C/C++ review and programming tips

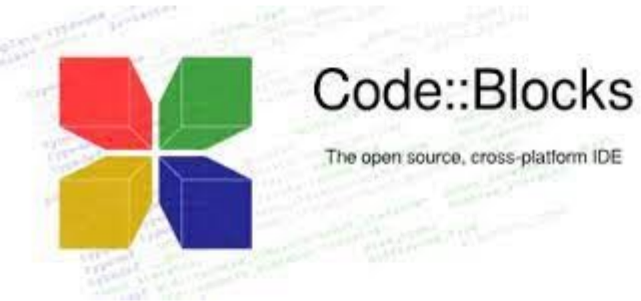
meowmeowRanger

Programming environment

- Text editor + Compiler
- IDE

Text editor + Compiler

IDE (Integrated Development Environment)



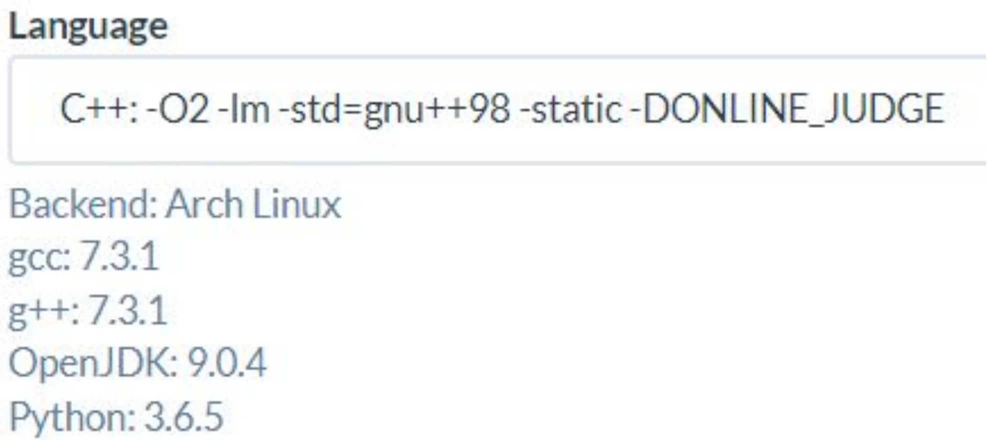
How about Visual Studio Code?

How to compile/run a code

```
$ gcc a.c

$ g++ a.cpp

$ g++ a.cpp

$ ./a.out

$ ./a.out < in > out
```

Useful Compile flag

- -std=c++17
- -Wall
- -O2
- -o



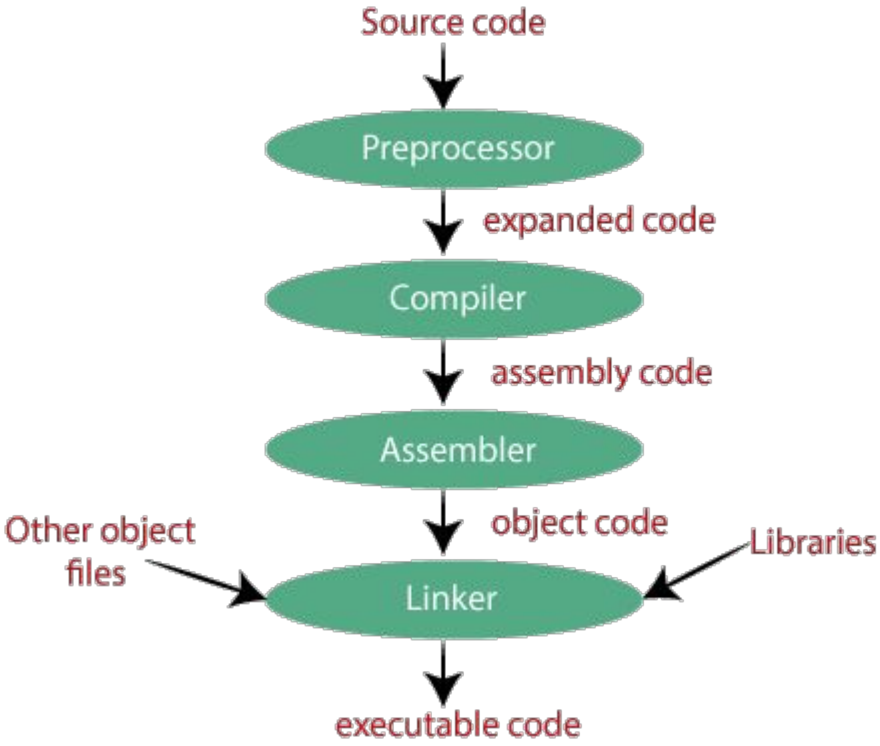
Language

C++: -O2 -lm -std=gnu++98 -static -DONLINE_JUDGE

Backend: Arch Linux
gcc: 7.3.1
g++: 7.3.1
OpenJDK: 9.0.4
Python: 3.6.5

Compiling process

Compiling process



Preprocessor

- #include
- #define
- #ifdef
- #pragma
- g++ -E

#include

```
#include <cstdio>
#include "a.h"
int main() {
    printf("%d", n);
#include "meow"
    return 0;
}
```

```
// a.h
int n = 10;
```

```
// meow
puts("meow")
```

