Learn
For
Cause

DATA STRUCTURES WITH C++ INTERNSHIP TRAINING

DAY 2 - BASIC CONCEPTS OF C++

Literals :

Traditionally, a literal is an expression denoting a constant whose type and value are evident from its spelling. For example, 42 is a literal, while x is not since one must see its declaration to know its type and read previous lines of code to know its value.
However, C++11 also added user-defined literals, which are not literals in the traditional sense but can be used as a shorthand for function calls.

a) this

Within a member function of a class, the keyword this is a pointer to the instance of the class on which the function was called. this cannot be used in a static member function.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

The type of this depends on the cv-qualification of the member function: if X::f is const, then the type of this within f is const X*, so this cannot be used to modify non-static data members from within a const member function. Likewise, this inherits volatile qualification from the function it appears in.
Version ≥ C++11
this can also be used in a *brace-or-equal-initializer* for a non-static data member.

```
struct S;
struct T {
    T(const S* s);
// ...
};
struct S {
// ...
T t{this}; };
```

this is an rvalue, so it cannot be assigned to.

b) Integer literal :

An integer literal is a primary expression of the form -

decimal-literal
It is a non-zero decimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9), followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
int d = 42;

octal-literal
It is the digit zero (0) followed by zero or more octal digits (0, 1, 2, 3, 4, 5, 6, 7)
int o = 052

hex-literal
It is the character sequence 0x or the character sequence 0X followed by one or more hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F)

int x = 0x2a;int X = 0X2A;

binary-literal (since C++14)
It is the character sequence 0b or the character sequence 0B followed by one or more binary digits (0, 1)
int b = 0b101010; // C++14

Integer-suffix
if provided, may contain one or both of the following (if both are provided, they may appear in any order:
unsigned int u_1 = 42u;
long-suffix (the character l or the character L) or the long-long-suffix (the character sequence ll or the character sequence LL) (since C++11)
The following variables are also initialized to the same value:
unsigned-suffix (the character u or the character U)
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14

Notes :

Letters in the integer literals are case-insensitive: 0xDeAdBaBeU and 0XdeadBABEu represent the same number (one exception is the long-long-suffix, which is either ll or LL, never lL or Ll)
There are no negative integer literals. Expressions such as -1 apply the unary minus operator to the value represented by the literal, which may involve implicit type conversions.
In C prior to C99 (but not in C++), unsuffixed decimal values that do not fit in long int are allowed to have the type unsigned long int.
When used in a controlling expression of #if or #elif, all signed integer constants act as if they have type std::intmax_t and all unsigned integer constants act as if they have type std::uintmax_t.

c) true

A keyword denoting one of the two possible values of type bool.
bool ok = true;
if (!f()) {
ok = false;
goto end; }

d) false

A keyword denoting one of the two possible values of type bool.

e) nullptr

Version ≥ C++11
A keyword denoting a null pointer constant. It can be converted to any pointer or pointer-to-member type, yielding a null pointer of the resulting type.

Note that nullptr is not itself a pointer. The type of nullptr is a fundamental type known as std::nullptr_t.

```
bool ok = true;
if (!f()) {
ok = false;
goto end; }
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
void f(int* p);
template <class T>
void g(T* p);
void h(std::nullptr_t p);
int main() {
   f(nullptr); // ok
   g(nullptr); // error
   h(nullptr); // ok
}
```

Operator Precedence

I) Logical && and || operators: short-circuit
&& has precedence over ||, this means that parentheses are placed to evaluate what would be evaluated together.
c++ uses short-circuit evaluation in && and || to not do unnecessary executions.
If the left hand side of || returns true the right hand side does not need to be evaluated anymore.

```
#include <iostream>
#include <string>
using namespace std;
bool True(string id){
   cout << "True" << id << endl;
   return true;
}
bool False(string id){
   cout << "False" << id << endl;
   return false;
}
int main(){
   bool result;
   //let's evaluate 3 booleans with || and && to illustrate operator precedence
   //precedence does not mean that && will be evaluated first but rather where
   //parentheses would be added
   //example 1
   result =
      False("A") || False("B") && False("C");
            // eq. False("A") || (False("B") && False("C"))
   //FalseA
   //FalseB
   //"Short-circuit evaluation skip of C"
   //A is false so we have to evaluate the right of ||,
   //B being false we do not have to evaluate C to know that the result is false
   result =
      True("A") || False("B") && False("C");
            // eq. True("A") || (False("B") && False("C"))
   cout << result << " :====================" << endl;
   //TrueA
   //"Short-circuit evaluation skip of B"
   //"Short-circuit evaluation skip of C"
   //A is true so we do not have to evaluate
   //        the right of || to know that the result is true
   //If || had precedence over && the equivalent evaluation would be:
   // (True("A") || False("B")) && False("C")
   //What would print
   //TrueA
   //"Short-circuit evaluation skip of B"
   //FalseC
   //Because the parentheses are placed differently
   //the parts that get evaluated are differently
   //which makes that the end result in this case would be False because C is false
}
```

