ECE2800J Mid RC Part 3

SJTU-GC

Yang, Jingwen

Warnings

- Always remember that RCs only contain some key points, they can not be used as a substitute of lectures.
- If there is any difference between RC slides and lecture slides, always refer to lecture slides.
- Optional contents are not required, and there are potential errors.
- If you find anything wrong in this part, tell me as soon as possible!

Exam Suggestions

- Do review the concepts and examples in the lecture slides.
- Do review the exercises and projects.
- Be really careful when you are reading the question requirements.
- Do not stick to one difficult question for too long time. Learn to skip and manage your time wisely.
- Be familiar with writing code by hand.
- Be confident. The midterm exam is not very difficult.

L9: Program Arguments

Grammar

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

- `argc` (argument counts): the number of arguments, **including** the name of the program!
- `argv` (argument vectors): the array of arguments (C-style strings).
- Although you can change the names of `argc` and `argv` to any other names you like, you are highly recommended to **follow the conventional naming**, especially in the exam!

Example

```
./my_program arg1 arg2 arg3
```

- `arg1`, `arg2`, `arg3` are the arguments passed to the program.
- The arguments (including `./my_program`) are passed to the `main` function as parameters.
- If we run the above command to the following program,

```cpp
#include <iostream>
int main(int argc, char* argv[]) {
    std::cout << "Number of arguments: " << argc << std::endl;
    for (int i = 0; i < argc; i++) {
        std::cout << "Argument " << i << ": " << argv[i] << std::endl;
    }
    return 0;
}
```

You will get

```
Number of arguments: 4
Argument 0: ./my_program
Argument 1: arg1
Argument 2: arg2
Argument 3: arg3
```

Others

`stoi` and `atoi`

Here is something interesting.

```cpp
#include <iostream>
#include <string>
#include <cstdlib>
int main(int argc, char* argv[]) {
  int v1 = std::stoi(argv[1]); // <string>, 'std::' necessary, c++11
  int v2 = atoi(argv[2]); // <cstdlib>, 'std::' optional
  int v3 = std::atoi(argv[3]); // <cstdlib>, same to the above
  std::cout << "The first value is: " << v1 << std::endl;
  std::cout << "The second value is: " << v2 << std::endl;
  std::cout << "The third value is: " << v3 << std::endl;
  return 0;
}
```

If we run

```
./others 1 2 3
```

We will get

```
The first value is: 1
The second value is: 2
The third value is: 3
```

You know you can use `atoi` function to convert a C-style string into an integer is enough.

About Program Name (Optional)

Why we need to type `./` to run our programs, but we can directly use commands like `diff`?

- Environment Variable: `$PATH`.

  Run `echo $PATH` to see what are in the environment variable.

- Linux system will not check current directory `.` by default for security considerations.

Exercise 9

- Finish the following program `ex9.cpp`. This program reads any number of `int` program arguments and sums them up. Don't worry about illegal input.

  ```
  #include <iostream>
  #include <cstdlib>
  int main(__________(1)__________) {
      int sum = 0;
      for (________(2)________) {
          sum += std::atoi(____(3)____);
      }
      std::cout << sum << std::endl;
      return 0;
  }
  ```

- Suppose `ex9.cpp` is compiled to a program named `ex9`, which is stored in the directory `~/ECE2800J/FA2025/RC3/`. Assume now you are in the `~/ECE2800J/FA2025/` directory. Please write a **single** Linux command, which can run `ex9` in current directory. The command sums up three numbers `1 2 3`. Do not use `cd` command or any control operators (`&&`, `||`, etc).

  ```
  ________________(4)_________________
  ```

- According to (4), what is the content of `argv[0]`?

  ```
  ____________(5)_____________
  ```

L10: IO

Stream Operations

- Insertion Operator `<<`: It is used to insert things into the output stream. It knows how to convert all of the other standard data types to characters before inserting them into the stream.
- Extraction Operator `>>`: It is used to extract things from the input stream. It knows how to convert characters into values of simple types and strings.

`std::cout`

Basic Knowledge

- Buffered: there is a region of memory that holds data during `std::cout` operations.
- The buffer content is written to the output only when:
  - A newline (e.g., `std::endl` or `'\n'`) is inserted into the stream. E.g., `std::cout << "ok" << std::endl;`.
  - The buffer is explicitly flushed. E.g., `std::cout << "ok" << std::flush;`.
  - The buffer becomes full.
  - The program decides to read from cin.
  - The program exits.
- Once the buffer content is written to the output, the buffer is cleaned.
- If some content is not printed out, it may be still in the buffer.
- In contrast, output sent to `std::cerr` is **NOT** buffered.

