CS 162
Intro to Computer Science II

Lecture 20

Vector

Template

Standard Template Library

3/4/24

Oregon State University

Odds and Ends

- Lab 9 posted

- Assignment 4 posted

Recap: Vocabulary:

- **Polymorphism**
  - Treat an object of one class as an object of a different class
    - A call to a member function executes different code depending on the type of calling object

- **Virtual function**

```
virtual void fun();
```

  - A base-class function that is declared as virtual, indicating to the compiler that it should wait until **run-time** to determine which version of that function to run
  - A virtual function can be overridden if it is redefined in a child class

Recap: Vocabulary:

- **Dynamic binding (late binding)**
  - Used when the type of object is evaluated at **runtime**. The compiler generated code (vpointer, vtable) will check to determine the object type and then execute the correct version of code
  - This allows C++ to support polymorphism
  - We do this using the **virtual** keyword

- **Static binding (Early binding)**
  - The default behavior in C++. A function call always executes the same version of code

Recap: Vocabulary:

- **Pure virtual function** (also known as abstract function)

```
virtual void fun() = 0;
```

  - A virtual function that has no definition in the base class
  - Used when you are intending for child classes to implement the function

- **Abstract class**
  - Any class that has one or more pure virtual functions
  - An abstract class **cannot** be instantiated (i.e. you cannot create an object of an abstract class

Use of Abstract Classes

- Note: each pure virtual function needs a definition in all its derived class(es)

- All the common code in derived classes is written in abstract class
  - Same as normal inheritance, why we need abstract class?

Use of Abstract Classes

- Let's consider our demo…
- Make `make_noise()` pure virtual in `Animal` class
  - Why? Because every animal can make different noises
  - We wanted all derived class to define this function in their class to make noises
- Demo …

- `Animal` now has become abstract class
- Is there any use of `Animal` class objects?
  - No, they represent nothing.
  - So we need abstract class to prevent making objects of that class
- If you let any 3rd party to implement a `Tiger` class, making `Animal` abstract will enforce them to implement the `make_noise()` in the `Tiger` class

Vector: Example 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 (STL)
- Has a base type and stores a collections of this base type: `vector <int> v;`
- Still starts indexing at 0, can still use `[]` to access things
- Use `push_back()` to add one element to the end
- Number of elements == `size`
- How much memory currently allocated == `capacity`

More vectors

- We need to `#include <vector>` to use `std::vector`
- We use `push_back()` to add elements
- Use `pop_back()` to get rid of the last element
- `size()` – how many elements inside the vector
- `capacity()` – how many elements it can hold (allocated memory)
- We can use `operator[]` or `at()` to access specific elements
  - i.e.

    `vec[1]` or `vec.at(1)`
  - Note: `[]` does not throw an exception for an `out-of-range` that `at()` does

More vectors

- To make 2D vectors:

```
vector <vector<int> > vec_2d;
for (int i = 0; i < row; i++){
      vector<int> row_vec;
      for (int j = 0; j < col; j++)
            row_vec.push_back(i * j);
      vec_2d.push_back(row_vec);
}
```

- Note:
  - We need the extra space between angle brackets in the declaration of `vec_2d`, to tell it from the `>>` operator

More vectors

- `std::vector` has a lot more functionalities:
  - It has constructors that allow us to initialize the vector with a specified size and even a specified initial value:

```
vector <int> vec1(20); // Allocate vector of size 20
vector <int> vec2(10,7); // Fill vector with 10 7s
```

More vectors

- `std::vector` has a lot more functionalities:
  - It has constructors that allow us to initialize the vector with a specified size and even a specified initial value:

```
vector <int> vec1(20); // Allocate vector of size 20
vector <int> vec2(10,7); // Fill vector with 10 7s
```

  - `.size()` – returns the size of the vector
  - `.resize()` – changes size
  - `.empty()` – test whether the vector is empty
  - `.front()` – access the first element
  - `.back()` – access the last element
  - `.clear()` – clear content
  - `.swap()` – swap content

- More: https://cplusplus.com/reference/vector/vector/

Today's topic(s)

- Templates
- Standard Template Library (STL)
- Linked List

Templates

- How would you write a function to swap two ints?

```
void swap (int& a, int& b){
        int temp = a;
        a = b;
        b = temp;
}
```

- What if we also want to swap two floats?

```
void swap (float& a, float& b){
        float temp = a;
        a = b;
        b = temp;
}
```

- Two doubles? Two chars? Two strings? Two Animal objects?...

Function Templates

- Useful when 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 <class T>
void swap (T& a, T& b){
        T temp = a;
        a = b;
        b = temp;
}
```

Function Templates

- `template <class T>`
  - Referred to as template prefix
  - Tells the compiler that the definition that follows is a template
  - T is a type parameter
- To call this function template, we can explicitly specify our template parameter using angle brackets:
  - `swap<int>(i, j); // where i and j are ints`
  - `swap<float>(x,y); // where x and y are floats`
  - `swap<Animal>(a1, a2); //where a1 and a2 are Animals`
- Since swap() takes parameters of the template type T, we don't need to explicitly specify the template type, i.e. these also work:
  - `swap(i, j); // where i and j are ints`
  - `swap(x,y); // where x and y are floats`
  - `swap(a1, a2); //where a1 and a2 are Animals`

Function Templates

- We can write function templates that include any number of template parameters, e.g:

```
template <class T, class U>
void print_two_things(T first, U second){
    cout << first << second << endl;
}
```

And we can call it as before:

```
print_two_things<string, int>("number: ", 1);
print_two_things(2.5, 'e');
```

Note:

- The compiler generates a new implementation of the template for each type with which it is used.
  - This means concrete implementations of templates (i.e. int, float) are not created until compile time
- Therefore, we cannot explicitly compile template implementations into object files from .cpp files.
  - In fact, we can’t separate template implementations into separate .cpp files at all
  - Instead, we need to write template implementations either in the same file in which they are used or else in a header (.h) file

Template 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

Today’s topic(s)

- Template
- Standard Template Library (STL)

Standard Template Library (STL)

- C++ STL can be broken down into:
  - **Containers** – general purpose data structures (templates) for holding things
  - **Iterators** – special classes for traversing containers
  - **Algorithms** – sorting, searching, etc.

- Iterators make it possible to run the algorithm on the containers

- The STL is a great resource:
  - It contains a wide variety of very useful structures and algorithms
  - It is well-implemented, which means the structures and algorithms perform very efficiently
  - In general, it allows us to avoid re-inventing the wheel

Introducing STL Containers

- Predefined templates that can store any type of data
- The appropriate container will be dictated by the application requirements
- Example considerations:
    - Does the data need to be stored?
    - How will the data be accessed?
        - Front to back
        - Randomly?
    - Will additional data ever need to be added or removed?
- Careful planning will allow you to write clean, efficient code

Types of Containers

- Sequential containers (vector, deque, list)
  - Programmer controls the order of the elements
- Associative containers (map, set, multimap, multiset)
  - Position of elements is controlled by container
  - Elements are generally accessed by using a “key”
- Adapters (stack, queue)
  - Use an existing type of container to build other types
    - In this context, we call these “Abstract Data Types”

Examples of C++ Containers

- <array> - stores a constant amount of data in contiguous memory
- <vector> - An array that can be resized
- <list> - Linked list that stores data in non-contiguous memory
- <set> - An ordered collection of items
- <queue> - Stores data & returns it in the order it was received
  - First in, first out
- <stack> - Stores data & returns it in the opposite order that it was received
  - First in, last out

- Generally, it is a good idea to refer to the STL documentation before starting a project