14. Процедурно програмиране - основни информационни и алгоритмични структури *(C++)*.

Принципи на структурното програмиране. Управление на изчислителния процес. Основни управляващи конструкции – условни оператори, оператори за цикъл. Променливи – видове: локални променливи, глобални променливи; инициализация на променлива; оператор за присвояване. Функции и процедури. Параметри – видове параметри. Предаване на параметри – по име (по указател), по стойност. Типове и проверка за съответствие на тип. Типовете като параметри на функция. Едномерни и многомерни масиви. Основни операции с масиви – индексиране. Сортиране и търсене в едномерен масив – основни алгоритми за сортировка. Рекурсия – пряка и косвена рекурсия, линейна и разклонена рекурсия.

Принципи на структурното програмиране.

Основните принципи на структурното програмиране са принципа за **модулност** и принципа за **абстракция на данните**.

Съгласно **принципа за модулност**, програмата се разделя на подходящи взаимосвързани части, всяка от които се реализира чрез определени средства. Целта е промените в представянето на данните да не променят голям брой от модулите на програмата.
Функцията е самостоятелен фрагмент на програмата – отделна програмна единица, съдържаща описание на променливи и набор от оператори на езика. Те се затварят между фигурни скоби и се наричат тяло на функцията. Функцията има възможност да предава и получава информация към и от други функции. За да се предаде в извиканата функция стойност на една променлива, дефинирана в извикващата функция, е необходимо тази променлива да се включи в списъка на предаваните стойности. Този списък се нарича списък на аргументите. Обикновено след изпълнение на дадена функция в извикващата функция се връщат резултатите от някакви изчисления. Тези резултати имат определен тип. Възможно е една функция да не връща резултат и тогава тя се нарича **процедура.**
Във функционално отношение функцията е част от програмата, с изпълнението на която се получават определени резултати. Чрез използването на функции една програма може да се раздели на отделни модули. Предимствата на модулния подход са следните:

- модулите могат да се програмират независимо един от друг;
- модулите могат много лесно да се тестват за грешки и да се модифицират;
- могат да се използват вече готови програми, оформени като модули;
- логиката на цялата програмата става по-разбираема.

Съгласно **принципа за абстракция на данните**, методите за използване на данните се отделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с абстрактни данни – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени конструктори, мутатори и функции за достъп (селектори), които реализират абстрактните данни по конкретен начин. Така при решаването на даден проблем се оформят следните нива на абстракция:

1. Приложения в проблемната област.
2. Модули, реализиращи основните операции над данните.
3. Примитивни операции – конструктори, мутатори, селектори.
4. Избор на представянето на данните.

Реализацията на подхода трябва да е такава, че всяко ниво използва единствено средствата на непосредствено следващото го ниво. По този начин, промените, които възникват на едно ниво ще се отразят само на предходното ниво.

Информационни структури

При изпълнение на компютърна програма се извършват определени действия над данните, дефинирани в програмата. Тези данни се съхраняват в **информационните структури**, допустими в съответния език за програмиране. Най-общо типовете информационни структури могат да бъдат разделени на два вида: вградени и абстрактни. Вградените типове са предварително дефинирани и се поддържат от самия език, а абстрактните типове се дефинират от програмиста. Друга класификация на типовете е следната: скаларни и съставни типове. Скаларните типове представят данни, които се състоят само от една компонента.
При съставните типове данни, данните представляват редица от компоненти. Скаларните типове, поддържани в C++ са следните: булев, цял, реален, символен, изброен, указател, псевдоним. Съставните типове, поддържани в C++ са следните: масив, вектор, запис.

