Local Variables

```cpp
string getUsername() {
    string user;

    cout << "Enter username: " << endl;
    cin >> user;

    return user;
}
```

- What happens to **user** when function returns?
  - It gets cleaned up automatically when the function returns
Returning References

```cpp
string & getUsername() {
    string user;

    cout << "Enter username: " << endl;
    cin >> user;

    return user;
}
```

- Now, what does `getUsername()` return?
  - A reference to the local variable `user`
  - Big mistake – `user` goes away, then reference becomes invalid!
- *Never* return references (or pointers) to local variables!
- In general, *be very careful* when returning references.
  - The referenced object must outlive the caller’s use of it
References and Arithmetic Operators

- From Lab 4B:
  ```cpp
  // Implement + for sparse-vectors.
  const SparseVector & SparseVector::operator+(const SparseVector & rhs) const {
    return SparseVector(*this) += rhs;
  }
  ```

- Does this work?
  - This also returns a reference to a local variable.
  - More subtle, since the variable is an unnamed temporary
  - Will eventually result in a catastrophic failure of some kind

- Simple arithmetic operators should always return a const-object value (e.g. `const SparseVector`)
Multiple Header Files

- Lab 6 has 13 classes in two header files:
  - Expression etc. - expressions.hh
  - Command etc. - commands.hh

- Managing relationships between header files can be annoying!
  - Commands depend on expressions
  - Other files depend on both of these
  - Easily leads to “multiple declaration” errors
Header Files and Multiple Inclusion

- Can’t declare something multiple times!
  - Classes, functions, variables, ...
  - Generates compiler errors.
- Solution is to use include-guards
  - Use the C++ preprocessor to make sure that each include file’s contents are only included once.
  - Uses the preprocessor directives:
    - #ifndef – if not defined …
    - #define – define some symbol …
    - #endif – end of #ifdef, #ifndef, or #if preprocessor block
The Preprocessor

- Preprocessor directives start with `#` character
- Performs simple text-level processing before the actual compilation phase begins
  - Inclusion of named files
    - `#include "expressions.hh"`
  - Macro substitution
    - `#define MAX_SPEED 65`
    - Our old friend `assert(...)` is also a macro
  - Conditional compilation
    - Can optionally include or exclude chunks of code
    - Great for writing applications that run on multiple platforms
Include-Guards

- Enclose contents of include file with a guard

  `#ifndef SOMEFILE_HH
  #define SOMEFILE_HH

  ...  // Normal header file contents

  #endif  // SOMEFILE_HH`

- Comment after `#endif` is just for readability

- First time file is included, symbol isn’t defined.
  - Symbol will be defined on subsequent inclusions, so contents won’t be repeated.

- Choose a symbol name that is likely not to be used elsewhere!
  - Some variant of header file’s name is usually safe
Results of preprocessing sometimes saved into temporary files
- Not so often anymore, but common in the past!
- If you are writing macros, may need to look at preprocessor output to debug your work
- Most compilers provide a switch for that
  - `g++ -save-temps` ...

Including lots of header files can *dramatically* increase compile times
- Only include header files that you need.
Inside the `assert()` Macro

- **C/C++** `assert()` macro uses the preprocessor
  ```c
  #ifdef NDEBUG
  #define assert(e) ((void) 0)
  #else
  #define assert(e) \
    (((void) ((e) ? 0 : __assert(#e, __FILE__, __LINE__)))
  ...
  // Definition of __assert()
  #endif
  ```

- If `NDEBUG` symbol is defined during compilation, `assert()` becomes a no-op
  - `g++ -DNDEBUG ...`
  - `-D...` defines the specified symbol when compiling
Inside the **assert()** Macro (2)

- **C/C++ assert()** macro:
  ```c
  #ifdef NDEBUG
  #define assert(e) ((void) 0)
  #else
  #define assert(e) \
  ((void) ((e) ? 0 : __assert(#e, __FILE__, __LINE__)))
  ...
  // Definition of __assert()
  #endif
  ```

- **__FILE__** and **__LINE__** are special symbols
  - Managed/updated by the preprocessor
  - **__FILE__** is file currently being compiled (.cc or .hh)
  - **__LINE__** is the current line of file being compiled
Inside the `assert()` Macro (3)

