Expressions. Operations and operators. Operator precedence. Logical and Bitwise operations. Constants. Casting.

Machine arithmetic

Information in a computer is stored in binary form.

Representation of **unsigned** integers:

- Number sizes are 1, 2, 4 or 8 bytes;
- All the bits of a number are used to store the digits of the number;
- Maximum value: 2^{8}-1, 2^{16}-1, 2^{32}-1 or 2^{64}-1 respectively.

Machine arithmetic

Representation of signed integers:

- Number sizes 1, 2, 4 or 8 bytes
- The most significant bit is used to indicate the sign of the number. 1 – negative number, 0 – positive
- Negative numbers are stored in two's complement
- Value range -2^{7} to $2^{7}-1$; -2^{15} to $2^{15}-1$; -2^{31} to $2^{31}-1$; -2^{63} to $2^{63}-1$ respectively

Machine arithmetic

Two's complement code is a way of representing negative integers that allows you to replace the subtraction operation with addition.

Converting a number to additional code:

- Invert the digits of a number;
- Add one to the resulting number.

Machine arithmetic

Representation -1 in two's complement

1. 00000001
2. 11111110
3. 11111111

Representation of -128 in two's complement

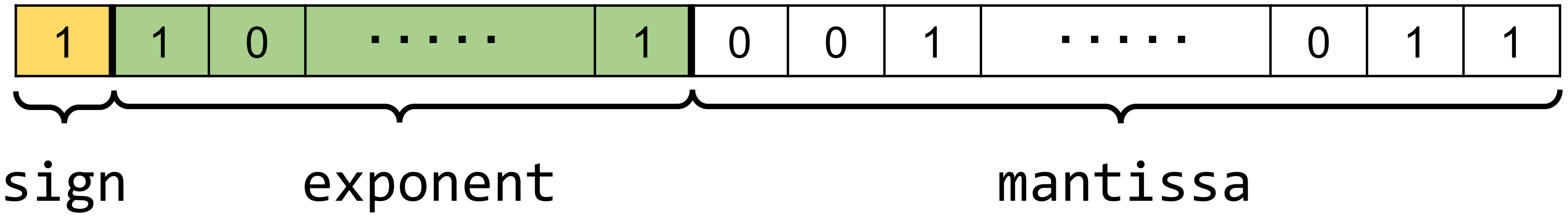
1. 10000000
2. 01111111
3. 10000000

Machine arithmetic

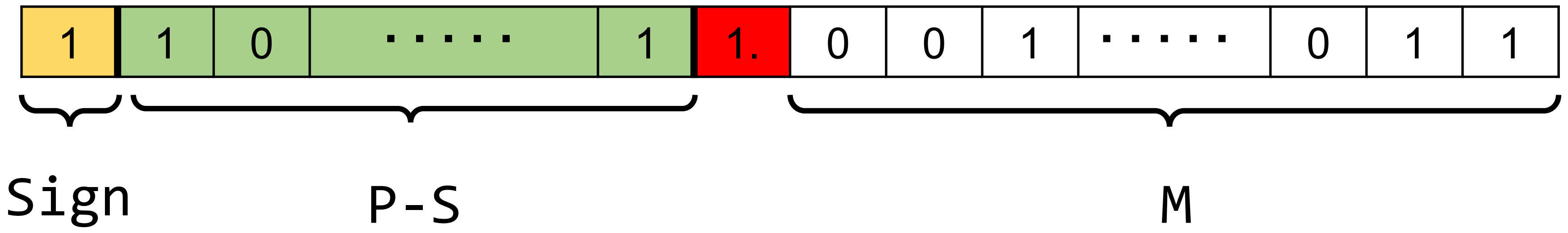
A floating point number is a form of representing real numbers in which the number is stored as a mantissa and an exponent.

- $$x = (-1)^{Sign} \times (1.M) \times B^{P-S}$$

Floating point representation (IEEE 754):



Machine arithmetic

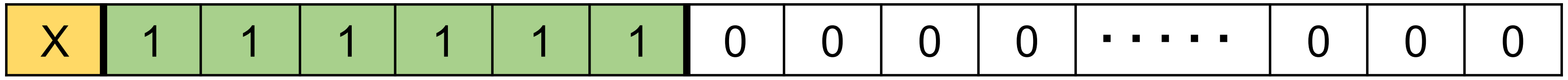


The most significant bit of the mantissa is always 1 and is not stored.