Нека T е име или дефиниция на тип. За типа T, T* е тип, наречен **указател** към T. Множеството от стойностите на типа T* се състои от адресите на променливите от тип T, заедно със специалната константа NULL (нулев указател), която може да бъде свързана с всеки указател, независимо от типа T. За променливите от тип указател се разпределят 4 байта памет.
Нека T е име или дефиниция на тип. За типа T, T& е тип, наречен **псевдоним** на T. Множеството от стойностите на типа T& се състои от всички вече дефинирани променливи от тип T.
Дефинирането на променлива от тип T& задължително е с инициализация – дефинирана променлива от тип T. След това не е възможно променливата-псевдоним да стане псевдоним на друга променлива.
Променлива, това е място за съхранение на данни, което може да съдържа различни стойности от някакъв определен тип. Идентифицира се със зададено от потребителя име. Една променлива се дефинира като се зададат нейното име и типа на стойностите, които може да приема. Типът определя броя на байтовете, в които ще се съхранява променливата. Мястото в паметта, където е записана стойността на променливата се нарича **адрес** на тази променлива. По-точно адресът на променливата е адресът на първия байт от множеството байтове, отделени за променливата. Намирането на адреса на една променлива става чрез унарния префиксен оператор **&** (амперсанд) със следния синтаксис: &<променлива>, където <променлива> е име на вече дефинирана в програмата променлива.
Адресите на променливите от един тип T могат да се присвояват на променливите от тип T* - указател към T.
Извличането на съдържанието на един указател става чрез унарния префиксен оператор ***** със следния синтаксис: *<променлива_от_тип_указател>. С други думи, този оператор извлича стойността на адреса, записан в променливата от тип указател с име <променлива_от_тип_указател>. Ако променливата-указател е от тип T*, то резултатът от прилагането на оператора * е данна от тип T – всички операции допустими за типа T са допустими и за нея.

Променливите от тип указател могат да участват като операнди в следните аритметични и логически операции +, -, ++, --, ==, !=, <, >, <=, >=. Изпълнението на аритметични операции върху указатели е свързано с някои особености, заради което аритметиката с указатели се нарича още **адресна аритметика**. Особеността се изразява в т.н. мащабиране. Ще го изясним с пример.
Да разгледаме следния фрагмент:

```
int *p;
double *q;
…
p = p + 1;
q = q + 1;
```

Операторът p = p+1; свързва p не с предишната стойност на p, увеличена с 1, а с предишната стойност на p, увеличена с 1*4, където 4 е броят на байтовете, необходими за записването на данна от тип int (p е указател към int). Аналогично, q = q+1; увеличава стойността на q с 1*8, а не с 1, тъй като q е указател към тип double (8 байта са необходими за записване на данна от този тип). Общото правило е следното: ако p е указател от тип T*, стойността на p+i е p+i*sizeof(T), където sizeof(T) е функция, която намира броя на байтовете, необходими за записване на данна от тип T. При изваждането нещата стоят по аналогичен начин.

В C++ е възможно да се дефинират указатели, които са константи, а също и указатели, които сочат към константи. И в двата случая се използва запазената дума const, която се поставя пред съответните елементи от дефинициите на указателите. Стойността на елемента дефиниран като const (указателя или обекта, към който сочи) не може да се променя. Ще разгледаме пример:

```
int i, j, k;
int * const b = &i;
const int *c = &j;
const int * const d = &k;
```

b е константен указател към тип int и той не може да променя стойността си, но чрез него може да се променя стойността на i; c е указател към константа от тип int и той може да променя стойността си, но чрез него не може да се променя стойността на j; d е константен указател към константа от тип int – той не може да променя стойността си и чрез него не може да се променя стойността на k, може да служи единствено за извличане на стойността на k.

В C++ има интересна и полезна връзка между указателите и масивите. Тя се състои в това, че имената на масивите са указатели към техните "първи" елементи. Това позволява указателите да се разглеждат като алтернативен начин за обхождане на елементите на даден масив. Ще разгледаме пример - нека a е масив, дефиниран по следния начин:
int a[100];
Тъй като a е указател към a[0], то *a е стойността на a[0], т.е. *a и a[0] са два различни записа на стойността на първия елемент на масива. Тъй като елементите на масива са разположени последователно в паметта, a+1 е адресът на a[1], a+2 е адресът на a[2] и т.н. a+99 е адресът на a[99]. Тогава *(a+i) е друг запис на a[i] (i = 0, 1, …, 99). Има обаче една особеност. Имената на масивите са константни указатели. Заради това, някои от аритметичните операции, приложими над указатели не могат да се приложат за масиви. Такива са ++, -- и присвояването на стойност. Използването на указатели е по-бърз начин за достъп до елементите на масива и заради това се предпочита. Индексираните променливи правят кода по-ясен и разбираем. В процеса на компилация всички конструкции от вида a[i] се преобразуват в *(a+i), т.е. операторът за индексиране се обработва от компилатора чрез адресна аритметика.
Името на двумерен масив е константен указател към първия елемент на едномерен масив от константни указатели. Ще

изясним с пример казаното. Нека а е двумерния масив, дефиниран така:
int a[10][20];
Тогава а е константен указател към първия елемент на едномерния масив a[0], a[1], …, a[9], като всяко a[i] (i = 0, 1, …, 9) е константен указател към a[i][0]. При това положение имаме следните еквивалентни записи: **a ⇔ a[0][0], *a ⇔ a[0], *(a+1) ⇔ a[1], …,*(a+9) ⇔ a[9], по-общо a[i] ⇔ *(a + i) (i = 0, 1, …, 9), a[i][j] ⇔ (*(a + i))[j] ⇔ *(*(a + i) + j).

