Contents

Java vs C++

https://softwareengineering.stackexchange.com/questions/269447/will-a-profound-knowledge-of-c-help-you-in-learning-other-languages-faster-eas

Learning C++ before Java is more beneficialy because if you know the WHY"s (how something is done at a fundamental level) then you have the roots to grow.

Java is different from C++ from three ways:

1. Automated memory management
2. Simplified syntax and no preprocessor
3. Java standard libraries

Deep Understanding of Syntax

<iostream> Here iostream is a library. COUT and CIN are OBJECTS. The cout stands for "console out"

0.1. Summary « is an insertion operator
:: is a scope resolution
= is an "ASSIGNMENT OPERATOR"
== is a "comparison operator"
< is less than
5 < 3 five is less than 3
> is greater than
2 > 1
2 is greater than 1

Program is set of two ingredients:

1. data
2. operations performed on data

1 byte = 8 bits
Maximum integer in 1 byte (8 bits) is 255.
Formula for maximum binary value for N bits

0.2. Formula to calculate maximum decimal value for N bits. Our standard equation

$$M = 2^N - 1$$

Example:

$$255 = 2^8 - 1$$

25. Primitive Data Types

How to memorize range for 0.2 floats and 0.2 doubles
Realize that doubles have an extra **zero** in the power
floats are 10^{38} and doubles are 10^{308}

Float range: -3.4×10^{-38} to 3.4×10^{38}
Double range: -1.7×10^{-308} to 1.7×10^{308}
multiply double's range and get float's number;
$1.7 \times 2 = 3.4$ Other data types: 1. long 2.long long (for really long decimals)

0.3. Proving the int range is -32,768 -to 32,767

This is our array for 2 bytes (16 bits)
Assume the numbers inside each index are it's index value
Notation: Most significant byte (MSB), Least significant byte (LSB)
-ve stands for negative +ve stands for positive

Known: 1 byte = 8 bits.
If we have two bytes, we will have 16 bits. Let's assume that our int data type takes in two bytes (in some old compiler).

1. Assume 2 bytes (16 bits)

2. Most significant bit (MSB) reserves the first spot for the sign; In other

Data Type	Size (bytes)	Range
int	**2 or 4**	**-32,768 -to 32,767**
float	4	-3.4×10^{-38} to 3.4×10^{38}
double	8	-1.7×10^{-308} to 1.7×10^{308}
char	**1**	**-128 to 127**
bool	undefined	true/false

16 15

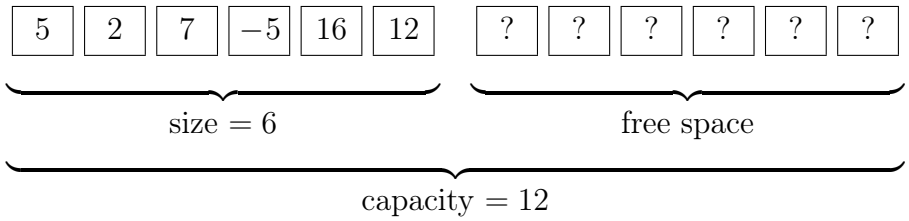
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

words, MSB takes one spot for a (1) -ve or (0) +ve

3. Excluding the MSB, there are 15 bits left for the integer.
4. Therefore $2^{15} = 32768$
   (a) $32,768$ is the maximum number from the *range 1 to 32,768*
   (b) So, our int data type can take in a max number, 32,768.
5. If we want a range starting from 0, our range is now *range of 0 to 32,767*
6. Range visually represented as: 0 ... 32,767
7. Reminder: we know that one bit is reserved at MSB (first index) for positive and negative signs.
8. So if we include the negative side from -32767 to -0, our range looks like this
9. -32,767 -0 0 32,767
10. But realize that -0 does not exist; we add one more value to the negative side to account for our -0
11. -32,767 + 1 0 32,767
12. -32,768 0 32,767
13. Therefore our int data type range is from *-32,768 to 0 to 32,767*

