Цілі типи

В Сі та Сі++ існує декілька цілих типів, що відрізняються розміром та типом арифметики (знакова або беззнакова):

- short int (також можна позначити short, можна використати ключове слово signed)
- unsigned short int (також можна позначити unsigned short)
- int (також можна позначити signed int). Це найбільш оптимальний цілий тип для даної платформи, і він гарантовано має розмір не менше 16 bits. Більшість сучасних моделей використовуй 32 біти (див. таблицю 2.1).
- unsigned int (також можна позначити unsigned), натуральний або беззнаковий еквівалент int, що використовуй модульну арифметику при арифметичних операціях. Зручний для маніпулювання бітами.
- long int (також можна позначити long)
- unsigned long int (також можна позначити unsigned long)
- long long int (також можна позначити long long)
- unsigned long long int (також можна позначити unsigned long long) (з C99)

Відмітимо що порядок слів в опису може бути довільним: unsigned long long int та long int unsigned long позначають той самий тип.

Наступна таблиця містить всі цілі типи :

Цілі типи Сі

Таблиця 2.1

Специфікатор типу	**Еквівалент**	**Розмір типу**				
		C standard	**LP32**	**ILP32**	**LLP64**	**LP64**
short short int signed short signed short int	short int	мінімум **16**	**16**	**16**	**16**	**16**
unsigned short unsigned short int	unsigned short int					
int signed signed int	int	мінімум **16**	**16**	**32**	**32**	**32**
unsigned unsigned int	unsigned int					
long long int signed long signed long int	long int	мінімум **32**	**32**	**32**	**32**	**64**
unsigned long unsigned long int	unsigned long int					
long long	long long int	мінімум **64**	**64**	**64**	**64**	**64**

long long int	(C99)
signed long long	
signed long long int	
unsigned long long	unsigned long long int
unsigned long long int	(C99)

Стандарт Сі не гарантує розміри бітів, єдине що він гарантує, це те що

1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long).

Цілочисельна арифметика знакових типів та беззнакових типів відрізняється зокрема в тих випадках, коли розмір результату більше ніж розмір вхідних даних, отже в випадках, коли це можливо треба бути особливо уважним

Також вкрай небажано застосовувати цілий та натуральний операнди одночасно в арифметичних виразах.

Моделі даних

Вибір розмірів типів Сі визначається моделлю даних(*data model), що в свою чергу визначається архітектурою комп'ютера та його операційною системою*. Найбільш популярні наступні моделі даних:

32 бітові системи:

LP32 або **2/4/4** (int - 16-bit, long та вказівник 32-bit)

Win16 API

ILP32 або **4/4/4** (int, long, та вказівник 32-bit);

Win32 API

Unix та Unix-like systems (Linux, Mac OS X)

64 бітові системи:

LLP64 або **4/4/8** (int та long - 32-bit, вказівник 64-bit)

Win64 API

LP64 або **4/8/8** (int - 32-bit, long та вказівник 64-bit)

Unix та Unix-like systems (Linux, Mac OS X)

Цілі типи фіксованої довжини

Інколи виникай потребі зафіксувати розмір типів для будь-якої платформи. Для цього можна або перевизначити самому типи за допомогою typedef або скористатись бібліотекою <stdint.h> , яка включена в стандарт з С99.

Цілі типи визначені в <stdint.h>

Таблиця 2.2

int8_t **int16_t** **int32_t** **int64_t**	Знаковий цілий тип з розміром 8, 16, 32 і 64 біт відповідно
int_fast8_t **int_fast16_t** **int_fast32_t** **int_fast64_t**	Найбільший знаковий цілий тип з розміром мінімум 8, 16, 32 та 64 біт відповідно
int_least8_t	Найменший знаковий цілий тип з розміром 8, 16, 32 та 64 біт відповідно

int_least16_t **int_least32_t** **int_least64_t**	
intmax_t	Максимальний за розміром цілий тип
intptr_t	Цілий тип в який можна записати вказівник
uint8_t **uint16_t** **uint32_t** **uint64_t**	Натуральний тип з розміром 8, 16, 32 і 64 біт відповідно (лише якщо платформа їх підтримує)
uint_fast8_t **uint_fast16_t** **uint_fast32_t** **uint_fast64_t**	Найбільший натуральний тип з розміром 8, 16, 32 та 64 біт відповідно
uint_least8_t **uint_least16_t** **uint_least32_t** **uint_least64_t**	Найменший натуральний тип з розмірами 8, 16, 32 та 64 біт відповідно
uintmax_t	Максимальний за довжиною натуральний тип
uintptr_t	Натуральний тип тип в який можна записати вказівник

Цілочисельні константи

Для того щоб визначити константи цілого типу можна обрати наступні варіанти:

- звичайний ***десятковий*** тип:ненульова десяткова цифра (1, 2, 3, 4, 5, 6, 7, 8, 9), за якою слідуй довільна кількість десяткових цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- ***вісімкова*** константа: цифра (0) за якою довільна кількість вісімкових цифр (0, 1, 2, 3, 4, 5, 6, 7)
- ***шістнадцятирічна*** константа: двійний спецсимвол 0x або 0X за яким слідуй довільна кількість шістнадцяткових цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F).