Optional

- Is the buffer really flushed when a `'\n'` is inserted into `std::cout`?
  - Line buffered or fully buffered.
  - Things become different when it comes to redirection operator `>`. See `output.cpp`.
  - **Ignore it in your exam. Just be consist with lecture slides.**

`std::cin`

Basic

- Buffered.
- Characters typed (which are to be gathered by `std::cin`) are stored in a buffer until the enter key is pressed.
- Ignore leading blanks.
- `std::cin.get`: It reads a single character, whitespace or newline:

  ```
  char c;
  std::cin.get(c);
  ```

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

Optional

- Line buffer: Store what you typed. You can modify your input easily.

- Input buffer: When you press enter key, all the content in the line buffer are moved to the input buffer. `std::cin` reads from here, and leaves the remaining in the input buffer. (You'd better know it.)
- Is the following `while` loop OK when we use a redirection operator `<`? Yes. See `input_redir.cpp`.

  ```
  #include <iostream>
  #include <string>
  int main() {
      std::string str;
      while (std::getline(std::cin, str)) {
          std::cout << "Read string: " << str << std::endl;
      }
      return 0;
  }
  ```

Exercise 10

The following program reads your age and name from `stdin`, and prints them to `stdout`. Is the program correct? If yes, provide an example input and output. If no, modify the program to make it correct.

```
#include <iostream>
#include <string>
int main() {
    int age;
    std::string name;
    std::cout << "Enter your age: ";
    std::cin >> age;
    std::cout << "Enter your name: ";
    std::getline(std::cin, name);
    std::cout << "Age: " << age << ", Name: '" << name << "'" << std::endl;
    return 0;
}
```

File Stream

Basic

- Basic usage:

  ```
  #include <iostream>
  #include <string>
  #include <fstream>
  int main() {
      std::ifstream infile("std_input.txt");
      std::ofstream outfile("std_output.txt");
      std::string str;
      while (std::getline(infile, str)) {
          outfile << str << std::endl;
  ```

```
    }
    infile.close();
    outfile.close();
    return 0;
}
```

- Close the file stream once you do not need to use the current file.
- Open multiple files in the same file stream is undefined and very **dangerous**!
- Multiple file streams open the same file is also very **dangerous**!
- The following code may cause the `while` loop to run an extra loop, because when the pointer points to right before `EOF`, `iFile` returns `true`.

  ```
  std::ifstream iFile("some_file");
  std::string line;
  while(iFile) {
      std::getline(iFile, line);
      std::cout << line << std::endl;
  }
  ```

  Use `getline` instead.

  ```
  while(std::getline(iFile, line)) {
      std::cout << line << std::endl;
  }
  ```

Optional

- File stream modes:
  - `std::ios::in`: Open the file in read mode. Equivalent to 8. Default for `std::ifstream` and `std::fstream`.
  - `std::ios::out`: Open the file in output mode. Equivalent to 16. Default for `std::ofstream` and `std::fstream`.
  - `std::ios::trunc`: Truncate an existing stream when opening (i.e. clear the content). Equivalent to 32. Default for `std::ofstream`. Only affect `std::ofstream` and `std::fstream`.
  - `std::ios::app`: Open the file in append mode. Equivalent to 1.
  - Leave others for you to explore.
- `|`: bitwise or operation. Used to bind multiple modes together.
- File stream `std::fstream`. Can both do input and output. Default as `std::ios::in | std::ios::out`. Note that the default writing mode is overwriting.
- Some differences:
  - `std::ofstream` is default as `std::ios::out`. However, either choosing `std::ios::out`, `std::ios::trunc` or `std::ios::out | std::ios::trunc` will lead to same behavior (i.e. clear the original content and write what you want).

    - `std::ofstream` is default as `std::ios::in | std::ios::out`. It will not clear the original content. In `std::ios::out` mode, the write pointer points to position 0 (i.e. the beginning), and we rewrite the content from position 0 (not insert). For example, the file content is `"Hello World!"`, then we use the default `std::fstream` to open it, and write `"New"` into it. The final content is `"Newlo World!"`.
- When using `std::fstream`, be **very careful** with the read pointer and the write pointer. Note that the operating system usually provides only one file offset to indicate the current position. Most of the time the two pointers will move together (I failed to find a counter example).
- `seekp`: Change the current write position. `tellp`: Tell the current write position. `seekg`: Change the current read position. `tellg`: Tell the current read position. They are mostly used for binary files.
- More details are left for you to explore.