Question at step #9: How do we have enough memory for the negative side if our maximum number is 32,768?
Answer: Look at two's complement; when we reach our one more than maximum value of 32,768, (e.g. 32,769), we actually overflow and get -1.
Two's Complement

In practice, it looks like this:



0.4. Proving the char range is -128 to 127

1. The data type, char, is 1 byte or 8 bits.
2. We reserve the first bit (MSB) for the positive and negative signs.
3. Therefore, we are left with 7 bits.
4. So we have $2^7 = 128$
   (a) our values for char is the range *1 to 128*
5. When we begin at 0, our range becomes *0 to 127*
6. To account for the negative side, we have *-127 -0 0 127*
7. To fix the -0, we add one more value to our negative side, our final range is *-127 0 127*

This had many similiar to our int range proof.

For unsigned(only positive) integers, add back the first bit that stores the negative and positive values. Aka. use all 16 bits (2 bytes in integer data type) One way:

1. Instead of $2^{15} = 32,768$, we will have $2^{16} = 65,536$.
2. With a range starting from 0, *0 to 65,535*
3. Therefore, unsigned int has a range from *0 to 65,535*

Other way: Add the min and max values of an int range *-32,768 to 0 to 32,767*

1. $32spoq, 768 + 32,767 = 65,535$
2. Therefore, unsigned int has a range from *0 to 65,535*

0.5. Remember these ASCII Codes a = 97 ... z = 122; 0 -> 49 ... 9 -> 57

A = 65 a = 97 '0' = 48
B = 66 b = 98 '1' = 49
.
.
.
Z = 90

Data Types

Data Type	
(15 bits used with 1 reserved) char	-128 to 127 (8 bits used with 1 reserved) unsigned int

0.6. Division with data types Integer division / int / int works float / int will result in a float

NOTE: % can only divide with modular int % int works

0.7. Operator precedence Operator associativity Associativity is the grouping of the first operation to take place If two operators have the same precedence, then associativity would be left to right. For example, + and - have the same precedence.

0.8. Random Numbers Gaddis Textbook - Understanding random numbers include two standard libraries randomize with srand()

MIN and MAX values

```
const int MIN_VALUE = 1;
const int MAX_VALUE = 10;
y = (rand() \% (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE);
```

1. $MAX_VALUE - MIN_VALUE + 1$ in our equation gets the number of integers within our range.
2. y = (rand()
3. If our max value was 11, we will have a remainder of 1;
4. (rand()
5. this part of the equation is equivalent of checking if an integer is divisible by some value
6. e.g Checking if an integer is even. (someInt % 2 == 0)
7. We add MIN_VALUE because

30. Sum of all Natural Numbers N

$sum = n * (n + 1)/2$

31. Finding the roots of a quadratic equation
$ax^2 + bx + c = 0$
Polynomic power of two means it's quadratic
Root equation to find out the possible values of x
$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

35. Increment and Decrement

y = ++x 1. First Increment 2. Then ASSIGNMENT
Undeclared and Uninitialized variables are random

```
int my_variable; % this is random
```

0.9. Prefix vs Postfix Prefix assigns before incrementing ++x Postfix increments then assigns x++

```
 int x = 10,y;
y = x++; // before you increment, you assign (y = 10)
cout << y << endl;


int a = 10, b;
b = ++a;
cout << b << endl;
```

0.10. 39. Bitwise Operators XOR (x-or) (exclusive or)
both must be exclusive (one is 1 and other is 0) –> 1
if both are the same –> 0

```
    // int x=5, z = 10, y;
    // // IDEA: multiply these x and z together and
       increment x, but we want the initial value of x (
       start at 5)
    // y = ++x * z;

    // cout << y << endl;

    // int a=5,b;

    // b = a++;
```

```
    // BITWISE OPERATORS (electronics)
    // char x = 11, y = 5, z;
    // z = x << 1;
    // cout << (int) z << endl;
    day d;
    d = tues;
    cout << d << endl;
```

