Lecture 9: Program Argument

Arguments in C++

Many Linux commands are programs that take arguments:

```
diff file1 file2
g++ test.cpp
```

You could take a look at `/usr/bin` and `/bin` directories if you like.

In C++, program arguments are passed to the program through `main()`:

```cpp
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
    cout << argc << endl;
    for(int i = 0; i < argc; ++i){
        cout << argv[i] << endl;
    }

    return 0;
}
```

- `argv`: an array of C-strings: stores all arguments including program name;
- `argc`: the number of strings in `argv`

You may run the above program in command:

```
./test ve280 midterm
```

The output would be:

```
3
./test
ve280
midterm

```

Manipulating C-Strings

Converting C-strings to numerical types.

```cpp
int main(int argc, char* argv[]) {
    // Method 1: atoi.
    int x = atoi(argv[1]);
    cout << x << endl;
    // This is valid, but BAD.
    // Clang-tidy: `atoi` converts a string to an integer value, but will not report conversion errors.

    // Method 2: strtol.
    char * pEnd = nullptr;
    long int y = strtol(argv[1], &pEnd, 10);
    cout << y << endl;

    // Method 2: stoul.
    unsigned long z = stoul(argv[1]);
    cout << z << endl;
}
```

Converting C-strings to C-arrays and `std::string`s.

```cpp
int main(int argc, char* argv[]) {
    // Incorrect:
    char carr[strlen(argv[1])] = argv[1];
    // Error: Need compound array initializer.

    // Correct:
    char *carr = new char[strlen(argv[1])+1];
    // Clang-tidy: use auto.

    // Better:
    auto *carr = new char[strlen(argv[1])+1];

    // Use `strcpy` to copy
    strcpy(carr, argv[1]);
    cout << carr << endl;

    // Remember to delete after new
    delete[] carr;

    // Can be assigned to a std::string
    string str = argv[1];
    cout << str << endl;
}
```

Lecture 10: Streams

A *stream* is just a sequence of data with functions to put data into one end, and take them out of the other.

- I/O streams
- File stream
- String stream

In C++, streams are **unidirectional**, which means that data is always passed through the stream in one direction.

In general there are 2 types of stream data:

- Characters data
    - Communicating between your program and a keyboard or screen;
    - Reading and writing files.
- Binary data: represent text and files in binary.
    - More efficient, less interpretable.

I/O Streams

The input stream `cin` is an instance of class `std::istream`. Similarly, `cout` is an instance of `std::ostream`).

There is another output stream object defined by the `iostream` library called `cerr`. This stream is identical in most respects to the `cout` stream. In particular, its default output is also the screen. By convention, programs use the `cerr` stream for error messages.

`>>` and `<<` operators

This operator (`>>` and `<<`) applied to an I/O stream is known as *extraction/insertion operator*. It is overloaded as a member function for arithmetic types, stream buffers and manipulators.

For the scope of VE280, you should be familiar with how these operators handles arithmetic types, *i.e.* how they extracts/inserts and parses characters sequentially from the stream to interpret them as the representation of a value of the proper type, which is stored as the value of `val`.

`std::istream ::operator>>` (the extraction operator) is one of it's member function.

```
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>> (float& val);
istream& operator>> (double& val);
istream& operator>> (long double& val);
istream& operator>> (void*& val);
```

`std::istream ::operator>>` (the insertion operator) is one of it's member function.

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

Note that the return value is a reference to `istream` / `ostream`, so it can be cascaded like `cin >> foo >> bar >> baz;`

Buffer

`cout` and `cin` streams are buffered, in contrast, output sent to `cerr` is not buffered.

This means input/output inserted into a stream is saved by the underlying operating system in a region of memory called a *buffer*. The content in the buffer is written to the output only when specific actions are taken, and once the buffer content is written to the output, the buffer is *cleaned*.

When using a buffered output stream, the following events make the buffer content written:

- A newline is inserted into the stream;

  `cout << “ok” << endl;`

  `cout << “ok\n”;`

- The buffer is explicitly flushed;

  `cout << “ok” << flush;`

- The buffer becomes full;
- The program decides to read from `cin`;
- The program exits.

Similarly, characters gathered by `cin` are stored in a buffer until the enter key is pressed.

Useful Functions

Print with Fixed Field Width

In `iomanip` library, the `setw()` manipulator sets the width of the following number to the specified number of positions and right-aligns the number within that field. It pads with spaces.

For example, the output of

```
int x = 1;
int y = 2;
cout << x << setw(4) << y << endl;
```

would be `1 2`.

`getline()`

If you need to read strings including blanks or tabs, `getline()` would be useful.

```
istream& getline (istream& is, string& str);
```

`getline()` reads all characters up to but not including the next newline and puts them into the string variable, and then **discards the newline**.

Assume the input is `it's\tan\tinput\n`.

```
string x, y, z;
cin >> x >> y;
getline(cin, z);
```

What would be the value of `z`?

In fact, `z` would be " input", with a leading space and no `\n` at the end.

`get()`

The `get()` function reads a single character, whitespace or newlines.

```
istream& get (char& c);
```

Assume the input is `it's\tan\tinput\n`.

```
char ch;
string x, y, z;
cin >> x >> y;
cin.get(ch);
getline(cin, z);
```

What would be the value of `z` now?

In fact, `z` would be "input", without a leading space nor `\n` at the end.

Failed Input Streams

You can test the state of a stream by using it where a bool is expected.

```
std::ios::operator bool;
```

It is often the case to write `while(cin)` and `if(cin)`.

Alternate Streams

You can also use the Linux I/O redirection facility to move the input end of the stream from the keyboard or the output end of the stream from screen to a file.

```
$ ./program < input > output
```

File Stream

Files streams are defined in `fstream` library.

Declare an input/output file stream object, we write

```
ifstream iFile;
ofstream oFile;
```

The file stream object must be connected to a file. Connecting a stream to a file is opening the file for the stream. After using a file, it should be closed, which disconnects the stream from the file.

Although the system will automatically close files if you forget as long as your program ends normally, you should always explicitly close the file. This reduces the chance of a file being corrupted if the program terminates abnormally.

```
iFile.open(“myText.txt”);
iFile.close();
```

The methods of file stream is similar to that of I/O streams:

```
iFile >> foo;
oFile << bar;

if(!iFile)
    ...
while(getline(iFile, line))
    ...
```

The file stream enters the failed state if:

- It cannot be opened.
- You attempt to read past the end of the file.

Failed file stream state may be checked by evaluating the stream object, by using it where a bool is expected.

```
iFile.open(“a.txt”);
if(!iFile) {
    cerr << “Cannot open a.txt\n”;
    return -1;
}
```

String Stream

Files streams are defined in `sstream` library.

Declare an input/output file stream object, we write

```
istringstream iStream;
ostringstream oStream;
```

When we use input string stream, it is usually assigned a string it will read from by `str(std::string)`.

```
iStream.str(line);
```

The methods of file stream is similar to that of I/O streams:

```
iStream >> foo;
oStream << bar;
```

We fetch the string value of the string stream using the member function `str(void)` of a string stream.

```
string result;
result = oStream.str();
```

Lecture 11: Testing

Difference between testing and debugging:

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

Five Steps in testing:

1. Understand the specification
2. Identify the required behaviors
3. Write specific tests
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
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();
}
```

Step 5: Implement

```
int fact(int n) {
    if (n < 0) return -1;
    int ret = 1;
    for (int i = 2; i <= n; ++i)
        ret *= i;
    return ret;
}
```

Credit

SU2019 & SU2020 VE280 Teaching Groups.