Основни управляващи конструкции – условни оператори, оператори за цикъл

Основните алгоритмични структури в C++ са операторът за присвояване, разклонените алгоритмични структури – операторите за условен преход if в кратка форма, if-else в дълга форма и switch, оператор за безусловен преход goto и операторите за цикли, чрез които се пораждат циклични изчислителни процеси – for, while и do-while. Има и още един вид оператори – операторът break, който се използва за излизане от цикъл или от switch и оператор continue, чрез които безусловно се прескача към следващата итерация на цикъл.

Функции и процедури. Параметри – видове параметри. Предаване на параметри – по име (по указател), по стойност.

Добавянето на нови оператори в приложенията, реализирани на езика C++, се осъществяват чрез **функциите**. Те са основни структурни единици, от които се изграждат програмите на езика. Всяка функция се състои от множество от оператори, оформени подходящо, за да се използват като обобщено действие или операция. След като една функция бъде дефинирана, тя може да бъде изпълнявана многократно за различни входни данни.

Програмите на C++ се състоят от една или повече функции. Сред тях задължително трябва да има точно една с име **main**, която се нарича **главна функция**. Тя е първата функция, която се изпълнява при стартиране на програмата. Главната функция, от своя страна, може да се обръща към други функции. Нормалното изпълнение на програмата завършва с изпълнението на главната функция (възможно е изпълнението да завърши принудително по време на изпълнение на функция, различна от главната).
Ще разгледаме най-общото разпределение на оперативната памет за изпълнима програма на C++. Това разпределение зависи от изчислителната система, от типа на операционната система, а също от модела памет. Най-общо се състои от: програмен код, област на статичните данни, област на динамичните данни и програмен стек.
В частта **програмен код** е записан изпълнимият код на всички функции, изграждащи потребителската програма. В областта на **статичните данни** са записани глобалните обекти на програмата. За реализиране на динамични структури от данни се използват средства за динамично разпределяне на паметта. Чрез тях се заделя и се освобождава памет в процеса на изпълнение на програмата, а не преди това (при нейното компилиране). Тази памет е областта на **динамичните данни**.
Програмният стек съхранява данните на функциите на програмата. Неговите елементи са блокове от памет, съхраняващи данни, дефинирани в някаква функция, които се наричат **стекови рамки.**
Ще разгледаме примерен фрагмент, който изчислява най-големият общ делител на две естествени числа. За целта дефинираме две функции: gcd (x, y), която намира най-големия общ делител на x и y и main, която се обръща към gcd

```
int gcd (int x, int y)
{
        int h;
        while (y != 0)
        {
                h = x;
                x = y;
                y = h% y;
        }
        return x;
}
int main ()
{
        int a = 14, b = 21;
        int r = gcd (a, b);
        return 0;
}
```

Описанието на функцията gcd прилича на това на функцията main. Заглавието определя, че gcd е двуаргументна целочислена функция с цели аргументи. Името е произволен идентификатор, в случая е направен мнемоничен. Запазената дума int пред името на функцията е нейният тип (по-точно е типът на резултата на функцията). В кръгли скоби и отделени със запетая са описани параметрите x и y на gcd. Те са различни идентификатори и се предшестват от типовете си. Наричат се **формални параметри** за функцията. Тялото на функцията е блок, реализиращ алгоритъма на Евклид за намиране на най-