0.11. Bitwise Operations Finding the value of

```
int z = 5;
cout << (~z);
// Calculating the value of ~z
// 00000110 --> 5

// 11111001 --> ~5
//+        1
// --------
// 00000110--> Two's Complement
//
```

```
//     int n = 1;
//     int fact = 1;
//     for (int i = 1; i <= n; i++)
//     {
//         fact *= i;
//     }
//     while (n>0)
//     {
//         fact=fact*n;
//         n = n-1;
//     }
//     cout << fact << endl;
//
//     cout << 6*5*4*3*2*1 << endl;
```

```
    #include <iostream>
using namespace std;

enum day {mon=3, tues, wed, thurs, fri, sat, sun}; // these
    are constants

int main() {
  typedef int marks;

  marks m1, m2, m3;
```

```
|| }
```

For Loops

```
    // initialize with values
    int A[] = {0,1,2,3,4,5};

    cout << A << endl;
    // iterating for each element in x;iterates the entire
        array

    for(auto x:A) cout <<  x << endl; // assigns each value
         in A to the variable x

    // For each loop vs for loop | for each loop doesn't
        need to know length of array, it's automatic
    for(int x: A)

    vs


    }

% CODE
```

```
int main() {
    int A[] = {8,6,3,9,7,4};
    cout << A << endl; // prints out memory address

    for(auto x:A) cout <<  x << endl; // assigns each value
         in A to the variable x

    for (auto x : A)
    {
        // cout << ++x << endl;
        cout << x + 2 << endl; // NOTE: x is a copy of the
            variables inside A. the variables inside A are
            not changed unless we use a reference &
```

```
        // for (auto &x : A)
        cout << x << endl;
    }
```

```
        int number = 6;
    int x = 0;
    x = number--;
    cout << x << endl;

        int number = 6;
    int x = 0;
    x = --number;
    cout << x << endl;
}
```

```
#include <iostream>
#include <chrono>
using namespace std;

int main() {
    int A[] = {8,6,3,9,7,4};

    // FIND THE MAXIMUM NUMBER IN THE ARRAY
    int max = A[0]; // first item on our array
    for (int i=1;i<6;i++) {
        if (max < A[i])
        {
            max = A[i];
            cout  << "Found larger than max " << max << "
                \n";
        }
    }
```

```
//     for(int i=0;i<5;i++)
//         {
//         for(int j = 0; j < i; j++)
//             {
//             cout<<" * ";
//             }
//             cout << "p";
//             cout<<endl;
//         }
```

Searching

```
  #include <iostream>
#include <chrono>
using namespace std;

int main() {
    int A[] = {8,6,3,9,7,4};

    // LINEAR SEARCH

    // int key = 393;
    // for (int i=0; i<4; i++)
    // {
    //     if (key == A[i])
    //     {
    //         cout << "Found the key at index " << i;
    //         exit(0); // want to exit after you found the
        key
    //     }
    // }
    // cout << "Couldn't find the key";
```

```
    #include <iostream>
#include <chrono>
using namespace std;

int main() {
    int A[] = {6, 8, 13, 17, 20, 22, 25, 28, 30 ,55};
    int max = INT_MIN;
    cout << max << endl;
    // Binary Search: MUST BE SORTED LIKE A BOOK
```

```
    // int key = 30;
    // int first = 0, last = 9;
    // // IDEA: find a number by like searching a book
    // // Search low, modify right hand high side; search
       high, modify low
    // // Step 1: Go into the middle
    // int middle = (first + last) / 2;
    // if (middle == key) {
    //     cout << "Found the key";
    // }
    // else if (key > A[middle]) {
    //     first = middle;
    //     middle = (first + last) / 2;
    // }

    // else if (key < middle) {
    //     last = middle;
    //     middle = (first + last) / 2;
    // }
    }
```

108. Pointers

Stack vs Heap
Use the stack when your variable will not be used after the current block is finished
Use the heap when the data in the variable is needed beyond the lifetime of the current function.