#define

```
#include <cstdio>
#define MAXN 100
#define int long long
#define FOR(x, n) for(int x = 0; x < n; ++x)
#define MAX(x, y) (x > y ? x : y)

signed main() {
        int n; scanf("%lld", &n);
        int a[10];
        int mx = 0;
        FOR(i, n) scanf("%lld", a + i);
        FOR(i, n) mx = MAX(a[i], mx);
        printf("%lld", mx);
        return 0;
}
```

#define

```
#define EOF (-1)

#define NULL (0)
```

Pitfall of marco function

```
#define triple(x) x+x+x

printf("%d", 3 * triple(5));
```

typedef vs using vs #define

- typedef long long ll;
- using ll = long long;
- #define ll long long

Pitfall of marco function

```
#define square(x) (x * x)

printf("%d", square(2 + 3));
```

Pitfall of marco function

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
for (int i = 0; i < n; i++) {
    ans = MAX(ans, DFS(i));
}
```

#ifdef

```
#include <stdio.h>
int main() {
#ifdef DEBUG
      puts("debug mode on");
#endif
#ifndef DEBUG
      puts("debug mode off");
#endif
      return 0;
}
```

#ifdef

```
#ifndef FUNCTION_H_

#define FUNCTION_H_

......

.....

....

#endif
```

#ifdef

```
#include "test1.h"

#include "test2.h"


int main() {

        return 0;

}
```

```
// test1.h
#include "test2.h"
```

```
// test2.h
#include "test1.h"
```

#pragma

```
#include "test1.h"

#include "test2.h"


int main() {

        return 0;

}
```

```
// test1.h
#pragma once
#include "test2.h"
```

```
// test2.h
#pragma once
#include "test1.h"
```

Assembly code

- g++ -S
- Human can read it
- You’ll learn it in I2P(II) or Computer Architecture

Object code

- g++ -c
- Human can’t read it

Linking

- link every object code
- separate the definition and implementation

Separate definition and implementation

```
#include <stdio.h>
#include "print.h"

int main() {
        print();
        return 0;

}
```

```
// print.h
#pragma once
void print();
```

```
// print.c
#include <stdio.h>
void print() {
        puts("CSST");

}
```

linking

- g++ -c linking.c
- g++ -c print.c
- g++ linking.o print.o
- // alternative
- g++ linking.c print.c

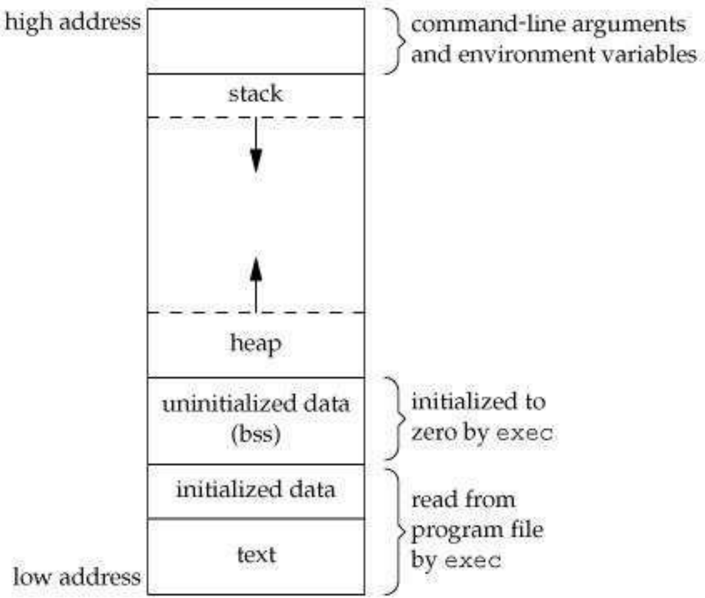
Memory & scope

Why???

```
#include <stdio.h>
// int a[1000000]; AC
int main() {
        // int a[1000000]; RE/WA
}
```

process memory layout

- stack: function call, local variable
- heap: dynamic allocate memory
- bss: uninitialize global variable


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Why???

```
#include <stdio.h>
// int a[1000000]; AC bss
int main() {
        // int a[1000000]; RE/WA stack
        // int* a = malloc(sizeof(int) *1000000) heap
}
```

How about

```
#include <stdio.h>

int main() {

    int n = 100;

    int a[n];