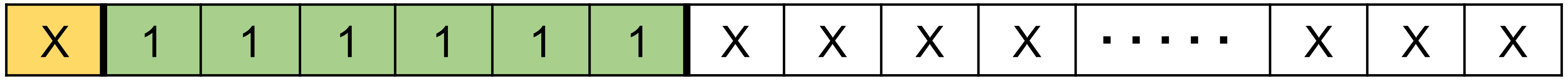
P-S is stored as an integer

Machine arithmetic

- Infinity



- NaN (Not a number) – special state of a floating point number. Can be interpreted as uncertainty. The mantissa can be anything except 0.



Machine arithmetic

Accuracy

- Single (4 bytes)
- Double (8 bytes)
- Extended (10 bytes)

Logic functions

Boolean variable – a variable that takes one of two values true (1) or false (0).

The simplest logical functions:

- Inversion (negation)
- Conjunction (logical AND, multiplication)
- Disjunction (logical OR, addition)

Logic functions

- Inversion (negation)

X	F(X)
0	1
1	0

Logic functions

- Conjunction (logical AND, multiplication)

X	Y	F(X, Y)
0	0	0
0	1	0
1	0	0
1	1	1

Logic functions

- Disjunction (logical OR, addition)

X	Y	F(X, Y)
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise logical operations

Bitwise logical operations

Operator	Description
&	Bitwise conjunction (AND)
|	Bitwise disjunction (OR)
^	Bitwise exclusive OR (XOR)
~	Bitwise negation
<<	Shift left
>>	Shift right

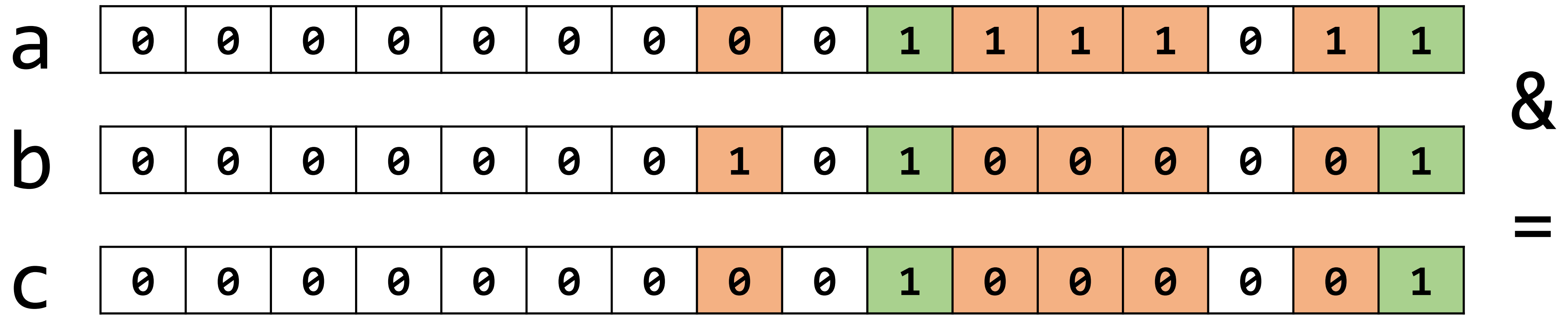
These operations can be combined with assignment

For unsigned numbers, shifting left n places is quickly multiplying the number by 2n. Shift to the right - division by 2n.

