CS 246 Fall 2021 — Tutorial 3

September 22, 2021

Summary

1 Strings **1**

2 Streams **1**

3 Parameters **3**

4 Example: Complex number multiplication **4**

5 Tips of the Week: Adding Scripts to Path **5**

6 Vim Tips of the Week: Visual Mode and `.vimrc` **6**

1 Strings

- In C++, there is a `std::string` type to replace C-style character arrays.
- `#include <string>`
- **Note:** In general, `std::string` in this course refers to C++-style strings. Any time that C-style character arrays are used will we say so explicitly.
- Common supported operations include (some as member functions of `std::string`):
  - indexed access using `[]` or `at()`.
  - concatenation using `+, +=` (both with `std::string` and with C-style strings). For `+`, at least one side must be a `std::string`. For `+=`, it must have a `string` on its left.
  - lexicographical comparison using `==, !=, <, >, <=, >=` (also supports C-style strings)
  - others: `length, clear, substr, find, rfind, replace`
  - see `https://en.cppreference.com/w/cpp/string/basic_string` for more details
- Use the `c_str()` member function to access a C-style version (`const char *`[1]) of the string.

2 Streams

- In C++, streams are used to handle I/O from stdin/stdout/stderr, files, and strings.

[1] i.e. you should not modify the contents of what this pointer points to. In fact, that has undefined behavior.

2.1 Input Streams

- An input stream is a stream from which information can be read.
- By default, reading from an input stream is whitespace-delimited.
- Functions common to all input streams:
  - `<stream> >> <string>`: reads the next word from `<stream>` and stores it in `<string>` where `<string>` is the name of a variable of type `string`.
  - `<stream> >> <int>`: reads the next integer from `<stream>` and stores it in `<int>` where `<int>` is the name of a variable of type `int`. The fail bit is set to true if no characters in the stream beyond the current position can be interpreted as an `int`.

    Similar functions exist for all built in C++ types, e.g. `bool`, `char`, `float`, etc.
  - `eof()`: returns true if the stream has reached end-of-file (EOF).
  - `fail()`: returns true if a read from the stream has failed, including having reached EOF.
  - `clear()`: sets the fail bit to false.
  - `ignore()`: skips the next character in the stream.

2.2 Output Streams

- An output stream is a stream to which information can be sent.
- Functions common to all output streams:
  - `<stream> << <var>`: puts the information stored in `<var>` in `<stream>`. This function exists for all built-in C++ types.

2.3 IO Streams

- `#include <iostream>`
- Includes `std::cin` (stdin), `std::cout` (stdout), and `std::cerr` (stderr).
- As previously described, these are the three streams that all programs have. Input and output can be read from and written to these streams.

2.4 File Streams

- `#include <fstream>`
- Types of file streams:

`std::ofstream` file stream only for output

`std::ifstream` file stream only for input

- For example, to open a file to read in from:

  ```
  std::ifstream file{"file.txt"};
  ```

- By default, creating an `std::ofstream` to a file which already exists will overwrite the data in the file. If the file doesn't exist, it will be created.

2.5 String Streams

- `#include <sstream>`
- String streams are streams in which formatted information can be stored, and from which a string matching the stored information can be obtained.

  `std::ostringstream` string stream only for output

  `std::istringstream` string stream only for input

- `str()`: This obtains a C++-style string matching the information stored in a string stream.

  Note: the following expression results in a dangling pointer:

  ```
  std::ostringstream oss{...};
  const char *p = oss.str().c_str();
  ```

  The string returned from `str()` is temporary and the memory allocated for the string is freed once this statement finishes.

3 Parameters

- Parameters are variables which are passed to a function.

3.1 Overloading

- In C++, we can have multiple functions with the same name as long as the number of parameters and/or the types of parameters are different.

  ```
  int foo(char c, int n);        int foo(int n);

  int foo(char& c, int n);       int foo(const char& c, int n);
  ```

- **Note:** Functions cannot be overloaded based on return type alone.

3.2 Default Parameters

- The parameters of a function can be given default values.