Наприклад, наступні змінні ініціалізовані тим самим значенням:

```
int d = 42;
int o = 052;
int x = 0x2a;
int X = 0X2A;
```

Суфікси для форматованого введення/виведення цілих типів

Для того, щоб позначити програмний тип можуть використовуватися суфікси (вони можуть бути в будь-якому порядку):

- натуральний суфікс (unsigned-suffix) (символ u або символ U)
- суфікс довжини (long-suffix) (символ l або символ L) або long-long-suffix (the суфікс ll або LL) (since C99)

Специфікатори для цілочисельних констант

Таблиця 2.3

суфікс	Основа числення 10	Основи числення 8 та 16
no suffix	int long int unsigned long int (until C99) long long int (since C99)	int unsigned int long int unsigned long int long long int(since C99) unsigned long long int(since C99)
u або U	unsigned int unsigned long int unsigned long long int(since C99)	unsigned int unsigned long int unsigned long long int(since C99)
l або L	long int unsigned long int(until C99) long long int(since C99)	long int unsigned long int long long int(since C99) unsigned long long int(since C99)
l/L та u/U	unsigned long int unsigned long long int(since C99)	unsigned long int unsigned long long int(since C99)
ll або LL	long long int(since C99)	long long int(since C99) unsigned long long int(since C99)
ll/LL та u/U	unsigned long long int(since C99)	unsigned long long int(since C99)

Літери в числових константах case-insensitive: 0xDeAdBaBeU та 0XdeadBABEu те саме число (виключення long-long-suffix, який або ll або LL, але ні lL ні Ll)

Відмітимо, що в Сі немай від'ємних констант. Вирази такі як -1 визначають унарний оператор (unary minus operator) що застосовується до константи, та можливо період до іншого типу (type conversions).

Константи, що закінчується e або E, якщо за ними й знаки + або – повинні бути відокремлені пробілами (щоб не переплутати з науковим представленням дійсного типу):

```
int x = 0xE+2;   // error
int y = 0xa+2;   // OK
int z = 0xE +2;  // OK
int q = (0xE)+2; // OK
```

Приклад:

```
#include <stdio.h>
#include <inttypes.h>
int main(void)
{
    printf("123 = %d\n", 123);
    printf("0123 = %d\n", 0123);
    printf("0x123 = %d\n", 0x123);
    printf("123456789012345678901ull = %llu\n",
123456789012345678901ull);
    // the type is a 64-bit type (unsigned long long or possibly
unsigned long)
    // even without a long suffix
```

```
    printf("12345678901234567890u = %"PRIu64"\n",
12345678901234567890u );

//  printf("%lld\n", -92233720368547758O8); // ERROR
// the value 9223372036854775808 cannot fit in signed long long, which
is the
// biggest type allowed for unsuffixed decimal integer constant
    printf("%llu\n", -9223372036854775808ull );
    // unary minus applied to unsigned value subtracts it from 2^64,
    // this gives unsigned 9223372036854775808

    printf("%lld\n", -9223372036854775807ull - 1);
    // correct way to form signed value -9223372036854775808
}
```

Результат:

```
123 = 123
0123 = 83
0x123 = 291
12345678901234567890ull = 12345678901234567890
12345678901234567890u = 12345678901234567890
9223372036854775808
-9223372036854775808
```

Символьний тип

Ще одним варіантом цілого типу й символьні типи:

- signed char – знакове представлення символьного типу.
- unsigned char – беззнакове представлення знакового типу. Може використовуватись для object representations (raw memory).
- char – тип для представлення символів. Є еквівалентним або signed char або unsigned char (що залежить від реалізації та може бути визначено через командний рядок), але це окремий тип, що відрізняється і від signed char та від unsigned char.

Також в сучасному Сі за допомого typedef визначають типи wchar_t , char16_t, та char32_t (since C11) для представлення символів за допомогою юнікоду.

Макроси бібліотеки limits.h

Бібліотека **limits.h** визначай різні властивості різних типів Сі. Макроси визначені в цій бібліотеці обмежують значення визначені в char, int та long.

Ці межи зокрема визначають кількість та область різних значень, що може приймати даний тип, наприклад unsigned char може приймати від 0 до 255.

Дані значення можуть прийматись різними та визначені директивою #define , але не можуть бути нижче визначених.

Таблиця 2.4

Макрос	**Значення**	**Опис**
CHAR_BIT	8	Кількість бітів в байті.
SCHAR_MIN	-128	Мінімальне значення signed char.
SCHAR_MAX	+127	Максимальне значення signed char.
UCHAR_MAX	255	Максимальне значення unsigned char.

CHAR_MIN	-128	Мінімальне значення типу char і його значення буде рівне SCHAR_MIN якщо char буде приймати від'ємні значення, інакше воно рівне 0.
CHAR_MAX	+127	Максимальне значення типу char і його значення буде рівне SCHAR_MAX якщо char буде приймати від'ємні значення, інакше воно рівне UCHAR_MAX.
MB_LEN_MAX	16	Максимальна кількість бітів в двубайтовому типі
SHRT_MIN	-32768	Мінімальне значення short int.
SHRT_MAX	+32767	Максимальне значення short int.
USHRT_MAX	65535	Максимальне значення unsigned short int.
INT_MIN	-2147483648	Мінімальне значення int.
INT_MAX	+2147483647	Максимальне значення int.
UINT_MAX	4294967295	Максимальне значення unsigned int.
LONG_MIN	-9223372036854775808	Мінімальне значення long int.
LONG_MAX	+9223372036854775807	Максимальне значення long int.
ULONG_MAX	18446744073709551615	Максимальне значення unsigned long int.

