Упражнение №3

СТАНДАРТНИ ТИПОВЕ ДАННИ. ОПЕРАЦИИ

1. Стандартни типове данни

Езикът С включва стандартен набор от базови типове данни, към които се отнасят цял тип и реален тип, съответно за представяне на цели числа и на числа, които могат да имат дробна част. Целите типове данни обхващат: **char**, **short**, **int** и **long**. Типове данни, позволяващи представяне на числа с дробна част, включват: **float** и **double**.

В програма на С типът данни определя количеството на заделената памет (в байтове), начина на представяне на числата, допустимия диапазон на числата и символичните начални адреси (идентификатори на променливи), например:

```
char sym;     // 1 байт памет, цели числа от -128 до 127
short a;      // 2 байта памет, цели числа от -32,768 до 32,767
int b;        // 1 дума (= 4 байта - зависи от системата)
long c;       // 4 байта, цели числа от -2,147,483,648 до 2,147,483,647
float d;      // 1 дума (= 4 байта), реални числа 3.4E +/- 38 (7 цифри)
double e;     // 2 думи (= 8 байта), реални числа 1.7E +/- 308 (15 цифри)
```

Целите типове данни могат да бъдат допълнително квалифицирани като: **signed** и **unsigned**. (За всички стандартни цели типове **signed** се приема по подразбиране.) Добавянето на **unsigned** пред типа води до промяна в диапазона на целите числа, които могат да бъдат представени (само положителни цели числа), например:

```
unsigned char;                  //диапазон от 0 до 255
unsigned short;                 // диапазон от 0 до 65,535
unsigned int;                   // диапазонът зависи от системата
unsigned long;                  // диапазон от 0 до 4,294,967,295
```

Не е необходимо броят на заделяните за всеки от стандартните типове данни байтове да се помни наизуст, тъй като информацията може да се получи от средата за програмиране чрез използване на **sizeof(<стандартен_тип>)**.

Пример 1**.** Да се състави програма, която определя броя заделени байтове за всеки от разгледаните стандартни типове (чрез **sizeof()**) и за всеки тип извежда този брой на екрана, предшестван от названието на типа.

```
#include <stdio.h>
int main()
{
  printf("char=%d, short=%d\n", sizeof(char),sizeof(short));
  printf("int=%d, long=%d\n", sizeof(int),sizeof(long));
  printf("float=%d, double=%d\n", sizeof(float), sizeof(double));
  return 0;
}
```

Въвеждането и извеждането на данни от определен стандартен тип изисква използването на форматни спецификации, които за най-популярните стандартни типове са:

Стандартен тип данни	Форматна спецификация
char	**%c** (интерпретира цялото число като код на ASCII символ и извежда символа, а не числото)
int	**%d**
long	**%ld**
float	**%f**
double	**%lf**

Пример 2. Да се състави програма, която заделя памет за данни от тип: **char**, **int**, **long**, **float**, **double**, инициализира по време на компилация съответните променливи съответно в: a='a', b=102, c=50000, d=6, e=-48.907. Програмата да извежда на екрана съдържанието на заделената за данни памет, форматирано в съответствие с типа на отделните елементи данни.

```
#include <stdio.h>
int main()
{
   char a = 'a';
   int b = 102;
   long c = 50000;
   float d = 6;//ще бъде добавена нулева дробна част
   double e = -48.907;
   printf("%c %d %ld %f %lf\n",sym,b,c,d,e);
   return 0;
}
```

Тъй като за всеки от стандартните типове данни се заделя точно определен брой байтове, което определя и допустимия диапазон на представяните числа, то при опит за записване на число извън диапазона се получава препълване. (Препълване може да се получи и в резултат на аритметични операции). Реакцията на програмата може да бъде различна, взависимост от използваната среда за програмиране.

Пример 3: На променливата **b** от тип **unsigned char** се присвоява стойност, която очевидно надвишава максималното допустимо цяло число за този тип. Реакцията на средата за програмиране се свежда до извеждане на предупреждение (warning). Програмата обаче се изпълнява, като резултатът, т.е. съдържанието на променливата **b** видимо е грешно.

Програмен код:
```
unsigned char b = 10000;
printf("overflow => %d\n", b);
```

Резултат:
```
Overflow => 16
```

2. Операции

2.1. Аритметични операции

Аритметичните операции се използват при съставянето на аритметични изрази, заедно с операнди от числов тип (променливи, константи, подизрази). Могат да бъдат еднооперандни или двуоперандни. Имат приоритет и се изпълняват в указаната последователност. Умножението и делението са с по-висок приоритет от събирането и изваждането. Ако в един аритметичен израз участват повече от една двуоперандна аритметична операция, те ще се изпълнят отляво надясно в съответствие с указания приоритет; ако трябва да се промени редът на изчисление в израза, трябва да се използват кръгли скоби за ограждане на подизразите, които да се изчислят с преди останалите части от израза.

