CS 246 Spring 2018 — Tutorial 3

May 23, 2018

Summary

1 Strings

- In C++, there is a `string` type to replace C-style character arrays.
- `#include <string>`
- **Note:** In general, `string` in this course refers to C++-style strings. Any time that C-style character arrays is used will be referred to explicitly.
- Common supported operations include (some as member functions of `string`):
  - indexed access using `[]` or `at()`.
  - concatenation using `+`, `+=` (both with `string` and with C-style strings). For `+`, at least one side must be a `string`. For `+=`, it must have a `string` on its left.
  - lexicographical comparison using `==, !=, <, >, <=, >=` (also supports C-style strings)
  - others: `length, clear, substr, find`
- 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 content of what this pointer points to. In fact, that has undefined behavior.

2.1 Input Streams

- An input stream is a stream which information can be read from.
- 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 failbit 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 reaching EOF.
  - `clear()`: sets the failbit to false.
  - `ignore()`: skips the next character in the stream.

2.2 Output Streams

- An output stream is a stream which information can be sent to.
- 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 `cin` (stdin), `cout` (stdout), and `cerr` (stderr).
- As previously described, these are the three streams which 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:

`ofstream`	file stream only for output
`ifstream`	file stream only for input

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

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

- By default, creating an `ofstream` to a file which already exists will overwrite the data in the file. If the file doesn't exist, it will be created.
- **Note:** An `ofstream` object is buffered. To prevent the output file from missing data, it must be closed if that was not done automatically.

2.5 String Streams

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

`ostringstream`	string stream only for output
`istringstream`	string stream only for input

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

  Note: the following expression will result in a dangling pointer:

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

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

2.6 Example: Complex number multiplication using string streams

- We now see a real life application of string streams.
- For a complex number of the form `a+ib`, we will consider a representation of it as a string in C++.
- To do this, we will need a way to convert substrings of this string representation into integers.
- In C, there is a function that converts a (C-style) string to an int (`int atoi(const char *str)`), and some compilers have a function that converts an integer to a C-style string (although that is not in the C/C++ standard).
- Note that in CS 246, `atoi()` is forbidden, since `<cstdlib>` is not allowed to be included in the headers. How do we achieve conversion between integer and string in C++?

- Turns out that we can use string streams!
- To make this easier, we will assume that both the real and imaginary parts are positive.

```
string complexNumberMult(string comp_a, string comp_b){
  int real_a, im_a, real_b, im_b;
  char buffer;

  istringstream stream_a(comp_a), stream_b(comp_b);
  ostringstream ans;

  stream_a >> real_a;
  stream_a >> buffer; // buffer is '+'
  stream_a >> buffer; // buffer is 'i'
  stream_a >> im_a;


  stream_b >> real_b;
  stream_b >> buffer; // buffer is '+'
  stream_b >> buffer; // buffer is 'i'
  stream_b >> im_b;

  ans << real_a * real_b - im_a * im_b; // putting in the real part
  ans << "+i"; // putting in "+i"
  ans << real_a * im_b + real_b * im_a; // putting in the imaginary part

  return ans.str();
}
```

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);
```

- Functions that differ by a constant parameter are also overloadable **only if the parameter is also a reference**.

```
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 References

- Syntax:

  ```
  int x = 42;
  int &rx = x;
  ```

- A reference is basically a automatically-dereferenced constant pointer to data. What does this mean?

  – **Constant pointer to data:** the data which a reference is refering to cannot be changed after initialization.

  - **Dereferenced:** When working with pointers, the pointer must be dereferenced to access the data. For example, we can define `int *xp = &x;`. Then, to access the value pointed by `xp`, you must use `*xp`.

    References don't need to be dereferenced and cannot be dereferenced (unless they're a reference to a pointer).

- Consider the code below:

  ```
  int x = 10, y = 5;

  int &rx = x;
  int &ry = y;

  int *px = &x;
  int *py = &y;

  int res1 = (*px + *py) * (*px - *py);
  int res2 = (rx + ry) * (rx - ry);
  ```

  The two variables `res1` and `res2` contain the same value but the calculation with references looks simpler.

- A reference is an alias to data; the reference and the data share the address which they are refering to in memory.

  ```
  int x = 17;
  int &rx = x;

  // these two lines print the same address
  cout << &x << endl;
  cout << &rx << endl;
  ```

- **Note:** references cannot "point" to nothing — they must always be initialized, and you can't directly initialize them to `nullptr`.

4.1 Pass-by-Reference

- **Pass-by-value:** makes a copy of the parameter passed for use during the function. Changes to the parameter do not exist outside of the scope of the function.
- **Pass-by-reference:** creates an alias to the parameter.
- Writing a function which take a pointer to a variable simulates **pass-by-reference**. (A copy of the address is made but changes to the variable persist after the function call.)
- **Passing-by-reference** is usually faster than **passing-by-value** because copying the parameter usually takes more time than copying an address.

- Literals cannot be passed by reference since it is not a *lvalue* (something that has an address), with the exception of **pass-by-const-reference**.
- **Pass-by-const-reference** occurs when we pass an argument as a const reference. You can pass a literal as a const reference since there is no danger of the function changing the literal.
- By doing so, we get 2 main benefits:
  - Large structures are not copied and can't be changed
  - Can pass in literal values
- Example of **pass-by-reference** vs. **pass-by-const-reference**:

```
int foo(int &x, const int &y) { ... }
int main() {
    int a = 42;
    foo(a, a);
    foo(a, 43);
    foo(43, a); // invalid, what does it mean to change a literal?
    foo(43, 43); // as above
}
```

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
  ```