Bitwise logical operations

```
short int a = 123;
short int b = 321;

short int c = a & b;        c = 65
```


a 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1
&
b 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1
=
c 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1

Bitwise logical operations

```
short int a = 123;
short int b = a << 2;      b = 492
short int c = a >> 3;      c = 15
```

a	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1
b	0	0	0	0	0	0	0	1	1	1	1	0	1	1	0	0
c	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Bitwise logical operations

The shift operation takes into account t
sign of the number:

```
short int a = -87;
short int b = a << 2;      b = -348
short int c = a >> 3;      c = -11
```

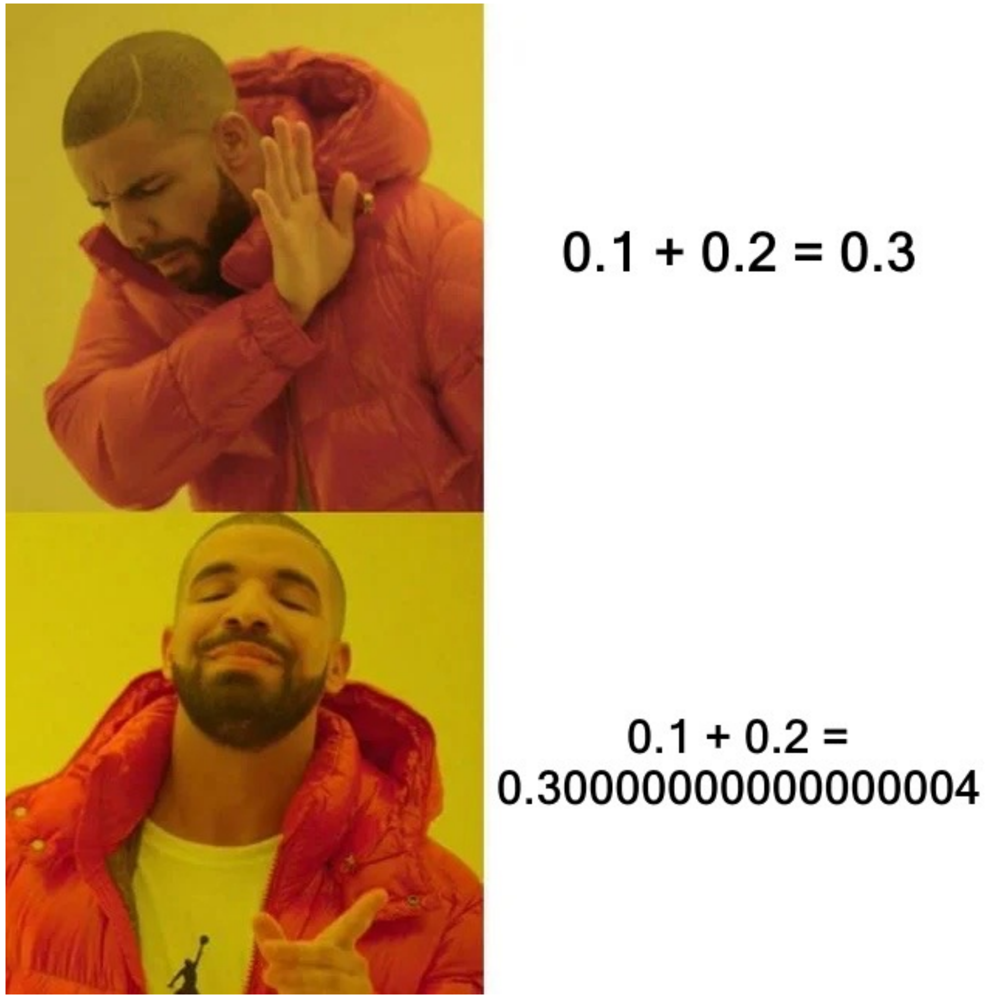
a	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	1
b	1	1	1	1	1	1	1	0	1	0	1	0	0	1	0	0
c	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1

Some magic or not?

```
double a = 0.1;
double b = 0.2;

double c = a + b;        c = ?
```

Some magic or not?



Standard C++ Data Types

- Integer (signed/unsigned);
- Real (floating point);
- Symbolic;
- Logical;
- **void**;
- Pointer;

Standard C++ Data Types

Character types:

- **char**
- **wchar_t**

char – used to represent characters in an encoding of no more than 255 characters. For example, ASCII.

