The C++ programming language

August 3, 2024Ramasamy Kandasamy

1. Compile and Debug

1.1. G++

Usage:

```
g++ option hello.cpp -o hello
g++ hello.cpp -o hello
g++ -O3 hello.cpp -o hello
```

Options:

`-o`	Output file name.
`-I`	Include additional directories in the search path. Eg: `g++ -I /home/user/libs foo.cpp`
`-O<level>`	Optimization level, starts from 0. Eg: `-O3`
`-g`	Creates breakpoints for debugging using GDB.
`-lm`	Missing link. Used to link certain libraries like `math.h`.
`-E`	Output preprocessed but uncompiled code.
`-S`	Compile but do not assemble.
`-c`	Compile and assembly, but do not link.

1.2. GDB

To run: `gdb hello`

`break`	Set break point. Eg: `break function` or `break line number`.
`run`	Run the program. The program stops at every break point.
`next`	Run until next breakpoint.
`print`	Eg: `print i` or `print &i`
`sizeof`	Eg: `print sizeof(i)`
`&i`	Address of i.
`*j`	Content of memory location j.
`ptype`	get type of a variable. Eg: `ptype(i)`
`set var`	Reassign variable. Eg: `set var i = 1`
`disassemble`	Use after break and run. Gives assembly code.

Accessing memory location using x

Usage: `x/nfs`. `nfs` describes the format.

n - Number of units to display.

f - Number format.

s - Size of each unit.

`x`	Hex.
`o`	Octal.
`t`	Binary.
`d`	Decimal.
`u`	Unsigned decimal.
`i`	Instruction.
`c`	Character.
`s`	String.

`b`	byte.
`h`	Halfword.
`w`	Word.
`g`	Giant.

2. Pre-processor directives

Pre-processor directive always start with a '#'.

- `#include filename`
  Replace with contents of *filename*
  - files with double quotes: `# include "file.h"`
    First pre-processor looks for `file.h` in the same directory as the source file and then in pre-configured list of standard system directories.
  - files with angle bracket: `# include <stdio.h>`
    The pre-processor looks only in the pre-configured list of standard system directories.
  - Additional directories can be include
- `#define NAME replacement text`
  `NAME` is replaced with *replacement text*
- `#define token(arg1,arg2) statement`
  Defining a macro. Eg:
  `#define max(A,B) ((A) > (B) ? (A) : (B))`
- `#undef NAME`
  Nullifies existing definition of `NAME`. Used to ensure a routine is a function.
- `#if`, `#elif`,`#else`, `#endif`
  Eg-1:

```
#if !define(HDR)
#define HDR
/*contents of hrd.h*/
#endif
```

  NOTE: `define()` after `#if` returns `1` if its argument is already defined.
  Here `#define HDR` is first line of `hrd.h` and this file is included only if it was not already included. Eg-2:

```
#if SYSTEM == SYSV
#define HDR "sysv.h"
#elif SYSTEM == BSD
#define HDR "bsd.h"
#elif SYSTEM == MSDOS
#define HDR "msdos.h"
#else
#define HDR "default.h"
#endif
```

- `#ifdef` and `#ifndef`
  Test whether a name is already defined.
  Eg-1 in above point can be replaced by: `#ifndef HDR`

```
#define HDR
/*contents of hdr.h*/
#endif
```

3. Variables and Constants

3.1. Variable types

Type	Location	memory location	Scope
`Global`	Outside functions	data/bss	Global
`Static`	Outside functions	data/bss	Source file
`Static`	Inside a function	data/bss	Local
`Local`	Inside a function	stack	Local
`Register`#	Inside a function*	register	Local

* causes error if declared in global space.

This is not strict. The compiler might choose not to keep the variable in register.

3.2. Variable declaration and initialization

- Eg: `int num;` Local or global depending on context.
- Eg: `static int num;` Static variable.
- Eg: `const int num;` Causes error if `num` is modified.

3.3. Data types

Major variable types

`char`	1 byte.
`short int`	2 bytes.
`int`	4 bytes.
`long int`	4 to 8 bytes.
`long long int`	8 bytes.
`__int128_t`	16 bytes, not part of standard definition, but is supported by g++.
`float`	4 bytes.
`double`	8 bytes.
`long double`	80 bit floating point supported by g++.
`size_t`	unsigned type.

Their actual size might vary depending on the system.

Modifiers for variable types

`unsigned`	With int and char.
`signed`	With int and char.

3.4. Constants

`1234`	`int`
`1234567890L`	`long int`. When should I use L?
`010`	Octal ≡ 8.
`0x10`	Hexadecimal ≡ 16.
`0b10`	Binary ≡ 2. Binary format is not part of standard C, but is supported by gcc.
'x'	Character constant, has an integer value.
'\ooo'	Specify ASCII code of the character in octal.
'\xhh'	Specify ASCII code of the character in hexadecimal.

Escape sequences:

`\a, \b, \f, \n, \r, \t, \v, \\, \?, \', \"`.

Enumeration constants: List of constant integers.

- Eg: `enum ans {NO, YES};`
  By defaults integers are assigned from 0.
- Eg: `enum days {MON=1,TUE,WED,THU,FRI,SAT,SUN};`
  Here, `MON` is assigned 1, and by default, `TUE` is 2.

