Език С. Указатели. Дефиниция. Инициализиране. Аритметика с указатели. Примери.

Едно от *най-съществените достойнства* на този език от нисоко ниво е, че той притежава и качествата на език от ниско ниво и може да замества асемблерните езици в 90% от случаите.

Елементите на езика за програмиране *С*, които го определят като достоен заместник на асемблерните езици са *указателите*, *операции с указателите*
,
операции с отделните битове
(
поразрядни
или
побитови операции
). Тези елементи на езика
С
ни позволяват да работим с адресите на променливите и да манипулираме с отделни битове на използваните числени стойности, без да е необходимо да слизаме на най-ниско машинно ниво, както е при асемблерните езици.

1. Определение и дефиниция на указателите:

Всеки обект на езика за програмиране *С – променлива*, *константа*, *масив*, *елемент на масив* и *т.н.*,
се съхранява в определени клетки от паметта на компютъра или, както е прието да се казва, на определен
адрес
.

Променлива, чиято стойност е адрес от паметта на компютъра, се нарича *указател*. Стойността на указателя посочва (
указва
) местоположението на дадена променлива и позволява косвен достъп до стойността на

тази променлива, за разлика от директния достъп с помощта на нейното име, който използвахме досега. В някои случаи косвеният достъп чрез указатели е единственият начин за реализиране на ефективна и компактна програма. В същото време така се прехвърля цялата отговорност на програмиста, който вече с „развързани" ръце, използва указатели, сочещи (
указващи
) там, където трябва. Ако програмистът в този момент сбърка, тези
указатели
могат да доведат до крах в изпълнението на програмата, а освен това дори до разрушаване на някои системни файлове на КС. Това е само предупреждение, а не заплаха, че започва боравене с професионални инструменти (средства за програмиране) и те изискват внимание и отговорност от програмиста – професионалист.

И така *указателят* е *променлива*, чиято *стойност* е *адрес* от паметта. Както всяка променлива в езика за програмиране
C
,
указателят
също трябва да се дефинира, преди да се използва. Това се извършва, като се запише тип на обекта, към който той сочи и се използва символът
[*]
преди неговия идентификатор.
Указател
се дефинира в следния формат:

тип_на_данни*име_на_указател

като *[тип_на_данни]* и *[име_на_указател]* могат да бъдат съответно всеки валиден тип данни и идентификатор.Например, за да дефинираме
указател
към променлива от тип
[
int
]
, трябва да запишем:

*int*p;*

където *[p]* е името на указателя.

Тъй като обектите, към които сочат *указателите*, могат да бъдат от различни типове, при дефинирането *указателите* също могат да бъдат от всички валидни за езика *C* типове данни:

*int*p;*

*float*p1;*

*double z, *p2;*

В този пример са дефинирани *указателите p*, *p1*, *p2*, като е показано, че дефиницията на *указателите* може да се обедини с дефинициите на други променливи, както е показано на третия ред (последен) от горния пример.

Най – важното при дефинирането на *указателите* е използваният в дефиницията тип да съответства на типа на обекта, към който те сочат (*указват)* . Ако това задължително условие не е изпълнено, то всички възможни последици от неправилното използване на *указателите* са за сметка на програмиста. Но да не забравяме и че повечето съвременни компилатори предупреждават за подобни несъответствия.

2. Инициализиране на указателите:

Указателите не са цели числа, а адреси от паметта на компютъра. След като се дефинира даден указател, той съдържа произволна стойност и трябва да бъде използван едва след като е инициализиран, т.е. след като му е присвоена *пр*
авилна стойност
–
валиден адрес
от паметта на компютъра. По определение
указателите
са
променливи
и следователно могат да бъдат инициализирани по общите правила за инициализация на променливите. Но единствените правилни стойности, които те могат да приемат, са валидни адреси от паметта или нула. Нулевата стойност, използвана при инициализиране на
указателите
, има име
[
NULL
]
, дефинирано в стандартния заглавен файл
. Целта е да се подчертае, че това е специален случай на присвояване на цяло число на
указател
. С изключение на този специален случай, присвояването на цели числа на
указателите
е грешно (нецелите числа се изключват съгласно с определението за
указател
, тъй като не могат да бъдат адреси). Инициализирането на
указател
с
[
NULL
]
се използва, за да се означи, че той е свободен и не посочва нищо. Използването му като посочващ (
указващ
) към адрес с нулева стойност обаче ще бъде катастрофално за програмата. Тази стойност на
указател
може да се използва само като маркер или флаг за край, грешка и др. Инициализирането с конкретни стойности на валидни адреси се извършва с адресната операция
[&]
.

3. Специални операции за указатели:

Имаме две специални *едноместни* операции за *указателите*, които ни позволяват да ги използваме ефективно.

1.1. *Първата* от тях е *[&] – определяне адреса на операнда*. Форматът на операцията е:
&операнд
.

Единственият операнд тук може да е само име на променлива или на елемент от масив. Неправилни конструкции са:

&(x + y/2)

&155

Неправилно е да се използва операцията *[&]* за определяне на адрес на променлива от клас *[register]*. Ето защо в списъка на аргументите на функцията *[sc anf]*, която ще се разгледа в следващата тема, пред имената на променливите се поставя *[&]*.

1.2. Другата (*втората*) специална операция е *[*] - определяне стойността* на променливата, чийто адрес е стойността на
указателя операнд
. Форматът на тази операция е:

**операнд*
.

[Операнд] може да е всеки валиден за езика за програмиране *С указател*, като програмистът отговаря стойността на този
указател
да е валиден адрес от паметта на КС. Като
[операнд]
може да се използва и израз, съставен от
указател
и валидно аритметично действие.

Горните две специални операции за указатели се изпълнява *отдясно наляво*.

Когато *указателят* участва в израз, например в израз за присвояване, той може да бъде не само *отляво*, но и *отдясно*:

```
int x, *p1, *p2;

 unsigned y;

 p1 = &x;

 p2 = p1;

 y = p1;

 printf („Адресът на x(дес) е %u”, y);  /*(дес)- (десетична стойност*)/
```

printf („Адресът на x(дес) е %u”, p1);

Съгласно определението на операцията *[*]*, това е определяне стойността на променливата, посочена с операнда – *указател*, ако *[px]* е валиден указател към променливата *x*, използването на *[* px *]* има същия смисъл, както и използването на *[x]*. Разликата е само в начина на достъп до стойността на променливата – чрез *указател* косвено, чрез името й директно. Оттук следва, че навсякъде, където е възможно използването на *[x]*, може да се използва конструкцията *[* px *]*.

Едноместните операции за *указатели - [*]*, *[&]* и едноместната аритметична операция с указатели *[- -]* са с *еднакъв приоритет* и се изпълняват отдясно наляво. Ето защо в някои случаи трябва да се поставят скоби, за да се осигури желаният ред на изпълнение.

4. Аритметика с указатели (адресна аритметика): Една от силните страни на езика за програмиране *C* са в ефективното използване на *указателите* за аритметични действия с тях. Същността на *указателите*, като особен вид променливи определя и кои от аритметичните действия са позволени при тях. При адресната аритметика са валидни само операциите събиране и изваждане. При това, има изисквания и към операндите на тези операции.

Едноместните операции [+ +] и *[- -]* - (вж. Тема 5, Точка 4), са напълно валидни за *указат елите* .
Най–съществено е, че те притежават необходимата „интелигентност” и увеличаването, респ. намаляването с
[1-ца]
, в действителност е увеличение, респ. намаление с толкова байта, колкото заема в паметта обектът, към който сочи
указателят
. С други думи, автоматично се изчислява необходимият брой байтове, които се добавят, респ. изваждат, в зависимост съответно от типа на данните и размера им в байтове:

```
int x, *px

 px = &x;

 printf (“%u %u”, px, ++px);
```

Указателят [px] е дефиниран от тип *[int]* и в зависимост от типа на КС се представя например с 2(два) байта в паметта на КС. Така всеки адрес се отличава от съседния точно с 2(две) единици. Ако *указателят [px]* е дефиниран към данни от тип
[float]
, при всяко увеличение, респ. намаление с едноместните операции
[+ +]
или
[- -]
ще се добавят или извадят 4(четири) байта – най-често за
[float]
за различните КС се използват 4 (четири) байта.