Standard C++ Data Types

wchar_t – used to store characters in encodings containing a wider range of characters, such as UNICODE. The size of the type is implementation dependent.

type	Size, bytes
char	1
wchar_t	Depends on implementation

Standard C++ Data Types

Control sequences:

- ‘\n’ – line break
- ‘\r’ – carriage return
- '\0' – end of line
- ‘\t’ - horizontal tab
- ‘\’’ – apostrophe
- ‘\”’ – quotation mark
- '\\' - slash

Standard C++ Data Types

```
char a = ‘a’;       code 97 corresponding to
                    'a' is stored
char b = 75;        the code corresponds to the
character K
wchar_t c = L‘\n’;
```

Boolean type

Boolean type:

bool

- Takes one of two values: true or false.
- Typically, false is interpreted as 0, true as any non-zero value.
- The type size is 1 byte.

Logical operations

Logical operations

operator	description
&&	and
||	or
!	inversion

Logical operations can be applied to other standard data types. In this case, a zero value is interpreted as false, a non-zero value – true.

void

void type:

The set of values of this type is empty

Used only as part of more complex types:

- To define functions that have no return value
- As a base type of pointers to objects of arbitrary types

Pointer

A pointer is a variable that stores the address of an object in a memory area.

The size of the pointer depends on the bit depth of the operating system.

There are signs:

- To the object;
- Per function;
- On void.

Pointer

Pointer to an object – contains the address of a memory area in which data of a certain type is stored.

Function pointer – contains the address of the memory area where the executable code of the function is located.

Pointer to void – contains the address of a memory area in which data of an arbitrary type is stored.

Pointer

Operations with pointers:

- Initialization;
- Dereferencing;
- Increment;
- Decrement;
- Adding a constant;
- Subtraction of a constant;
- Comparison.

Pointer

Pointer initialization:

```
type* name = address;

int a = 7;
int* ptr = &a;                          ptr = 0x11223344
void *ptr2 = &a,                     ptr2 = 0x11223344
      *ptr3 = ptr;                      ptr3 = 0x11223344
int** ptr4= &ptr;                       ptr4 = 0xAABBCCDD
```

& - operator for obtaining the address of a variable.

Pointer

Dereferencing is an access to a memory area whose address is stored by a pointer.

```
int a_val = *ptr;      a_val = 7
```

* - dereference operator

Pointer

Increment and decrement - move to the next or previous element of the same type in memory.

In this case, the value of the pointer changes by the amount sizeof(type)

```
int* ptr = &a;      ptr = 0x11223344
ptr++;              ptr = 0x11223348
```

Pointer

Adding and subtracting a constant N - moves N elements forward or backward relative to the element to which the pointer points.

```
ptr++;                  ptr = 0x11223348
ptr += 7;               ptr = 0x11223364
```

(7_{10} * sizeof(int) = 28_{10})

Pointer

The operations of dereferencing, incrementing, decrementing, adding and subtracting a constant cannot be performed with pointers to `void`.

Pointer

A null pointer is a special pointer value that indicates that it does not refer to a valid memory location.

Designation:

NULL (C or C++ … C++11)

nullptr (C++11 and …)

```
ptr = nullptr;          ptr = 0x00000000
```

typedef

Creating an alias for a data type is done using the operator typedef

```
typedef old_type_name new_type_name;
```

```
typedef unsigned char BYTE;
typedef void* ADDRESS;
```

Cast

Type casting is the conversion of a value of one type to a value of another.

Type casts:

- Implicit;
- Explicit.

Cast

Explicit type casting is specified by the programmer in the program text using appropriate language constructs.

Implicit type casting is performed by the compiler or interpreter according to the rules described in the language standard.

Explicit casting

In the C++ language, the following constructs can be used for explicit type casting:
`static_cast`;
- `const_cast`;
- `reinterpret_cast`;
- Casting in C style.

Explicit casting

```
static_cast<new_type>(expression)
```

Performs a valid conversion from expression to new_type.

If the conversion is not valid, a compilation error is thrown.

Explicit casting

It is acceptable to convert one standard type to another.

It is not acceptable to convert a number to a pointer or a pointer to a number.

It is possible to convert a pointer of any type to a pointer to void and vice versa.

Pointers of different types cannot be cast to each other.

