

Целые типы данных

👉 x86-64 и ARM64

Тип	Байты	Диапазон значений
начиная с C89 && C++98		
char	1	signed or unsigned
unsigned char		[0; 255]
signed char		[−128; 127]
int == signed int	4	[−2 ³¹ ; 2 ³¹ − 1]
unsigned int		[0; 2 ³² − 1]
short int	2	[−32768; 32767]
unsigned short int		[0; 65535]
long int	8	[−2 ⁶³ ; 2 ⁶³ − 1]
unsigned long int		[0; 2 ⁶⁴ − 1]
начиная с C99 && C++11		
long long int	8	[−2 ⁶³ ; 2 ⁶³ − 1]
unsigned long long int		[0; 2 ⁶⁴ − 1]

Представление целых чисел

- Целые числа без знака представляются в двоичной системе счисления

4-bit unsigned integer

$$0101 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 5$$

$$11 = 8 + 2 + 1 = 1011$$

$$\text{max number} = 2^n - 1, \text{ где } n - \text{число битов}$$

- Отрицательные числа представляются в «дополнительном коде»

☞ two's complement will be required in C23

① инвертируются биты: $0 \rightarrow 1, 1 \rightarrow 0$

② добавляется 1

	представление	-5	
1)	$5 = 0101 \rightarrow 1010$		
2)	$1010 + 1 \rightarrow 1011 = -5$		

Представление целых чисел

- Целые числа без знака представляются в двоичной системе счисления

4-bit unsigned integer

$$0101 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 5$$

$$11 = 8 + 2 + 1 = 1011$$

$$\text{max number} = 2^n - 1, \text{ где } n - \text{число битов}$$

- Отрицательные числа представляются в «дополнительном коде»

☞ two's complement will be required in C23

❶ инвертируются биты: $0 \rightarrow 1, 1 \rightarrow 0$

❷ добавляется 1

	представление	-5	
1)	$5 = 0101 \rightarrow 1010$		
2)	$1010 + 1 \rightarrow 1011 = -5$		

Преимущества дополнительного кода:

- Имеется знаковый бит (*most significant bit*) 0 для + , 1 для –
- Обратное преобразование такое же:

$$1) -5 = 1011 \rightarrow 0100 \quad 2) 0100 + 1 \rightarrow 0101 = 5$$

- Простота сложения:

$$1 \leftarrow \begin{array}{r} 1011 \\ 0101 \\ \hline 0000 \end{array} = \begin{array}{r} -5 \\ +5 \\ \hline 0 \end{array}$$

$$\begin{array}{r} 1011 \\ 0011 \\ \hline 1110 \end{array} = \begin{array}{r} -5 \\ +3 \\ \hline -2 \end{array}$$

$$\begin{array}{r} 0111 \\ 0001 \\ \hline 1000 \end{array} = \begin{array}{r} +7 \\ +1 \\ \hline -8 \end{array}$$

- Ноль имеет единственное представление

Недостатки дополнительного кода:

- Модули наибольшего и наименьшего чисел различаются:

$$\text{max positive number} = 2^{(n-1)} - 1 \quad (7 \text{ для 4-bit})$$

$$\text{min negative number} = -2^{(n-1)} \quad (-8 \text{ для 4-bit})$$

- $-(\text{min negative}) = \text{min negative}$:

$$1) -8 = 1000 \rightarrow 0111 \quad 2) 0111 + 1 \rightarrow 1000 = -8$$

Преимущества дополнительного кода:

- Имеется знаковый бит (*most significant bit*) 0 для + , 1 для –
- Обратное преобразование такое же:

$$1) -5 = 1011 \rightarrow 0100 \quad 2) 0100 + 1 \rightarrow 0101 = 5$$

- Простота сложения:

$$1 \leftarrow \begin{array}{r} 1011 \\ 0101 \\ \hline 0000 \end{array} = \begin{array}{r} -5 \\ +5 \\ \hline 0 \end{array}$$

$$\begin{array}{r} 1011 \\ 0011 \\ \hline 1110 \end{array} = \begin{array}{r} -5 \\ +3 \\ \hline -2 \end{array}$$

$$\begin{array}{r} 0111 \\ 0001 \\ \hline 1000 \end{array} = \begin{array}{r} +7 \\ +1 \\ \hline -8 \end{array}$$

- Ноль имеет единственное представление

Недостатки дополнительного кода:

- Модули наибольшего и наименьшего чисел различаются:

$$\text{max positive number} = 2^{(n-1)} - 1 \quad (7 \text{ для } 4\text{-bit})$$

$$\text{min negative number} = -2^{(n-1)} \quad (-8 \text{ для } 4\text{-bit})$$

- $-(\text{min negative}) = \text{min negative}$:

$$1) -8 = 1000 \rightarrow 0111 \quad 2) 0111 + 1 \rightarrow 1000 = -8$$

Альтернативное представление целых чисел

“Offset binary” or “excess-K”