}
```

variable-length array

- supported by C99
- allocate the memory on stack
- It’s no valid in C++

What’s the output?

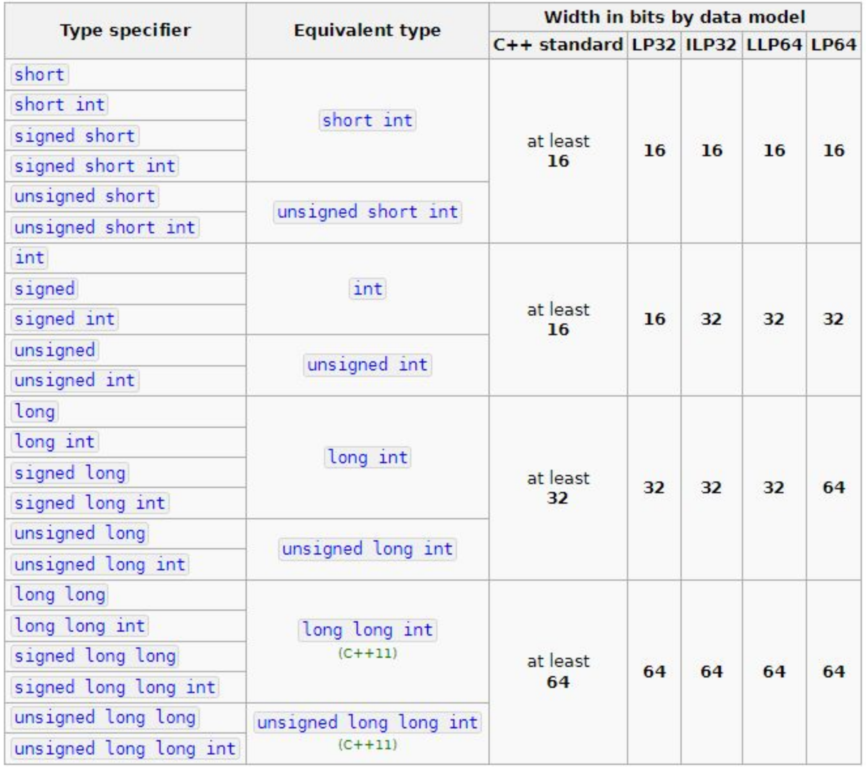
```
int i = 5;
int main() {
    int i = 3;
    for (int i = 0; i < 5; i++) {
        for (int i = 0; i < 5; i++) {
            printf("%d ", i);
        } printf("\n");
        printf("%d\n", i);
    }
    printf("%d\n", i);
}
```

Calculate the memory usage

```
int a[1000];

char c[1000];

int main() {

    for (int i = 0; i < 1000; i++)

        scanf("%c", c + i);

    return 0;

