указател към първия елемент от масива. Например, ако имаме масив arr, то arr[0] и *arr имат една и съща стойност равна на първия елемент от масива. Аналогично може да достъпим и $n$-тия елемент от масива: p[n-1] или *(p+n-1).

Освен масивите, които разгледахме до сега, съществуват и динамично заделени масиви, които се намират на различно място в паметта. Когато една C++ програма се стартира, компилаторът автоматично заделя памет за кода на програмата, друга памет за глобални променливи и стекова памет, в която се съхранява реда на извикване на функции и съответнитните им аргументи и локални променливи. Останалата част от паметта е heap-а и е достъпна за всички потребители на системата.




В heap-а може да заделяме памет чрез оператора new, който след заделяне на памет връща указател, сочещ към първия елемент, за който е заделил памет. Пример: int *arr = new int[10]. Тази команда заделя 10*4 байта от heap паметта (достатъчна за 10 числа от тип int) и връща указател, който сочи към първото число от масива. За масивите заделени в heap-а и изобщо за всички обекти създадени с оператора new, трябва да се погрижим, като освободим паметта, когато решим, че вече няма да ни е обходима и е удачен момент да заделим процесорно време за тази операция по освобождаване на паметта. Това става с оператора delete за една данна или delete[ ] за масив от данни. Например delete[ ] arr; След освобождаването на паметта, указателят arr ще сочи към nullptr.

**Указатели и двумерни масиви**
Променливата от тип двумерен масив е указател към първия ред на матрицата. Заделянето на двумерен масив в heap паметта става по следния начин: T** matrix = new T[10][10]; Но за да освободим тази памет е необходимо да траверсираме по всички редове на матрицата и да изтрием масивите, които ги репрезентират, след което да изтрием масива, който съхранява указателите към тези редове. Т.е.

```
for(int i = 0; i < 10; ++i) {
        delete[ ] matrix[i];
}
delete[ ] matrix;
```

Достъпването на елементи от $n$-мерен масив не прави разлика в зависимост от паметта, в която е заделен.

**Указатели и низове**
В C++ съществуват данни от тип char (символ). Те репрезентират една буква, закодирана с помощта на ASCII стойност с цяло число от интрвала [0, 255]. Тоест, броят на символите в C++, поддържани от типа char са само 255 и затова този тип данни заемат 1 байт от паметта. Един низ представлява масив от символи от тип char и се декларира чрез специалната дума string. Например string h = "hello" се репрезентира като едномерен масив от ASCII символи като char ch[ ] = {'h', 'e', 'l', 'l', 'o'} и за него е в сила всичко което споменахме за указатели и едномерни масиви. Масивите от ASCII символи могат да бъдат заделяни и в heap паметта като например char* arr = new char[n].

**Функции**
Когато дадено парче код ще се използва на повече от едно място в кода е добра практика това парче код да се изнесе във функция, която може да се извиква. По този начин ще намалим значително обема на кода като го преизползваме. Функциите се характеризират със сигнатура, която включва: име на функцията, брой и тип параметри, тип на връщана та стойност. Всяка една програма на C++ се състои от поне една функция – main, която

приема масив от входни аргументи от тип масив от ASCII символи и целочислен брой на тези аргументи и връща целочислен тип, който се ползва за код за изпълнение на функцията. Други функции могат да се дефинират извън main-а и да се извикват в него или в други функции, но с приключването на main функцията приключва и програмата. Добра практика е една функция да изпълнява точно една задача.

```
int sum(int a, int b) {
        return a+b;
}

int main(int cnt, char** args) {
        int a = 1, b = 2;
        int res = sum(a, b);
        std::cout << res << std::endl;
        return 0;
}
```

**Дефиниране на функция**

Обща дефиниция на функция:

```
[<модификатор>][<тип на функция>]<име на функция>(<формални параметри>) {
    <тяло>
}
```