- Eg: `enum months {JAN =1, FEB, APR=4,MAY};`
  Here, `MAY` is 5.

4. Arithemetic and logical operation

	Operators	Associativity
1	`() [] -> .`	left to right
2	`! ~ ++ -- + - * & (type) sizeof`	right to left
3	`* / %`(reminder)	left to right
4	`+ -`	left to right
5	`<< >>`	left to right
6	`< <= > >=`	left to right
7	`== !=`	left to right
8	`&`	left to right
9	`^`	left to right
10	`|`	left to right
11	`&&`	left to right
12	`||`	left to right
13	`?:`	left to right
14	`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
15	`,`	left to right

4.1. Explanation for selected operators

- `->` and `.`
  See **Structures**.
- **Casting**
  Changes the type of a variable.
  Eg: `float a = (int) 3/2;// a` is 1.0.
- **sizeof**
  `sizeof num;` // Give size of the variable num.
  `sizeof (int)` // Gives size of int, i.e. 4.
- **Bitwise operators**
  - `>>`
    Eg: `a = b >> 1` //Shift `b` bitwise to the right by 1 bit.
    Fill left most bits with the original left most bit.
  - `<<`
    Eg: `a = b << n` //Shift `b` bitwise to the left by n bits.
    Fill right most bits with zero.

  Others:

`&`	Bitwise AND
`|`	Bitwise inclusive OR
`^`	Bitwise EXOR
`~`	Bitwise NOT

- **Comma operator**
  - `expr1,expr2`
    Use sparingly, Eg: `i++,j--`
  - `x = (Conditional expresion)? expr1: expr2`
    `x =expr1` if `Conditional expression` is true, else `x = expr2`.

5. Control Flow

If-else

```
if (expression)
    statement
else if (expression)
    statement
.
.
.
else
    statement
```

Switch

```
switch (expression)
{
    case const-expr:  statements; break;
    case const-expr:  statements; break;
    default:  statements; break;
}
```

Without break statement the execution will *fall through* all the cases.

```
This can be used as:
switch (expression)
{

    case 1:  case 2:  case 3:
        group = 0;
        break;
    case 4:  case 5:
        group = 1;
        break;
    default:
        group = -1;
        break;
}
```

While

```
while (expression)
    statement
```

For

```
for (expr1; expr2; expr3)
    statement
```

This for loop is equivalent to:

```
expr1;
while (expr2)
{
    statements;
    expr3;
{
```

Do While

```
do
    statement
while (expression);
```

Break

`break` : Exit the the loop or control.
`Works for while, for, do while and switch`

Continue

`continue`: Continue next iteration.
Works for `while, for, do while`
If `switch` is within a loop and if `continue` is used inside `switch` and if it is executed, then the loop skips to next loop.

Goto and label

Example:

```
if (disaster)
{
    goto error;
}
...
error:
{
    statement
}
```

Shorter alternative for `if-else` statement:
`x = <cond-expr> ? <out-if-true> : <out-if-false>;`
Eg: `char is_pos = (n > 0) ? 'y' : 'n';`

6. Functions

6.1. int main function: command line arguments

```
main(int argc, char *argv[])
{
    ...
}
```

argc	Number of arguments including the program name.
argv[0]	Pointer to program name. NOTE: argv[0] represents name of the compiled program and not the source file and how it is called eg: a.out vs ./a.out
argv[i]	Pointer to i^{th} argument.
argv[argc - 1]	Pointer to last argument.

6.2. Function definition

```
type function_name(argument list)
{
    statements
    ...
    return expression
}
```

eg:

```
int foo(int num, int cars[],char *pn)
{
    statements
    ...
    return expression
}
```

6.3. Function declaration

Implicit declaration:
The function is assumed to return `int`. Nothing is assumed about the arguments.
Explicit declaration:
Examples:

```
int foo(int, int [], int *);
int foo(int x, int y[], int *p);
```

Explicit declaration is not necessary if the function is defined before `main()`.

6.4. Passing functions as arguments

Example: passing `foo2` to `foo`.
Declaration:

```
int foo(int x, int (*foo2)(int y));
```

Usage:

```
foo(x, foo2);
```

Also see: *7.7 Pointers to functions*

7. Pointers and arrays

7.1. Pointers

Declaration	*type *ptr*; Eg: `int *p;`
Initialization	*type *ptr = val*; Eg: `int *p = 0;` Eg: `int *p = NULL;// Null pointer.` Eg: `int *p = (int *) 100;` NOTE: Casting is required for integer other than zero. Also, the above memory location may not be available.
&	Gives address. Eg: `ptr = &x;`
*	Dereferencing operator. Eg, `*ptr` refers to x.

7.2. Arrays

- **Character arrays**
  ```
  char s[100];
  s = "abc"; //ERROR.
  *s = "abc"; // OK.
  char s[] = {'a','b','c'};
  char s[] = "abc";
  ```
- **Integer, float arrays**
  ```
  int num[100]
  num = {1,2,3} // ERROR
  *num = {1,2,3} // ERROR
  int num[] = {1,2,3}
  float num[] = {1.0,2.0,3.0}
  ```

7.3. Character pointers

Examples:

```
char *pmessage;
pmessage = "Hello, World!\n";
printf("%s",pmessage); # Prints the string
```

*pmessage refer to 'H'

7.4. Pointer to pointers

Examples:

`int **p`	`p` is a pointer to a pointer to `int`
	`*p` is poiner to a pointer to `int`
`char *line[MAXLEN]`	`line` is a pointer to a character array.