String Stream

- Very similar to file stream.
- `std::istringstream`: It reads characters in a string and convert them into values of proper types.
- `std::ostringstream`: It writes to a string. It knows how to convert standard data types into characters and insert them into the string.
- `std::stringstream` (optional): It can both read and write. If you want to reuse the existing `std::stringstream` object `ss`, remember to call `ss.str("")` and `ss.clear()` to refresh it. More details are left for you to explore.
- Example usage:

  ```
  #include <sstream>
  #include <string>
  #include <iostream>
  int main(){
      std::istringstream is;
      std::ostringstream os;
      std::string foo;
      int bar;
      std::string s = "ECE 2800J";
      is.str(s);
      is >> foo >> bar;
      os << foo << bar;
      s = os.str();
      std::cout << s << std::endl;
      return 0;
  }
  ```

  Question: What is the final output?

  ```
  __________(6)___________
  ```

L11: Testing

Concepts

- Testing: Discovering that something is broken.
- Debugging: Fixing something once you know it is broken.
- Incremental testing: Test individual pieces of your program (such as functions) as you write them. This will often require you to write extra code (**the driver program**) to test your program effectively.

Steps

- Understand the specification. For example, the specification comments `REQUIRES`, `EFFECTS`, etc.
- Identify the required behaviors.
  - For any specification, boil the specification down to a list of things that must happen. See examples in the lecture slides.
- Write specific tests.
  - Simple inputs (simple cases).
  - Boundary conditions (boundary cases). For example, `0`, `INT_MAX`, etc.
  - Nonsense (nonsense cases).
- Know the answers in advance.
- Include stress tests.
  - large test cases.
  - long running test cases.

Assert

- Usage: `assert(judgment);` If `true`, nothing happens. If `false`, the program exits with an error message.
- If you want to test some function using `assert`, the judgment should test exactly what the function does. Partial correctness is not enough.
- Disable `assert()`:
  - Define `NDEBUG` before including `<cassert>`:

    ```cpp
    #define NDEBUG // disable assert()
    #include <cassert>
    ```

  - Specify it on the command line of the compiler:

    ```
    g++ -DNDEBUG ...
    ```

Others

Here are some other useful variables:

- `__func__`: Defined by compiler, and holds function's name.
- `__FILE__`: Name of current file.
- `__LINE__`: Current line number.

L12: Exceptions

Concepts

- Recognize and handle: partial function with `REQUIRES`. Check at runtime check.
  - Modify the inputs. / Return default outputs.
  - `assert(condition)` terminates the program if `condition` is not true.
  - Encode "failure" in the return values.
- Exception: something bad that happens in a block of code, preventing the block from continuing to execute.
- Mechanism: if the exception occurs, the program will move to the handler.

Try-Catch Block

Basic

```
void foo() {
    try { Block }
    catch (Type var) { Handler }
}
```

- `try`: The code in the `try` block may throw some type of exception.
- `catch`: Each `catch` block tries to catch some type of exception, unless it is a default handler `catch (...) {}`.
- If the exception is successfully handled in the catch block, execution continues normally with the first statement following the catch block.
- `catch (Type var) {}`(Optional): Here `var` is passed by value, so it is a copy of the original object. Sometimes using `(const Type& var)` is more recommended.
- You **cannot** write a `catch` block unless you have a `try` block before it.
- Exception will be propagated along the calling function stack. Only the first catch block with the same type as the thrown exception object will handle the exception.
- You can have multiple `catch` blocks following a `try` block.

  ```
  try {
      if (foo) throw 2.0;
      // some statements go here
      if (bar) throw 4;
      // more statements go here
      if (baz) throw ‘a’;
  }
  catch (int n) {}
  catch (double d) {}
  catch (char c) {}
  catch (...) {}
  ```

Exercise 12

- What is the output of the following program?

```
#include <iostream>
void bar(double x){
    throw x;
    try{
        throw x;
    }
    catch(double a){
        std::cout << "double in bar\n";
    }
    std::cout << "exit bar\n";
}
void foo(int x){
    try {
        bar(x);
    }
    catch(int a){
        std::cout << "int in foo\n";
    }
    catch(double b){
        std::cout << "double in foo\n";
    }
    std::cout << "exit foo\n";
}
int main(){
    int x = 6;
    foo(x);
}
```

```
______________(7)_______________
______________(8)_______________
```

References

[1] Qian, Weikang, ECE2800J 25FA Lectures 9-12.

[2] ECE2800J 24FA Mid RC Part 3.