големия общ делител на естествените числа x и y. Завършва с оператора return x; ,чрез който се прекратява изпълнението на функцията като стойността на израза след return се връща като стойност на gcd в мястото, в което е направено обръщението към нея.
Ще опишем как се изпълнява фрагментът. Дефинициите на функциите main и gcd се записват в областта на паметта, определена за програмния код. Изпълнението на програмата започва с изпълнението на функцията main. Чрез първия ред от тялото на main се дефинират целите променливи a и b, като на a се присвоява стойност 14 и на b стойност 21.
В тази последователност те се записват в дъното на програмния стек в стековата рамка на main. Чрез следващия оператор се дефинира цялата променлива r, като в стековата рамка на main, веднага след променливата b се отделят четири байта, в които ще се запише резултатът от обръщението gcd (a, b) към функцията gcd. Променливите a и b се наричат **фактически параметри** за това обръщение. Забелязваме, че типът им е същия като на съответните формални параметри x и y.
Сега ще опишем как се изпълнява обръщението gcd (a, b).
В програмния стек се генерира нов блок памет – стекова рамка за функцията gcd. Обръщението се осъществява на два етапа. Първият етап е **свързването** на формалните с фактическите параметри. В стековата рамка на gcd се отделят по четири байта за формалните параметри x и y в обратен ред на реда, в който са записани в заглавието. В тази памет се откопират стойностите на съответните им фактически параметри. Отделят се също четири байта за т.н. return-адрес, адреса на мястото в main, където ще се върне резултатът, а също се отделя памет, в която се записва адресът на стековата рамка на извикващата функция (в случая main). И така на този етап формалните параметри x и y се свързват съответно със стойностите 14 (стойността на a) и 21 (стойността на b). Вторият етап е изпълнението на тялото на функцията gcd. На първата стъпка се заделя памет четири байта за дефинираната променлива h във върха на стековата рамка за gcd. Тъй като е в сила y != 0, стойността на h става 14, след това стойността на x става 21 и след това стойността на y става
14 % 21 = 14. С други думи, тук x и y размениха стойностите си.
На следващата итерация на цикъла отново y != 0, така че стойността на h става 21, стойността на x става 14 и след това стойността на y става 21 % 14 = 7. Отново y != 0, така че отново се изпълнява тялото на цикъла – стойността на h става 14, стойността на x става 7, стойността на y става 14 % 7 = 0. Това е последната итерация на цикъла и той приключва.
Изпълнението на оператора return x; преустановява изпълнението на gcd като връща в main в мястото на прекъсването (return-адреса) стойността 7 на обръщението gcd (a, b). Отделената за gcd стекова рамка се освобождава. Указателят към върха на стека сочи края на стековата рамка на main. Изпълнението на програмата продължава с инициализацията на променливата r. Резултатът от обръщението gcd (a, b) се записва в отделената за r памет. След това се изпълнява оператора return 0; след което се освобождава и стековата рамка на main.
Функцията gcd реализира най-простото и чисто дефиниране и използване на функции – получава входните си стойности единствено чрез формалните си параметри и връща резултата си чрез оператора return. Забелязваме, че обръщението gcd (a, b) работи с копия на стойностите на a и b, запомнени в x и y, а не със самите a и b. В процеса на изпълнение на тялото на gcd, стойностите на x и y се променят, но това не оказва влияние на стойностите на фактическите параметри a и b. Такова свързване на формалните с фактическите параметри се нарича **свързване по стойност** или още **предаване на параметрите по стойност(1)**.
При него фактическите параметри могат да бъдат не само променливи, но и изрази от типове, съвместими с типовете на съответните формални параметри.
В редица случаи се налага функцията да получи входа си чрез някои от формалните си параметри и да върне резултат не по обичайния начин (чрез оператор return), а чрез същите или други параметри. Ще разгледаме примерен фрагмент, който разменя стойностите на две реални променливи. Идеята е следната – ако дефинираме функция swap (double *x, double *y), която разменя стойностите на реалните променливи, към които сочат указателите x и y, то обръщението swap (&a, &b) би разменило стойностите
на a и b.

```
void swap (double *x, double *y)
{
        double work = *x;
        *x = *y;
        *y = work;
        return;
}
int main ()
{
        double a = 1.5, b = 2.75;
        swap (&a, &b);
        return 0;
}
```

Функцията swap има подобна структура като на gcd. Но и заглавието и тялото и са по-различни. Типът на swap е указан чрез запазената дума void. Това означава, че функцията не връща стойност чрез оператора return. Затова в тялото на swap е пропуснат израза след return. Формалните параметри са указатели към double, а в тялото се работи със съдържанието на тези указатели. Забелязваме също, че обръщението към swap във функцията main не участва като аргумент на операция, а е оператор. Изпълнението се осъществява подобно на изпълнението на по-горния фрагмент с малки разлики – при първия етап на изпълнението на обръщението към swap в отделената памет за формалните параметри x и y се записват **адресите** на съответните им фактически параметри. Поради тази причина, промяната на стойностите, към които сочат формалните

