# VE280 2022FA RC4


**VE280 2022FA RC4**

- L10: I/O Streams
  - `cin` , `cout` & `cerr`
    - >> and <<
    - Stream Buffer
    - Cerr
      - Exercise
  - File Stream
  - String Stream
- L8: Enum
  - Example
    - Note
  - Nice Usage
- L9: Program Arguments
- L11: Testing
  - Concepts
  - Example
- Exam Like Exercises
- Reference


## L10: I/O Streams

### `cin` , `cout` & `cerr`

#### >> and <<

In C++, streams are **unidirectional**, which means you could only `cin >>` and `cout <<`.

If we look into `cin`, it's an object of class `istream` (input stream). `operator>>` (the extraction operator) is one of it's member function.

Check `std::istream::operator>>`, similar for `cout` & `cerr` (`ostream`)

```
istream& operator>> (bool& val);
istream& operator>> (short& val);
istream& operator>> (unsigned short& val);
istream& operator>> (int& val);
istream& operator>> (unsigned int& val);
istream& operator>> (long& val);
istream& operator>> (unsigned long& val);
istream& operator>> (long long& val);
istream& operator>> (unsigned long long& val);
istream& operator>> (float& val);
istream& operator>> (double& val);
istream& operator>> (long double& val);
istream& operator>> (void*& val);
```

Many type of parameter it takes -> it knows how to convert the characters into values of certain type

Return value also a reference of `istream` -> it can be cascaded like `cin >> foo >> bar >> baz;`

Some other useful functions:

```
istream& getline (istream& is, string& str);
std::ios::operator bool // member of istream -> if(cin), while(cin)
istream& get (char& c); // member of istream
```

Differences:

- `>>` will read until reaching the next space or `\n`, and the space and `\n` will still be left in the buffer. And space and `\n` won't be stored into the parameter.
- `getline` would read a whole line and discard the `\n` at the end of the line directly.
- `get()` reads a single character, whitespace or newlines.

### Stream Buffer

`cout` and `cin` streams are buffered (while `cerr` is not).

You need to run `flush` to push the content in the buffer to the real output.

`cout << std::endl` is actually equivalent to `cout << '\n' << std::flush`

When the buffer becomes full, the program decides to read from `cin` or the program exits, the buffer content will also be wrriten to the output.

### Cerr

There is another output stream object defined by the iostream library called `cerr`.

By convention, programs use the `cerr` stream for error messages.

On JOJ, `cerr` message is printed out in `stderr` block and is not counted for output.

This stream is identical in most respects to the `cout` stream; in particular, its default output is also the **screen**. Its output redirection can be used as `./program 2><filename>` e.g. `./program 2>test_error.txt`

#### Exercise

Redirect the compile error message of a Makefile into `a.txt`.

```

```

## File Stream

```
#include <fstream>

std::ifstream iFile; // inherit from istream
std::ofstream oFile; // inherit from ostream
iFile.open("myText.txt");  // if unsuccessful to open, iFile would be in failed state, if(iFile) returns false. But member function open() has void return type!!!

iFile >> bar;
while(getline(iFile, line)) // simple way to read in lines.

oFile << bar;
```

**Question:** Is anything missing in the above program?

> Hint 1: "Deductions will follow!" "Really?"
>
> Hint 2: https://en.cppreference.com/w/cpp/io/basic_fstream

## String Stream

```
#include <sstream>

istringstream iStream; // inherit from istream
iStream.str(line); // assigned a string it will read from, often used for getline
iStream >> foo >> bar;
iStream.clear(); // Sometimes you may find this useful for reusing iStream

ostringstream oStream; // inherit from ostream
oStream << foo << " " << bar;
string result = oStream.str(); // method: string str() const;
```

# L8: Enum

Enum is a type whose values are restricted to a set of integer values. Advantages:

- Use less memory than `std::string`
- More readable than `const int` or `char`
- Limit valid value set, so compiler help you find spelling mistakes.

