C++ Streams

- C++ provides general-purpose, stream-based facility for program input and output
  - Output: converting variables/objects into char sequences
  - Input: converting char sequences into variables/objects
  - Locale affects formatting in the conversion process
- C++ Stream IO is extensible to user-types
  - Primitive types and standard classes are supported
  - Easy to incorporate user-defined types into C++ Stream IO
- Supports console IO, file IO, etc.
- Can also treat strings as streams
- Can write stream wrappers for networking sockets, etc.
Console-IO Streams

- Standard program input/output uses:
  - `cin` Standard input stream
  - `cout` Standard output stream
  - `cerr` Unbuffered output for error messages
  - `clog` Buffered output for error messages

- Also `wchar_t` versions!
  - `wcin`, `wcout`, `wcerr`, `wclog`

- Defined in `<iostream>` header
Stream State

- All streams have state associated with them
- Streams provide flags indicating “what happened” or “what might happen”
  
  ```
  bool good() // Next operation might succeed
  bool eof() // End of input seen
  bool fail() // Next operation will fail
    // (a previous operation failed)
  bool bad() // Stream is corrupted
  ```

  ```
  iostate rdstate() // Get IO-state flags
  void clear(iostate f = goodbit) // Set flags
  void setstate(iostate f) // Add f to flags
  ```
Streams also provide test operations

```
operator void*(); // Nonzero return value if !fail()
bool operator!() const { return fail(); }
```

Can use streams in loop-conditions

```
string word;
while (cin >> word) {
    ... // Do stuff with each word.
}
```

Remember, `>>` returns `istream&`

Then `istream&` is cast to `void*` (standard C++ behavior)

- Return-value of cast depends on stream's status
- Loop terminates when there are no more words to read
Stream-State Flags

- State flags are defined in `ios_base` class
  - `ios_base::badbit`
  - `ios_base::eofbit`
  - `ios_base::failbit`
  - `ios_base::goodbit`
- Can use `rdstate()` and flags to do stuff:
  - `ios_base::iostate s = cin.rdstate();`
  - `if (s & ios_base::badbit) {
      ... // Handle input errors.
  }
- Or just use `fail()`, `bad()`, etc.
- Setting flags is a little simpler:
  - `cin.setstate(ios_base::failbit);` // state |= failbit
  - `cin.clear(ios_base::goodbit);` // state = goodbit
Stream-State and Exceptions

- Testing stream-state can be annoying
- Can configure streams to throw exceptions when state changes

```cpp
void exceptions(iostate except);
```
- Specify the states that should cause exceptions
- Example: `ios_base::badbit | ios_base::failbit`
- When stream goes into those states, `ios_base::failure` exception is thrown
- To find out what states will throw an exception:
  ```cpp
  iostate exceptions();
  ```
- Throwing exceptions on IO errors is off by default
The `>>` operator is for formatted input
- Whitespace is automatically skipped

`istream` also provides several `get()` member-functions for reading unformatted input

```cpp
int get();                // Reads one character
istream & get(char &ch); // Reads a character into ch
istream & get(char *p, int max)
istream & get(char *p, int max, char term)
istream & getline(char *p, int max)
istream & getline(char *p, int max, char term)
```

`get()`, `getline()` read characters, up to a terminator
- Default terminator is newline

`get()` does not remove terminator from stream!
- `getline()` is preferred to `get()` because of this
The `<string>` header also defines several `getline()` functions (not member functions)

```cpp
istream & getline(istream &is, string &s)
istream & getline(istream &is, string &s, char term)
```

Preferred to `iostream` member functions, so you can use a `string` instead of a `char*` buffer
Stream IO for User-Defined Types

- Implement `<<` operator for user-type output
  
  ```cpp
  ostream & operator<<(ostream & os,
                      const UserType & u);
  ```

  - Should not be a member function of anything

- Implement `>>` operator for user-type input
  
  ```cpp
  istream & operator>>(istream & is,
                      UserType & u);
  ```

  - Remember the non-const user-type reference!
Stream-Input Example: Complex

- Stream-input operator for reading complex values
- Supported formats: \( f \) or \((f)\) or \((f,f)\)
  - \( f \) is a decimal number
  - Whitespace padding can be included or excluded
- Implementation:
  ```cpp
  // Handles f or (f) or (f,f) formats
  istream & operator>>(istream &s, complex &a) {
    double re = 0, im = 0; // Components of complex number
    char ch = 0;           // Chars read from stream

    if (!s)
      return s;  // Stream already in fail state!
  ...
  ```
Stream-Input Example: Complex (2)

```
...
    s >> ch;       // Get first non-whitespace char
    if (ch == '(') {  // Value(s) surrounded in parens
        s >> re >> ch;

        if (ch == ',' ') s >> im >> ch;  // Found comma

        if (ch != ')') s.clear(ios_base::failbit);
    }
    else {          // Value not surrounded in parens
        s.putback(ch);  // "Unread" the char we just read
        s >> re;        // Try reading a number instead
    }

    if (s) a = complex(re, im); // Stream state still good

    return s;
```
File IO is almost as easy as console IO
- `#include <fstream>
- `fstream` for reading and writing to a file
- `ifstream` for reading from a file
- `ofstream` for writing to a file
- Filename, mode can be passed to constructor
- Example 1: open a word list for reading
  ```cpp
  ifstream wordList("words.txt");
  ```
- Example 2: open a result-file for appending
  ```cpp
  ofstream resultData("result.dat",
    ios_base::append);
  ```
- Also has member functions for opening and closing a file
  ```cpp
  void open(const char *p, openmode m = out);
  void close();
  bool is_open();
  ```
File IO Modes