7.5. Multi-dimensional arrays

I think, multi-dimensional arrays can be thought of as pointers to pointers etc.
Example:
Given: `int a[2][3] = {{1,2,3},{4,5,6}}`
The following are equivalent:

`a[0][0]`	`**a`
`a[1][1]`	`*(*(a + 1) + 1)`

Here `a` is a pointer to an array of pointers.

7.6. Arrays and pointers

The following two are equivalent:

`pa = &a[0];`	`pa = a;`
`a[i]`	`*(a + i)`
`p[i]`	`*(p + i)`
`f(int arr[])`	`f(int *arr)`

Legal	Illegal
`pa++`	`a++`
`pa[-1]`	`a[-1]`

7.7. Pointer to functions

Eg:

```
int sum_f(int size, int arr[], int (*foo)(int )) // Definition.
{
    int sum = 0;
    for (int i = 0; i < size; i++)
        sum += (*foo)(arr[i]);
}
```

Here `foo` is a pointer to a function and `(*foo)` is the function.
`sum_f(size, arr, square)` // Usage.Sum of squares.
`sum_f(size, arr, cube)` // Usage.Sum of cubes.
NOTE: Function names act as pointers to the function.

8. Structures, unions and typedefs

8.1. Structure

- **Definition:**
```
struct point
{
    int x;
    int y;
};
```
NOTE-1: Structure tag is optional??
NOTE-2: In C a function cannot be a member of a structure.

- **Declaration examples:**
  - `struct {···} a,b,c;`
  - `struct point {···} a,b,c;`
  - `struct point a,b,c`
  - `struct point *p;`

- **Accessing members**
```
a.x = 5;
printf("%d\n",a.x);
```

- **Accessing members with pointer**
`struct point *p = &a;`
The following are equivalent:
  - `p -> x;`
  - `(*p).x;`
  - `a.x;`

- **Arrays of structure:**
`struct points pts[100];`
`struct {···} pts[] = {{···},{···},···,{···}};`
NOTE: In above assignment, in the RHS, elements within the braces could be of different types matching the members of the structure.
In case of simple members, each members need not be enclosed within braces.

8.2. typedef

enditemize

- `typedef int Length;`
`Length len, maxlen;`
`Length *length[];`
`typedef struct tnode *Treeptr;`
`typedef struct tree Treenode;`

- The above examples could also be implemented by `#define`
The following can only be implemented by `typedef`:
`typedef int (*PFI) (char*, char*);`

8.3. Union

A union is a variable that may hold objects of different types and sizes.
The syntax is based on structures:
```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```
For example, integer value of u can be accessed as:
`u.ival`

8.4. Bit fields

Eg:
```
 struct {
    unsigned int is_keyword :  1;
    unsigned int is_extern :  1;
    unsigned int is_static:  1;
} flags;
```
This defines `flags` that contains three 1-bit fields.
The number following the colon represents the field width.
Namespace `using namespace std;`
With this statement standard library functions can be used without the prefix `std::`

9. Lambda Functions

Format:
`[capture](parameters) -> return_type {body}`

- **Capture:** Captures variable from the surrounding scope by value or reference.
Eg: `[a,&b]`
- **Parameters:** Parameters of the function.
Eg: `(int a, int b)`
Return type: Return type of the function.

Examples

Sorting in reverse order:

```
sort(v.begin(), v.end(), [](int a, int b) -> bool {
    return a > b;
});
```

Counting number of elements in a vector:

```
int count = count_if(v.begin(), v.end(), [](int a) -> bool {
    return a > 5;
});
```

Using capture:
Lambda functions can also be stored in variables like the add in the following example.

```
#include <iostream>

 int main() {
     int x = 10;
     int y = 20;
     auto add = [x, y]() -> int {
         return x + y;
     int result = add();
 };
}
```

10. Bit operations

10.1. Builtin functions

`__builtin_clz`	Count leading zeros.
`__builtin_ctz`	Count trailing zeros.
`__builtin_popcount`	Count number of ones.
`__builtin_parity`	Parity of number of ones.

Bit hacks

11. Input output

11.1. File Access

Format:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
int fclose(FILE *fp);
fp = fopen(name, mode);
```

Allowable modes include:

"r"	read.
"w"	write.
"a"	append.
"b"	binary files: "rb", "wb", "ab".

NOTE: Difference between `rb` and `r` etc:
While using just "r" might work in writing binary files, it might cause trouble due misinterpretation of special characters like LF and CR.
See: https://stackoverflow.com/q/2174889/5607735

Standard I/O

Can be treated in the same way as file pointer.
`stdin`, `stdout`, `stderr`

11.2. Reading and writing a file

```
size_t fread{const void* ptr, size_t size,/
size_t count, FILE* stream};

size_ t fwrite{const void* ptr, size_t size,/
size_t count, FILE* stream};
```

11.3. Character I/O

`getchar`	Takes input from standard input. `int getchar()` Equivalent to `getc(stdin)`
`putchar`	Output to standard output. `int putchar(int)` Equivalent to `putc(c), stdout`
`getc`	`int getc(FILE *fp)`, Eg: `getc(stdin)`
`putc`	`int putc(int c, FILE *fp)`
`ungetc`	

11.4. Line I/O

`gets`	Reads until `EOF`. `gets` deletes terminal \n
`puts`	Writes line to stdout. `puts` adds terminal \n
`fgets`	`char *fgets(char *line, int maxline, FILE *fp)`
`fputs`	`int *fputs(char *line, FILE *fp)`
`getline`	Eg: `getline(cin, s)`. Where `s` is a string.