## Example

```
#include <iostream>
enum A {
    a, b, c=-1, d, e=5, f, g=a + e, h
};
int main() {
    std::cout << a << ' ' << b << ' ' << c << ' ' << d << ' '
        << e << ' ' << f << ' ' << g << ' ' << h << '\n';
    // Output is 0 1 -1 0 5 6 5 6
    return 0;
}
```

- By default the enum value starts from 0, and increments for each value
    - But you can also assign any integer value to them
- Values in enum (a, b, c,...) can be treated as global `const int`
    - Can be compared (<, >, ==, !=)

### Note

Since enum A is a new type, `std::cin` and `std::cout` cannot identify them. You need to cast the enum variable to int before print it.

```
#include <iostream>
enum A {
    a, b, c=-1, d, e=5, f, g=a + e, h
};
int main() {
    A A1=a;
    std::cout << A1 << '\n'; //wrong, but works
    std::cout << static_cast<int>(A1) << '\n'; //correct
    return 0;
}
```

## Nice Usage

Enum type can serve as array index (same as `const int`). More to come in later projects.!

```
#include <iostream>
enum suit {
    DIAMOND,SPADE,HEART,CLUB
};
const char* suit_name[4] = {"DIAMOND", "SPADE", "HEART", "CLUB"};
int main () {
    std::cout << suit_name[DIAMOND] << '\n';
    std::cout << suit_name[SPADE] << '\n';
}
```

# L9: Program Arguments

Write a main function that takes program arguments:

```
int main(int argc, char *argv[]) {
    /* code here */
}
```

Or in a way easier to memorize:

```
int main(int argc, char ** argv) {
    /* code here */
}
```

Both are acceptable.

• `arg` for argument, `c` for count, `v` for value or vector.

• `argv` is a 1-D array of c-strings (equivalent to char*), so we need two `\*` and get `char** argv`

• You can consider `argv` as a pointer to (pointer to `char`), or an array of (pointer to `char`)

# L11: Testing

## Concepts

Difference between testing and debugging

- Testing: discover a problem ~~(and more problems)~~
- Debugging: fix a problem ~~( and at the same time create new problems)~~

Five Steps in testing:

- Understand the specification (Design requirement)
- Identify the required behaviors (Specification boil down; abstract; Party A)
- Write specific tests (**Simple**+ **Normal**+ **Nonsense** )
- Know the answers for those tests (The correct output; concrete; Party B)
- Stress tests (**large** and **long running** )

Who knows... this is also inside the scope of the exam.!!!

## Example

- Step 1. Specification

  ```
  Write a function to calculate factorial of non-negative integer, return -1
  if the input is negative.
  ```

- Step 2. Behavior
  - Normal: return `_____` for input `_____`
  - Boundary: return `_____` for input `_____`
  - Nonsense: return `_____` for input `_____`
- Step 3: Test Cases

  ```
  void testNormal() {
      assert(fact(5) == 120);
  }
  void testBoundary() {
      assert(fact(0) == 1);
  }
  void testNonsense() {
      assert(fact(-5) == -1);
  }
  ```

# Exam Like Exercises

1. Consider the following codes:

```
int i1,i2;
double d;
std::string s;
std::cin >> i1 >> s >> d >> i2;
```

If the user input

```
20.22 -10 17.20 280 rc.!
```

with a newline. Then what's the value of each variable?

- `i1` = _____, `i2` = _____, `d` = _____, `s` = _____

2. Write `two statements` that first try to open `a.txt`, then if the file is not opened successfully, assert the program.

```

```

3. What is a partial function? Can you give an example of a partial function and explain why it is a partial function? Whenever possible, it is much better to write a complete function instead of a partial one. Why?

_____

_____

_____

_____

_____

_____

4. A function `args_to_list` will transfer program arguments into `list_t`. Complete the following functions:

```
list_t args_to_list(_________(a)_________);
// Implementation Omitted

int main(_________(b)_________) {
    list_t arg_list = args_to_list(_______(c)_______);
    return 0;
}
```

5. This is a function in your project 2, write 3 different boundary cases. Each case should test a different boundary situation. For each test case, you must provide: a **description** of the test case, the expected **behavior** for a correct implementation of the function, and **why** the case is a boundary test case. Use the provided example as format guideline.

```
list_t insert_list(list_t first, list_t second, unsigned int n){
/*
// REQUIRES: n <= the number of elements in "first".
//
// EFFECTS: Returns a list comprising the first n elements of
//          "first", followed by all elements of "second",
//           followed by any remaining elements of "first".
//
// EAXMPLE: insert (( 1 2 3 ), ( 4 5 6 ), 2) returns ( 1 2 4 5 6 3 ).
//
*/
```

Your answers here:

(1)_____

(2)_____

(3)_____

# Reference


[1] Weikang, Qian. VE280 Lecture 8-11.

[2] Jiajun, Sun. VE280 Midterm Review Slides. 2021FA.

[3] Pingchuan, Ma. VE280 RC 5. 2021 FA.