VE280 2021FA Mid RC Part4

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
```

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`

File Stream

```
#include <fstream>

ifstream iFile; // inherit from istream
ofstream oFile; // inherit from ostream
iFile.open("myText.txt");  // if successful to open, iFile would store
true, false otherwise.


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

oFile << bar;
```

What to notice is that `>>` 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. While `getline` would read a whole line and discard the `\n` at the end of the line directly.

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
iStream.close();

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

Exercise:

Convert the message with ',' and **arbitrary number of spaces (could be 0)** as the delimiter

```
Ann ,Peter , Owen,Alice , Bob, Tim
```

to

```
Ann
Peter
```

```
Owen
Alice
Bob
Tim
6 // number of names
```

Solution

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

int main()
{
    string line;
    getline(cin, line);
    for (auto it = line.begin(); it != line.end(); it++) // you could also use line[i]
        *it = (*it == ',') ? ' ' : *it; // change ',' to ' '
    stringstream ss(line);
    int count = 0;
    string name;
    while (ss >> name)
    {
        cout << name << endl;
        count++;
    }
    cout << count << endl;
}
```

L11: Testing

Difference between testing and debugging

- Testing: discover a problem
- Debugging: fix a problem

Five Steps in testing:

1. Understand the specification
2. Identify the required behaviors (program that outputs differently on 1/2 arguments)
3. Write specific tests (for each required behavior)
4. Know the answers in advance
5. Include stress tests

General steps for Test Driven Development

1. Think about task specification carefully
2. Identify behaviors

3. Write one test case for each behavior (normal, **boundary** and **nonsense**, distinguish them)
4. Combine test cases into a unit test
5. Implement function to pass the unit test
6. Repeat above for all tasks in the project

A simple example

Step 1: Specification

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

Step 2: Behaviors

```
Normal: return 120 for input = 5
Boundary: return 1 for input = 0
Nonsense: return -1 for input = -5
```

Step 3: Test Cases

```
void testNormal() {
    assert(fact(5) == 120);
}

void testBoundary() {
    assert(fact(0) == 1);
}

void testNonsense() {
    assert(fact(-5) == -1);
}
```

Step 4: Unit Test

```
void unitTest() {
    testNormal();
    testBoundary();
    testNonsense();
}
```

Another example

Step 1: Specification

```
Given 2 non-negative integers n>=1 and 0<=r<=n, write a function to
calculate all the binomial coefficient starting from C(n,r) to C(n,n),
output -1 if the input is invalid
```

Step 2: Behaviors

```
Normal: output 6, 4, 1 for input (n,r) = (4,2)
Boundary_1: output 1 / 1, 1 for input n = 1 (testing n = 1)
Boundary_2: output C(n,0), ..., C(n,n) for input r = 0 (testing r = 0)
Boundary_3: output 1 for input r = n (testing r = n, only one output)
Nonsense_1: return -1 for input n < 1; (n == 0)
Nonsense_2: return -1 for input r < 0; (r == -1)
Nonsense_3: return -1 for input r > n; ((n,r) = (2,3))
```

L12: Exception

`try` & `catch`

1. the **first** catch block with the **same** type as the thrown exception object will handle the exception
2. `catch(...) {}` is the default handler that matches any exception type
3. You cannot write a catch block unless you have a try block before it
4. You can throw an exception from anywhere, instead of just within a try or catch block
5. It is always good coding styles to specify in the EFFECTS clause that when a function would throw an exception:

```
int factorial (int n);
    // EFFECTS: if n>=0, return n!; otherwise throws n
```

Backpropagation: locate the handler

1. browse through the remaining part of the function, if not found, proceed to step 2
2. browse through caller of the function, if not found, proceed to step 3
3. browse through caller of the caller... (until reach main function)

   Once found: run all the commands **in and after** the catch block **that catch the exception.**

```
void foo()
{
    try
    {
        throw 'a'; // try int and double
    }
    catch (char c)
    {
        cout << "Char " << c << " caught!" << endl;
    }
    cout << "Foo executed." << endl;
}

void bar()
{
    try
    {
        foo();
    }
    catch (int i)
    {
        cout << "Int " << i << " caught!" << endl;
    }
    cout << "Bar executed." << endl;
}

int main()
{
    try
    {
        bar();
    }
    catch (...)
    {
        std::cout << "Default catch!" << endl;
    }
    cout << "Main executed." << endl;
    return 0;
}
```

Reference

[1] Weikang, Qian. VE280 Lecture 10-12.

[2] Changyuan, Qiu. VE280 Midterm Review Slides Part 3. 20FA.