NOTE: Never use `gets`
`gets` does not check for buffer overrun, and keeps reading until it encounters new line or `EOF`. It has been used to break computer security.

11.5. printf and scanf

Conversion formats for printf and scanf

Character	Argument type; Printed as
d,i	int; decimal number
o	int; unsigned octal number (without leading zero)
x,X	int: unsigned hexadecimal number (without a leading 0x or 0X), using `abcdef` or `ABCDEF`.
u	int; unsigned decimal number.
zu	`size_t`.
c	int; single character.
s	char *; Print string, scan word. NOTE: `scanf` reads only a word and not the entire string.
f	double; [-]*m.dddddd*.
e,E	double; [-]*m.dddddd*±*xx*, or [-]*m.dddddd*±*xx*.
g,G	double; %e or %E if exponent is < -4 or >= precision, else use %f.
p	void *; pointer.
%	no argument; Print %.

Between % and conversion character there may be in order:

- Minus sign: Left justification.
- Number: Minimum field width.
- Period: Separates field width from precision.
- Number: Precision.
  For float: number of digits after decimal.
  For int: minimum number of digits.
  For string: maximum number of characters to be printed.
- h: for short integer. l: for long integer.

Formatting rules

Examples

	Wildcard
%*d	Here precision can be specified dynamically. Eg: `printf("%*d", 6, foo);`, this is equivalent to `printf("%6d",foo);`
	Integers
%d	print as decimal integer.
%6d	print as decimal integer, at least 6 characters wide.
	Floating point numbers
%6f	print as floating point.
%.2f	print as floating point, 2 characters after decimal point.
%6.2f	print as floating point, at least 6 characters wide and 2 characters after decimal point.
	String: example `hello, world` (12 chars).
:%s:	:hello,␣world:
:%10s:	:hello,␣world:
:%.10s:	:hello,␣wor:
:%-10s:	:hello,␣world:
:%.15s:	:hello,␣world:
:%-15s:	:hello, world␣␣␣:
:%15 .10s:	:␣␣␣␣␣hello, wor:
:%-15 .10s:	:hello,␣wor␣␣␣␣␣:

Printf and variants

- `int printf(char *format, arg1, arg2, ···);`
  Print output to stdout.
- `int sprintf(char *string, char *format arg1, arg2, ···);`
  Print output to `string`.
- `int fprintf(FILE *fp, char *format arg1, arg2, ···);`
  Print output to file pointed by the file pointer `fp`.

Example:

```
printf("%s\n", string_var);
printf("%s", "hello, world");
printf("square of n is %d\n", n_square);
```

Scanf and variants

- `int scanf(char *format, arg1, arg2, ···);`
  Scan input from `stdin`.
- `int sscanf(char *string, char *format arg1, arg2, ···);`
  Scan input from string.
- `int fscanf(FILE *fp, char *format arg1, arg2, ···);`
  Scan input from the file pointed by the file pointer `fp`.

NOTE1: unlike printf, in scanf the variable are pointers.
NOTE2: In `scanf` `%s` reads a word and not the entire string.
Also see: https://stackoverflow.com/a/1248017/5607735
Examples:

```
scanf("%d", &n);
sscanf(string, "%d", &n);
fscanf(fp, "%d", &n);
```

11.6. cin and cout

`cin`	Read input until space, or tab or LF.
`cout`	Output.

NOTE:
- Use the following to increase speed ot I/O.
  `ios_base::sync_with_stdio(false);`
  `cin.tie(nullptr); cout.tie(nullptr);`
- Use `cin.ignore()` after `cin` and before using `getline`.

11.7. Miscellaneous

`int fflush(FILE *stream)`
Causes any buffered but unwritten data to be written. The effect is undefined on an input stream.
`int fclose(FILE *stream)`
Flushes any unwritten data for `stream`, discardes any unread buffered input.

12. Libraries

Usually in Linux, the library header files are stored in `/usr/include`
NOTE: Use `#include <bits/stdc++.h>` to include all standard libraries.