Приклад

Приклад застосування констант з **limits.h**.

```
#include <stdio.h>
#include <limits.h>

int main() {
   printf("The number of bits in a byte %d\n", CHAR_BIT);

   printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
   printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);

   printf("The minimum value of CHAR = %d\n", CHAR_MIN);
   printf("The maximum value of CHAR = %d\n", CHAR_MAX);

   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);

   return(0);
}
```

Результат повинен бути:

```
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -2147483648
```

```
The maximum value of INT = 2147483647
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -9223372036854775808
The maximum value of LONG = 9223372036854775807
```

Переповнення цілих чисел

При додаванні цілих чисел може виходити переповнення. Проблема полягає в тому, що комп'ютер не видає попередження при їх появі: програма продовжить виконання з невірними даними. Більше того, поведінка при переповненні є визначеною стандартом та фіксованою лише для цілих без знаку (натуральних).

Переповнення може привести до великих проблем: обнулінню та затиранню даних, можливим експлойтам, помилкам, що буде трудно воспроізвести та знайти при відладці, та ці помилки можуть накопичуватися з часом. Розглянемо деякі прийоми боротьби з переповненнями для цілих чисел зі знаком та натуральних чисел.

1. Попередня перевірка даних. З файлу limits.h можна узнати максимальне і мінімальне значення для чисел типу int. Якщо обидва числа додатні, то їх сума не більша за INT_MAX, якщо різниця INT_MAX і одного з чисел менше другого числа. Якщо обидва числа від'ємні, то різниця INT_MIN і одного з чисел повинна бути більше іншого. Якщо обидва числа мають різні знаки, то їх сума не більша INT_MAX або INT_MIN.

```
int sum1(int a, int b, int *overflow) {
int c = 0;
if (a > 0 && b > 0 && (INT_MAX - b < a) || a < 0 && b < 0 && (INT_MIN
- b > a)){
    *overflow = 1;
}
else{
    *overflow = 0;
    c = a + b;
}
return c;
}
```

В цій функції змінній overflow буде присвоєно значення 1, якщо було переповнення. Функція повертає суму, незалежно від результату додавання.

2. Другий спосіб перевірки – взяти для суми тип, максимальне й мінімальне значення якого відомо що більше суми двох цілих. Після додавання необхідно перевірити, щоб сума була не більшою ніж INT_MAX і не меншою INT_MIN.

```
int sum2(int a, int b, int *overflow) {
signed long long c = (signed long long) a + (signed long long) b;
if (c < INT_MAX && c > INT_MIN) {
*overflow = 0;
c = a + b;
}
else{
```

```
    *overflow = 1;
}
return (int) c;
}
```

Зверніть увагу на явне приведення типу. Без нього спочатку пройде переповнення і неправильне число буде записано в змінну c.

3. Третій шлях перевірки платформозалежний, більш того, його реалізація буде різною для різних компіляторів. При переповненні цілих (зазвичай) ставиться флаг переповнення в регістрі флагів. Можна на асемблері перевірити значення флагу відразу ж після виконання додавання.

Робота з натуральними числами без знаку значно простіша: при переповненні виходить обнуління і відомо, що отримане число буде менше кожного з доданків.

```
unsigned usumm(unsigned a, unsigned b, int *overflow) {
    unsigned c = a + b;
if (c < a || c < b) {
    *overflow = 1;
}
else{
    *overflow = 0;
}
return c;
}
```

Операція sizeof()

Дана операція обчислюй розмір пам'яті, необхідний для розміщення в ній виразів або змінних вказаних типів.

Операція має дві форми:

1). **ім'я_типу A; sizeof (A);**

Приклад:

```
unsigned long x;
unsigned long  y = sizeof(x);
```

2).**sizeof (ім'я_типу);**

Приклад:

```
unsigned long x;
unsigned long  y = sizeof( unsigned long );
```

Операцію sizeof() можна застосовувати до констант, типів або змінних, у результаті чого буде отримано число байт, що відводяться під операнд. Приміром, sizeof(int) поверне число байт для розміщення змінної типу int.

Операції над цілими числами

Над цілими числами можна виконувати майже всі дії, що перелічені в таблицях 1.6 -1.7:

	>>>
!, ~, +, - (унарні), ++, --, *, (тип), sizeof	<<<
*, /, % (бінарні)	>>>
+, - (бінарні)	>>>
<<, >>	>>>
<, <=, =>, >	>>>
==, !=	>>>
& (порозрядна)	>>>
^	>>>
| (порозрядна)	>>>
&& (логічна)	>>>
|| (логічна)	>>>
?: (тернарна)	<<<
=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=,>>=	<<<

Примітка. Відмітимо важливий момент, пов’язаний з операцією ділення. Якщо обидва операнди є цілими числами, то результат ділення також цілий, тобто операція "/" позначай цілочисельне ділення. Відповідно % - це остача від цілочисельного ділення. Для того, щоб отримати дійсне число як результат ділення цілих чисел потрібно якимось чином привести один з операндів до дійсного числа. Наприклад

```
int x,y;
scanf("%d %d",&x, &y);

float z = (float)ч/y;
printf("z=%f",z);
z = (x+0.0)/y;
printf("z=%f",z);
z = x/(float)y;
printf("z=%f",z);
```

Булевий тип

Булевий тип з'явився в мові Сі з стандарту С99. Для його підключення потрібна бібліотека <stdbool.h>. Він дозволяй зберігати змінні 2 значень 0 або 1, які доступні також за допомогою макросів (літералів) true та false.
Ініціалізація або присвоєння може виглядати так:

```
bool x1 = true;   // bool x1 =1;
bool y1 = false;   // bool y1 =0;
```

Відмітимо також, що перетворення до цього типу робиться за тим правилом, що будь який ненульовий вираз перетворюється на 1(true), і лише вираз рівний нулю перетворюється на 0(false), зокрема:
bool(0.5) дорівнює 1
bool(4) дорівнює 1
bool(0) дорівнює 0.