  For example,

  ```
  void foo(int n = 75);
  ```

  There are now two ways to call `foo`:

  ```
  foo();
  foo(10);
  ```

  Using default parameters is equivalent to having two functions with the same body and different parameters (and it's a way to reduce code duplication).

- In a function declaration, all default variables must come last.

  Example:

  ```
  void foo(int n = 75, char c); // invalid
  void foo(int n = 75, char c = 'a'); // ok
  ```

- **Question:** Which of the following is not a valid overload of `bool foo(int x, char c);`?

  1. `int foo();`
  2. `char foo(char x, int c);`
  3. `bool foo(int c);`
  4. `int foo(int x, char c, int y = 10);`
  5. None of the above.

4 Example: Complex number multiplication

We would like to write a small C++ program that can read in multiple pairs of complex numbers either from the file whose name was specified on the command-line, or from standard input if no such file name was given. If there is more than one command-line argument, or the input file cannot be successfully opened for reading, an error message will be produced to standard error, and the program terminated with a non-zero exit code. For each pair of numbers successfully read in, the numbers will be multiplied and the results printed to standard output. You may assume that the numbers will be correctly formatted, and will either look like $a + bi$, $a - bi$, or a, where a and b are real numbers. (Having no spaces on either side of the plus/minus sign will let us read in each complex number as a single `std::string` value. Remember that the multiplication of two complex numbers is defined as: $(x + yi)x(u + vi) = (xu - yv) + (xv + yu)i$. Let's start by breaking the problem down into what we need to do:

- check the number of command-line arguments provided

  - How do we do this in C++?
  - What values do we need to have?
- if have a file name, open a file and check that was successful
- if no file name, read from `std::cin`
- loop, reading in pairs of complex numbers and stop when reach end-of-file
- for each string value read in, call a helper `convert` function that is initially just the stub that creates a complex number with zeroes
- create an output operator for `Complex`
- create a multiplication operator stub for two `Complex` numbers that returns $0 + 0i$ for now

```
struct Complex {
    float real, imag;
};

Complex convert( string s ) {
    Complex c;
    c.real = c.imag = 0.0;
    // fill in later
    return c;
}
```

- fill in `convert`
- fill in the `operator*` stub

5 Tips of the Week: Adding Scripts to Path

- Since the test suite format for due date 1 on assignments is compatible with the `produceOutputs` and `runSuite` scripts you wrote in assignment 1, it would be nice to use these scripts without typing the full path to the scripts every time.
- Remember when you type a command in bash, it searches the directories in the `PATH` variable to look for the executables.
- So copy the scripts that you wrote into the `~/bin` directory, and check if your `PATH` variable contains the `bin` directory. If it doesn't, add this to the bottom of your `~/.bash_profile`:

  ```
  PATH="$HOME/bin:$PATH"
  ```

6 Vim Tips of the Week: Visual Mode and `.vimrc`

- In most text editors you can select text and copy/cut/paste them.
- In Vim you do this by entering visual mode by pressing `v`. You can also select whole lines of text by pressing `V` (Shift + v), and blocks of text by pressing `C-v` (Ctrl + v).
- To go back to normal mode, press `Esc`.
- In visual mode you can use normal movement commands like `w`, `b`, but a few keys are different:

  `y` copies the selection

  `d` copies the selection and deletes it

  `c` copies the selection, deletes it, and enter insert mode

- Vim comes with a lot of functionality, but a lot of them are not enabled by default. Put the following in the file `~/.vimrc` for some useful configuration (double quote begins a comment in Vim configuration):

```
set nocompatible          " disable vi-compatible behaviour
filetype plugin indent on " enable functionality based on file type
syntax enable             " enable syntax highlighting

set autoindent            " automatic indentation
set incsearch             " jump to next match when typing during search
set laststatus=2          " always show status line
set mouse=a               " enable mouse support
set expandtab             " use space for indent
set number                " show line numbers
set scrolloff=5           " always show 5 lines before / after cursor
set shiftwidth=0          " use value of tabstop for indent
set smarttab              " tab / backspace adjusts indents
set tabstop=4             " width of a tab / indent
```