12.1. Character class tests: <cctype>

`isalpha(c)`
`isupper(c)`
`islower(c)`
`isdigit(c)`
`isalnum(c)`
`isspace(c)` `true` for ASCII codes: 9-13 & 32.
`toupper(c)`
`tolower(c)`

12.2. String functions: <cstring>

`strcat(s,t)`	Concatenate `t` to end of `s`
`strncat(s,t,n)`	
`strcmp(s,t)`	
`strncmp(s,t,n)`	
`strcpy(s,t)`	Copy `t` to `s`
`strncpy(s,t,n)`	
`strlen(s)`	
`strchr(s,c)`	Return pointer to first `c`
`strchr(s,c)`	Return pointer to last `c`

12.3. Mathematical function: <cmath>

`sin(x)`
`asin(x)`
`hsin(x)`
`exp(x)`
`log(x)`
`log10(x)`
`pow(x,y)` x^y
`sqrt(x)`
`ceil(x)`
`floor(x)`

`abs(x)`	Abs. val. of x. Returns integers types.
`fabs(x)`	Absolute value of x. Returns `double`
`frexp`	`frexp(int x, int *exp)`. Returns significand (y) in the range $[0.5, 1)$ and the stores the exponent in exp, such that $x = y * e^{exp}$
`ldexp`	`ldexp(double y, int exp)`. Inverse of `frexp`.
`modf`	`double modf(x, double *ip)`. Splits `x` into integer and fractional parts. Returns the fractional part and stores integer parts at `ip`.
`fmod(x,y)`	Floating point remainder of x/y with the same sign as x.

12.4. Utility Functions: <cstdlib>

`system(char *s)`
Run system commands.
Eg: `system("date")`

String ¡-¿ numbers

String to numbers

From C library `int atoi(char *s)`.
Variants: `atof`, `atol`, `atod`
From C++ library `int stoi(string s, size_t *p = 0, int base = 10)`.
Variants: `stol`, `stod`, `stof`, `stold`
Eg:

```
int n = stoi("1234"); // n = 1234.
int n = stoi("12", 0, 16); // n = 18.
int n = stoi("A", 0, 16); // n = 10.
```

Number to string

`to_string` Eg: `string s = to_string(1234);`\\s = "1234".

Memory management

- `void *calloc(size_t nobj, size_t size)`
  Return pointer to array of `nobj` of size `size`.
  The space is initialized to 0.
- `void *malloc(size_t size)`
  Returns pointer to an object of size `size`.
  The space is uninitialized.
- `void *realloc(void *p, size_t size)`
  Change size of the object pointed to by p to `size`.
  Return pointer to new space.
- `free (void *p)`
  Deallocates space point to by `p`.

Memory management in C++

- `void* operator new(size)`
  Eg: `myClass* p1 = new myClass;`
  `myClass* p1 = new(sizeof(myClass));`
- `void delete(void* ptr)`
  Eg: `delete p1; delete(p1);`

limits

Examples:

- `numeric_limits<int>::min()`
- `numeric_limits<int>::max()`
- `numeric_limits<double>::min()`
- `numeric_limits<double>::infinity()`

12.5. Algorithms

Sort and search

Search and sort from C

- `void *bsearch(const void *key, const void *base, size_t n, size_t size, int (*cmp) (const void *keyal, const void *datum))`
  Searches `base[0]` $\cdots$ `base[n-1]` for `key`.
  Comparison function must return negative if its first argument (search key) is less than its second argument (a table entry) and so on.
- `qsort(void *base, size_t n, size_t size, int (*cmp)(const *void, const *void))`

Search and sort from C++

Compare functions:

```
bool compare(x, y){
    return(x strictly precedes y)
}
```

In all the below, a could be an iterator or a pointer and all of them can use custom compare functions.

- y = `lower_bound(a, a+n, x)`
  Returns an iterator to the first element whose value is $\geq$ x.
- y = `upper_bound(a, a+n, x)`
  Returns an iterator to the first element whose value is $>$ x.
- y = `equal_range(a, a+n, x)`
  Returns a pair of iterators pointing to the upper bound and lower bound of x.
- y = `equal_range(a, a+n, x, compare_function)`
  Returns a pair of iterators pointing to the upper bound and lower bound of x.
- `sort(it1, it2, compare_function);`
- `stable_sort(it1, it2, compare_function);`
- `is_sorted(it1, it2, compare_function);`
- `is_sorted_until(it1, it2, compare_function);`
  Returns the iterator to the element until which the array is sorted.
- `find_if(start_it, end_it, func)`
  Returns iterator to the first element in the range `[start_it, end_it)` for which `func` returns true.
- `find_if_not(start_it, end_it, func)`
  Returns iterator to the first element in the range `[start_it, end_it)` for which `func` returns false.

Min Max

- `max(a, b, comp); min(a, b, comp);`
- `max({a1, a2, a3}, comp); min({a1, a2, a3}, comp);`
- `max_element(a, a+n, comp);`
  Returns the iterator for the largest element in a. If there are multiple largest element, returns the iterator/pointer for the first one.
- `min_element(a, a+n, comp);`

Merge

- `merge(InputIterator F1, InputIterator L1, InputIterator F2, InputIterator L2, OutputIterator R);`
  Returns the iterator to the end of output.
  NOTE: This operator does not allocate memory. The `OutputIterator R` should point to a vector with sufficient memory to store the concatenated vector.
- `set_intersection` Arguments and output are the same as that for `merge`.
- `set_union` Arguments and output are the same as that for `merge`.
- `set_difference` Arguments and output are the same as that for `merge`.
- `set_symmetric_difference` Arguments and output are the same as that for `merge`.