}
```

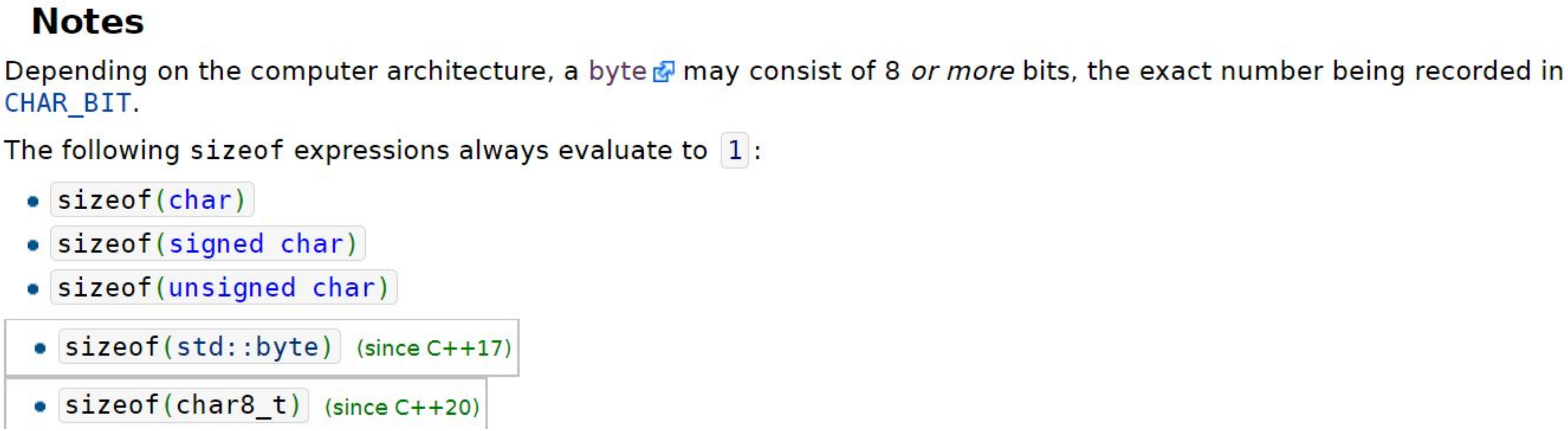
Calculate the memory usage

- sizeof(int)
- sizeof(char)
- **The number of bytes of differnt type is vary from different environment!**
- **char is always 1**



Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
short	short int	at least 16	16	16	16	16
short int						
signed short						
signed short int						
unsigned short	unsigned short int					
unsigned short int						
int	int	at least 16	16	32	32	32
signed						
signed int						
unsigned	unsigned int					
unsigned int						
long	long int	at least 32	32	32	32	64
long int						
signed long						
signed long int						
unsigned long	unsigned long int					
unsigned long int						
long long	long long int (C++11)	at least 64	64	64	64	64
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int (C++11)					
unsigned long long int						

Calculate the memory usage



Notes

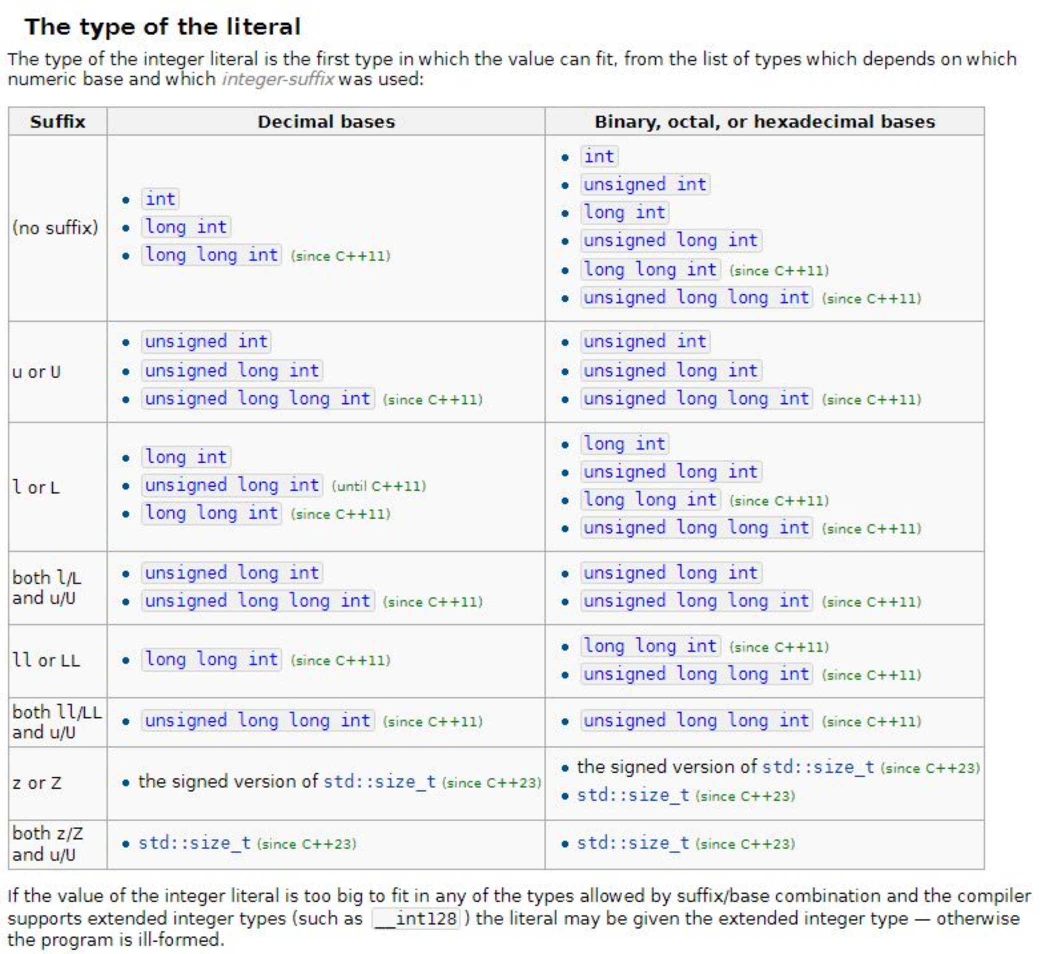
Depending on the computer architecture, a byte may consist of 8 *or more* bits, the exact number being recorded in `CHAR_BIT`.

The following `sizeof` expressions always evaluate to `1`:

- `sizeof(char)`
- `sizeof(signed char)`
- `sizeof(unsigned char)`
- `sizeof(std::byte)` (since C++17)
- `sizeof(char8_t)` (since C++20)

Expression & Statement

literal

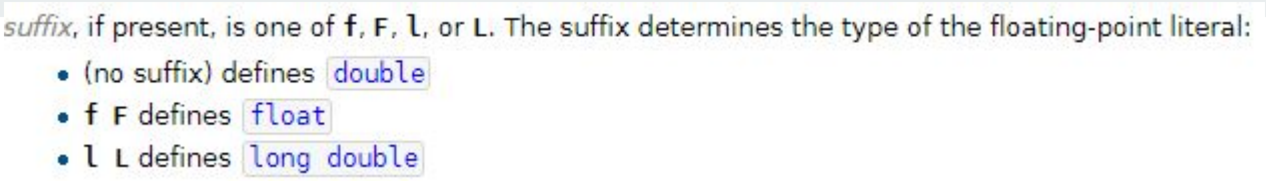


The type of the literal

The type of the integer literal is the first type in which the value can fit, from the list of types which depends on which numeric base and which *integer-suffix* was used:

Suffix	Decimal bases	Binary, octal, or hexadecimal bases
(no suffix)	• `int` • `long int` • `long long int` (since C++11)	• `int` • `unsigned int` • `long int` • `unsigned long int` • `long long int` (since C++11) • `unsigned long long int` (since C++11)
u or U	• `unsigned int` • `unsigned long int` • `unsigned long long int` (since C++11)	• `unsigned int` • `unsigned long int` • `unsigned long long int` (since C++11)
`l` or L	• `long int` • `unsigned long int` (until C++11) • `long long int` (since C++11)	• `long int` • `unsigned long int` • `long long int` (since C++11) • `unsigned long long int` (since C++11)
both `l`/L and u/U	• `unsigned long int` • `unsigned long long int` (since C++11)	• `unsigned long int` • `unsigned long long int` (since C++11)
`ll` or LL	• `long long int` (since C++11)	• `long long int` (since C++11) • `unsigned long long int` (since C++11)
both `ll`/LL and u/U	• `unsigned long long int` (since C++11)	• `unsigned long long int` (since C++11)
z or Z	• the signed version of `std::size_t` (since C++23)	• the signed version of `std::size_t` (since C++23) • `std::size_t` (since C++23)
both z/Z and u/U	• `std::size_t` (since C++23)	• `std::size_t` (since C++23)

If the value of the integer literal is too big to fit in any of the types allowed by suffix/base combination and the compiler supports extended integer types (such as `__int128`) the literal may be given the extended integer type — otherwise the program is ill-formed.

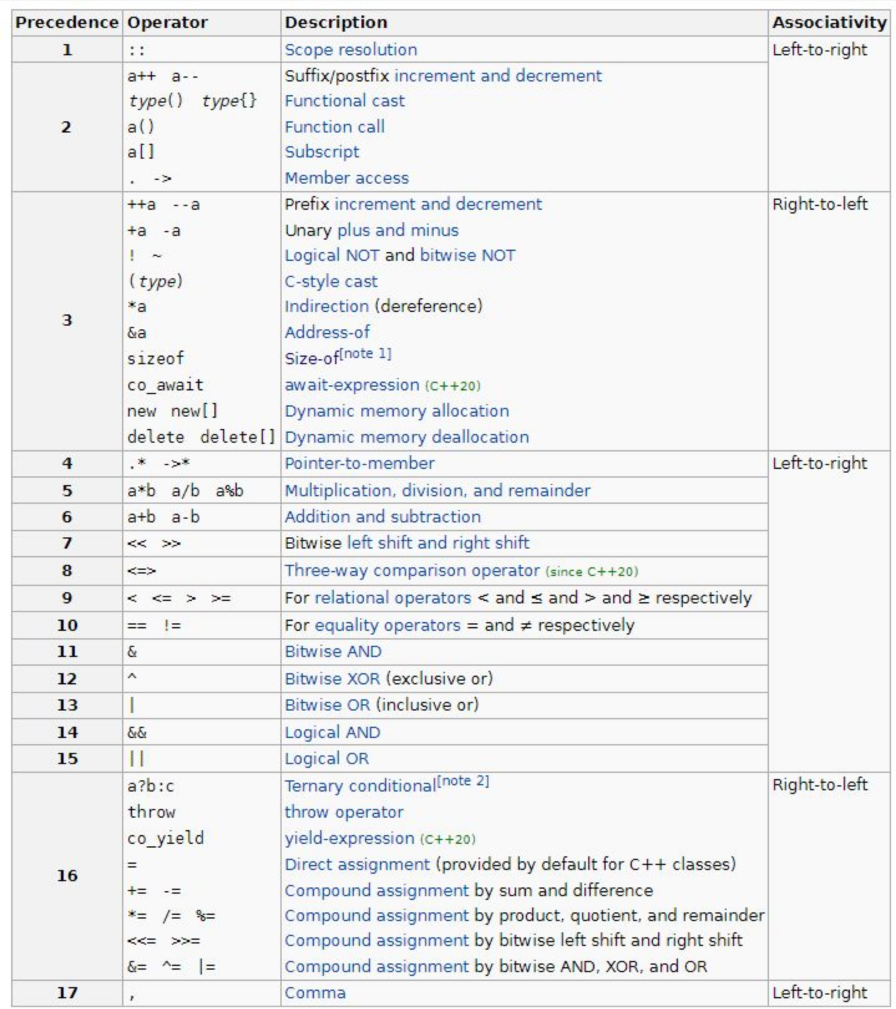


suffix, if present, is one of `f`, `F`, `l`, or `L`. The suffix determines the type of the floating-point literal:

- (no suffix) defines `double`
- `f` `F` defines `float`
- `l` `L` defines `long double`

literal