Операції порівняння

Ще одним варіантом присвоєння значення булевому виразу й присвоєнням за допомогою порівняння двох змінних або констант.

Перелік операцій порівняння наведено в таблиці 2.5.

Таблиця 2.5

Операція	**Значення**
<	менше
<=	менше або рівне
==	перевірка на рівність
>=	більше або рівно
>	більше
!=	перевірка на нерівність

Операції порівняння здебільшого використовуються в умовних виразах.
Приклади умовних виразів:

```
b<0,
'b'=='B',
'f'!='F',
 201>=205.
```

Кожна умова перевіряється: істинна вона чи хибна. Точніше слід сказати, що кожна умова приймає значення "істинно" (true) або "хибно" (false).
Результатом умовного виразу й цілочисельне арифметичне значення. "Істинно" - це ненульова величина, а "хибно" - це нуль. В більшості випадків в якості ненульового значення "істинно" використовується одиниця. Тобто 1- true, 0 – false.

Приклад:

```
#include<stdio.h>
main()
{
   int tr, fal;
   tr=(11<=15); /* вираз істинний */
   fal=(1>1); /* вираз хибний */
   printf("true - %d false - %d ",tr,fal);
   return 0;
}
```

Логічні операції

Логічні операції &&, ||, ! використовуються здебільшого для "об'єднання" виразів порівняння у відповідності з правилами логічного І, логічного АБО та логічного заперечення (таблиця 2.6.).

Логічні операції

Таблиця 2.6

Операція	**Значення**
&&	логічне І (and)
||	логічне АБО (or)
!	логічне заперечення (not)

Зауваження. Складні логічні вирази обчислюються "раціональним способом" (lazy evaluation). Наприклад, якщо у виразі (A<=B)&&(B<=C) виявилось, що А більше В, то всі вирази, як і його перша частина (A<=B), приймають значення "хибно", тому друга частина (B<=C) не обчислюється.
Таблиця істинності логічних операцій наведена в таблиці 2.7.

Таблиця 2.7

E1	**E2**	**E1&&E2**	**E1||E2**	**!E1**
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Порозрядні операції (побітові операції)

Порозрядні операції застосовуються тільки до цілочисельних операндів і "працюють" з їх двійковими представленнями. Ці операції неможливо використовувати із змінними типу double, float, long double. Перелік порозрядних операцій наведено в таблиці 2.8.

Порозрядні операції

Таблиця 2.8

Операція	**Значення**
~	порозрядне заперечення
&	побітова кон'юнкція (побітове І)
|	побітова диз'юнкція (побітове АБО)
^	побітове додавання за модулем 2
<<	зсув вліво
>>	зсув вправо

Таблиця істинності логічних порозрядних операцій

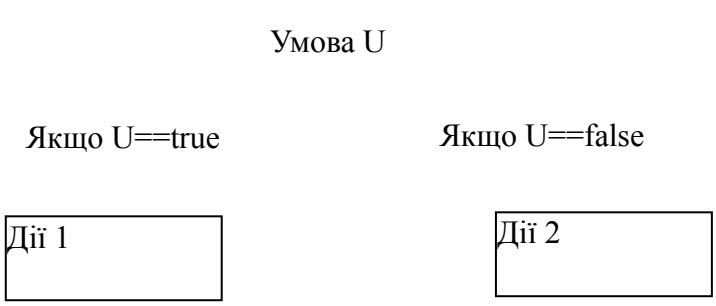
Таблиця 2.9

E1	**E2**	**E1&E2**	**E1^E2**	**E1|E2**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

- Порозрядне заперечення ! заміняє змінює в бітовому (двійковому) представленні числа кожну 1 на 0, а 0 на 1.
Приклад: ~(0x9A)=(0x65) тобто ~ (10011010) == (01100101)
- Порозрядна кон'юнкція & (порозрядне І) порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо тільки два відповідних розряди операндів рівні 1, в інших випадках результат 0.
Приклад: (0x93)&(0x3D)=(0x11) тобто (10010011) & (00111101) == (00010001)
- Порозрядна диз'юнкція | (порозрядне АБО) порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо хоча б один з відповідних розрядів рівний 1.
Приклад: (0x93)|(0x3D)=(0xBF) тобто (10010011) | (00111101) == (10111111)
- Побітове додавання за модулем 2 порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо один з двох (але не обидва) відповідних розряди рівні 1.
Приклад: (0x93)^(0x3D)=(0xAE), бо (10010011) ^ (00111101) == (10101110)
На операції побітового додавання за модулем 2 ґрунтується метод обміну значень двох цілочисельних змінних.
a^=b^=a^=b;
Операція зсуву вліво (вправо) переміщує розряди першого операнду вліво (вправо) на число позицій, яке задане другим операндом. Позиції, що звільняються, заповнюються нулями, а розряди, що зсуваються за ліву (праву) границю, втрачаються.
Приклади:
(10001010) << 2 == (00101000);
(10001010) >> 2 == (00100010)

Розгалуження

Для написання програм потрібно вміти керувати подіями та операторами програми в залежності від результатів попередніх дій та вхідних даних. Для цього використовуються розгалуження.
Розгалуження будуй частину програмного коду, що перевіряй певну умову або булевий вираз та визначай які дії відбуваються якщо цей вираз дорівнює true, а які дії якщо він й false.