параметри x и y води до промяна на самите фактически параметри. Функцията swap получава входните си стойности чрез формалните си параметри и връща резултата си чрез тях. Забелязваме, че обръщението swap (&a, &b) работи не с копия на стойностите на a и b, а с техните адреси. В процеса на изпълнение на тялото на swap се променят стойностите на фактическите параметри a и b. Такова свързване на формалните с фактическите параметри се нарича **свързване по указател** или **предаване на параметрите по указател** или **свързване по адрес(2)**. При този вид предаване на параметрите, фактическите параметри задължително са променливи указатели или адреси на променливи.
Освен тези два начина на предаване на параметри, в езика C++ има още един – **предаване на параметри по псевдоним(3).** Той е сравнително по-удобен от предаването по указател и се предпочита от програмистите. Ще го илюстрираме с фрагмент, еквивалентен на горния (т.е. изпълняващ същите действия), но реализиращ функцията swap, в която предаването на параметрите е по псевдоним.

```
void swap (double &x, double &y)
{
        double work = x;
        x = y;
        y = work;
        return;
}
int main ()
{
        double a = 1.5, b = 2.75;
        swap (a, b);
        return 0;
}
```

Специфичното при изпълнението на фрагмента е следното – при първия етап от изпълнението на обръщението към swap, тъй като формалните параметри x и y са псевдоними на променливите a и b, за тях памет не се отделя в стековата рамка на swap. Това, което се прави, е параметърът x да се закачи за фактическия параметър a и аналогично параметърът y се закача за фактическия параметър b. Така всички действия с x и y в swap всъщност се изпълняват с фактическите параметри a и b от main съответно. Да забележим, че фактическите параметри, съответстващи на формални параметри-псевдоними при всички случаи са променливи. Тази реализация на swap е по-ясна от съответната с указатели. Тялото й реализира размяна на стойностите на две реални променливи без да се налага използването на адреси.

Обощение:

- предаване по указател: копират се адресите на фактическите параметри и промените на фактическите параметри влият на формалните
- предаване по стойност – копират се стойностите на фактическите параметри в стек и после копието се разрушава, няма променя за формалните параметри
- предаване по псевдоним – адресите на фактическите параметри се съхраняват в паметта; не се правят копия на формалните параметри; промените на фактическите параметри влият на формалните

Възможно е някои параметри да се предават по стойност, други по псевдоним или указател, а също функцията да получи резултат и с оператора return.
Възможно е в една програма, към една функция да се извършват обръщение на място, което предшества нейната дефиниция. В този случай дефиницията на функцията, която извършва това обръщение трябва да се предшества от **декларацията** на използваната функция. Декларацията на една функция се състои от нейното заглавие, последвано от ‘;’. В него имената на формалните параметри могат да се пропуснат.
Типът на една функция е произволен без масив, но се допуска да е указател. Ако е пропуснат, се подразбира int.
Списъкът от формалните параметри на една функция (още се нарича **сигнатура**) може да е празен или void. В случай, че е непразен, имената на параметрите трябва да са различни. Те заедно с името определят еднозначно функцията. Възможно е също един формален параметър да се специфицира със запазената дума const, което означава, че неговата стойност няма да бъде променяна в рамките на тялото на функцията.
Операторът **return** връща резултата на функцията в мястото на извикването. Синтаксисът е следния: return [<израз>];. Тук return е запазена дума, <израз> е произволен израз от тип, съвместим с типа на функцията. Ако типът на функцията е void, <израз> се пропуска. Семантиката е следната: пресмята се стойността на <израз>, конвертира се до типа на функцията и връщайки получената стойност в мястото на извикването на функцията, прекратява нейното изпълнение. Да отбележим, че ако функцията не е void, тя задължително трябва да върне стойност. Това означава, че операторът return трябва да се намира във всички клонове на тялото.
Функциите могат да се дефинират в произволно място на програмата, но не и в други функции. Така за разлика от други езици за процедурно програмиране, в C++ не се допуска влагане на функции.
При стартирането на една програма се създава стековата рамка на главната функция, която се поставя в дъното на програмния стек. Във всеки момент от изпълнението на програмата, във върха на стека е стековата рамка на функцията, която се обработва в момента. Непосредствена под нея в стека стои стековата рамка на функцията, извършила обръщението към функцията, която се обработва в момента. Когато изпълнението на една функция завърши, стековата рамка на функцията, съответна на това обръщение се отстранява от стека.

Променливи – видове: локални променливи, глобални променливи; инициализация на променлива; оператор за присвояване.