- **C/C++ assert() macro:**
  ```c
  #ifdef NDEBUG
  #define assert(e) ((void) 0)
  #else
  #define assert(e) \
    ((void) ((e) ? 0 : __assert(#e, __FILE__, __LINE__)))
  ...
  // Definition of __assert()
  #endif
  ```

- **assert() returns no value:** `((void) ... )`

- **expr ? true_val : false_val**
  - A ternary (3-arg) operator, an “inline if-statement”
  - If `expr` is true, use `true_val`; otherwise use `false_val`
Inside the **assert()** Macro (4)

- **C/C++ assert() macro:**
  ```c
  #ifdef NDEBUG
  #define assert(e) ((void) 0)
  #else
  #define assert(e) \n    ((void) ((e) ? 0 : __assert(#e, __FILE__, __LINE__)))
  ...
  #endif
  ```

  ...// Definition of __assert()  
  #endif

- Finally, the definition contains no semicolons!

- Provided by the caller:
  ```c
  assert(curr->value != 0);
  ```
Lab 7 is a brief introduction to the capabilities of the Standard Template Library (“the STL”)

- Generate a search index for a text document
  - Make a list of unique words
  - Keep a count of each word’s occurrences
  - Find out if a word appears in a set of stop-words

- The STL is a very powerful set of tools for managing and processing information
  - Definitely want to learn this library!
What Is the STL?

- A set of generic containers, algorithms, and iterators that provide many of the basic algorithms and data structures of computer science.

- Generic
  - Heavily parameterized; lots of templates

- Containers
  - Collections of other objects, with various characteristics.

- Algorithms
  - For manipulating the data stored in containers.

- Iterators
  - “A generalization of pointers.”
  - Cleanly decouple algorithms from containers.
Some STL Containers

- First category of containers: **Sequences**
  - Use indexing for access
  - Keep their values in a specific order

- **vector** – a growable array
  - Constant indexing, linear cost for insert/resize

- **deque** – double-ended queue
  - Constant prepend/append time, linear insert time

- **list** – a doubly linked list
  - Constant insert time, linear cost for indexing
  - Supports forward and backward traversal

- **slist** – a singly linked list
  - Only supports forward traversal
More STL Containers

- The other category: **Associative Containers**
  - Use *keys* for access – unique values, any type
  - Tend to keep values in a specific order, but not required to!
- **set, multiset** – ordered collection of keys
  - In *set*, keys are unique; in *multiset*, can appear multiple times
  - \( \log(n) \) time for insert, lookup
- **map, multimap** – ordered collection of *(key, value)* pairs
  - In *map*, keys are unique; in *multimap*, can appear multiple times
  - \( \log(n) \) time for insert, lookup
  - Entries are ordered by key
- **hash_set, hash_multiset** – unordered collection of keys
  - Like *set, multiset*, but with constant insert/lookup time
- **hash_map, hash_multimap** – unordered *(key, value)* pairs
  - Like *map, multimap*, but with constant insert/lookup time
STL Customization

- STL containers are templates
  - Can apply them in your programs very easily
  - Can customize them in many ways
- Parameters specify the container element types

- `vector<float> numbers;`
  - A vector (growable array) of floating-point values
  - Individual values can appear multiple times
- `set<string> wordSet;`
  - Each word appears only once in the set
  - Words are kept in alphabetical order
STL Customization (2)

- `set<string, greater<string> > wordSet;`
  - Second parameter specifies how to order the elements in the set!
  - Default value is `std::less<T>` function-template, uses `<` for comparison
  - Can reverse the order the values are kept by specifying `std::greater<T>`, which uses `>` for comparison
Example: Tracking Word-Counts

- Want to generate a search-index from a text document
- What does program need to keep track of?
  - A mapping from each word we’ve seen, to the number of times that word appears
- Can use the STL `map` template:
  ```cpp
  #include <map>
  map<string, int> wordCounts;
  ```
  - First argument is the key type – a given word
  - Second argument is the value type – the number of times that word appears
Example: Tracking Word-Counts (2)

- **STL maps** support array-indexing syntax:
  ```cpp
  wordCounts["foo"]++;  // Increments the count for the word “foo”
  cout << wordCounts["bar"] << endl;  // Outputs the current count for the word “bar”
  cout << wordCounts["bar"] << endl;  // If haven’t set `wordCounts["bar"]` before, this will evaluate to 0.
  ```
C++ Strings