Why do we use pointers? In our regular main program. our scope is in our current program and stack. Stack variables are deleted automatically out of the scope. We can use pointers to access the heap memory. Heap memory must be deallocated manually.

0.19. 109. Pointers How can we access the heap memory?
example Initializing an array puts it inside the stack and using the keyword new puts it inside the heap

```
int A[] = {6, 8, 13, 17, 20, 22, 25, 28, 30 ,55}; // Stack

int *p; // Heap
p = new int[5]
OR

int *p = new int[5] // Heap
```

new means memory allocated to the heap

Whatever is in the heap, stays in the heap (as long as the program stays running). Things inside the stack will be deleted after that block has finish running HEAP MEMORY MUST BE DEALLOCATED!!! (This is called a memory leak because the memory is not being used, but still using memory)
delete []p

```
int x = 10;
int *p;
      p = &x; // assigns address of x
  cout << "The value of x is " << x << endl;
      cout << "The memory location of x is " << &x << endl;
      cout << "The value of p is " << p << endl; // some
          address
      cout << "The memory location of p is " << &p << endl;
      cout << "The value of pointer p is " << *p << endl;
          //what the value of the address of x is.
```

```
        int a = 3;
        int abe; // declared
        if (false)
        {
                int abe = 10;
                abe = 3;
        }
        cout << abe << endl;
```

```
int *p = new int [5]; // creates a (variable) POINTER p,
    that points to a NEW ARRAY in the HEAP
p[0] = 12;

delete []p; // delete array in memory
p = nullptr; // get rid of pointer; modern C++ uses nullptr
    , not NULL
cout << p[0];
```

0.22. Changing dynamic array size during a run.

D

```
// when you want a new array with a new size

int *p = new int[10]; // array on the heap; dynamic array
    with a pointer p
for (int i =0; i<10; i++)
{
        p[i] = i;
        cout << p[i] << " ";
}


delete []p; // want to make array larger; delete array;
    avoid memory leaks
```

```
p = new int[20]; // makes a new array with size 500 with
    existing pointer p (the arrow of pointer p is changed.
    now points to a different array)

for (int i =0; i<20; i++)
{
        p[i] = i;
        cout << p[i] << " ";
}
```

0.23. 111. Pointer Arithematic, 5 operations

```
int A[5] = {2, 4, 6, 8, 10}  \\
int *p = A; // p is pointing in array A
int *q = &A[3] // pointer q is pointing to address of index 3 in array A
```

Assume p is pointing to an array; There are five operations.

1. p++; // Goes to next element in array
2. p–; // Goes to previous element in array
3. p = p + 2; // Goes two more elements from current position
4. p = p - 2;
5. d = q - p; // address of position q - address of position A

```
int A[]= {2,4,6,8,10,12};
//    int *q = A

int *p = A; // the pointer starts at the first memory
    location (index 0)
// Pointer currently is at A[0]

// move pointer to next location to print 4
// p = &A[1];
p++; // Pointer is currently at A[1]
cout<<*p;

p=p+3; // pointer will be pointing on 10
// Pointer is currently at A[4]
cout<< p[-4];    // print 2 without moving pointer
```

0.25. Three Pointers Problems Pointers may pass the compile phase, but problems may appear during runtime Common Problems

1. Uninitialized Ptr (pointer is some random address)

   ```
   int *p;
   ```

   Ways to solve this problem‘

   (a) Existing address,

   (b) New address’

   (c) to the heap

0.26. Reference (r-value vs l-value) Reference does not consume any memory at all. r-value: variable on right hand side means,assigning data Example:

```
a = x;
```

x is on the right-hand side. The **data**value of x is assigned to a. l-value: variable on the left hand side means, assigning address Example:

```
x = 25;
```

x is on the left-hand side. The x is giving the location where the value 25 should be stored. Or the value 25 is being assigned to the address of x;