Операция	Описание
+	Сумиране
-	Изваждане
*****	Умножение
/	Деление (ако и числителят и знаменателят са от цял тип, остатъкът от делението се губи – целочислено деление)
%	Деление по модул (може да се използва само с данни от цял тип и резултатът е само остатъкът от делението; цялата част се губи)
++	Увеличаване с 1 (унарна операция)
--	Намаляване с 1 (унарна операция)
	++x (**--x**) – префиксна форма – резултатът е новата стойност на променливата, т.е. стойността на **x**, увеличена (намалена) с 1. **x++** (**x--**) – постфиксна форма – резултатът е старата стойност на променливата **x**, а като вторичен ефект се увеличава (намалява) стойността на променливата **x** с 1.

При изчислението на аритметични изрази често се налага преобразуване на типа на операндите. Преди да се изпълни аритметична операция, типът на левия и десния операнд трябва да се уеднаквят. Правилото за преобразуване е от типа данни, за който се заделени по-малко байтове към типа данни, за който се заделени повече байтове (т. нар. „разширително“ преобразуване на типа, изпълнявано автоматично от средата за програмиране). Има и „стесняващо“ преобразуване на типа, което ако бъде изпълнено по необходимост от средата за програмиране, води до извеждане на предупреждаващо съобщение за загуба на точност, поради „орязване” на разряди. „Стесняващо“ преобразуване може да бъде изпълнено явно, като се посочи съответният тип в кръгли скоби непосредствено пред операнд, израз или подизраз.

Пример 4. Какво ще изведе на екрана следващата програма на C? Защо стойността на **pi**, изчислен с израз 1 и 2 е неточен, а с изрази 3 и 4 е верен? Каква е разликата между изрази 3 и 4, които осигуряват еднакво точен резултат? Защо резултатът от изрази 5 и 6 е различен, при условие, че участват едни и същи операнди и операции?

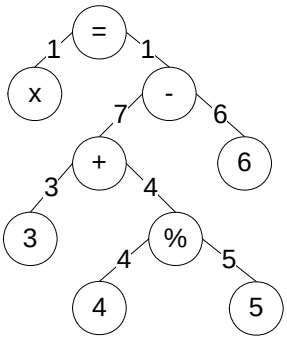
```
#include <stdio.h>
int main()
{
   int a = 22, b = 7;
   float c = 22, d = 7;
   printf("1: pi = %d\n", a/b);
   printf("2: pi = %f\n", (float)(a/b));
   printf("3: pi = %f\n", c/d);
   printf("4: pi = %f\n", ((float)a)/b);
   printf("5: %f\n",a+b/c+d);
   printf("6: %f\n", (a+b)/(c+d));
   printf("7: %d\n", ++a + b++);
   printf("8: a = %d, b = %d\n", a, b);
   return 0;
}
```

Пример 5: Изчислете аритметичните изрази и начертайте дървото на изпълнение на операциите.

а) `int x;`
`x = 3 + 4 % 5 - 6;`

Решение:

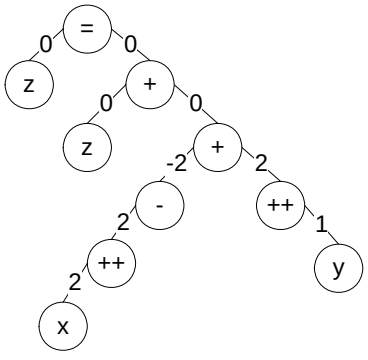
```
x = 3 + (4 % 5) - 6
x = (3 + (4 % 5)) - 6
x = ((3 + (4 % 5)) - 6)
x = ((3 + 4) - 6)
x = (7 - 6)
x = 1
```



б) `int x = 2, y = 1, z = 0;`
`z += - x ++ + ++ y;`

Решение:

```
z += - (x++) + (++y)
z += (- (x++)) + (++y)
z += ((-(x++)) + (++y))
z += ((-2) + 2)), x = 3, y = 2
z += 0
z = (0 + 0)
z = 0
```



в)
```
int x;
x = (7 + 6) % 5 / 2;
```

г)
```
x = 1; y = 2; z = 3;
z -= x-- - ++ y;
```

2.2. Операции за отношение

Операциите за отношение се използват за представяне на прости условия. Операциите са двуоперандни, а резултатът е 0 или 1. Аналогично на аритметичните операции се изпълняват отляво надясно, като операциите **>** (по-голямо), **>=** (по-голяма или равно), **<** (по-малко) и **<=** (по-малко или равно) са с по-висок приоритет от операциите **==** (равно) и **!=** (различно).

Пример 6. Какво ще изведе на екрана следващата програма на С, изчисляваща изрази за отношение? Защо резултатът от изрази 2 и 3 е противоположен? Защо резултатът от изрази 5 и 6 е противоположен, след като участват едни и същи операнди и операции за отношение?

```
#include <stdio.h>
int main()
{
   int a = 5;
   int b = 6;
   printf("1: %d\n", a > b);
   printf("2: %d\n", ++a == b);
   a = 5;
   printf("3: %d\n", a++ == b);
   a = 5;
   printf("4: %d\n", a+2 == b+1);
   printf("5: %d\n", a < b != 6);
   printf("6: %d\n", a < (b != 6));
   return 0;
}
```

2.3. Логически операции

Езикът С поддържа една едноопередна **!** и две двуоперандни логически операции **&&** и **||**.