Explicit casting

```
int a = 0xAABBCCDD;   -1430532899
char b = 7;

unsigned int res1 =
static_cast<unsigned int>(a);   2864434397

int res2 = static_cast<int>(b);     7
```

Explicit casting

```
float c = 7.07;
double d = 8.98;

int res3 = static_cast<int>(c);  7

float res4 = static_cast<float>(d);
                                 8.97999954
int res5 = static_cast<int>(d);  8
```

Explicit casting

```
int a = 0xAABBCCDD;    -1430532899
char b = 7;
float c = 7.07;
double d = 8.98;

void* ptr0 = static_cast<void*>(&a);

int* ptr1 = static_cast<int*>(ptr0);
```

Explicit casting

```
unsigned int* ptr2 = static_cast<unsigned
int*>(ptr1);
char* ptr3 = static_cast<char*>(&a);
float* ptr4 = static_cast<float*>(&a);

int* ptr5 = static_cast<int*>(&b);

int* ptr6 = static_cast<int*>(&c);

float* ptr7 = static_cast<float*>(&d);
```

ERROR

Explicit casting

```
void* ptr8 = static_cast<void*>(&a);
float* ptr9 = static_cast<float*>(ptr8);

*ptr9 = 7.777;

std::cout << a << std::endl;        1090051375
std::cout << *ptr9 << std::endl;      7.777
```

Explicit casting

The const modifier is used when declaring constants.

```
const_cast<new_type>(expression)
```

Clears or sets the constancy of an expression expression.

Explicit casting

```
const int a_c = 7;

int* ptr_c = const_cast<int*>(&a_c);
(*ptr_c)++;              a_c = 8

std::cout<<a_c<<std::endl;          7
std::cout << *ptr_c << std::endl;   8
```

Explicit casting

```
reinterpret_cast<new_type>(expression)
```

Allows you to convert any pointer to a pointer of any other type. Also allows you to convert any integer type to any pointer type and vice versa.

The reinterpret_cast operator does not remove constancy.

Explicit casting

```
float a = 7.777;

int* ptr0 = reinterpret_cast<int*>(&a);
int b = reinterpret_cast<int>(&a);
void* ptr1 =
reinterpret_cast<void*>(0xDEADBEEF);
```

Explicit casting

```
float a = 7.777;

int c = reinterpret_cast<int>(a);

const int* ptr2 = &b;
int* ptr3=reinterpret_cast<int*>(ptr2);
```

ERROR

Explicit casting

C-style cast

```
(new_type)expression
```

Converts expression to new_type.

Reduction of real numbers to integers occurs with the discarding of the fractional part.

Explicit casting

```
float a = 7.777;
int b = (int)a;                            b = 7

const int a_c = 7;
int* ptr0 = (int *)&a_c;
(*ptr0)++;

std::cout << *ptr0 << std::endl;      8
```

Explicit casting

```
float a = 7.777;

int c = *((int*)&a);  1090051375
```

Float and int sizes are the same. The variable c is bitwise equal to the variable a.

Explicit casting

```
int a = 0xAABBCCDD;

*(((char *)&a)+2) = 0xEE;

a = 0xAAEECCDD
```

Implicit type casting

Implicit type casting occurs in the following situations:

- Before performing the assignment;
- When comparison operators are executed, the operands are cast to the same type;
- When performing bitwise logical and operations;
- When performing arithmetic operations;

Implicit type casting

- After evaluating the expression of the switch statement, it is cast to type int;
- After evaluating the expression of the if, while, do-while and for statements, it is cast to type bool;
- Before passing an argument to a function;
- Before returning a value from a function.

Implicit type casting

```
float a = 1;                      1.0

char b = 7;
int c = b + 5;                    12

float a_f0 = 1 + 7/2;             4.0
float a_f1 = 1.0 + 7/2;           4.0
float a_f2 = 1 + 7.0/2;           4.5
```

Implicit type casting

```
char d = 0xDD;      -35
int g = d;          -35 (0xFFFFFFDD)

int k = 0xAABBCCDD;
int h = static_cast<char>(k); -35 (0xFFFFFFDD)

unsigned long long f = d + 0xCAFECAFE;
                                  0xCAFECADB
```

Implicit type casting

```
int a = 117440517;
float b = 117440520.0;
std::cout << std::setprecision(9);
if(a == b)
    std::cout << a << “==” << b << std::endl;
else
    std::cout << a << “!=” << b << std::endl;
```

Implicit type casting

```
117440517 == 117440520.0
```