Hash Function

Usage:
`hash<T> h;`
Eg:

```
hash<int> h; size_t x = h(10);
hash<string> h; size_t x = h("hello");
```

The standard hash function can be used for user defined types by combining hash values. Two strategies for combining hash values are:

- Simple: just use XOR: `h1` $\hat{}$ `h2`.
- Used by Boost(?):
  `h2 ≙ h2 + 0x9e3779b9 + (h1<<6) + (h1>>2);`
  0x9e3779b9 is a prime number.
  See: https://stackoverflow.com/a/2595226/5607735
  How efficient is this for a smaller prime number that can be committed to memory.

Custom hash function. Custom hash function for a user defined type can be defined as follows:

```
class CustomHash {
 size_t operator () (const pair<string, string>) const {
     hash<string> hasher;
     auto h1 = hasher(p.first);
     auto h2 = hasher(p.second);
     return h1 ^ h2;
 }
};
```

This can be used for custom types in unordered_map and unordered_set.

Others

- `swap(T& a, T&b);`
- `reverse(Iterator first, Iterator last);`
- `next_permutation(a.begin(), a.end());`
- `prev_permutation(a.begin(), a.end());`
- `random_shuffle(a.begin(), a.end());`
  Gives the next permutation for a given vector of integers.

13. Data structures

13.1. Pair

Eg: `pair<int, int> x = 1,2;`

`x.first`	Returns 1.
`x.second`	Returns 2.

13.2. Tuple

Fixed length collection of heterogenous values.
Eg: `tuple<int, string, double> t;`
Operations related to tuple

`get<i>`	Return i^th value of t. Eg: `get<0>(t);`.
`make_tuple()`	Returns tuple out of individual values. Eg: `make_tuple(5, "Paul", 23.4)`.
`tie()`	Unpacking tuple. Eg: `tie(n, s, x) = t;` `tie(n, std::ignore, x) = t;`

13.3. Vectors

Definiton:
`vector<T> v;`
Element access

`operator[]`	
`v.back()`	Last element
`v.front()`	First element

Modifiers

`v.push_back(5)`	Add `5` to the vector. Simillar to push in stack data structure.
`v.insert()`	Inserts elements / range in `v`. Eg: Append `u` to `v`. `v.insert(v.end(), u.begin(), u.end());`
`v.emplace`	`v.emplace{iterator pos, Args}`. New element is constructed in place using `Args` and inserted at pos.
`v.emplace_back`	`v.emplace_back{Args}`. New element is constructed in place using `Args` and inserted at the end. IMPORTANT: Note the differences between pushback, insert, emplace and emplaceback.
`v.pop_back()`	Remove element on top of the stack.
`v.erase()`	Remove element / range from `v`.
`v.clear()`	Clears all the elements in `v`.

Capacity

`v.size()`	Returns the size of the vector.
`v.capacity()`	Current memory allocated to `v` in terms of number of elements.
`v.max_size()`	Maximum memory available for `v`.
`v.reserve()`	Reserve a minimum size for `v`. Does not affect v.size().
`v.shrink_to_fit()`	Shrink the capacity to fit the size.
`v.empty()`	Test if `v` is empty.

`tie` Can be used to unpack a vector.
Example: `tie(a, b, c) = v;`

13.4. String

A `string` is like a vector of characters.
Definition:
`string s = "Hello";`
String operations:

`a + b`	Concatenate.
`s.append(t)`	append `t` to `s`.
`s.compare(t)`	returns `0` if `s == t`, returns < 0 if $s \prec t$, else returns > 0.
`s.substr(start, len)`	Select a substring.
`s.find(str2)`	find first occurence of `str2`.
`s.find(str2, pos)`	Find first occurence of `str2` by searching from the positon `pos`.

13.5. Set

Definition
`set<type> s;`
`multiset<type> s;`
`unordered_set<type> s;`
`unordered_multiset<type, hasher> s;`
`hasher`: Refer Custom Hash Function.
Set operations

`s.insert(item);`	
`s.erase(item);`	
`s.count(item);`	
`s.find(item);`	Returns the iterator that points to `item`.
`s.insert()`	Inserts elements in s. Eg: Insert elements from t to s. `v.insert(v.end(), u.begin(), u.end());`
`s.size()`	Not availabe for `unordered_set` and `unordered_multiset`.
`s.lower_bound(x)`	Returns iterator to the smallest element that is larger than or equal to x.
`s.upper_bound(x)`	Returns iterator to the smallest element that is larger than x.

NOTE: erase method remove all instances of the item in the multiset.

13.6. Bit set

Definiton
`bitset<size> s;`
Examples:
`bitset<5> s; bitset<5> s(string("1100110"));`
Operations on Bit set:

`a & b`	Bitwise AND.
`a | b`	Bitwise OR.
`a` $\hat{}$ `b`	Bitwise EXOR.
`a`	Bitwise negation.

13.7. Map

`map` : Balanced binary tree.
`unordered_map`: Hash list.
Definition:
`map<type1, type2> m;`
Eg: `map<string, int> m;`
`unordered_map<type1, type2, hasher> m;`

`hasher`: Refer Custom Hash Function.

Operations on map

`m["banana"]`	Retrive value associated with `banana`.
`m.count("banana")`	Return 1 if the key is present else 0.
`m.size();`	

13.8. Stack

Eg: `stack<int> s;`

s.push(5)
s.top()
s.pop()

13.9. Queue

Eg: `queue<int> q;`

`q.push(5)`	Adds element to the end.
`q.pop()`	Removes first element.
`q.front()`	Returns first element.
`q.empty()`	True if q is empty.