II) Unary Operators

Unary operators act on the object upon which they are called and have high precedence. (See Remarks)
When used postfix, the action occurs only after the entire operation is evaluated, leading to some interesting arithmetics:

```
int a = 1;
++a;           // result: 2
a--;          // result: 1
int minusa=-a;  // result: -1
bool b = true;
!b; // result: true
a=4;
int c = a++/2;      // equal to: (a==4) 4 / 2   result: 2 ('a' incremented postfix)
cout << a << endl;  // prints 5!
int d = ++a/2;      // equal to: (a+1) == 6 / 2 result: 3
int arr[4] =  {1,2,3,4};
int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;     // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl;  // prints  2
int e = arr[0]++;       // receives the value of arr[0] before it is incremented
std::cout << e << std::endl;      // prints 1
std::cout << *ptr2 << std::endl;  // prints arr[0] which is now 2
```

III) Arithmetic operators

Arithmetic operators in C++ have the same precedence as they do in mathematics:
Multiplication and division have left associativity(meaning that they will be evaluated from left to right) and they have higher precedence than addition and subtraction, which also have left associativity.
We can also force the precedence of expression using parentheses ( ). Just the same way as you would do that in normal mathematics.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;
//Addition:
int a = 2+4/2;
int b = (3+3)/2;
//With Multiplication
int c = 3+4/2*6;
int d = 3*(3+6)/9;
//Division and Modulo
int g = 3-3%1;
int h = 3-(3%1);
// equal to: 2+(4/2)
// equal to: (3+3)/2
// equal to: 3+((4/2)*6)
// equal to: (3*(3+6))/9
```

```
// equal to: 3 % 1 = 0
// equal to: 3 % 1 = 0
    result: 4
    result: 3
    result: 15
    result: 3
3 - 0 = 3 3 - 0 = 3
int i = 3-3/1%3;
int l = 3-(3/1)%3;
int m = 3-(3/(1%3));
// equal to: 3 / 1 = 3
// equal to: 3 / 1 = 3
// equal to: 1 % 3 = 1
3 % 3 = 0
3 % 3 = 0 3 / 1 = 3
3 - 0 = 3
3 - 0 = 3 3 - 3 = 0
```

IV) Logical AND and OR operators

These operators have the usual precedence in C++: AND before OR.
This code is equivalent to the following:
Adding the parenthesis does not change the behavior, though, it does make it easier to read. By adding these parentheses, no confusion exist about the intent of the writer.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;

// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

Floating Point Arithmetic

The first mistake that nearly every single programmer makes is presuming that this code will work as intended:
The novice programmer assumes that this will sum up every single number in the range 0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99, to yield the result 199—the mathematically correct answer.
Two things happen that make this untrue:
1. The program as written never concludes. a never becomes equal to 2, and the loop never terminates.
2. If we rewrite the loop logic to check a < 2 instead, the loop terminates, but the total ends up being **something different from 199. On IEEE754-compliant machines, it will often sum up to about 201 instead. The reason that this happens is that Floating Point Numbers represent Approximations of their assigned**
values.
The classical example is the following computation:

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
total += a; }
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
   //This never prints on IEEE754-compliant machines
   std::cout << "This Computer is Magic!" << std::endl;
else
   std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Though what we the programmer see is three numbers written in base10, what the compiler (and the underlying hardware) see are binary numbers. Because 0.1, 0.2, and 0.3 require perfect division by 10—which is quite easy in a base-10 system, but impossible in a base-2 system—these numbers have to be stored in imprecise formats, similar to how the number 1/3 has to be stored in the imprecise form 0.333333333333333... in base-10.
//64-bit floats have 53 digits of precision, including the whole-number-part.