Идентификаторите в C++ означават имена на константи, променливи, формални параметри, функции, класове. Най-общо казано има три вида области на идентификаторите – **глобална, локална** и **област на клас**. Областите се задават неявно – чрез позицията на дефиницията на идентификатора в програмата. Област на клас няма да разглеждаме.
Глобалните идентификатори са дефинираните извън дефинициите на функции променливи и константи и могат да се използват във всички функции на програмата, чиято дефиниция следва тяхната такава.
Повечето константи и променливи са локални идентификатори и имат локална област на действие. Те са дефинирани вътре във функциите и не са достъпни за кода в другите функции на модула. Областта им се определя според следното общо правило – започва от мястото на дефинирането и завършва в края на оператора (блока), в който е дефиниран идентификатора. Формалните параметри на функцията също имат локална видимост. Областта им на действие е цялото тяло на функцията. В различните области могат да се използват еднакви идентификатори. Ако областта на един идентификатор се съдържа в областта на друг идентификатор, първият се нарича нелокален за последния. В този случай е в сила правилото: Локалният идентификатор скрива нелокалния в областта си.
Освен чрез механизма за свързване на формалните параметри с фактически и чрез предаване на резултат чрез оператора return, функциите могат да обменят помежду си данни чрез глобални променливи. Първите два начина са основни и са за предпочитане, тъй като с тях се постигат по-ясни и разбираеми програми. От друга страна, използването на глобални променливи е по-ефективно, тъй като не изисква заместване.
Използването на глобални променливи обикновено се счита за лоша практика поради тяхната нелокалност: една глобална променлива потенциално може да се модифицира от всяко място в програмата и всяка част от програмата може да зависи от нея. Въпреки това, в някои случаи, глобалните променливи са подходящи за употреба, тъй като те могат да се използват за избягване на предаването на често използвани променливи в рамките на няколко функции. Ще разгледаме пример с една проста програма.

```
#include <iostream.h>
int global = 3;
void ChangeGlobal ()
{
        global = 5;
}
int main ()
{
        cout << global << endl;
        ChangeGlobal();
        cout << global << endl;
        return 0;
}
```

Тъй като променливата global е глобална, няма нужда тя да се предава като параметър за да може да се използва от функции, различни от главната функция. Глобалната променлива може да се използва във всяка функция на програмата.
Изходът от изпълнението на програмата:

```
3
5
```

Използването на глобални променливи прави софтуерът по-труден за четене и за разбиране. Тъй като всяка част от кода на програмата може да се използва за промяна на стойността на глобална променлива, разбирането на използването на променливата може да включва разбирането на голяма част от самата програма.

Рекурсия – пряка и косвена рекурсия, линейна и разклонена рекурсия.

Най-общо един обект е **рекурсивен**, ако се съдържа в себе или е дефиниран чрез себе си. В математиката има много примери за рекурсивни обекти, например функцията n!, която се дефинира така: 0! = 1, (n+1)! = n * n!.
Езикът C++ поддържа две конструкции, които позволяват да се реализират рекурсивни алгоритми – това са структури с рекурсия и рекурсивни функции.

Едно интересно свойство на структурата е това, че нейните елементи могат да имат тип указател и следователно тип указател към структура. Ще отбележим изрично, че не се допуска елемент на структура да бъде структура от същия тип. Тъй като указателят не дефинира променлива, а само сочи към такава, за него ограничението не е в сила. Така елементи на една структура могат да бъдат указатели към структура от същия тип. По този начин се получава рекурсивно използване на структури. Чрез него могат да се съставят сложни описания на данни като линейни списъци и дървовидни структури.
Например, в C++ е валидна следната дефиниция на структура:

```
struct list {
        int value;
        list *next;
};
```

Указателят next, описан в тялото на структурата list сочи към променлива – структура от тип list. Тази структура може да се използва за реализация на едносвързан списък.