```
#include <stdio.h>
int main() {
	int bin = 0b10;
	int oct = 010;
	int hex = 0x10;
	printf("%d, %d, %d", bin, oct, hex);
	return 0;
}
```



Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a--	Suffix/postfix increment and decrement	
	type() *type*{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
	. ->	Member access	
3	++a --a	Prefix increment and decrement	Right-to-left
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of[note 1]	
	co_await	await-expression (C++20)	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	<=>	Three-way comparison operator (since C++20)	
9	< <= > >=	For relational operators < and ≤ and > and ≥ respectively	
10	== !=	For equality operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13	|	Bitwise OR (inclusive or)	
14	&&	Logical AND	
15	||	Logical OR	
16	a?b:c	Ternary conditional[note 2]	Right-to-left
	throw	throw operator	
	co_yield	yield-expression (C++20)	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= |=	Compound assignment by bitwise AND, XOR, and OR	
17	,	Comma	Left-to-right

Operation on differnt types

1. int vs float -> float
2. int vs long long -> long long
3. signed vs unsigned -> unsigned

Implicit type conversion

```
#include <iostream>
int main() {
        double x = 1 / 3;
        double x1 = 1 / 3.0;
        long long t1 = (1 << 60);
        long long t2 = (1ll << 60);
        int y = 2147483647;
        std::cout << x << ' ' << x1 << ' ' << t1 << ' ' << t2 << '\n';
        std::cout << y + 1u << '\n';
}
```

Implicit type conversion