```
double a = 0011111110111001100110011001100110011001100110011001100110011010; //imperfect representation of 0.1
double b = 0011111111001001100110011001100110011001100110011001100110011010; //imperfect representation of 0.2
double c = 0011111111010011001100110011001100110011001100110011001100110011; //imperfect representation of 0.3
double a + b = 0011111111010011001100110011001100110011001100110011001100110100; //Note that this is not quite equal to the "canonical" 0.3!
```

Bit Operators Section | - bitwise OR

A bit wise OR operates on the bit level and uses the following Boolean truth table:
When the binary value for a (0101) and the binary value for b (1100) are OR'ed together we get the binary value of 1101:
The bit wise OR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator |=:

bitwise XOR (exclusive OR)

A bit wise XOR (exclusive or) operates on the bit level and uses the following Boolean truth table:

```
int a = 5;     // 0101b  (0x05)
int b = 12;    // 1100b  (0x0C)
int c = a | b; // 1101b  (0x0D)
std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

true OR true = true

```
true OR false = true
false OR false = false
int a = 0 1 0 1
int b = 1 1 0 0 |
        ---------
int c = 1 1 0 1
int a = 5;  // 0101b  (0x05)
a |= 12;    // a = 0101b | 1101b
int a = 5;     // 0101b  (0x05)
int b = 9;     // 1001b  (0x09)
int c = a ^ b; // 1100b  (0x0C)
std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
true OR true = false
true OR false = true
false OR false = false
```

Notice that with an XOR operation true OR true = false where as with operations true AND/OR true = true, hence the exclusive nature of the XOR operation.
Using this, when the binary value for a (0101) and the binary value for b (1001) are XOR'ed together we get the binary value of 1100:
The bit wise XOR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator ^=:

The bit wise XOR can be utilized in many ways and is often utilized in bit mask operations for encryption and compression.

Note: The following example is often shown as an example of a nice trick. But should not be used in production code (there are better ways std::swap() to achieve the same result).
You can also utilize an XOR operation to swap two variables without a temporary:

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
        ---------
int c = 1 1 0 0
int a = 5;  // 0101b  (0x05)
a ^= 9;    // a = 0101b ^ 1001b
int a = 42;
int b = 64;
// XOR swap
a ^= b;
b ^= a;
a ^= b;
std::cout << "a = " << a << ", b = " << b << "\n";
```

To productionalize this you need to add a check to make sure it can be used.

```
void doXORSwap(int& a, int& b)
```

```
{
   // Need to add a check to make sure you are not swapping the same
   // variable with itself. Otherwise it will zero the value.
   if (&a != &b)
   {
// XOR swap
       a ^= b;
       b ^= a;
       a ^= b;
} }
```

So though it looks like a nice trick in isolation it is not useful in real code. xor is not a base logical operation,but a combination of others: a^c=~(a&c)&(a|c)

```
 int cn=0b0111;
```

& - bitwise AND

```
int a = 6;     // 0110b  (0x06)
int b = 10;    // 1010b  (0x0A)
int c = a & b; // 0010b  (0x02)
std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

a = 6, b = 10, c = 2

A bit wise AND operates on the bit level and uses the following Boolean truth table:
When the binary value for a (0110) and the binary value for b (1010) are AND'ed together we get the binary value of 0010:
The bit wise AND does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator &=:

<< - left shift

The left bit wise shift will shift the bits of the left hand value (a) the number specified on the right (1), essentially padding the least significant bits with 0's, so shifting the value of 5 (binary 0000 0101) to the left 4 times (e.g. 5 <<
4) will yield the value of 80 (binary 0101 0000). You might note that shifting a value to the left 1 time is also the same as multiplying the value by 2, example:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
int a = 0 1 1 0
int b = 1 0 1 0 &
       ---------
int c = 0 0 1 0
```