13.10. Dequeue

Eg: `dequeue<int> d;`

`d.push_back(5)`	Add to the top.
`d.push_front(2)`	Add to the bottom.
`d.pop_back()`	Remove from the top.
`d.pop_front()`	Remove from the bottom.

13.11. Heap

Format:

Constructing a heap:

```
make_heap(iterator f, iterator s);
make_heap(iterator f, iterator s, comp);

pop_heap(iterator f, iterator s);
pop_heap(iterator f, iterator s, comp);

push_heap(iterator f, iterator s);
push_heap(iterator f, iterator s, comp);
```

NOTE:
`f` points to the first element and `s` points next to the second element.
`f` and `s` can be pointers when the heap is constructed over a array.

Compare function:

```
bool compare(x, y){
return(x < y); // For max heap
return(x > y); // For min heap
}
```

13.12. Priority queue

Format:

```
priority_queue(type, vector<type>, compare)
```

Max priority queue [Default]

Eg: `priority_queue<int> q;`

Min priority queue

```
priority_queue<int, vector<int>, greater<int>> q;
```

`q.push(3)`	
`q.top()`	Returns the largest element.
`q.pop()`	Removes the largest element.

Compare function for priority queues

```
class compare{
public:
    bool operator()(const T& x, const T& y) const {
        return(x < y); // Max priority queue.
        // return(x > y); // Min priority queue.
    }
}
```

13.13. Iterators

Iterators are like pointers but more specific to a collection.
Not applicable for `unordered_set`, `unordered_map` and `priority_queue`.

Example definition:

```
vector<int>::iterator it;
set<string>::iterator it;
```

`s.begin()`	Returns iterator pointing to the first element in the collection.
`s.end()`	Returns iterator pointing next to the last element in the collection.

Iterating over a map:

Here `*it` is a `pair` of the key and value. Eg: `pair<string, int> p = *it;`
Eg:

```
    map<string, int>::iterator it = m.begin();
    // Accessing key and value
    string key = it->first;
    int value = it->second;
```

13.14. Representing graphs

13.15. Adjacency list

Eg:

```
vector<int> adj[N]// Unweighted graph.
vector<pair<int,int>> adj[N]// Weighted graph.
```

13.16. Adjacency matrix

Eg: `int adj[N][N];`

13.17. Edge list

Eg:

```
vector<pair<int,int>> edgesUnweighted graph.
vector<tuple<int,int,int>> edges // Weighted graph.
```

14. Code snippets, etc.,

14.1. GCD:

```
int gcd(int x, int y){
    return y ? gcd(y, x%y) : x;
}
```

14.2. Middle of an array

n: length of array

0-based indexing

if n is odd: $\left\lfloor \frac{n}{2} \right\rfloor$
if n is even: $\left\lfloor \frac{n}{2} \right\rfloor - 1$ and $\left\lfloor \frac{n}{2} \right\rfloor$

1-based indexing

if n is odd: $\left\lceil \frac{n}{2} \right\rceil$
if n is even: $\left\lceil \frac{n}{2} \right\rceil$ and $\left\lceil \frac{n}{2} \right\rceil + 1$

14.3. Decimal to binary

```
int n;
cin >> n;
auto a = bitset<64>(n);
```

`a[0]` contains zeroth bit and so on.

15. Previous Mistakes:

15.1. Using proper variable types

Eg: using `int` instead of `long long int` would cause erroneous output.

15.2. Be careful of unsigned int

Eg

```
unsigned x; cin >> x;
while(x >= 0){
    // Do something
}
```

The above will never terminate, since `x` never becomes negative.

15.3. Upper bound and lower bound

Check for edge cases. While searching for `x`, if `x` is not found, then `lower_bound` points to an element larger than `x`. The following might be helpful.

```
if(lower_bound > begin && *lower_bound > x) lower_bound--;
if(lower_bound == end) lower_bound--;
if(upper_bound == end) upper_bound--; // Might be
necessary sometimes.
```

15.4. Loops

- Using variable in boundary condition. Eg:

```
// value of x changes within the loop.
for(int i = 0; i < (d[x] - c); i++) // Dangerous !!

k = d[x] - c;
for(int i = 0; i < k; i++) //
```

- When the variable in boundary condition is updated by mulitiplication.
  In the following loop when n > 1 and a = 1, this loop won't terminate.

```
int x = 1;
while(x < n){
    x = a * x;
}
```

15.5. Bugs in compiler directives

In Bioinformatics contest 2021, a bug was because I use `bitset<M>`, where `M` was specified as compiler directive. While debugging I did not pay attention to `#define` statements.
Lesson: While debugging pay attention to the entire code, particularly compiler directives. They are particulary important because they are hidden and not directly noticeable in the code.

15.6. Bugs due to augmented data

Ref: ABC_207 problem C. There were two vectors, `v` and `t` associated by index. The vector `v[i]` corresponds to `t[i]`. In this problem, I sorted v independently from t. Thus this broke the association between `v` and `t` resulting in erroneous results.

15.7. min_element and max_element

Do not use these two functions to find minimum and maximum elements in an unsorted array. Read documentation to know more about them.