- Число получается вычитанием из обычного беззнакового представления заданного K
- В n -bits системе обычно выбирается «посередине»: $K = 2^{(n-1)} - 1$

Пример для 4-bits: $K = 2^3 - 1 = 7$

0000	-	минимальное число	-7
0001	-		-6
...			
0111	-	ноль	0
...			
1110	-		+7
1111	-	максимальное число	+8

 Используется в стандарте IEEE-754 для показателя чисел с плавающей точкой

Поддержка целых типов в C

Заголовочный файл `<limits.h>`

Ограничения и параметры переменных целых типов:

- `CHAR_BIT` — размер `char` в битах
- `SCHAR_MIN`, `SHRT_MIN`, `INT_MIN`, `LONG_MIN`, `LLONG_MIN`
`SCHAR_MAX`, `SHRT_MAX`, `INT_MAX`, `LONG_MAX`, `LLONG_MAX`
— минимальные и максимальные значения для `signed ints`
- `UCHAR_MAX`, `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX`, `ULLONG_MAX`
— максимальные возможные значения `unsigned ints`

Беззнаковый тип `size_t` возвращаемый оператором `sizeof()`

- ☞ Может хранить максимальный размер объекта любого типа, включая массивы
- Для печати в `printf()` надо использовать суффикс `zu` (C23)

Поддержка целых типов в C++

Заголовочный файл `<limits>`

Шаблон класса `std::numeric_limits<>`, позволяет запрашивать свойства арифметических типов

- `min()`: функция возвращающая наименьшее значение
`std::numeric_limits<int>::min(); // -2147483648`
`std::numeric_limits<unsigned int>::min(); // 0`
- `max()`: функция возвращающая наибольшее значение:
`std::numeric_limits<int>::max(); // 2147483647`
`std::numeric_limits<unsigned int>::max(); // 4294967295`
- `is_signed`: член класса, истинна для типов с знаком:
`std::numeric_limits<char>::is_signed; // 1`

Беззнаковый тип `std::size_t`

👉 Такой же как в C, определен в заголовочном файле `<cstdlib>`

Декларация, инициализация, печать

```
char c = 'A';           // внимание: одинарные кавычки!
printf("c= %c\n",c);    // c= A %c - печать одного символа

int i = 10, j = -10;
printf("i= %i, j= %d\n",i,j); // i= 10, j= -10 в printf %i == %d

unsigned int ui = 10U;   // можно без int: unsigned ui = 10U;
printf("ui= %u\n",ui);  // ui= 10 %u значит unsigned

long int li = 10L, lj = -10L; // можно: long li
unsigned long int uli = 10UL; // можно: unsigned long uli
printf("li=%ld, lj=%li, uli=%lu\n",li,lj,uli); // li=10, lj=-10, uli=10

long long int lli = 10LL;    // можно: long long
unsigned long long int ulli = 10ULL;
printf("lli=%lli, ulli=%llu\n",lli,ulli); // lli=10, ulli=10
```

Восьмеричные и шестнадцатеричные константы


- Восьмеричная константа должна начинаться с **0** (нуля):

```
unsigned int oct = 012;    // 10 в десятичной
printf(" oct= %o\n",oct); // oct= 12
```

- Шестнадцатеричная константа начинается с **0x** (нуль-экс), символы от A до F обозначают цифры от 10 до 15:

```
unsigned int hex1=0xA, hex2=0xf;    // 10 и 15 в десятичной
printf(" hex1= %x,"                // hex1= a,
       " hex2= 0x%X\n",hex1,hex2); // hex2= 0xF
```

Nota bene

-  Все константы рассматриваются как положительные, а знак минус перед ними как унарный оператор минус

Расширенный набор целых типов в C99 и C++11

```
#include <inttypes.h>    // new integer types in C99
#include <cstdint>        // new integer types in C++11
```

N может принимать значения 8, 16, 32, 64

Цель	Тип	min	max
Точный размер	intN_t uintN_t	INTN_MIN 0	INTN_MAX UINTN_MAX
Не менее чем	int_leastN_t uint_leastN_t	INT_LEASTN_MIN 0	INT_LEASTN_MAX UINT_LEASTN_MAX
Самые быстрые	int_fastN_t uint_fastN_t	INT_FASTN_MIN 0	INT_FASTN_MAX UINT_FASTN_MAX
Наиболее длинные	intmax_t uintmax_t	INTMAX_MIN 0	INTMAX_MAX UINTMAX_MAX
Для указателей	intptr_t uintptr_t	INTPTR_MIN 0	INTPTR_MAX UINTPTR_MAX