```
int a = 5;  // 0101b  (0x05)
a &= 10;    // a = 0101b & 1010b
int a = 1;      // 0001b
int b = a << 1; // 0010b
std::cout << "a = " << a << ", b = " << b << std::endl;
int a = 7;
while (a < 200) {
   std::cout << "a = " << a << std::endl;
   a <<= 1;
}
a = 7;
while (a < 200) {
   std::cout << "a = " << a << std::endl;
a *= 2; }
```

But it should be noted that the left shift operation will shift *all* bits to the left, including the sign bit, example:
Possible output: a = 2147483647, b = -2

While some compilers will yield results that seem expected, it should be noted that if you left shift a signed number so that the sign bit is affected, the result is undefined. It is also undefined if the number of bits you wish to shift by is a negative number or is larger than the number of bits the type on the left can hold, example:

The bit wise left shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator <<=:

>> - right shift

The right bit wise shift will shift the bits of the left hand value (a) the number specified on the right (1); it should be noted that while the operation of a right shift is standard, what happens to the bits of a right shift on a *signed negative* number is *implementation defined* and thus cannot be guaranteed to be portable, example:
It is also undefined if the number of bits you wish to shift by is a negative number, example:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;     // 1111 1111 1111 1111 1111 1111 1111 1110
std::cout << "a = " << a << ", b = " << b << std::endl;
int a = 1;
int b = a << -1;  // undefined behavior
char c = a << 20; // undefined behavior
int a = 5;  // 0101b
a <<= 1;    // a = a << 1;
int a = 2;      // 0010b
int b = a >> 1; // 0001b
```

```
std::cout << "a = " << a << ", b = " << b << std::endl;
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
int a = 1;
int b = a >> -1;  // undefined behavior
```

The bit wise right shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator >>=:

```
int a = 2;  // 0010b
a >>= 1;    // a = a >> 1;
```

Bit Manipulation

Remove rightmost set bit
Explanation

if n is zero, we have 0 & 0xFF..FF which is zero
else n can be written 0bxxxxxx10..00 and n - 1 is 0bxxxxxx011..11, so n

Set all bits
C-style bit-manipulation

& (n -
1) is 0bxxxxxx000..00.

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
   // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
unsigned"); // For c++11 and later
   return n & (n - 1);
}
x = -1; // -1 == 1111 1111 ... 1111b
```

(See here for an explanation of why this works and is actually the best approach.)

Using std::bitset

Toggling a bit
C-style bit-manipulation
A bit can be toggled using the XOR operator (^).
Using std::bitset

Checking a bit
C-style bit-manipulation

```
std::bitset<10> x;
x.set(); // Sets all bits to '1'

// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip();  // num is now 1110 (flips all bits)
```

The value of the bit can be obtained by shifting the number to the right x times and then performing bitwise AND (&) on it:

```
 (number >> x) & 1LL;  // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

The right-shift operation may be implemented as either an arithmetic (signed) shift or a logical (unsigned) shift.

number in the expression number >> x has a signed type and a negative value, the resulting value is implementation-defined.

If we need the value of that bit directly in-place, we could instead left shift the mask:

(number & (1LL << x)); // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise Either can be used as a conditional, since all non-zero values are considered true.

Using std::bitset

Counting bits set

The population count of a bitstring is often needed in cryptography and other applications and the problem has been widely studied.
The naive way requires one iteration per bit:
A nice trick (based on Remove rightmost set bit) is:

It goes through as many iterations as there are set bits, so it's good when value is expected to have few nonzero bits.
The method was first proposed by Peter Wegner (in CACM 3 / 322 - 1960) and it's well known since it appears in *C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1);  // bit_val value is set to true;
unsigned value = 1234;
unsigned bits = 0;  // accumulates the total number of bits set in `n`
for (bits = 0; value; value >>= 1)
  bits += value & 1;
unsigned bits = 0;  // accumulates the total number of bits set in `n`
```

```
for (; value; ++bits)
  value &= value - 1;
unsigned popcount(std::uint64_t x)
{
  const std::uint64_t m1  = 0x5555555555555555;  // binary: 0101...
  const std::uint64_t m2  = 0x3333333333333333;  // binary: 00110011..
  const std::uint64_t m4  = 0x0f0f0f0f0f0f0f0f;  // binary: 0000111100001111
  x -= (x >> 1) & m1;             // put count of each 2 bits into those 2 bits
  x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
  x = (x + (x >> 4)) & m4;        // put count of each 8 bits into those 8 bits
  return (x * h01) >> 56;  // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