1. Модификатор – спецификатори, които се използват от компилатора като например extern и inline. extern е модификаторът по подразбиране и може да се пропуска. С него указваме на компилатора, че функцията е декларирана някъде на друго място в кода и за това липсва тяло. inline казва на компилатора да замести с тялото на функцията навсякъде където тази функция се използва в кода. Това ще подобри производителността на кода.
2. Тип на функция – това е типа на резултата, който функцията връща. Той може да е скаларен или съставен. В случай, че типа е съставен се връща указател към него (или първия му елемент, ако е масив). Функциите, които извършват своята работа без да се налага да връщат резултат са от тип void. Всяка функция, кото връща даден тип може да се пренапише като void функция, като данните, които връща се подадат като аргументи с адреси на вече създадени типове в тялото където се извиква функцията. По този начин може да „връщаме“ повече от един тип данни след изпълнението на функцията.
3. Име на функция – името на функцията служи за извикването ѝ. Няколко функции могат да имат едно и също име и това коя се извиква се определя или от останалата част на сигнатурата и или от контекста. Контекст може да е каст операция, тип на връщана стойност и други.
4. Формални параметри (аргументи) – списък от аргументи, които функцията получава при своето начално изпълнение. Списъкът може да е празен или да съдържа аргументи по подразбиране, които компилаторът автоматично присвоява, ако извикващият функцията не предостави експлицитно стойност за тях. Ако има аргументи по подразбиране, те задължително се слагат в края на списъка.
5. Тяло – списък от инструкции, които да се извършат в указания ред и да доведат до някакъв желан резултат.

Пример за функция с аргументи по подразбиране:

```
int sum(int x, int y, int z = 1, int w = 2) {
        return x + y + z + w;
}
```

Функциите може и да са анонимни без име или наричани още ламбда функции.

Обща дефиниция на ламбда функция:

```
[<прихващащ клас>] (<формални параметри>) {
    <тяло>
}
```

Прихващащият клас може да е [&] или [=]. [&] означава, че променливите са прихванати по референция, а [=] – по стойност. По подразбиране прихващащият клас е [=]. Пример:

| Сортиране на масив с анонимна функция функция |
|---|
| int arr[ ] = {3, 1, 2};<br>std::sort(begin(arr), end(arr), **[&]**( int a,  int b) { return a < b; }); |

**Обръщение към функция**
Функция може да се извика като се зададе името ѝ и формалните ѝ параметри. Списъкът с параметри при извикването на функцията се нарича списък от фактически параметри. Тези фактически параметри трябва да съответстват с формалните параметри по брой (ако няма аргументи по подразбиране), тип и вид:

- По брой – фактическите параметри при извикването на функцията трябва да са същия брой като формалните;
- По тип – фактическият параметър на $i$-та позиция трябва да си съответства по тип с формалния параметър на $i$-та позиция;
- По вид – ако формалният параметър е от тип указател, то фактическия параметър задължително е адрес на променлива. Ако пък формалният параметър е от тип псевдоним (тоест адрес на променлива), фактическият параметър задължително е променлива.

Обръщението към функция поражда образуването на нова стекова рамка (тоест в стековата памет се създава нов сегмент, обособен за дадената функция), след което формалните параметри се свързват с фактическите (всеки формален параметър се свързва със съответния си фактически) и накрая се изпълнява функцията. След приключването на изпълнението на функцията, стековата ѝ рамка се зачиства. Трябва да се отбележи, че стековата памет е организирана като стек (откъдето идва и името ѝ). Тоест на дъното на тази памет лежи стековата рамка за функцията main, над нея – стековата рамка на функцията, извикана от main, над нея – стековата рамка, извикана от предишната функция и т.н. до върха. На върха на стека се намира стековата рамка на последно извиканата функция.

**Рекурсивни функции**
Рекурсивните функции са функции, които в дефиницията си извикват себе си. В избираемия предмет „Сложност и изчислимост“ се разглежда теорема, която гласи, че всяка рекурсивна функция може да се пренапише като итеративна. Тази теорема е много мощна, тъй като много от действията в реалния свят могат лесно да се опишат чрез използването на рекурсия, тъй като се разбиват на по-прости аналогични действия.

Рекурсията може да се раздели на два вида:

- **пряка** – когато една функция извиква самата себе си;
- **непряка** (chain recursion) – когато една функция извиква друга, а тя от своя страна извиква първата.

Примери:

| Пряка рекурсия | Непряка рекурсия (Chain Recursion) |
|---|---|
| long long **factorial**(int n) {<br>  if (n < 2) {<br>  return 0;<br>  }<br><br>  return n * **factorial**(n - 1);<br>} | bool **hacksaw1**(int *a, int k, int n) {<br>  if (k == n - 2) return true;<br>  return ((a[k - 1] > a[k] && a[k] < a[k + 1]) && **hacksaw2**(a, k + 1, n));<br>}<br><br>bool **hacksaw2**(int *arr, int k, int n) {<br>  if (k == n - 2) return true;<br>  return ((a[k - 1] < a[k] && a[k] > a[k + 1]) && **hakcsaw1**(a, k + 1, n));<br>} |

Етапи от изпълнението на рекурсията:
- Разгъване – когато се извикват вложените функции;
- Свиване – когато вложените функции една по една връщат резултатите си.