Същото е в сила и при използването на аритметичните операции *[+]*, *[-]*, както и на специалните съкратени операции
[+ =]
и

[- =]
. Най-важното, което трябва да се знае е, че само
[събиране]
, респ.
[изваждане]
на
указател
с цяло число са
валидни аритметични адресни операции
.

При добавянето или изваждането на цяло число към или от указател, всяка единица всъщност е равна на броя байтове, отделени в паметта за дадения тип на елемента от данните - *[int]* или *[float]*, към който сочи указателят.

Например, след изпълнението на *[px - = 6]*, *указателят* ще съдържа стойността на шестия адрес (на обекти от типа *[int]*), преди текущата му стойност, т.е. в действителност се извършва действието
нов адрес = стар адрес – 12
(12 = 6 пъти x 2 байта за всеки елемент данни от тип
[int]
за някои типове КС).

Ако имаме за изпълнение *[px + = 6]*, *указателят* ще съдържа стойността на шестия адрес(на обекти от типа *[int]*), след текущата му стойност, или
нов адрес = стар адрес + 12
.

Както беше отбелязано вече, операциите *[+ +]* и *[- -]* са едноместни операции и могат да се комбинират с други операции. Например, ако
[p]

е валиден указател, може да се запише:

p + + /първо се извлича стойността, сочена от *p* и след това се увеличава указателят *p* към следващия адрес
**/*

p - - /първо се извлича стойността, сочена от *p* и след това се намалява указателят *p* към предишния адрес
**/*

+ + p /първо се увеличава указателят *p* към следващия адрес и след това се извлича стойността, сочена от *p */*

- - p /първо се намалява указателят *p* към предишния адрес и след това се извлича стойността, сочена от *p */*.

Тук навсякъде се използва фактът, че едноместните операции *[*]*, *[+ +]* и *[- -]* са с еднакъв приоритет и се изпълняват отдясно наляво.

Присвояването на един *указател* на друг *указател*, ако те сочат към еднотипни обекти, е допустимо. По-точно казано, присвояването е възможно, дори и ако *указателите* сочат към обекти от различен тип, но това е свързано с проблеми и не се препоръчва за практическо използване.

Аритметичните операции: *събиране на указатели*, *събиране* или *изваждане на указатели* с числа от тип *[float]* или *[double]*; *умножение*

или
деление на указатели
, са невалидни аритметични адресни операции.

Изваждане на *указател* от друг *указател* може да се извършва, ако те сочат към променливи, между които съществува връзка – както е към различните елементи на един масив. Например, ако *p1* и *p2* са *указатели* към елементи от един и същ масив, разликата им:
[
p
1 –
p
2]
ще определи
броя
на елементите между двата елемента от масива, към които сочат
p1
и
p2
.

За такива подобни *указатели* към взаимно-свързани елементи могат да се използват *оп ерациите за отношение*
и
логическите операции
. Разрешено е да сравним два
указателя
p1
и
p2
. Отношението
[p1
има стойност
[
TRUE
]
, ако
p1
сочи към елемент от масива с по-малък индекс от индекса на
p2
. Ако това не е така, стойността на това отношение е
[

FALSE
]
.

