CS 246 Fall 2016 - Tutorial 3

September 28, 2016

1 Summary

- Streams
- References
- Parameters

2 Vi Tip of the Week: Tabs

- When programming, it is often useful to have multiple files open at the same time. Multiple files can be opened with tab both when opening vi and when it is already opened.
- To open multiple tabs when openin vi, type multiple file names after vi followed by `-p`.
- To open additional files, enter command mode and type `:tabedit file`.
- When multiple tabs are open, enter `gt` to switch to the tab to the right of the current tab. `gT` switches to the tab to the left of the current tab.
- If multiple files are opened, all files can be closed and saved by entering `:wqa`.

3 Vi Tip of the Week: Matching Brackets

- `%`: when the cursor is on a bracket, pressing `%` moves the cursor to the matching bracket

4 Streams

In C++, streams are used to handle IO and files.

4.1 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 strings:
  - `eof()`: returns true if the stream has reached an end-of-file.
  - `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.
  - `<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 int from `<stream>` and stores it in `<int>` where `<int>` is the name of a variable of type int. If the next characters in `<stream>` cannot be interpreted as an int, the failbit is set to true.
  - Similar functions exist for all built in C++ types, e.g. bools, chars, floats, etc.

4.2 Output Streams

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

4.3 IO Streams

- `#include <iostream>`
- Includes `cin`, `cout`, and `cerr`.
  - `cin` is stdin
  - `cout` is stdout
  - `cerr` is stderr
- As previously described, these are the three streams which all programs have. Input and output can be redirected to and from these streams.

4.4 File Streams

- `#include <fstream>`
  - `ofstream`: file stream only for output
  - `ifstream`: file stream only for input
- To open a file:
  - `ifstream file{filename};`
- By default, opening a ofstream to a file which exists overwrites the data in the file. If the file doesn't exist, it is created.

4.5 String Streams

- `#include <sstream>`
- String streams are streams which formatted information can be stored in and from which a string matching the stored information can be obtained.
  - `ostringstream`: stringstream only for output
  - `istringstream`: stringstream only for input
- `str()`: obtain a C++ style string matching the information stored in a stringstream
- stringstreams are often used to safely convert the information stored in a string to an integer.

5 References

- Syntax:

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

- A reference is basically a dereferenced constant pointer to an object. What does this mean?
  - Constant pointer to an object: the object which a reference is refering cannot be changed after initialization.

- Dereferenced: When working with pointers, the pointer must be dereferenced to access the value of the objects, e.g. `int * xp = &x; *xp`. References 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 more simple.

- A references is an alias to to an object. The reference and the object share the address which they are refering to in memory.

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

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

- Note: references cannot be null.

6 Parameters

Parameters are variables which are passed to a function.

6.1 Overloading

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

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

- Note: functions cannot be overloaded based on return type.

6.2 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:

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

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

- Any number of parameters to a function can have default parameters but if a parameter has a value, all parameters to its right must have default values.

  Example:

```
void foo(int n = 75, char c); // not valid
void foo(int n = 75, char c = 'a'); // valid
```

- 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.

6.3 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 functions.
- Pass-by-reference: creates an alias to the object which is a parameter. Literals cannot be passed by reference (they do not have addresses).
- Writing a function whichs 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.)
- Pass-by-const-ref occurs when we pass an argument as constant reference
- By doing so, we get 2 main benefits:
  - Large structures are not copied and can't be changed
  - Can pass in literal values
- Passing-by-reference is usually faster than passing-by-value because copying the parameter takes more time than creating a 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
}
```