Пояснения

- «Точный размер» — точно **N**-бит
- «Не менее чем» — наименьшие с размером не менее **N**-бит
- «Самые быстрые» — самые быстрые с размером не менее **N**-бит
- «Наиболее длинные» — наибольшее количество бит
- «Для указателей» — гарантированно можно хранить указатели

Макросы для `printf` и `scanf`

Имя макроса строится по шаблону:

`PRI{SCN}` + *format specifier* + `suffix` + **N**

- **PRI** — для `printf`; **SCN** — для `scanf`;
- *format specifier* — одна из букв **d**, **i**, **o**, **x** (см. `%d`, `%i` ...)
- `suffix` — соответственно пусто или `LEAST`, `FAST`, `MAX`, `PRT`
- **N** — отсутствует для `MAX`, `PRT`

```
int64_t i64 = -123; int8_t i8 = 12;
printf("i64= %"PRIi64", i8= %"PRIi8"\n", i64, i8); // i64= -123, i8= 12

uint_least16_t uil16 = 123;
printf("uil16= %"PRIuLEAST16"\n", uil16); // uil16= 123

int_fast8_t if8 = 25;
printf("octal if8= %"PRIoFAST8"\n", if8); // octal if8= 31

uintmax_t uim = 0x1f0UL << 33; // left shift by 33
printf("hex uim= 0x%"PRIxMAX"\n", uim); // hex uim= 0x3e000000000

uintptr_t uiptr = (uintptr_t) &i64;
printf("uiptr= 0x%"PRIXPTR"\n", uiptr); // uiptr= 0x7FFE2659EB40
```

📌 Запись "abcd" "efgh" означает «сцепление» (concatenation),
результат: "abcdefgh"

Битовые операции

операция	название	гибрид с =
\sim	дополнение	унарная операция
$\&$	AND	$\& =$
$ $	OR	$ =$
\wedge	XOR	$\wedge =$
\ll	сдвиг влево	$\ll =$
\gg	сдвиг вправо	$\gg =$

- 👉 Эти операции можно применять **только** к целым типам
- 👉 Могут называться и поразрядными и побитовыми и логическими битовыми операциями

~ (дополнение или побитовое “NOT” – инвертирует все биты)

```
int i = 3;      /* 0011 */  
int j = ~i;     /* 1100 = (-4) */
```

& (битовое “И”)

```
int i = 5;      /* 0101 */  
int j = 3;      /* 0011 */  
int k = i & j;   /* 0001 */
```

| (битовое “ИЛИ”)

```
int i = 5;      /* 0101 */  
int j = 3;      /* 0011 */  
int k = i | j;   /* 0111 */
```

^ (“исключающее ИЛИ”, XOR)

```
int i = 5;      /* 0101 */  
int j = 3;      /* 0011 */  
int k = i ^ j;   /* 0110 */
```

☞ соответствует булевой функции «сложение по модулю 2» и имеет огромное число применений

<< (сдвиг влево)

Старшие биты исчезают, на место младших записываются нули:

```
unsigned int j = 3;      /* 0011 */  
unsigned int k = j << 2; /* 1100 = 12 */
```

>> (сдвиг вправо)

Младшие биты уходят, на место старших записываются нули:

```
unsigned int j = 3;      /* 0011 */  
unsigned int k = j >> 1; /* 0001 = 1 */
```

👉 Результат сдвига `UINT << E` или `UINT >> E` не определен для отрицательного `E` и для `E` большего чем число разрядов `UINT`

☞ Сдвиг вправо и расширение знакового бита

Что будет если сдвигать отрицательные целые?

right shift is unambiguous

```
int a = -5;      // 1011
int b = a << 1;  // 0110 = 6
```

left shift operator is implementation-dependend

```
//          logical or arithmetical
int c = a >> 1; // 0101 = 5      1101 = -3
int d = c >> 1; // 0010 = 2      1010 = -6
```

☞ Стандарт C не определяет какой тип сдвига используется

De facto, но не гарантированно стандартом:

- **unsigned** – логический сдвиг: заполнение нулями
- **signed** – арифметический сдвиг: повторение знакового бита

Наиболее часто встречающиеся действия с одиночным битом:

x – переменная с которой выполняется действие

n – номер бита в этой переменной (начиная с 0)

Проверка состояния бита

```
x & (0x1 << n) // проверка чётности: x & 0x1  
(x >> n) & 0x1 // второй способ
```

Установка бита в единицу

```
x |= (0x1 << n) // (0x1 << n) - часто называют маской операции
```