Характеристики и необходими условия на рекурсията:
- Дълбочина (характеристика) – броят на рекурсивните извиквания на функцията;
- Дъно (необходимо условие) – това е необходимо условие за достигане на край на рекурсията.

Въпреки силната изразителна мощ на рекурсията и по-простия код, в програмите, които се използват в работещи проекти, тя е много рядко срещано явление. Това е така, тъй като при нея може да се скрият много бъгове и да предизвика препълване на стековата памет, която расте при разгъването ѝ. Освен това, тя не е толкова интуитивна за разбиране, а по даден проект работят много програмисти и кода трябва да е възможно най-четим.

**Предаване на параметрите по стойност, чрез указател и чрез псевдоним**
В програмния език C++ съществуват три вида предавания на аргументи на функция. Най-простият и лесен начин е предаването на параметри по стойност. При предаването на стойност, параметърът, подаден към функцията се копира локално в нея и се използва копието при работата ѝ. По този начин всяка промяна върху параметъра в рамките на функцията няма да се отрази върху подавания параметър отвън. Например:

**по стойност**

```
int foo(int x) {
        x++;
        return x;
}

int main() {
        int num = 3;
        std::cout << num << std::endl;          \\ принтира 3
        std::cout << foo(num) << std::endl;     \\ принтира 4
        std::cout << num << std::endl;          \\ принтира 3
        return 0;
}
```

Този вид подаване на параметри е скъп, тъй като трябва да се извършва копие на променлива на локално ниво. Ако променливата заема голямо пространство от паметта, това намалява производителността и хаби допълнителна памет за локалното копие.

За да направим така, че една функция да промени подадените ѝ аргументи и тази промяна да се отрази на фактическата променлива е необходимо да подадем параметрите **по референция** (псевдоними). След типа на формалния параметър добавяме запазения символ **&**. Функцията директно работи с променливата, тъй като получава адреса ѝ и „знае“ къде да извършва манипулациите с нея.

Друг начин да се направи така, че промяна във функция на даден параметър да се отрази на външната променлива, подадена като фактически параметър, е чрез подаване на параметъра **чрез указател**. Това се извършва чрез поставяне на запазения символ * след типа на формалния параметър. Тук, обаче, функцията очаква да ѝ бъде подаден адрес като фактически параметър на мястото на формалния параметър маркиран със символа *.

| по референция | чрез указател |
|---|---|
| void swap(int& a, int& b) {<br>  int temp = a;<br>  a = b;<br>  b = temp;<br>}<br><br>int main() {<br>  int a = 3, b = 8;<br>  std::cout << a << " " << b << std::endl; \\ 3 8<br>  swap(a, b);<br>  std::cout << a << " " << b << std::endl; \\ 8 3<br>  return 0;<br>} | void swap(int* a, int* b) {<br>  *a ^= *b;<br>  *b ^= *a;<br>  *a ^= *b;<br>}<br><br>int main() {<br>  int a = 3, b = 8;<br>  std::cout << a << ' ' << b << std::endl;<br>  swap(&a, &b);<br>  std::cout << a << ' ' << b << std::endl;<br>  return 0;<br>} |

**Масиви като формални параметри**

**Едномерни масиви**

Масивите се предават като параметри единствено чрез указател. Тъй като самият масив е указател към първия елемент, тази концепция е логична. Това означава, че когато подадем масив като параметър, във функцията се работи директно с него, а не с негово копие. Тоест всяка промяна в масива в рамките на функцията ще се отбележи в масива извън функцията. Имаме три начина, по които да подадем един масив, два от които автоматично се преобразуват от C++ към предаване чрез указател:

int arr[ ], int arr[n] – и двете са еквивалентни на int* arr;

По този начин се пести време и памет, която иначе щеше да бъде използвана за локално копие на масива, което е излишно.

**Многомерни масиви**

Многомерните масиви отново се предават чрез указател, но при тях специфичното е че трябва да се специфицират размерите на дименсиите с изключение на първата. В двумерния случай например, трябва да се специфицират броя колони, но не и броя редове. Например следното е напълно валидно за двумерен масив: int arr[ ][20]. Масивите с повече дименсии следват същия шаблон. Тъй като многомерните масиви се предават като указател, това означава, че промените в рамките на функцията, ще се отразят и върху самият масив извън функцията.