На мові Сі будь-яке значення, що не дорівнює 0 або **null (NULL) вважається true**, а інакше вважається **false**.

Умова хибна, якщо вона дорівнює нулю, в інших випадках (навіть при від'ємних значеннях)вона істинна. *До того ж, умова, що перевіряється, повинна бути скалярною, тобто зводитися до простого значення, яке можливо перевірити на рівність нулю. Взагалі не рекомендується використання змінних типу float або double в логічних виразах з використанням рівне та нерівне для перевірки умов з причини недостатньої точності подібних виразів.*

На Сі можна записати розгалуження наступними шляхами

Умовна операція ? :

Умовна операція ?: - єдина тернарна операція в мові Сі. Її синтаксис:

умова ? вираз_1: вираз_2;

Принцип її роботи такий.

Спочатку обчислюється вираз умови. Якщо цей вираз май ненульове значення, то обчислюється вираз_1. Результатом операції ?: в даному випадку буде значення виразу_1. Якщо вираз умови рівний нулю, то обчислюється вираз_2 і його значення буде результатом операції. В будь-якому випадку обчислюється тільки один із виразів (вираз_1 або вираз_2).

Наприклад, дану операцію зручно використати для знаходження найбільшого з двох чисел x і y: max=(x>y)?x:y;

Приклад 1:

```
#include<stdio.h>
void main(){
int points;
printf("Введіть оцінку [2..5]:");
scanf("%d",&points);
printf("%s",points>3?"Ви добре знайте матеріал!":"Погано...");
}
```

Приклад 2. Неǒай c= 10. Тоді після виконання команди

```
x = (c == 3) ? 2 * c : c - 2;
```

отримаймо *x* = 8, оскільки не дорівнює 3, і тому тут обчислюється значення виразу 2

Оператор розгалуження if

Оператор розгалуження призначений для виконання тих або інших дій в залежності від істинності або хибності деякої умови. Основний оператор цього блоку в Сі - if ... else не має ключового слова then або двокрапки як в Python, проте обов'язково вимагай, щоб умова, що перевіряється, розміщувалася б у

круглих дужках. Оператор, що слідуй за логічним виразом, й then- частиною оператору if...else.
Синтаксис оператора:
if (<умова>) {<оператор1>; …;<операторN>;}
[else {<оператор2;> …, <операторM;>]

У випадку коли після умови чи ключового слова else слідує лише один оператор можна його не оточувати фігурними дужками, але більшість керівництв з стилю програмного коду радять робити це завжди, незалежно від кількості операторів для зручності змін в коді та єдиного стилю запису розгалуження.

Приклад 1.
```
/* програма виводить результат ділення двох дійсних чисел */
#include<stdio.h>
int main()
{
 float a,b,c;
 printf("Введіть число a: "); scanf("%f",&a);
 printf("Введіть число b: "); scanf("%f",&b);
 if (b==0) printf("Ділення на нуль ! ");
 else
 {
 c=a/b;
 printf("a: b == %g",c);
 };
}
```

Приклад 2.
```
/* застосування умовного розгалужування */
#include <stdio.h>
main()
{
 int number;
 int ok;
 printf("Введіть число з інтервалу 1..100: ");
 scanf("%d",&number);
 ok=(1<=number) && (number<=100);
 if (!ok)
 printf("Не коректно !! ");
 return ok;
}
```
Змінній ok присвоюється значення результату виразу: ненульове значення, якщо істина, і в протилежному випадку - нуль. Умовний оператор if(!ok) перевіряє, якщо ok дорівнюватиме нулю, то !ok дасть позитивний результат й відтоді буде отримано повідомлення про некоректність, виходячи з контексту наведеного прикладу.

```
Приклад 3.Нехай x= 9. Унаслідок виконання команд
if(x > 7)
        y = pow(x, 2);
```

```
else y = sqrt(x);
if(x <= 5)
          z= exp(x);
else z= ++x;
```
Отримаймо y=81, z= 10, x=10.

Приклад.
```
#include <iostream>
#include <math.h>
using namespace std;
int main(){
 float C=1.231;
 float x,y;
 std::cout<<"x="; std::cin>>x;
 std::cout<<"y="; std::cin>>y;
 double A;
 if(x<=y){A=x*y-C*y*sqrt(y);}else{A=cos(x)+log(y);}
 std::cout<<"A="<<A<<std::endl;
 std::cout<<"x="; std::cin>>x;
 if(x<=y){A=ч*y-C*y*sqrt(y);}else{A=cos(x)+log(y);}
 std::cout<<"A="<<A<<std::endl;
 getchar();
 return 0;
}
```

Результати обчислень:
```
x=1
y=1
A=-0.231
x=2
A=-0.416147
```

Оператор switch

switch(<вираз цілого типу>)
{
case <значення_1>: <послідовність_операторів_1>; break;
case <значення_2>: <послідовність_операторів_2>; break;
.....
case <значення_n>: <послідовність_операторів_n>; break;
[default: <послідовність_операторів_n+1>;]
}