Обнуление бита

```
x &= ~(0x1 << n)
```

«Переключение» бита

```
x ^= (0x1 << n)
```

Целые числа: математические функции

```
#include <stdlib.h> // Заголовочный файл
```

Абсолютное значение ($|n|$)

```
int abs(int n);  
long labs(long n);  
long long llabs(long long n);
```

пример для abs

```
printf("abs(-1)= %d\n",abs(-1)); // 1
```

Деление с остатком ($num/denom$)

```
div_t div(int num, int denom);  
struct div_t {int quot; int rem;};  
  
ldiv_t ldiv(long num, long denom);  
lldiv_t lldiv(long long n,  
               long long d);
```

пример для div

```
div_t q = div(-7, 3);  
printf("quot= %i rem=%i\n",  
       q.quot,q.rem); // -2, -1
```

Целые числа: деление на ноль

Тестовая программа

```
int one = 1, zero = 0;  
fprintf(stderr, "one/zero= ");  
fprintf(stderr, "%d\n", one/zero); // <- деление на ноль
```

X86

one/zero= Floating point exception

👉 Остановка по сигналу SIGFPE!

ARM

one/zero= 0

👉 Никакая проверка не осуществляется!

SIGnal Floating Point Exception

SIGFPE – сигнал отправляется процессу, когда в аппаратном обеспечении целых чисел или чисел с плавающей точкой обнаружено «что-то не то»

👉 неточное имя, но сохраняется для обратной совместимости кода

ASCII (American Standard Code for Information Interchange, 1963)

char хранит символы в ASCII-кодировке

- символы от 'A' до 'Z', от 'a' до 'z' и от '0' до '9' идут непрерывно
- для некоторых символов используются «escape последовательности»:
NUL → \0; BEL → \b; LF → \n; ...
- ☞ ASCII является подмножеством UTF8: таким образом обеспечена обратная совместимость со всеми старыми программами

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Пример

```
// print all small letters
for(char ch = 'a'; ch <= 'z'; ch++ ) {
    printf(" %c", ch);
}
printf("\n"); // a b c d e f g h i j k l m n o p q r s t u v w x y z

// print int as a char
int ic = 'A';
printf(" ic= %i -> %c\n",ic,ic); // ic= 65 -> A

// print char as an int
char c = 'c';
printf(" c= %c -> 0x%x\n",c,c); // c= c -> 0x63
```

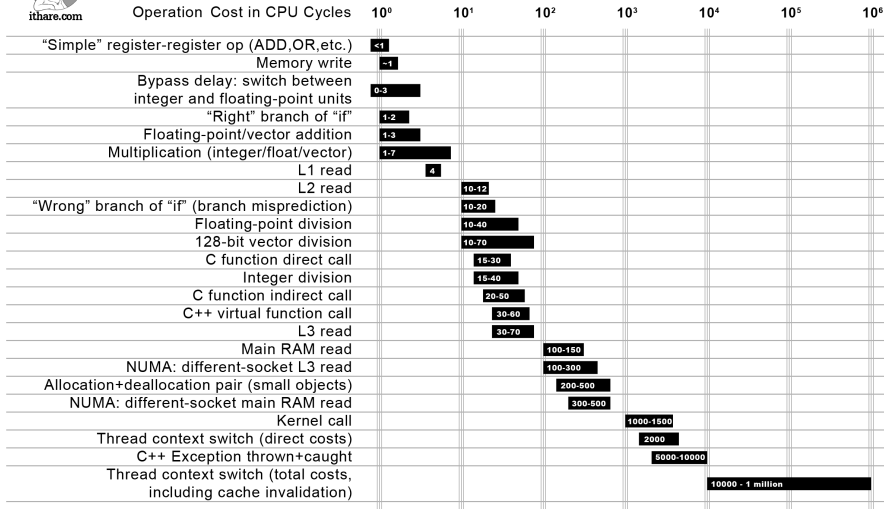
Дополнительные слайды

Целые типы: X86-64 vs X86-32

Тип	размер в байтах		диапазон значений
	x86-32	x86-64	
C89 && C++98			
char	1	1	signed or unsigned
signed char			[−128;127]
unsigned char			[0;255]
int == signed int	4	4	[−2 ³¹ ;2 ³¹ − 1]
unsigned int			[0;2 ³² − 1]
short int	2	2	[−32768;32767]
unsigned short int			[0;65535]
long int	4	8	
unsigned long int			
начиная с C99 && C++11			
long long int	8	8	[−2 ⁶³ ;2 ⁶³ − 1]
unsigned long long int			[0;2 ⁶⁴ − 1]



Not all CPU operations are created equal



Distance which light travels while the operation is performed