```
int x = 10;
if (0 < x < 5) {
}
if (true < 5) {
}
if (1 < 5) {
}
```

Short-circuit evaluation

- Builtin operators && and || perform short-circuit evaluation

```
int visited[10][10];
int gx[] = {0, 1, 0, -1};
int gy[] = {1, 0, -1, -1};
for (int i = 0; i < 4; i++) {
    int to_x = cur_x + gx[i];
    int to_y = cur_y + gy[i];
    if (xx >= 0 && xx < 10 && yy >= 0 && yy < 10 && visited[xx][yy]) {
        .....
    }
}
```

floating point comparison

```
#include <stdio.h>
int main() {
        double x = 1;
        double y = 3 / 3 / 5 * 5;
        if (x == y) puts("csst");
        else  puts("no csst");
}
```

floating point comparison

```
#include <stdio.h>
const double eps = 1e-4;
int main() {
    double x = 1;
    double y = 3.0 / 3 / 5 * 5;
    if (fabs(x - y) < eps) puts("csst");
    else puts("no csst");
}
```

lvalue vs rvalue

- lvalue: an expression that yields an object reference, such as a variable name, an array subscript reference, a dereferenced pointer, or a function call that returns a reference. An lvalue always has a defined region of storage, so you can take its address.
- rvalue: not an lvalue

++i vs i++

```c
#include <stdio.h>
int main() {
        int x = 1;
        int y1 = ++x + 1;
        x = 1;
        int y2 = x++ + 1;
        printf("%d %d\n", y1, y2);
}
```

C vs C++

- C++
  - ++i returns lvalue
  - i++ returns rvlaue
- C
  - i++, ++i both return rvalue

```
#include <stdio.h>


int main() {
        int x = 1;
        ++++++++x;
        printf("%d", x);
        return 0;
}
```

Ternary operator

- C++ return lvalue
- C return rvalue

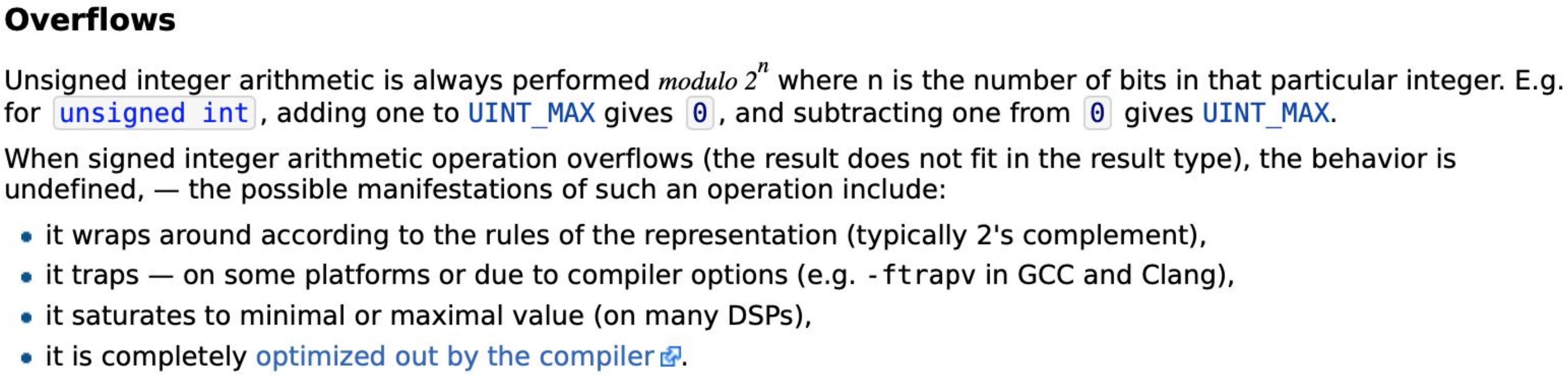
```
#include <stdio.h>


int main() {
        int l = 3, r = 5;
        (l > r ? l : r) = 10;
        printf("%d %d", l, r);
        return 0;
}
```

Undefined behavior

- behavior for which this International Standard imposes no requirements
- It’s unvalild
- The result may be unpredictable
- The standard didn’t define what will happen
- ~~It’s possible that undefined behavior cause system(“rm -rf /”)~~

Undefined behavior

```c
#include <stdio.h>

int f() {

    int x = 3 / 0;

}

int main() {

    f();

}
```



Overflows

Unsigned integer arithmetic is always performed *modulo* 2^n where n is the number of bits in that particular integer. E.g. for `unsigned int`, adding one to `UINT_MAX` gives `0`, and subtracting one from `0` gives `UINT_MAX`.

When signed integer arithmetic operation overflows (the result does not fit in the result type), the behavior is undefined, — the possible manifestations of such an operation include:

- it wraps around according to the rules of the representation (typically 2's complement),
- it traps — on some platforms or due to compiler options (e.g. `-ftrapv` in GCC and Clang),
- it saturates to minimal or maximal value (on many DSPs),
- it is completely optimized out by the compiler.

compiler optimization

```
int x;

cin >> x;

if (x + 1 > x) {

    cout << "meowmeow\n";

} else {

    cout << "cool\n";

}
```

Unspecified behavior

- For a well-formed program construct and correct data, that depends on the implementation
- The behavior is valid
- The standard accept multiple result according to the implementation

Order of Evaluation