This kind of implementation has the best worst-case behavior (see Hamming weight for further details). Many CPUs have a specific instruction (like x86's popcnt) and the compiler could offer a specific (**non standard**)

built in function. E.g. with g++ there is:

```
 int __builtin_popcount (unsigned x);
```

Check if an integer is a power of 2

The n & (n - 1) trick (see Remove rightmost set bit) is also useful to determine if an integer is a power of 2: bool power_of_2 = n && !(n & (n - 1));
Note that without the first part of the check (n &&), 0 is incorrectly considered a power of 2.

Setting a bit

A bit can be set using the bitwise OR operator (|).
Using std::bitset
set(x) or set(x,true) - sets bit at position x to 1.

Clearing a bit

A bit can be cleared using the bitwise AND operator (&).
Using std::bitset
reset(x) or set(x,false) - clears the bit at position x.

Changing the nth bit to x
C-style bit-manipulation Using std::bitset

set(n,val) - sets bit n to the value val.

```
// Bit x will be set
number |= 1LL << x;
```

```
std::bitset<5> num(std::string("01100"));
num.set(0);      // num is now 01101
num.set(2);      // num is still 01101
num.set(4,true); // num is now 11110
// Bit x will be cleared
number &= ~(1LL << x);
std::bitset<5> num(std::string("01100"));
num.reset(2);     // num is now 01000
num.reset(0);     // num is still 01000
num.set(3,false); // num is now 00000
// Bit n will be set if x is 1 and cleared if x is 0.
number ^= (-x ^ number) & (1LL << n);
```

```
std::bitset<5> num(std::string("00100"));
num.set(0,true);  // num is now 00101
num.set(2,false); // num is now 00001
```

Bit Manipulation Application: Small to Capital Letter

One of several applications of bit manipulation is converting a letter from small to capital or vice versa by choosing a **mask** and a proper **bit operation**. For example, the **a** letter has this binary representation 01(1)00001 while its capital counterpart has 01(0)00001. They differ solely in the bit in parenthesis. In this case, converting the **a** letter from small to capital is basically setting the bit in parenthesis to one. To do so, we do the following:

```
/**************************************
convert small letter to captial letter.
======================================
    a: 01100001
 mask: 11011111  <-- (0xDF)  11(0)11111
    :---------
a&mask: 01000001  <-- A letter
**************************************/
```

The code for converting a letter to A letter is

```
#include <cstdio>
int main() {
   char op1 = 'a';  // "a" letter (i.e. small case)
   int mask = 0xDF; // choosing a proper mask
printf("a (AND) mask = A\n");
printf("%c & 0xDF = %c\n", op1, op1 & mask);
return 0; }
```

The result is

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a   &   0xDF = A
```

Bit fields

Bit fields tightly pack C and C++ structures to reduce size. This appears painless: specify the number of bits for members, and compiler does the work of co-mingling bits. The restriction is inability to take the address of a bit field member, since it is stored co-mingled. sizeof() is also disallowed.
The cost of bit fields is slower access, as memory must be retrieved and bitwise operations applied to extract or modify member values. These operations also add to executable size.

Declaration and Usage

Here, each of these two fields will occupy 1 bit in memory. It is specified by **: 1** expression after the variable names. Base type of bit field could be any integral type (8-bit int to 64-bit int). Using unsigned type is recommended, otherwise surprises may come.
If more bits are required, replace "1" with number of bits required. For example:
The whole structure is using just 22 bits, and with normal compiler settings, sizeof this structure would be 4 bytes. Usage is pretty simple. Just declare the variable, and use it like ordinary structure.

```
struct FileAttributes
{
   unsigned int ReadOnly: 1;
   unsigned int Hidden: 1;
};
struct Date {
   unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time
   unsigned int Month: 4;  // 2^4 = 16, enough to represent 1-12 month values.
   unsigned int Day:   5;  // 32
};
Date d;
d.Year = 2016;
d.Month = 7;
d.Day =  22;
std::cout << "Year:" << d.Year << std::endl <<
       "Month:" << d.Month << std::endl <<
       "Day:" << d.Day << std::endl;
```