- C++ inherits C notion of `char*` as a “string”
  - Zero-terminated array of `char` values
  - Useful C functions for string manipulation in `<cstring>` header (C++ name for `string.h`)
- C++ introduces the `string` class
  - Dynamically allocated, resizable string
  - Provides many features and benefits over `char*` strings
  - Generally painless to use in very complex ways
  - Prefer `string` to `char*`, wherever possible!
  - `#include <string>`
Basic C++ String Operations

- C++ string objects can be initialized from other strings, or from `char*` values
  ```cpp
  string s1 = "green";       // Same as s1("green");
  string s2 = s1;            // Same as s2(s1);
  ```
  - `s2` is an independent copy of `s1`

- `string` supports assignment from `string` or `char*`
  ```cpp
  s2 = s1;
  s1 = "gray";
  ```

- `string` supports comparison operators
  - `==` `!=` `<` `>` etc.
  - Lexicographic comparison operations
  - Case sensitive by default
String Lengths and Indexes

- **length()** member-function reports number of characters in string
  
  ```
  string color = "chartreuse";
  cout << color << " has " << color.length() << " characters." << endl;
  ```

- (string also has a **size()** member-function)

- Characters have indexes 0 to **length() - 1**
Individual Characters

- Individual characters can be accessed or mutated with []

```cpp
string word = "far";
cout << word[2] << endl;
word[1] = 'o'; // now word == "for"
```
Classifying Characters

- Useful helper functions in `<cctype>` header
  - (from the C standard header `ctype.h`)
    - `int isalpha(int)` Any letter:  a..z or A..Z in C locale
    - `int isupper(int)` Uppercase letter:  A..Z in C locale
    - `int islower(int)` Lowercase letter:  a..z in C locale
    - `int isdigit(int)` Decimal digit:  0..9
    - `int isxdigit(int)` Hexadecimal digit:  0..9, a..f or A..F
    - `int isspace(int)` Any whitespace character
  - etc.
  - Note that these functions take an `int` and return an `int`!
  - They *actually* take a character value, and return true (nonzero) / false (zero) based on the character’s category
Also in `<ctype>` header:

- `int toupper(int)` Convert letter to uppercase
- `inttolower(int)` Convert letter to lowercase

- Again, the functions are declared to take an `int` and return an `int`
- They actually take a character value, and return another character value that is the uppercase (or lowercase) version of the input
String Manipulation

- `substr()` extracts a substring
  
  ```c++
  substr(size_type start, size_type length)
  ```
  - Returns a new `string` containing the substring
  - `start` is an index in range 0 to `length()` - 1

Example:

```c++
string s1 = "computer";
string s2 = s1.substr(3, 3);
```
- `s2` is “put”
STL Algorithms

- Another nifty feature of STL: **Algorithms**
  - Standard processing operations on collections
  - `for_each`, `find`, `count`, `copy`, ...
  - `swap`, `transform`, `replace`, `reverse`, `rotate`, `random_shuffle`, ...
  - `sort`, `stable_sort`, `merge`, `set_union`, `set_intersection`, `set_difference`, `min_element`, `max_element`, ...

- All customizable in a number of ways.
Using STL

- Just like C++, STL has many subtleties and gotchas
  - Simple tasks using STL are usually fine
  - More sophisticated use can be very problematic

- Read good books on the topic:
  - Effective STL by Scott Meyers
We have covered:
- Declaring and defining classes
- C++ object-references and const correctness
- Operator overloads for user-defined types
- Dynamic memory management in C++
- Writing generic template classes
- Introduction to exception handling
- Class hierarchies, inheritance, virtual functions

Primarily focused on C++ Core Language
Advanced C++ Track

- Focuses primarily on C++ Standard Library
  - Standard classes and features implemented on top of C++ Core Language
  - Strings, streams, standard exceptions
  - Standard Template Library

- Focuses even more on how to use C++ well
  - Managing allocated memory with smart pointers
  - Using exceptions safely and effectively
  - Build tools to facilitate larger projects
  - Other neat tools, libraries, C++ language topics!