CS11 Advanced C++

Spring 2018 – Lecture 3
Lab 3: Advanced Vector Features

• Last week, completed most of our `Vector<T>` functionality
  • Iterators, move semantics, nested typedefs

• There isn’t much left to do to our `Vector<T>`. Remaining features?

• List-initialization:
  ```cpp
  std::vector<int> month_lengths = 
  {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  ```

• Optimization for vectors of `bool`, e.g. `std::vector<bool>`
  • Why not use one bit for each `bool` value, rather than one byte/word/...?
C++ List Initialization

• C has used curly-braces for array / struct initialization for a long time
  
```
  const char *month_names[] = {
      "January", "February", "March", ..., "December",
      NULL
  };
  ```

• (Of course, C++ also supports this for array / struct initialization)

• C++ has moved to this pattern for collection / object initialization
  
```
  std::vector<int> month_lengths =
      {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  ```

• Values are provided via the `std::initializer_list<T>` class-template
  
```
  #include <initializer_list>
  ```

• To support list initialization, provide a constructor that takes a `std::initializer_list<T>` as its first (or only) argument
C++ List Initialization (2)

• `std::initializer_list<T>` is effectively a wrapper around an array
  • `size()` returns number of elements in the initializer list
  • `begin()` returns a pointer to first element in the initializer list
  • `end()` returns a pointer just past the last element in the initializer list
  • Also provides various typedefs

• Can use `initializer_list` with STL algorithms, range-based for loops, etc.
  • e.g. an easy implementation for `Vector<T>` would just `push_back()`
    each value in the sequence
Template Specialization

• Can provide specialized versions of templates
  • A separate definition of the template for a specific data type
• Example:
  • `Vector<T>` is a generic container of elements of type `T`
  • What about a special version of `Vector<T>` for storing `bool` values?
• Instead of one byte/word/etc. per `bool`, store each `bool` value in one bit
  
  ```cpp
template <> class Vector<bool> {
    ...
    // Specialization for storing vectors of bool
  };
```
• `Vector<bool>` variables will use the specialized version of the template
Template Specialization (2)

- Specialized version is a completely separate class-template 😞
  - Doesn’t inherit any implementation from generalized `Vector<T>`
  - Must reimplement entire collection from scratch
  - Specialized version may even have a different interface from the generalized version

- How can we simplify the implementation of `Vector<bool>`?

- Implement `Vector<bool>` in terms of e.g. `Vector<uint32_t>`
  - Use the generic implementation to handle memory management, vector growth, etc.
  - `Vector<bool>` can simply map accesses to the appropriate bit of the appropriate element within the `Vector<uint32_t>`
Vector<bool> Implementation

• Is `uint32_t` the best choice of element type for holding bits?
  • Maybe on 32-bit processors, but probably not on 64-bit processors
    • For optimal performance, want to choose an element type that matches the data-bus size

• Some possible choices:
  • `size_t` is often (but not always) the same size as a word
  • `uintptr_t` is the size of a pointer on the target architecture; usually is the size of a register

• To avoid being overly specific, can write `Vector<bool>` in terms of a generic type name, e.g.
  
  ```
  using block_t = uintptr_t; // These are members
  Vector<block_t> blocks;    // of Vector<bool>
  ```
  • Can change the definition of `block_t` later, if needed, without having to update the entire `Vector<bool>` template
Vector<bool> Implementation (2)

• Actually have two choices for implementing Vector<bool> in terms of Vector<block_t>

• **Option 1**: Vector<bool> contains a Vector<block_t> data-member
  • Models a “has-a” relationship, i.e. “Vector<bool> has a Vector<block_t>”
  • Composition

• This approach makes a lot of sense, and is straightforward to understand and to implement
Vector<bool> Implementation (3)

• Actually have two choices for implementing Vector<bool> in terms of Vector<block_t>

• **Option 2:** Can use private inheritance
  
  template <>
  class Vector<bool> : private Vector<block_t> {
      ...
  }

  • Does not model an “is-a” relationship!
  • Models an “is implemented in terms of” relationship – more like a “has-a” relationship, but subtly different
  • The parent class’ interface is only available to the child class; it is not made public on the child class
Vector<bool> Implementation (4)

• Actually have two choices for implementing Vector<bool> in terms of Vector<block_t>

• **Option 2:** Can use private inheritance
  ```
  template <>
  class Vector<bool> : private Vector<block_t> {
    ...
  };
  ```

• There are very few circumstances where private inheritance is necessary

• Usually can use composition to achieve the same results, and make things much easier to understand
Reading and Writing Bits

- Our `Vector<bool>` must support array-index operator `[]`
- (Iterators are **not** required – you can implement for extra credit, if you wish)

```cpp
Vector<bool> v;
...
cout << "Flag is " << v[i] << "\n";
```

- Implementation is straightforward:
  - `i / (8 * sizeof(block_t))` is the index of the block that holds the bit
  - `i % (8 * sizeof(block_t))` is the bit’s position within the block
  - (Can define a constexpr for `8 * sizeof(block_t)`, e.g. `BITS_IN_BLOCK`)
Reading and Writing Bits (2)

• Writing to bits is a bit more complicated

```cpp
Vector<bool> v;
...
v[i] = false;
```

• Individual bits aren’t addressable!

• How to support this functionality?
Reading and Writing Bits (3)

• Writing to bits is a bit more complicated

```cpp
Vector<bool> v;
...
```

```cpp
v[i] = false;
```

• For non-`const` array indexing, `Vector<bool>` can return a helper object that implements `operator=(bool value)`
  • The object handles the assignment operation on behalf of `Vector<bool>` by setting/clearing the appropriate bit
  • Must be initialized with the necessary data to perform the assignment, e.g. pass in a reference to the `Vector<block_t>`, and the desired index
Reading and Writing Bits (4)

- Writing to bits is a bit more complicated

```cpp
Vector<bool> v;
...
v[i] = false;
```

- To keep the `Vector<bool>` interface clean, any helper objects should be declared in the private section of the class-templat
And Now For Something Completely Different

• Next time we will start a new project! No more Vector<T>!