0.27. Functions

0.27.1 Function Overloading

```
int add(int x, int y) {
return x + y;
}

int add(int x, int y, int z) {
return x + y + z;
}

float add(float x, float y) {
return x + y;
}
```

```
int main(int argc, const char * argv[]) {
int a = 10, b = 15, c;
c = add(a,b);
cout << c << endl;
int myints = add(1,2);
cout << myints << endl;
cout << add(12.9f,8.3f) << endl;
}
```

```
int add(int x, int y) {
        return x + y;
}

int add(int x, int y, int z) {
        return x + y + z;
}

float add(float x, float y) {
        return x + y;
}
```

Let's call our int add() function.

```
int a = 10, b = 15, c;
c = add(a,b);
cout << c << endl;
```

Let's call our float add() function

```
cout << add(1.5f, 1.5f) << endl;
```

By default, the parameters will data type double. This is why we have the float notation (f) after the number. Assume we are calling add(10.5,10.5). This is an ERROR because 10.5 is recognized as a double

```
float add(float x, float y) {
        return x + y;
}
```

0.27.2 139. Function Templates

```
//write a Max() function template for 2 numbers
template<class T>
T Max(T x, T y)
{
        if (x > y) return x;
        else return y;
}
```

Now we call our maxim with data type int

```
 cout << maxim(6 ,6) << " ";
```

0.27.3 144. Call by Address

```
void swap(int* a, int* b) { // *a is address of x; call by
    address
        int temp = *a; // *a = pointer to value of x
        *a = *b; // *a is value deferences the pointer
        *b = temp; //
        cout << a << " " << b << " \n";
        cout << *a << " " << *b << " \n";


}

int main(int argc, const char * argv[]) {
        int x = 10, y = 20;
        swap(&x, &y);
        cout << x << " " << y << " \n";
        cout << &x << " " << &y << " ";
}
```

0.27.4 144. Call by Reference

function is NOT called, but replaced in the main function (in-line function) Beause the function is replaced by the function call, avoid doing these: no complex logic using call by reference avoid using loops with call by reference

```
void swap(int &a, int &b) { // *a is address of x; call by
    address
int temp = a; // *a = pointer to value of x
a = b; // *a is value deferences the pointer
b = temp; //
cout << a << " " << b << " \n";
//    cout << *a << " " << *b << " \n";
cout << &a << " " << &b << " \n";
for (int i=0; i<3;i++) cout << "hi";


}

int main(int argc, const char * argv[]) {
int x = 10, y = 20;
swap(x, y);
cout << x << " " << y << " \n";
cout << &x << " " << &y << " ";
}
```

0.28. Summary of Parameter calling When to use:

1. Call by value - no modification, just compute and return
2. Call by address - function to modify parameters
3. Call by reference - function to modify parameters (same as address), but only simple calculations (no loops)

0.29. 157. Object Oriented Programming encapsulation – putting data inside an object polymophism – ability to learn other objects when you know one // classification - based on criteria

```
#include<iostream>
using namespace std;
class Student
{
    private:
         int roll;
         string name;
         int mathMarks;
         int phyMarks;
         int chemMarks;
    public:
         Student(int r,string n,int m,int p,int c)
         {
         roll=r;
         name=n;
         mathMarks=m;
         phyMarks=p;
         chemMarks=c;
         }
     int total()
     {
         return mathMarks+phyMarks+chemMarks;
     }
     char grade()
     {
         float average=total()/3;
         if(average > 60)
             return 'A';
         else if(average>=40 && average<60)
             return 'B';
         else
             return 'C';
     }
```

```
};
int main()
{
 int roll;
 string name;
 int m,p,c;
 cout<<"Enter Roll number of a Student: ";
 cin>>roll;
 cout<<"Enter Name of a Student:";
 cin>>name;
 cout<<"Enter marks in 3 subjects";
 cin>>m>>p>>c;
 Student s(roll,name,m,p,c);
 cout<<"Total Marks:"<<s.total()<<endl;
 cout<<"Grade of Student:"<<s.grade()<<endl;

}
```

0.30.1 Constructors

0.30. OOP Practice with Student Example

Why do we have constructors? When we buy a car, it has predefined settings. (It has a color, and size.) When we create/construct an object, it should come with pre-defined settings.