- **ios_base** class defines file IO modes
  - **app** Open for appending
  - **ate** Open, seek to end of file ("at end")
  - **binary** Binary-mode IO (instead of text-mode)
  - **in** Open for reading
  - **out** Open for writing
  - **trunc** Truncate file to zero length

- Can bitwise-OR these values together
- **Example:**
  ```cpp
  fstream dictionary("dict.txt",
    ios_base::in | ios_base::out);
  ```
Strings as Streams

- `<sstream>` header declares string-streams
  - `string` objects are read from or written to, like a stream
- Three types (like file IO)
  - `stringstream` for read/write
  - `istringstream` for read-only
    - Useful for easily parsing data from a string
  - `ostringstream` for write-only
    - Useful for formatting output messages
    - Won’t overflow; grows as needed
- Can access/modify string-stream’s underlying data
  - Pass `string` value to `stringstream` constructor
  - `string str()` returns a copy of stream’s internal data
  - `void str(const string &)` sets stream’s contents
Automatic Document Generation

- Automating API-doc generation is a very powerful technique
  - Comment your code according to a specified style
  - Run a documentation-generator on your code
  - Produces API documentation of your code, in HTML, PDF, etc. formats, ready for distribution!
- The documentation is in one place – your source
  - Tools can use the code as well as your comments in the generated output
- Several different options for doc-generation
- We will use doxygen: http://www.doxygen.org
Doxygen Configuration

- Doxygen is driven by a config file
  - It will generate a template file for you:
    ```bash
doxygen -g [filename]
```
  - Default filename is `Doxyfile`
- Customize the config file for your project
  - Set different configuration parameters as needed
  - Parameters are well documented in the config file
- Parameter names are **ALL_CAPS**
  - (just like makefile variables)
  - Parameter-value can extend to next line, if current line ends with `\` (backslash) character
  - Switches are specified with **YES** or **NO**
Doxygen Config Tips

- You should set:
  - INPUT (input files/directories)
  - OUTPUT_DIRECTORY (where results go)
  - PROJECT_NAME

- Other good settings to use:
  - JAVADOC_AUTOBRIEF = YES
  - EXTRACT_ALL = YES
  - EXTRACT_PRIVATE = YES
  - EXTRACT_STATIC = YES
Several different formats are recognized
/**
 * This is a comment for my class. It is spiffy.
 */
class MyClass { ... };

/** starts the comment (javadoc style)
Can also start with /*! (Qt style)
Also several other options (see doxygen manual)

Classes, types, functions have a brief comment, and a detailed comment
If JAVADOC_AUTOBRIEF is defined in doxygen config, first sentence is used as brief comment.
Otherwise, must use \brief keyword in your comments
“Structural commands” specify what a comment is associated with

- “This is a comment for the source file.”
- “This is a comment for class C.”
- “This is a comment for parameter x of the function.”
- etc.

- Allows Doxygen comments to be separated from entities that are being commented. (Not always recommended...)

- Two different formats for structural commands
  
  - Doxygen format: `\cmd`
  - Javadoc format: `@cmd`
  
  - Can use either format, but be consistent!
What Can Be Commented?

- Files can be given comments
  - Must do this for doxygen to pick up certain comments
  - Examples:
    - /*! \file ... */ (Qt/Doxygen format)
    - /** @file ... */ (Javadoc format)

- Any type can be given a doxygen comment
  - Classes, structs, enums, typedefs, unions, namespaces
  - Comment should immediately precede the type
    - ...unless you are using structural commands
  - Preprocessor definitions can also be commented!
    - #define symbols, macros
Commenting Variables and Functions

- Global/static variables, and member variables
  - Comments can precede the variable:
    ```
    /** My special widget. */
    SpecialWidget sw;
    ```
  - Or they can follow the variable, on the same line:
    ```
    SpecialWidget sw; /**< My special widget. */
    ```
  - (Note the `<>` character)

- Functions and their parameters/return values
  - Parameters follow this pattern:
    ```
    @param name Description
    \param name Description
    ```
  - Return value is documented with `\return` or `@return`
Running Doxygen

- Doxygen is simple to run:
  ```
  doxygen [filename]
  ```
  - `doxygen` uses `Doxyfile` if no config file is given
  - Basically no command-line arguments; config file contains all the details!

- Results are stored in output directory
  - Each format gets its own subdirectory
  - `html` for HTML output, `latex` for LaTeX, etc.
  - Can specify alternate output directories if desired.
Doxygen References

- For more details, see the doxygen manual
  - http://www.doxygen.org