Известно е, че в тялото на всяка функция може да бъде извикана друга функция, която е дефинирана или декларирана до момента, в който се извиква. Освен това, в C++ е разрешено една функция да извиква в тялото си самата себе си. Функция, която пряко или косвено се обръща към себе си, се нарича **рекурсивна**.
Под **косвено** обръщение разбираме следното – функцията F_1 извиква функцията F_2, функцията F_2 извиква функцията F_3, …, функцията F_{n-1} извиква функцията F_n и след това функцията F_n извиква функцията F_1.
Сега ще разгледаме примерен фрагмент, който съдържа рекурсивна функция и ще проследим как той се изпълнява. Фрагментът ще пресмята m! за цяло число m, $m \geq 0$.

```
int fact (int n){
        if (n == 0) return 1;
        else return n * fact (n-1);
}
int main () {
        int m = 4;
        int n = fact (m);
        return 0;
}
```

В тази фрагмент е описана рекурсивна функция fact, която приложена към цяло неотрицателно число връща факториела на числото. При изпълнението на програмата интересно е извършването на обръщение към fact с фактически параметър m, който има стойност 4.
Ще проследим как се извършва това обръщение. Генерира се стекова рамка за обръщението към fact. В нея се отделят четири байта за формалния параметър n, в която памет се откопирва стойността на фактическия параметър m(4) и след това започва изпълнение на тялото на функцията. Тъй като n е различно от 0,се изпълнява операторът в else-частта, при което се извършва обръщение към fact с фактически параметър n-1, т.е. fact (3). По такъв начин преди завършването на първото обръщение към fact се прави второ обръщение към тази функция. За целта се генерира нова стекова рамка за новото обръщение към функцията fact, в която за формалния параметър n се откопира стойност 3. Тялото на функцията fact започва да се изпълнява втори път (временно спира изпълнението на тялото на функцията fact, предизвикано от първото обръщение към нея). По аналогичен начин възникват още обръщения към функцията fact. При последното от тях, стойността на формалния параметър n е равна на 0 и тялото на fact се изпълнява напълно. В резултат, изпълнението на това обръщение завършва и за стойност на fact се връща 1. Тази стойност се връща в непосредствено предишното извикване на fact, което изчислява 1*1 = 1 и аналогично се извършва връщане назад по веригата от извикванията на fact, докато се стигне до първото извикване, което изчислява 4*6 = 24 и връща тази стойност в главната функция. Естествено, след завършването на всяко изпълнение на fact, стековата рамка, която отговаря за това изпълнение се освобождава.

В този конкретен случай, рекурсивното дефиниране на функцията факториел не е подходящо, тъй като съществува лесно итеративно решение. Ще отбележим, че тъй като в тялото на рекурсивната функция fact има само едно обръщение към fact, то тогава всяко извикване на fact поражда най-много едно извикване на fact.
Затова функцията fact още се нарича **линейно рекурсивна**.
Когато в тялото на една рекурсивна функция има повече от едно обръщение към самата нея и при това съществува извикване на функцията, което поражда повече от едно извикване към същата функиция, то рекурсивната функция се нарича **разклонено рекурсивна**. Пример е рекурсивната функция за изчисляване на n-тото число на Фибоначи (n естествено), която се дефинира така:

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

В случая всяко извикване на fib поражда две независими извиквания на fib, което води до експоненциална сложност на рекурсивния алгоритъм. От друга страна, съществува елементарен итеративен алгоритъм, който решава задачата с линейна сложност.
Така стигаме до следната препоръка: ако за решаването на една задача може да се използва итеративен алгоритъм, той трябва да се реализира, не се препоръчва безпринципното използване на рекурсия, тъй като това води до загуба на време и памет.
Съществуват обаче задачи, които се решават трудно, ако не се използва рекурсия.
Простотата и компактността на рекурсивните функции проличава особено при работа с динамичните рекурсивни структури данни (които се дефинират чрез рекурсивни структури): свързан списък, стек, опашка, дърво. Основен недостатък на рекурсивните програми е намаляването на бързодействието поради загуба на време за копиране на параметрите в стека. Освен това се изразходва повече памет, особено при по-дълбока степен на вложеност на рекурсията.

Едно общо правило при създаването на рекурсивни функции е следното: когато реализираме рекурсия, във функцията трябва да има гранични условия за излизане от рекурсията, които задължително се достигат след краен брой вложени извиквания на рекурсивната функция. В противен случай се получава безкрайно зацикляне и обикновено програмата приключва аварийно поради препълване на програмния стек. За повишаване на ефективността на рекурсивните функции, всички локални променливи и параметри във функцията, които се използват само преди първото рекурсивно обръщение могат да се изнесат извън функцията като глобални.
По този начин се избягва излишното им повторно създаване в новата стекова рамка при изпълнение на рекурсивното обръщение.
Ще отбележим, че за всеки итеративен алгоритъм лесно може да се построи рекурсивен алгоритъм, който решава същата задача. Обратното също е вярно, тъй като е възможно рекурсията да се моделира итеративно, но само чрез използване на достатъчно голям потребителски стек.

