Kevin Chen

11/14/15

C++ Notes (Abbreviated)

Statements and Flow Control

* Compound statements is a group of statements all grouped together in a block, enclosed in curly braces. This is a compound statement:
  + {statement1; statement2; statement3}
* Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.
* Don’t need block (curly braces) if there is just one statement after a flow control statement. Example:
  + if (x == 0)

cout << “zero”;

* goto and labels
  + goto: an unconditional jump. After the keyword “goto”, specify which label you want to go to. Label syntax: label name followed by colon. (Example: mylabel:)
* switch: similar to if-loop. Checks if expression is equal to each case. Use “case” and “default” instead of “if/else if” and “else”

switch (expression)

{

case constant1:

group-of-statements-1;

break;

case constant2:

group-of-statements-2;

break;

.

.

default:

default-group-of-statements

}

* + switch will continue executing the cases below if there is no break statement.

Functions

* Syntax: [return type] [name] ( argument1, argument2 ...) { statements }
  + Example: int add(int a, int b) {return a + b};
* Return of main: main() returns an int.
  + 0 = success.
  + EXIT\_SUCCESS = success
  + EXIT\_FALIURE = failure
  + The bottom two values are defined in the header <cstdlib>
* Pass in references instead of values: type “&” after type name in parameter.
  + Example: void double(int& a) {a = 2 \* a}
* Passing in references takes less effort for computers than creating a copy of that value, esp. for objects such as strings.
* Can guarantee reference parameters won’t be changed by using “const” in front of parameter type.
  + Example: string concatenate (const string& a, const string& b)
* Inline functions: Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function. Add the word “inline” before the return type.
  + Example: inline string concatenate (const string& a, const string& b)
* Default values in parameters: used if no value is specified, default value is used instead. Must come last in method parameters.
  + Example: int divide (int a, int b=2) can be called divide(6) or divide(6, 3). First will return 3, second would return 2.
* Function declarations
  + Must be done before the code which calls them. (Thus, put them above main.)
  + Can specify actual code in body, or type it later. Optional: include names of parameters. Examples of declaration:
    - int protofunction (int first, int second);
    - int protofunction (int, int);

Overloads and Templates

* Two functions can have same name if their parameters are different in quantity or type.
* Templates: used for methods that can be overloaded with many types, but have same body. Syntax: template <template-parameters> function-declaration
  + template <class T>

T sum (T a, T b)

{

return a+b;

}

* + “class” can be replaced with “typename”.
  + “SomeType” represents a generic type that will be determined on the moment the template is instantiated.
* Calling template function: name <template-arguments> (function-arguments)
  + Example: x = sum<int>(10,20);
  + If the type is used as parameters, you don’t need to specify type: x = sum(10, 20);
* Non-Type Template Arguments: Example: template <class T, int N>
  + This call: std::cout << fixed\_multiply<int,2>(10) << '\n'; would return 20.
  + Can’t pass in variable for template argument because value of template parameters is determined on compile-time to generate instance of the method.

Scopes

* Declare variable outside of methods to make it global.
* In each scope, a name can only represent one entity. But name can represent two entities if they are from different scopes.
* Blocks define a new scope.
* Namespaces: allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope.
  + Syntax:

namespace identifier

{

named\_entities

}

* + Access namespace entities with [namespace name]::[entity]
  + Can be split into multiple lines.
  + Can extend across translation units (across different files of source code)
* “using” namespace can avoid having to type namespace name every time. Instead, just type actual namespace entities within the scope.
  + using first::x; // The variable “x” inside the namespace “first” can now be used by just calling “x” instead of “first::x”.
  + using first // All entities inside “first” can be called without having to type the namespace before it.
* Namespaces can be aliased with new names using this syntax:
  + namespace new\_name = current\_name;
* All the entities (variables, types, constants, and functions) are declared in “std” namespace.
* Storage classes
  + Static storage: storage for these variables are allocated for the entire duration of the program. Example: global variables
  + Automatic storage: storage for these variables is only available during the block in which they are declared. After that, storage may be used for local variable of another function. Example: local variables
  + Static storage automatically initialized to zero. Local variable automatically initialized to an undetermined (random) value.