Оператор-перемикач switch призначений для вибору одного з декількох альтернативних шляхів виконання програми. Виконання оператора switch починається з обчислення значення виразу (виразу, що слідуй за ключовим словом switch у круглих дужках). Після цього управління передається одному з <операторів>. Оператор, що отримав управління - це той оператор, значення константи варіанту якого співпадай зі значенням виразу перемикача.
Вітка default (може опускатися, про що свідчить наявність квадратних дужок) означає, що якщо жодна з вищенаведених умов не задовольнятиметься (тобто

вираз цілого типу не дорівнюй жодному із значень, що позначені у case-фрагментах), керування передається по замовчуванню в це місце програми. Треба також зазначити ***обов'язкове застосування оператора break у кожному з case-фрагментів*** (цей оператор застосовують для негайного припинення виконання операторів while, do, for, switch), що негайно передасть керування у точку програми, що слідуй відразу за останнім оператором у switch-блоці.

Приклад 1:

```
switch(i){
case -1: n++; break;
case 0: z++; break;
case 1: p++; break;
}
```

Приклад 2 :

```
switch(c){
case 'A': capa++;
case 'a': lettera++;
default: total++;
}
```

В останньому прикладі всі три оператори в тілі оператора switch будуть виконані, якщо значення с рівне 'A', далі оператори виконуються в порядку їх слідування в тілі, так як відсутні break.

Приклад. У п’ятиповерховому будинку на кожному поверсі по чотири квартири. Скласти програму для визначення поверху в залежності від номера квартири.

```
#include <iostream>
#include <math.h>
using namespace std;
int main(){
  short int n;
  cout<<"Введіть № квартири: "; cin>>n;
  switch(n){
    case 1:
    case 2:
    case 3:
    case 4: cout<<"1-й поверх"<<endl;
        break;
    case 5:
    case 6:
    case 7:
    case 8: cout<<"2-й поверх"<<endl;
        break;
    case 9:
    case 10:
    case 11:
    case 12: cout<<"3-й поверх"<<endl;
        break;
    case 13:
    case 14:
    case 15:
    case 16: cout<<"4-й поверх"<<endl;
        break;
```

```
    case 17:
   case 18:
   case 19:
   case 20: cout<<"5-й поверх"<<endl;
        break;
   default: cout<<"помилковий № квартири"<<endl;
 }
return 0;
}
```

Слід відмітити, що розгалуження можна вкладати одне до одного, та будувати послідовні блоки розгалужень.

Наприклад:

1) Обчислення сігнуму (послідовне розгалуження):

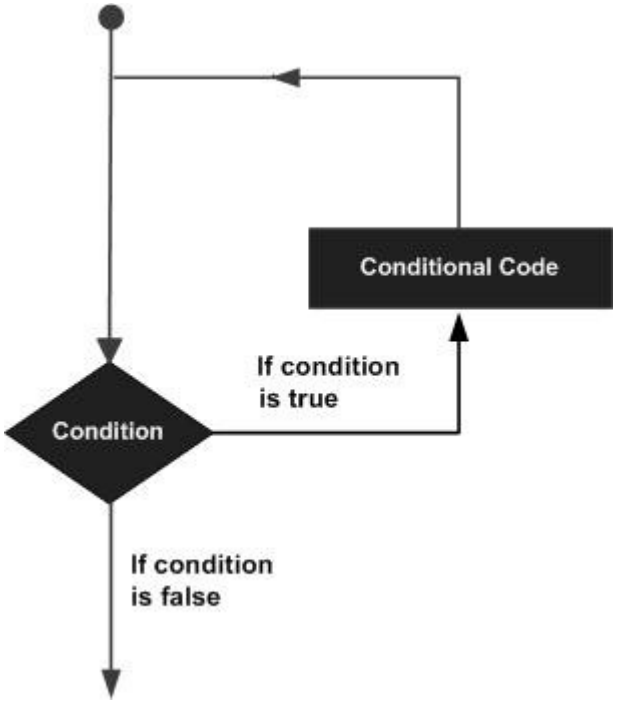
```
if(x>0){
     signum =1;
}
else if (x==0){
     signum=0;
}
else{ signum=-1;
}
```

2) Вкладене розгалуження:

```
if(x>0){
     if(y>0){
                    z= sqrt(x)+sqrt(y);
     }
     else{
          z =1;
     }
}
else{
     z=-1;
}
```

Цикли

Для виконання послідовних обчислень потрібно вміти вказувати команди, які виконуються доки не справджується певна умова виходу з циклу обчислень.



В Сі існують декілька варіантів написання циклів:

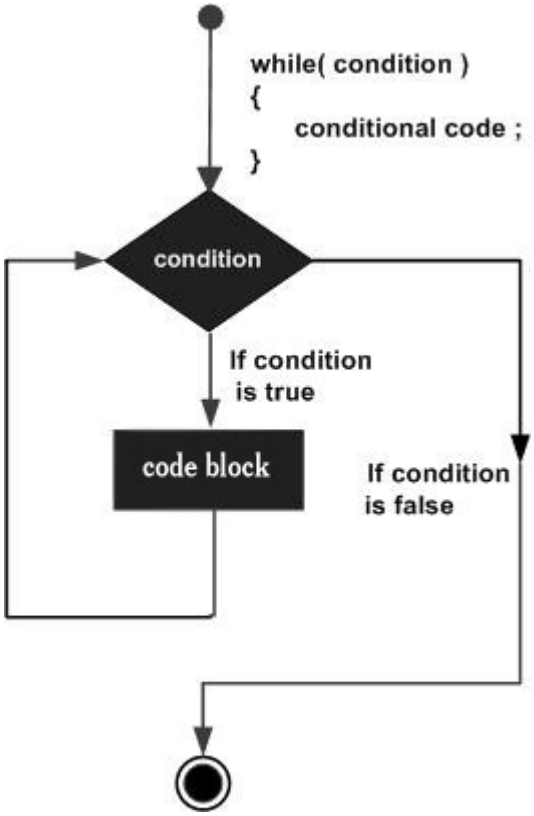
1) ***Цикл з передумовою (while loop)***
2) ***Цикл з лічильником (for loop)***
3) ***Цикл з післямовою (do...while loop)***
4) ***Вкладені цикли (nested loops)***

Цикл з передумовою

Цикл з передумовою (while loop) виконуй певний ланцюжок обчислень – тіло циклу, поки виконується певна умова.
Синтаксис цього циклу наступний

```
while((<умова> condition) {
    <дії>statement(s);
}
```

тут **<дії>statement(s)** може бути однією командою або серією команд.
Умова(**condition)** може бути виразом або булевою змінною так само як і умова розгалуження. Тіло циклу виконується поки умова й істинною, тобто дорівнюй true (або вираз умови не дорівнюй 0 або NULL).
Як тільки умова стає false, програма передай керування у місце, що слідуй за дужками що обмежують тіло циклу.



Важливою властивістю цього циклу й те що якщо з самого початку умова в циклі й хибною тіло циклу не виконується жодного разу.

Приклад

```
#include <stdio.h>
int main () {
   /* local variable definition */
   int a = 10;
   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }

   return 0;
}
```

Виконання коду видасть наступний результат:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

Цикл з післяумовою

На відміну від циклів **for** та **while**, які перевіряють умову перед виконанням циклу, цикл з післяумовою **do...while** виконуй перевірку умови після проходження тіла циклу.

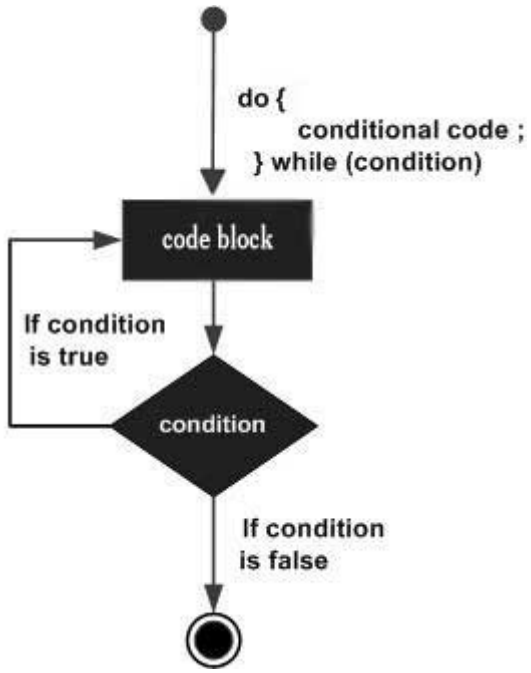
do...while схожий за структурою виконання на цикл з передумовою крім того, що він гарантуй виконання тіла циклу хоча б один раз.

Синтаксис **do...while** циклу

do {
<дії>statement(s);
} while(<умова> condition);

Помітимо, що спочатку відбувається виконання ланцюгу команд тіла циклу, а потім перевірка умови.

Якщо умова циклу виконується, тобто дорівнюй true, потік виконання передається в початок тіла циклу та виконання циклу починається з початку тіла циклу. Цей процес продовжується поки умова циклу не стає хибною, тобто рівною false.



Приклад.

```
#include <stdio.h>
int main () {
   /* визначення локальної змінної */
   int a = 10;
   /* виконується цикл */
   do {
      printf("value of a: %d\n", a);
      a = a + 1;
   }while( a < 20 );
   return 0;
}
```

Результат виконання:

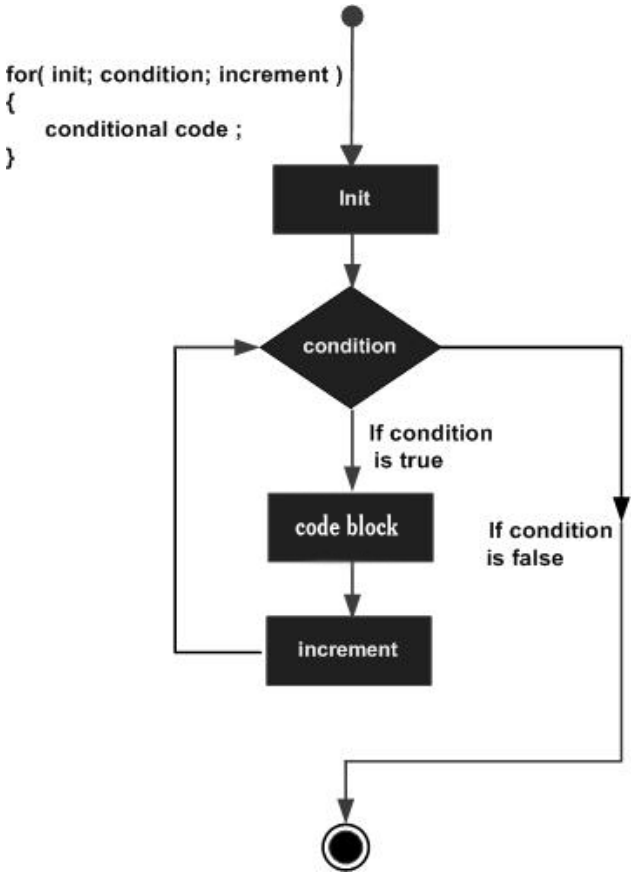
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