Типове и проверка за съответствие на тип. Типовете като параметри на функция.

C++ е строго типизиран език. По време на компилация се прави проверка за съответствието на типовете както на списъка от аргументи, така и на типа на резултата на извиканата функция. Ако бъде открито несъответствие между фактическите типове и типовете, декларирани в прототипа на функцията, ще бъде приложено неявно конвертиране, ако това е възможно. Ако не е възможно неявно конвертиране или е неправилен броя на аргументите ще се получи грешка по време на компилация. Прототипът на функцията предлага на компилатора информация за типовете, която му е необходима при проверката на типовете.

Сортиране и търсене в едномерен масив – основни алгоритми за сортировка.

Selection Sort – Сортиране чрез пряк избор/пряка селекция

На първа стъпка се намира минималния елемент на масива и се разменя с първия. На втора стъпка се намира минималния измежду останалите елементи и се разменя с втория и т.н. И така на всяка стъпка минималният елемент на несортираната част се разменя мястото с първия елемент от несортираната част.

```
int a[N];
for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){                    O(n^2)
                if(a[i] > a[j]) {swap(a[i], a[j]);}
        }
}
```

Bubble Sort -Метод на мехурчето

Сравняваме всеки два съседни елемента x_{i-1} и x_i и ако се окаже, че $x_{i-1} > x_i$ те си разменят местата (при възходящо сортиране) и накрая остава най-големия елемент и след n стъпки получаваме сортирания масив

```
int a[N];
for(int i = 0; i < n; i++){
        for(int j = 0; j + 1< n; j++){                    O(n^2) (“най-лошия алгоритъм за сортиране”)
                if(a[j] > a[j+1]) {swap(a[j], a[j+1]);}
        }
}
```

Оптимизация:
Извършваме сортирането в обратна посока (от i → n-1до 0) (т.е. започваме от „дъното“ и най-тежките елементи потъват към него – най-големите елементи отиват в десния край). Поддържа се променлива flag, указваща максималния индекс, при който е извършена размяна на предходна итерация. Елементите, които са разположени вдясно от размяната са на окончателните си места и не ги разглеждаме повече.

```
int flag; int i = 0; int j = 0;
do{
```

```
		for(j = 0, flag = 0; j < i; j++){
			if(a[j] > a[j+1]) {swap(a[j], a[j+1]); flag = j;}
			i = flag;
		}
	}while(i > 0)
```

Insertion sort – Сортиране чрез вмъкване

Последователно се сравнява всеки един елемент с x и евентуално x се разменя със стоящия вляво от него елемент. Процесът продължава до достигне на елемент, по-малък или равен на x или достигане на първия елемент на масива.
Разделяме условно масива на две последователности – готова и входна. В началото готовата последователност се състои само от първия елемент на масива, а входната – от втория до последния. За всеки елемент във входната последователност търсим подходящо място за вмъкване в готовата последователност и след това го вмъкване на това място.

```
	for(int i = 0; i < n; i++){
		int j = 0; int x = a[i];
		for(j = i - 1; j >= 0 ; j--){
			if(a[j] <= x) break;
			a[j+1] = a[j];
		}
		a[j+1] = x;
	}
```

QuickSort – Бързо сортиране на Хоор

Избираме елемент x и разделяме масива на две: лява част с елементи <= x и дясно с елементи > x. Прилагаме същия алгоритъм за лявата и дясната част до достигането на интервали, съдържащи единствен елемент: O(nlogn)

```
void qSort(int a[ ], int l, int r)
{
	if(l > r) return;
	int x = a[l], i = l, j = r;
	while(i <= j){
		while(a[i] < x) i++;
		while(a[j] > x) j--;
		if(i <= j){
			std::swap(a[i], a[j]);		// разменят се двата елемента – прегради, спрели разширяването
			i++; j--;			// и процесът се повтаря докато индексите се разминат
		}
	}
	qSort(a, l, j);
	qSort(a, i, r);
}
```

Търсене – двоично търсене (в сортирана редица :)) - O(n)

```
int BinarySearch(int a[ ], int n, int x)
{
	int l = 0, r = n-1, medium = 0;
	while(l <= r){
		int middle = (l + r)/2;
		medium = a[middle];
		if(medium > x) {r = middle -1;}
		else if(medium < x) {l = middle + 1;}
		else return middle;
	}
	return -1;
}
```