Kevin Chen

11/22/15

Other Language Features

Type Conversions

* Implicit conversions: don’t need explicit operator.
  + Standard conversion: affect fundamental types (e.g. int to float)
  + Promotion: same value in destination type. (e.g. int to short)
  + Other conversions might not result in exact same value (e.g. negative int to unsigned int)
  + Arrays/functions implicitly convert to pointers.
  + Pointer upcast: pointer to derived class converted to pointer of accessible and unambiguous base class, without modifying const or volatile qualification.
* Implicit Conversions with classes
  + single-argument constructors, assignment operator, and type-cast operator.
  + Type-cast operator: method to cast a particular type into another type within the original type’s class. Syntax: “operator” followed by destination type and empty set of parenthesis.
* “Explict” keyword prevents an implicit conversion of the parameter of a constructor or type-cast member function. Add the word “explicit” at the beginning of the method name.
  + Also prevents the method from being called using “=” operator.
* Type casting. Syntax: y = (int) x; – OR – y = int (x);
  + But unrestricted explicit type-casting castings can be applied too indiscriminately. Example: cast pointer to one type into a pointer of another type independent of what they point to.
  + Four specific casting operators help control these type of conversion. Syntax: <cast\_type> <new\_type> (expression)
* dynamic\_cast: used with pointers and references with class. Ensures pointer still points to correct type.
  + Includes pointer upcast (pointer-to-derived to pointer-to-base)
  + Includes downcast (only if pointed object is valid complete object of pointed type). Base class is not valid complete object type of derived class.
  + Returns null pointer if converting to a non-complete object of pointed type.
  + Throws bad\_cast exception if conversion is not possible.
  + Requires Run-Time Type Information (RTTI)
* static\_cast
  + Cast between two related types.
  + Includes upcast and downcast regardless if safe or not. More dangerous than dynamic\_cast…less check for security.
* Reintepret\_cast
  + Cast between any two types, related or unrelated. Simple binary copy.
* const\_cast: manipulates constness of the object pointed by a pointer.
* typeid: typeid(expression) returns a reference to a constant object of type type\_info (defined under <typeinfo>).
  + Can be used to compare types or get the type of an object using the name() member.
  + When typeid is used on a polymorphic class, it returns most derived complete object.

Exceptions

* Place a section of code under exception inspection by using try-block.
  + Use the throw keyword in the try-block. Once thrown, control is transferred into exception handler defined using the catch keyword.
  + throw accepts one parameter which is passed into catch event handler.
  + Multiple handlers (catches) can be chained, each with parameter type.
  + catch (…) handles any exception.
  + After exception occurs, code execution resumes after the try-catch block.
  + Can nest try-catch blocks so that the internal catch block forwards the exception into the external level. (Have throw inside internal catch.)
* Exception specification
  + Append a “throw()” specifier after normal declaration of method. Put type specifier inside parenthesis to indicate what type the method can throw.
  + If no type, then method cannot throw any exception.
  + If no “throw” clause, then method can throw any type. Normal exception handling
* Standard exception
  + std::exception defined under <exception> header is a base class designed to declare objects to be thrown as exceptions.
  + std:exception has virtual member function what()

Preprocessor Directives

* #define: replaces identifier with value. Example: #define TABLE\_SIZE 100 will replace any instance of TABLE\_SIZE with 100.
  + Can also work with parameters to define function macros. Example: #define getmax(a, b) a>b?a:b replaces any instance of getmax with two parameters with this method, replacing each argument with the actual value.
  + Defined macros last until the corresponding #undef preprocessor directive.
  + # is a special operator. This operator, followed by parameter name, is replaced by a string literal.
  + ## accepts two parameters. Concatenates the two arguments: no blank spaces.
* Conditional inclusions allow to include or discard sections of code if condition is met.
  + #ifdef allows section of code to compile only if the macro specified as the parameter is defined. #ifndef is opposite.
  + #if, #else, and #elif (i.e. “else if”) only allow portions of code to be compiled if condition is met.
  + defined and !defined returns if value is defined or not defined, respectively.
* Line control defines line number within a file name of source code.
  + Syntax: #line number "filename"
  + number is the new line number that will be assigned to the next line of code. Line numbers of successive lines will increased one by one.
  + filename is an optional parameter that redefines the file name.
* Error directive aborts compilation process, generating compilation error that can be specified as its parameter. Syntax: #error [message]
* Source file inclusion (#include)
  + #include <*header*> includes headers provided by the implementation.
  + #include “*file*” includes a file that is searched for in an implementation-defined manner (generally includes current path). If file not found, interprets it as header.
* Pragma directive (#pragma): specifies options that are specific to the platform and compiler you use.
* Pre-defined macros: \_\_LINE\_\_, \_\_File\_\_, \_\_DATE\_\_, \_\_TIME\_\_, \_\_cplusplus, \_\_STD\_HOSTED\_\_.

Input/Output with Files

* ofstream: stream class to write on files.
* ifstream: stream class to read from files.
* fstream: stream class to both read and write from/to files.
* Opening a file: open(filename, mode); Mode is an optional parameter that can take on the values: ios::in, ios::out, ios::binary, ios::ate, ios::app, ios::trunc. Use OR operator to combine these flags.
  + There exists a default mode parameter of the *open* member functions.
  + There also is a constructor that can open a file as well. The two parameters are the same parameters in the open() function.
  + binary mode performs input/output operations independently of any format considerations. non-binary files are known as text files. Some translations may occur because of special characters (e.g. newline and carriage return characters)
* Close method closes the file.
* getline(file, line) gets the next line of "file" and stores it in the String "line". If there wasn't a next line, then it returns false. Third parameter can be used as delimiter.
* State flags check for specific states of the stream:
  + bad() for reading/writing error
  + fail() for reading/writing error and incorrect format (ex. alphabet character extracted when we try to read an int),
  + eof() if file opened and file open for reading has reached the end.
  + good() returns false if any of the above flags are true.
  + clear() resets flags.
* Internal position for getting (reading) and putting (writing)
  + tellg() returns current getting position of type streampos, and tellp() returns current reading position.
  + seekg(streampos) and seekp(streampos) set current reading or writing position.
  + seekg(streamoff offset, seekdir direction) and seekp(streamoff offset, seekdir direction) set current reading or writing position with offset from reference point direction
  + seekdir is an enumerated type with one of three values: ios::begin, ios::cur, ios::end.
  + Each of the stream class member types (streamoff, streampos, and direction) can be just thought of as an int/long (implicitly convertible to/from int)
* Binary files can be read faster through write(memory\_block, size) and read(memory\_block, size)
* When we operate with file streams, they are associated with an internal object called streambuf. Represents an intermediate memory block between stream and physical file.
  + Buffer will be flushed under these cases: when file is closed, when buffer is full, when certain manipulators are called (endl and flush), and when member function sync() is called.