```
#include <iostream>
int a = 1, b = 2;
int A() { return a + b; }
int B() { return ++a+b++; }
int main() {
    printf("%d %d", A(), B());
    return 0;
}
```

Implementation-defined behavior

- for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents
- The behavior is valid
- The standard accept multiple results according to the implementation
- The implementation document should tell you what will happen
- Ex: sizeof(int)

Pointer

What is the pointer

```
#include <cstdio>
int main() {
    int a = 5;
    int* b = &a;
    printf("%d %d\n", a, *b);
    a = 6;
    printf("%d %d\n", a, *b);
    *b = 7;
    printf("%d %d\n", a, *b);
}
```

name	address	value
	…	
a	0x19c28a37	5
	…	
	…	
	…	
b	0x8c7a9912	0x19c28a37

What is the pointer

```
int* a = new int (10);
```

name	address	value
	…	
None	0x19c28a37	10
	…	
	…	
	…	
a	0x8c7a9912	0x19c28a37

Array

```
#include <cstdio>
int main() {
    int a[10] = {0};
    printf("%d\n", a[5]);
    printf("%d\n", *(a + 5));
    printf("%d\n", *(5 + a));
    printf("%d\n", 5[a]);
    return 0;
}
```

Can it compile?

```
#include <cstdio>
int main() {
    int a[3][3] = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}};
    printf("%d\n", a[1][0]);
    int *b = a;
    printf("%d\n", b[3]);
    return 0;
}
```

How to fix?

```
#include <cstdio>

int main() {

    int a[3][3] = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}};

    printf("%d\n", a[1][0]);

    int (*b)[3] = a;

    printf("%p\n", b[3]);

    return 0;

}
```

Pass by value

```
#include <cstdio>
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main() {
    int x = 1, y = 2;
    swap(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```

Pass by value

```
#include <cstdio>
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main() {
    int x = 1, y = 2;
    swap(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```

Pass array to function

```
#include <cstdio>
void func(int a[10] /* int a[] or int* a*/) {
    a[0] = 1;
}
int main() {
    int a[10] = {0};
    func(a);
    printf("%d\n", a[0]);
    return 0;
}
```

Pass array to function

```
#include <cstdio>
void func(int a[10][5] /* int a[][5] or int(*a)[5] */) {
    a[0][0] = 1;
}
int main() {
    int a[10][5] = {0};
    func(a);
    printf("%d\n", a[0][0]);
    return 0;
}
```

Why scanf need &?

```
#include <cstdio>

int main() {

    int a;

    scanf("%d", &a);

    int b[10];

    scanf("%d", b + 5);

    return 0;

}
```

return a value

```
#include <cstdio>
int func() {
    int a = 5;
    return a;
}
int main() {
    printf("%d\n", func());
    return 0;
}
```

return a local variable

```
#include <cstdio>
int* func() {
    int a[3] = {1, 2, 3};
    return a;
}
int main() {
    printf("%d\n", func()[1]);
    return 0;
}
```

Why not the below one

```
struct Node {
    int val;
    Node* next;
};


struct Node {
    int val;
    Node next;
};
```

NULL vs nullptr

```
void f(int);

void f(int*);



f(NULL);

f(nullptr);
```

Some programming tips and syntax candy

At least available in C++17

Dark magic

```
#include <bits/stdc++.h>
```

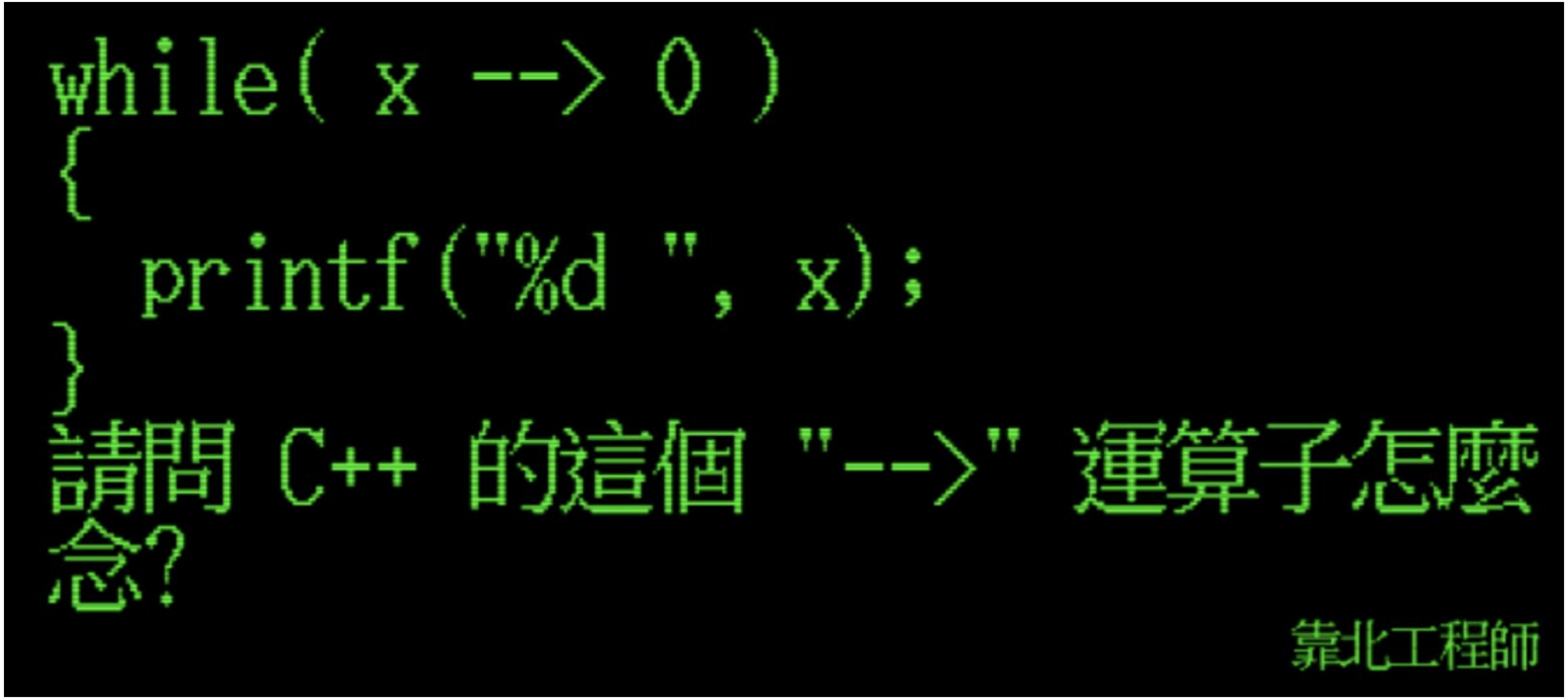
Why?