Операция	Описание
!	Логическо отрицание; има стойност 1, ако единственият операнд има стойност 0 и обратното; изпълнява се отдясно наляво
&&	Логическа операция И; има стойност 1, само ако и двата операнда са различни от 0 и стойност 0 във всички останали случаи; изпълнява се отляво надясно
||	Логическа операция ИЛИ; има стойност 0, само ако и двата операнда имат стойност 0; във всички останали случаи стойността е 1; изпълнява се отлява надясно

Операцията **!** е с по-висок приоритет от логическите операции **&&** и **||**, а от своя страна операцията **&&** е с по-висок приоритет от операцията **||**.

Характерно за двуоперандните логическите операции е, че редът за изчисление на операндите е строго определен (за сравнение, при двуоперандните аритметични операции редът на изчисление на операндите не е определен – изборът е на компилатора). За операция **&&** първо се изчислява левият операнд. Ако той има стойност 0, десният операнд не се изчислява и резултатът е равен на 0. Ако левият операнд е различен от 0, то се изчислява и десният операнд; ако десният операнд има стойност, различна от 0, резултатът е равен на 1; ако десният операнд има стойност 0, резултатът е 0.

За операцията **||** първо се изчислява левият операнд и ако той има стойност, различна от 0, то десният операнд не се изчислява и резултатът е равен на 1. Ако левият операнд има стойност 0, то се изчислява и десният операнд; ако десният операнд има стойност, различна от 0, резултатът е 1; ако десният операнд има стойност 0, резултатът е 0.

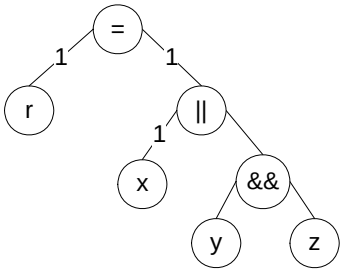
Пример 7. Какво ще изведе на екрана всеки оператор **printf()** от следващата програма на С, изчисляваща логически изрази? Защо операцията **a++** не се изпълнява като част от логически израз 3? Защо същата операция **a++** се изпълнява като част от логически израз 5?

```
#include <stdio.h>
int main()
{
  int a, b;
  a = b = 1; //операция '=' е с дясна асоциативност
  printf("1: %d\n", a+b<b+6||0);
  printf("2: %d\n", a+b<b+6&&0);
  printf("3: %d\n", a&&b||a++);//операцията a++ не се изпълнява
  printf("4: a = %d\n", a);
  printf("5: %d\n", --a==b||a++);//операцията a++ се изпълнява
  printf("6: a = %d\n", a);
  return 0;
}
```

Изразите могат да се представят с дървовидна структура, в която всяка операция (аритметична, за отношение, логическа) се представя с възел, чиито поддървета са операндите. За корен на дървото се избира операцията присвояване '=', която е с най-нисък приоритет. По-ниско приоритетните операции са по-близко до корена на дървото.

Пример 8. Да се представят с дървовидна структура следващите изрази:

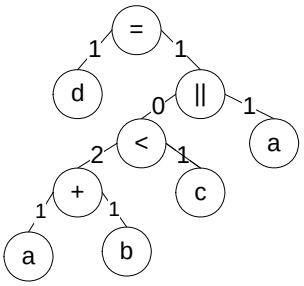
а) `int x = 1, y = 0, z = 0, r;`
`r = x || y && z;`



Решение:

```
r = x || (y && z)
r = (x || (y && z))
r = (1 || (y && z))
r = 1
```

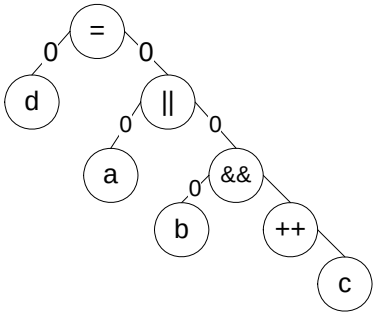
б) `int a=b=c=1, d;`
`d = a+b < c || a;`



Решение:

```
d=(((a+b)<c)||a) = 1;
a = 1;
b = 1;
c = 1;
```

в) `int a=b=c=0, d;`
`d = a|| b && ++c;`



Решение:

```
d=(a||(b&&(++c))) = 0;
a = 0;
b = 0;
c = 0;
```

г) `int x = 1, y = 0, z = 0, r;`
`r = x && y || z;`

д) `int y = 2009, d;`
`d = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;`

Приоритет на операциите

Операция	**Ред на изпълнение**
`() [] . ->`	отляво надясно
`! ~ + - ++ -- & * (тип) sizeof`	отдясно наляво
`* / %`	отляво надясно
`+ -`	отляво надясно
`<< >>`	отляво надясно
`< <= > >=`	отляво надясно
`== !=`	отляво надясно
`&`	отляво надясно
`^`	отляво надясно
`|`	отляво надясно
`&&`	отляво надясно
`||`	отляво надясно
`?:`	отляво надясно
`= += -= *= /= %= >>= <<= &= ^= !=`	отдясно наляво
`,`	отляво надясно