Templates & Vectors

Making your job easier

Hints & Advice

- Have you reviewed the readings from the Learning Materials page?
  - They provide more in-depth explanations

- Did you read and compile the example code posted on Canvas?
  - I often include more details in the posted code

Example: Overloading Details

- Which operators can you overload?
  - As discussed in class, most C++ operators can be overloaded
  - A few exceptions (especially **scope (::)** and **dot (.)**)
- Can you change the precedence of operators by overloading?
- What is the difference between the following:
  - Overloaded functions written as class members
  - Overloaded functions written as non-members (or as friends)
  - Reminder: https://www.learncpp.com/cpp-tutorial/94-overloading-operators-using-member-functions/

Extra Tidbit

- Are you familiar with short-circuit evaluation?
  - Consider this code:

```
if (is_active && arrow_missed())
   move_wumpus();
```

- With short circuit evaluation, the second part of the expression is evaluated only when it’s needed to determine the outcome
  - In the scenario when “is_active” is false, **arrow_missed()** will not be executed!
- C++ generally uses this strategy

Extra Tidbit (cont)

- Special note of interest:
  If you overload the **&&** or **||** operators, you lose short-circuit evaluation

Function Templates

- Useful when we have a general algorithm which doesn’t change even if types change
- Algorithm Abstraction: expressing algorithms in a very general way so that we can ignore incidental detail and concentrate on the substantive part of the algorithm
- Classic example: swap
  - We can create a template function which can take any type

```
template <typename T>
void swap(T& v1, T& v2) {
  T temp;
  temp = v1;
  v1 = v2;
  v2 = temp;
}
```

More Details

- template <typename T>
  - Referred to as the template prefix
  - Tells the compiler that the definition that follows is a template
  - T is a type parameter
- Template definition is a large collection of function definitions
- Compiler does not actually produce definitions for every single type
  - One will be produced for every type which uses the template in the program
- Compilers are not consistent in their treatment of templates
  - Needs to be defined in the same file it is invoked
  - One strategy is to put declaration and definition into a **.hpp** file
    - Example code is available on Canvas

Templated Classes

- Work the same way as templated functions
- All functions within the class will operate on the provided types
- Scope with ClassName<T>::functionname()
- Each function needs the Template prefix

Introducing vectors

- Vectors are able to contain objects/variables of any type, just like arrays
- Difference between a traditional array and a vector is related to the dynamic memory management
- In an array, **you** are responsible for managing memory
- With a vector, memory management is handled for you

http://www.cplusplus.com/reference/vector/vector/

Vector: practical application of a template class

- Arrays that can grow and shrink in length while the program is running
- Formed from template class in the Standard Template Library
- Has a base type and stores a collections of this base type
  `vector<int> v;`
- Still starts indexing at zero, can still use hard brackets to access things
- Use push_back to add one element to the end
- Number of elements == size
- How much memory currently allocated == capacity