```
cin.tie(nullptr);

ios_base::sync_with_stdio(false);
```

- tie, sync_with_stdio
- why
- endl vs ‘\n’
- Side effect
  - the outputs may show when the program ends
  - Can’t mix C-style C++-style IO

Use the stuff in standard library

- Readable
- Convenient
- Avoid mistake



```
while( x --> 0 )
{
  printf("%d ", x);
}
```

請問 C++ 的這個 "-->" 運算子怎麼念?

靠北工程師

auto

```
pair<int, int> f();
auto ret = f();


vector<int> v(5);
for (auto it = v.begin(); it != v.end(); it++) {
    cout << *it << '\n';
}
```

auto - template argument deduction

```
vector<vector<int>> v(10, vector<int>(10, 0));



auto v2 = vector(10, vector(10, 0));
```

reference

```
int a = 3;

int& b = a;

b = 4;

a = 5;
```

reference

```
swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
swap(&a, &b);

swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
swap(a, b)
```

Use reference to avoid copy

```
void f(vector<int>& vec);

int main() {
  vector<int> v;
  f(v);
}
```

Range base for

```
vector<int> v(10, 7122);

for(auto it = v.begin(),ed=v.end(); it!=ed; it++) {
  int val = *it;
  ...
}
for(size_t i = 0; i < v.size(); i++) {
  int val = v[i];
  ...
}
for(int val : v) {
  ...
}
```

Range base for

```
vector<int> v(10);
for (auto it : v) {
  it = 3;
}
for (auto& it : v) {
  it = 6;
}
```

Structure binding

```
pair<int, int> f();

auto ret = f();
cout << ret.first << ' ' << ret.second;

auto [x, y] = f();
cout << x << ' ' << y << '\n';
```

Structure binding

```
struct Meow {
    int val;
    string str;
};

vector<Meow> v(3);
for (auto& [val, str] : v) {
    cin >> val >> str;
}
for (auto& [val, str] : v) {
    cout << val << ' ' << str << endl;
}
```

List-initialization

```
pair<long long, double> p = {1, 2};

vector<int> v = {1, 2, 3, 4, 5};
```

functor

```
struct functor {
    int operator() (int a) {
        return a * a;
    }
}

functor func;
cout << func(5) << '\n';
```

lambda function (an anonymous functor)

```
auto sq = [](int x) {
    return x * x;
};
cout << sq(5) << '\n';
```

lambda function (an anonymous functor)

```
vector<int> v(5);
int x = 10;
auto meow = [&v, x]() {
    for (auto& it : v) {
        it = x;
    }
};
meow();
for (auto it : v) cout << it << endl;
```

lambda function (an anonymous functor)

```
vector<int> v = {1, 5, 7, 3, 4};
sort(v.begin(), v.end(), [](int a, int b) {
        return a > b;
});
```

Other stuffs

Time-limited exceed?

- General computer can roughly excecute 1’000’000’000 basic operations in 1 second.
- Predict whether it will get TLE by yourself

Will it TLE?

```
#include <stdio.h>
#include <string.h>
int main() {
    char s[1000005];
    scanf("%s", s);
    int ans = 0;
    for (int i = 0; i < strlen(s); i++) {
        if (s[i] == 'a') ans++;
    }
    printf("%d\n", ans);
    return 0;
}
```

Use Assert to get RE

```
#include <stdio.h>
#include <assert.h>
int main() {
    int a;
    scanf("%d", &a);
    assert(1 <= a && a <= 10);
    printf("1 <= a <= 10\n");
    return 0;
}
```

How to debug

- print the value of variable to trace the code
- gdb
- Experience
- Address sanitizer
    - -fsanitize=address -g

THE END