Цикл з лічильником

Цикл з лічильником або **for** loop це цикл що дозволяй ефективно виконувати ланцюжок команд, що виконуються фіксовану кількість разів.
Синтаксис звичайного циклу **for** наступний:

```
for ( <ініціалізація(init)>; <умова(condition)>;<інкеремент(increment)> ) {
    statement(s);
}
```

Потік виконання циклу наступний:
Крок ініціалізації **init** виконується спочатку та лише один раз. На цьому кроці ініціалізуються змінні (лічильники), що контролюють виконання циклу. Можливі варіанти, коли цей крок порожній тобто нічого не виконується.
Далі, умова (**condition)** підраховується. Якщо вона істинна (true), тіло циклу виконується. Якщо воно хибне (false), тіло циклу не виконується і потік виконання йде на команду що йде після тіла циклу 'for' (за фігурними дужками).
Виконується тіло циклу
Після виконання тіла циклу 'for' , потік виконання йде до виразу інкременту(**increment)**. Команди цього виразу дозволяють змінювати лічильники циклу.
Умова виконується знову. Якщо вона й істиною (true), то тіло циклу виконується знову (тіло циклу, крок increment , знову умова). Після того, як умова становиться false, цикл 'for' закінчує роботу.



Приклад

```
#include <stdio.h>
int main () {
   int a;
   /* for loop виконання */
   for( a = 10; a < 20; a = a + 1 ){
      printf("value of a: %d\n", a);
   }
   return 0;
}
```

Результат виконання

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

Нескінчений цикл

Цикл може стати нескінченим якщо умова циклу ніколи не стане false.
Це можна зробити наступними шляхами:

1) while(1){ ….}

2) do{…} while(1>0);

3) Часто роблять нескінчений цикл за допомогою **for:**

```
#include <stdio.h>
int main () {
   for( ; ; ) {
      printf("This loop will run forever.\n");
   }
   return 0;
}
```

Якщо умова циклу відсутня, вона вважаться, рівною істині .
Примітка: виконання нескінченого циклу зупиняється за допомогою Ctrl + C або (Ctrl+<Pause/Break>).

Керування циклом в тілі циклу

Іноді виникай потреба виходу з циклу посеред виконання ланцюгу команд тіла циклу. Зокрема, якщо умова циклу передбачай нескінчений цикл, а вийти з циклу все ж такі треба. На Сі існують наступні варіанти керування циклом:

1) Команда **break**. Завершуй **цикл** або вираз **switch** та передай виконання на місце що йде після циклу.
2) Команда **continue.** Завершуй виконання тіла циклу та передай керування на умову циклу (при while do..while) або в крок зміни лічильника (increment) в циклі for.
3) Команда **goto**. Передає керування на мітку що вказана в мітках (label). ***Примітка. Сучасні керівництва дуже радять не використовувати цю команду ніколи.***

Примітка: коли оператор керування залишає тіло циклу всі автоматично створені в тілі циклу змінні знищуються.

Операція слідування (кома)

Операція "кома" (,) називається операцією слідування, яка "зв'язуй" два довільних вирази. Список виразів, розділених між собою комами, обчислюються зліва направо.
Наприклад, фрагмент тексту:
a=4; b=a+5;
можна записати так:
a=4, b=b+5;
Операція слідування використовується в основному в операторах циклу for().
Для порівняння наводимо приклад з використанням операції слідування (приклад 2) та без неї (приклад 1):
Приклад 1:

```
int a[10],sum,i;
/* ... */
sum=a[0];
for (i=1;i<10;i++)
sum+=a[i];
```

Приклад 2:

```
int a[10],sum,i;
/* ... */
```

```
for (i=1,sum=a[0];i<10;sum+=a[i],i++) ;
```

Вкладені цикли

Як і розгалуження, цикли можна вкладати один в одний.
Синтаксис вкладеного циклу з лічильником **nested for loop:**

```
for ( <init1>; <condition1>; <increment1> ) {

   for ( <init2>; <condition2>; <increment2> ) {
      statement(s);
   }
   statement(s);
}
```

Синтаксис вкладеного циклу з передумовою **nested while loop**:

```
while(condition) {

   while(condition) {
      statement(s);
   }
   statement(s);
}
```

Синтаксис вкладеного циклу з післяумовою **nested do...while loop**:

```
do {
   statement(s);

   do {
      statement(s);
   }while( condition );

}while( condition );
```

Помітимо, що й цикли одного виду можна вкладати в цикли іншого та це вкладання можна робити декілька разів (хоча існує стилістичне обмеження таких вкладень – не більше чотирьох)
Приклад
Наступна програма рахуй прості числа від 2 до 100 −

```
#include <stdio.h>

int main () {
   /* local variable definition */
   int i, j;

   for(i = 2; i<100; i++) {

      for(j = 2; j <= (i/j); j++)
      if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) printf("%d is prime\n", i);
   }
```

```
   return 0;
}
```

Результат:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
```