Сега да си изясним защо е поставено ивискването за взаимна връзка (вж. по-горе „взаимно-свързани елементи”) между данните в случаите на изваждане или сравняване на указатели. Тази взаимна връзка означава, че данните са от един и същ тип, и съответните мащабни коефициенти ще бъдат еднакви. Само в този случай адресната аритметика ще води до смислени резултати. Ето защо – оттук става ясно, ука зателят
трябва да се дефинира като
указател
за определен тип данни -
[
int
]
или
[float]
. От тази дефиниция компилаторът определя необходимите му мащабни коефициенти (а те са еднакви за двата
указателя
) за правилното изпълнение на адресната аритметика.

Единственото смислено присвояване на цяло число на указател е присвояването на нулева константа [NULL]. Смисълът от такова присвояване е необходимо при означаване на особени случаи при използването на
указатели
: край на масив, грешка, т.н. Възприетото в езика за програмиране
C
договаряне тук е, че ако
указателят
има нулева стойност, той не сочи към никакви данни. Следователно сравняването на указател за равенство или неравенство с
[
NULL
]
е смислена операция. Например, валидни фрагменти от програмни оператори са:

if (p = = NULL) … ,

или

if (p ! = NULL)

Тема 9

Език С. Операции за вход и изход, “printf”, “scanf”, вход-изход на ниво байт, “inp”, “outp”. Примери за използването им.

Изпълнението на дадена програма започва с присвояване на начални стойности на променливите. Правилата и начините затова присвояване на начални стойности на различните видове данни се наричат с общото понятие инициализиране на данните.Едни от основните идеи при създаване на езика за програмиране
С
са:

 Осигуряване възможност за преносимост на програмите между различните КС

 Възможност за работа на малки изчислителни машини.

За да се постигнат тези две основни цели, езиковите конструкции са ограничени по брой, като са премахнати машиннозависимите конструкции. В езика С например не са предвидени оператори за вход и за изход, които по правило са машиннозависими. Вместо тези оператори са въведени и се използват
функции за вход
и
за изход
, които се разпространяват като стандартна библиотека, заедно с компилатора за езика.

Входно – изходните операции са едни от най-необходимите и от най-използваните. Те са
[
printf]
,
[scanf]
,
[getchar]
и
[
pu
tchar]
.

1. *Функция “printf”:*

Функцията [printf] е най-често използваната функция за показване, за извеждане на букви и цифри на екрана на КС. Форматът на функцията е:

printf („форматиращи_параметри”, списък от аргументи);

[Форматиращи параметри] – поставят се в двойни кавички и определят броя на показваните аргументи и начина на тяхното показване. Всеки от форматиращите параметри започва с [%]. Тези форматиращи параметри могат да бъдат два типа:

Низова константа,

 Низ от форматиращи параметри, които показват следващите в списъка аргументи, а те самите не се показват.

[Списък от аргументи] – това са променливите (или изразите), чиито стойности искаме да бъдат показани. Отделните аргументи се отделят със запетаи. Може да се направи най-обща аналогия между [printf] и операторите [write] и [format] в

един друг език за програмиране от високо ниво –
Fortran
, или операторът
[
print
using]
на езика за програмиране от високо ниво
Basic
.

Особеност на тази функция е, че след изпълнението й не се осигурява преминаването на нов ред, ако това не се укаже явно с [“n”].

Пример:

printf („направете вашия избор:n”);

Символната информация не изписва форматиращи параметри. Тези параметри са необходими основно при показването на числова информация.

Следващият пример е за форматиращи параметри, чийто брой е равен на броя на аргументите в списъка от аргументи:

printf (“ % 4d□□□ % 6,2fn ”,□□□ x,□□□ (x + y));

Това е пример за две числови стойности [x] и сумата [x + y]. Числото [x] ще се отпечата като десетично цяло число, съставено от четири цифри – символът
[
d
]
в първия форматиращ параметър показва, че първият аргумент -
[

x
]
ще се покаже като цяло десетично число. Резултатът от сумата
[
x
+
y
]
ще се отпечата (изобрази) като дробно число с десетична точка, съставено от 6(шест) цифри, 2(две) от които 6(шест) са след десетичната точка. Символът
[
f
]
определя показването на втория аргумент като дробно десетично число.