Types of constructors

1. Default
2. Non-parameterized
3. Parameterized (Rectangle(int l=0, int w=0);
4. Copy constructors

0.30.2 Constructor Example

```
// Constructor
Rectangle::Rectangle()
{
    length = 1;
    width = 1;
}
Rectangle::Rectangle(int l, int w)
```

```
{
    length = l;
    width = w;
}
// Copy Constructor
Rectangle::Rectangle(Rectangle &r)
{
    length = r.length;
    width = r.width;
}
// Facilitator
int Rectangle::area() {
    return length * width;
}
// Enquiry
bool Rectangle::isSquare()
{
    return length == width;
}
// Destructor
Rectangle::~Rectangle()
{
    cout << "Rectangle destroyed\n";
}
```

0.30.3 Summary of Constructors

Here is an example of a full class.

```
class Rectangle {
    // Data Hiding to prevent user-errors
    private:
        int length;
        int width;
    public:
        // Constructors
        Rectangle(); // no parameters (default)
        Rectangle(int l, int w); // set default arguments
        Rectangle(Rectangle &rect); // copy constructor
        // Mutators - change values or properties
        void setLength(int l);
        void setWidth(int w);
        // Accessor - access data/ reading properties
        int getLength() {return length;} // this is an
            inline function (copy/pasted machine code)
        inline int getWidth();
        // Facilitators - uses the data and calculates
        int area();
        int perimeter();
        // Enquiry
```

```
        bool isSquare();
        // Destructor
        ~Rectangle();
};
```

```
#include <iostream>
using namespace std;

class Complex {
    private:
        int real;
        int imag;

    public:
        Complex(int r = 0, int i = 0)
        {
            real = r;
            imag = i;
        }
        Complex add(Complex x)
        {
        Complex temp;
        temp.real = real + x.temp;
        temp.imag = imag + x.imag;
            return temp;
        }
};

struct Demo {
    int x;
    int y;
    void Display() {
        cout << "Hiiii\n";
    }
};


class Rectangle {
    // Data Hiding to prevent user-errors
    private:
        int length;
        int width;
    public:
        // Constructors
        Rectangle(); // no parameters (default)
        Rectangle(int l, int w); // set default arguments
        Rectangle(Rectangle &rect); // copy constructor
        // Mutators - change values or properties
        void setLength(int l);
```

```
        void setWidth(int w);
        // Accessor - access data/ reading properties
        int getLength() {return length;} // this is an
            inline function (copy/pasted machine code)
        inline int getWidth();
        // Facilitators - uses the data and calculates
        int area();
        int perimeter();
        // Enquiry
        bool isSquare();
        // Destructor
        ~Rectangle();
};

int main() {

//     Demo d1;
//     d1.x = 10;
//     d1.y = 5;
//     d1.Display();
//     d1.Display();

    Rectangle r(10,10);
    Rectangle r1(10,10);
    Rectangle r2;
    cout << "Area is " << r1.area() << "\n";

    if (r1.isSquare()) {
        cout << "Yes" << "\n";
    }
        r.setLength(3);
        r.setWidth(4);
        cout << r.area() << endl;
        Rectangle::Rectangle(r2(r));
}
```

0.30.4 In-line Function

In-line functions are functions that are embedded in the machine code with the main() instead of separate from the main function. The function machine code is copy and pasted in the function call.

```
// Example of definitions of an in-line function
int getLength()
{
return length;
}
```

```
inline int getLength()
```

0.31. Struct vs Class Struct by default, everything is public. In Class, everything by default is private (you have to write public:) In C, Struct could not take in functions. In C++, Struct can take in functions.

0.32. Inheritance