15.8. map vs multiset, NOT exactly a mistake

In one problem I used multiset to keep count of some values. It was very slow compared to using map to keep count of those values.

15.9. Math errors

Division by zero and logarithm of zero.

15.10. Not accessing out of bound elements

```
if(a[i] == 0) {do something;} // Wrong.
if(i < n && a[i] == 0){do something;}// Correct.
```

15.11. Order of precedence and brackets

Example:(CRF-732: 1546C)

```
if(l_pos[k] - l_pos[k-1] % 2 == 1){do something}// Wrong;
if((l_pos[k] - l_pos[k-1]) % 2 == 1){do something}// Correct;
```

15.12. Using braces appropriately in conditionals and loops

Example:(CRF-732: 1546C)

```
// Wrong if do_2 depends on x :
if(x) do_1;
    do_2;

// Correct if do_2 depends on x :
if(x){ do_1;
    do_2;
}
```

15.13. Errors due to 0-based and 1-based indexing

The data in most programming contests use 1-based indexing. Error could arise while reading or using such data.
Eg:

```
for(int i = 0; i < n; i++){
int temp; cin >> temp;
a[temp]++;
}
```

Here if a is 0-based and input temp is 1-based, then error will arise. The following is correct.

```
for(int i = 0; i < n; i++){
int temp; cin >> temp;
a[temp-1]++;
}
```

15.14. Error while using obj.size() as limit in a loop

Obj.size() return a value of type `size_t` which (probably) unsigned long long int. Therefore it inherits the same dangers associated with using `unsigned int`(13.2). Eg: Here if v.size() is zero, v.size()-1 will not return -1, rather it would return the largest `long long`.

```
for(int i = 0; i <= v.size()-1; i++){}
```

Probable a correct way to write the above code is as follows:

```
for(int i = 0; i <= (int)v.size()-1; i++){}
```

Another example:

```
for(int i = r.length()-2; i >=0; i++){}
```

Correct version.

```
int rl = r.length();
for(int i = rl-2; i >=0; i++){}
```

15.15. Loop condition dependent on string::find

Bugs could arise because `string::find` return -1 when the query is not found.
Eg:

```
for(int i = 0; i < size; i++){
    i = s.find(c, i);
    //More code...
}
```

Correct version:

```
for(int i = 0; i < size; i++){
    i = s.find(c, i);
    if(i == -1) break;
    //More code...
}
```

15.16. Semicolon typo

Eg:

```
for(int i = 0; i < n; i++);{
    cout << i << '\n';
}
```

In above code the for loop is prematurely terminated. This would result in runtime error.

15.17. Bugs due to using infinity

Eg:

```
const int inf = numerics::limits<int>max();
int x = inf;
cout << x + inf << '\n';
```

The output here would not be inf. I made a simillar mistake while using infinity in Floyd-Warshall's algorithm.

15.18. No code after return

Once I made the following mistake:

```
return r.top(); r.pop();
```

Obviously, that `r.pop()` did not work.

15.19. Beware when using logical operators with functions

The following is dangerous:

```
ans = ans && f(x);
```

If `ans` is initally false, f won't be executed.
Better one would be (I am not sure if this always works with all programming languages):

```
ans = f(x) && ans;
```

I made this mistake in LeetCode 130. Surrounded Regions.

15.20. Be careful with char vs int

I made the following mistake:

```
if(mat[i][j] == 1){
// do something.
}
```

Here `mat` was `vector<vector<char>>` I should have tested for `mat[i][j] == '1'`.

15.21. Be careful about pass by reference.

Ref: A student's code.
Not using pass by reference resulted in failure in the update of value of the target. This caused issue in an implementation of linked-list.

15.22. a = a + 1 vs a += 1

Ref: Discussion with a labmate. This is in the context of python programming. a = a + 1 creates a new object and assigns it to a. a += 1 does not create a new object. It modifies the existing object.

15.23. Failure to initialize all the variables

Ref: A student's code.
The student did not initialize all the variables. This resulted in a bug. This was particularly critical because one of the class variable was a pointer. This resulted in segmentation fault. It also cause TLE, but I am not sure about the exact mechanism.

16. Variable nomenclature for competitive programming

Category	Symbols
Counter	i, j, k; i1, i2
Limits	(h, l), (u, d), (l, r), (h1, l1)
Integers	n, m, k; n1, n2
Floats	x, y, z, x1, x2
Element	e: for(auto e: v)
Pair/points	p, q, r, p1, p2
Pointer	ptr; ptr1, ptr
String	s; s1, s2
Map	mp; mp1, mp2
Vector	a, b, c, u, v; u1, u2, v1, v2
Matrix	mat, mat1, mat2
Graph	g, g1, g2
DP mat/vect	dp; dp1, dp2
Priority queue	pq; pq1; pq2
Stack	stk; stk1, stk2
Set	set1, set2
Temporary	tmp; tmp1, tmp2

Using suffix

Eg:	Description
ai	Element of a
al	Left limit element of a (or) Left limit ptr of a
ar	Right limit element of a (or) Right limit ptr of a
aptr	Pointer to an element in a
ait	Iterator to a

Using prefix

Eg:	**Description**
na	Number of elements in a
ma	Map associated with a
sa	Set associated with a
qa	Queue or priority queue associated with a
ga	Graph associated with a