Възможно е смесване на двата типа форматиращи параметри – на низовите константи, които се показват без изменение и на преобразуващите спесификации. Това се изпълнява независимо от реда на смесване.

Спесификациите се означават с латинските букви: d, u, o, x, X, f, e, E, g, G, c, s. Отделните спесификации могат да се променят с т.наречените модификатори по схемата:

% модификатори спесификация.

Обикновено със записа [% f] броят на цифрите след десетичната точка на мантисата се подразбира равен на 6, но може да бъде променян.

Може да се използват някои общи модифициращи спесификационни параметри, поставени между символа за формат - [%] и съответната спесификация. Те са изброени последователно по-долу в реда, по който трябва да се задават:

1. По подразбиране всички числа се извеждат дясно изравнени, т.е. ако числото е с

по-малка разрядност от тази на отделеното му поле, то се изравнява отдясно, а излишните позиции вляво се запълват с интервали. Може да се зададе ляво изравняване, ако след [%] се постави символът [-]. Пр това ляво изравняване излишните позиции в полето за показване се оставят отдясно на числото и също се запълват с интервали;

2. Някои компилатори позволяват знакът [+], който обикновено се изпуска за положителните числа, да бъде показан явно. За целта се поставя
[+]
между символа
[%]
и съответната спесификация. Ако вместо знак
[+]
се постави
[интервал]
, числото се показва (изобразява) с един интервал пред него. По този начин могат да се подравняват числа със знак и без знак;

3. При някои компилатори чрез символа [#] е възможно да се извежда [0] пред осмично число или
[0x]
, или
[0X]
пред шестнадесетично число. Същият символ
[#]
, използван при спесификациите:
[%
f
]
,
[%
e
]
,
[%
E
]
,
[%
g
]
,

[%
G
]
, задава задължително показване на десетичната точка и на нули до зададената дължина на полето в края на числото.

Спесификациите, описани в горните т.т.2 и 3, както беше отбелязано за всяка от тях, не се поддържат от всички компилатори.

4. Минималната дължина на полето за показване на съответния аргумент може да бъде променена с помощта на цяло число, поставено между [%] и спесификацията. Ако аргументът изисква повече позиции, те се добавят автоматично. Ако този аргумент в дадения случай е по-малък от зададената дължина на полето, излишните позиции в ляво (в дясно при ляво подреждане) се запълват с интервали. Тези интервали могат да бъдат запълнени с нули. За целта се записва нула преди зададената минимална дължина на полето. При някои компилатори се допуска минималната дължина на полето да се задава с променлива, записана непосредствено пред аргумента, за който се отнася и отделена със запетая, като на мястото на дължина на полето се записва символът
[*]
.

Пример:

```
int i;

 printf (“ %4d |”, 25);

 i = 4;

 printf (“ | %*d |”, I, 25);
```

5.□□ Желаната точност на показване може да се зададе с десетична точка и цяло число. Те определят:

-□□□□□□□□ Броя на цифрите след десетичната точка при показването на десетични дробни числа и на експоненциални числа;

-□□□□□□□□ Броя на символите при показване на низ.

В първия случай, подразбиращият се брой цифри, ако не се зададе тази модификация, е равен на 6(шест). Във втория случай случая -на низ, ако низът е по-малък от зададената точност, оставащите позиции се запълват с интервали.

6.□□ Аргументи от тип [long int] се показват, като се зададе модификатор [l] пред съответната спесификация –
d
,
u
,
o
,
x
,
X
.

Както се убедихме, с помощта на функцията [printf] може да се реализира почти всеки възможен формат за показване на информацията. Но тук следва да отбележим, те тази универсалност се заплаща с увеличаване големината на кода на програмата – един входен файл (с разширение .C), съдържащ
[
printf
]
е с по-голям обем от този на съответния изпълним файл (с разширение

.EXE
). Затова, при критична големина на изпълнимия файл се препоръчват не толкова универсални функции за извеждане (
putchar
и др. – разглеждани в Тема 12).