Templates

Templates and the need for header only code

In one of our previous lessons, we built our own "vector", but it was specifically designed to only hold integers. But what if we wanted to store `float`, or `char`, or `bool`? This can be accomplished by using templates in C++.

Without a template

```
// file: vector.hh
class Vector {
private:
    int* m_array;   // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
    int m_size;     // Current size of the vector
public:
    Vector(int max_size);
    ~Vector();

    bool push_back(int value);
    int pop_back();

    // ...};
```

With a template

```
// file: vector.hh
template <typename your_type>
class Vector {
private:
    your_type* m_array;   // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
```

```
    int m_size;     // Current size of the vector
public:
    Vector(int max_size);
    ~Vector();

    bool push_back(your_type value);
    your_type pop_back();

    // ...};
```

Here is how you would use the code and have the C++ compiler multiply your code for different types:

```
int usage() {
  Vector<int> my_int_vector_v2(1);
  Vector<char> my_char_vector_v2(1);
  Vector<float> my_float_vector_v2(1);}
```

Sample 1

When you design your header file, especially with a template, your code can be input right within the class itself.

```
// file: vector.hh
template <typename your_type>
class Vector {
private:
    your_type* m_array;   // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
    int m_size;     // Current size of the vector
public:
    // code for function can be right here
    Vector(int max_size) {
      m_array = new int[max_size];
      m_max_size = max_size;
      m_size = 0;
    }
```

```
    void push_back(your_type value) {
      if (m_size < m_max_size) {
        m_array[m_size] = value;
        m_size++;
      }
    }

    // ...};
```

Sample 2

```
// file: vector.hh
template <typename your_type>
class Vector {
private:
    your_type* m_array;   // Pointer to dynamically allocated array
    int m_max_size; // Max size of the vector
    int m_size;     // Current size of the vector
public:
    // we can declare functions but "define" them below
    // this improves code readability
    Vector(int max_size);

    void push_back(your_type value);

    // ...
};
// We must use "scope" operator to define which class the function belongs to
Vector::Vector(int max_size) {
  m_array = new int[max_size];
  m_max_size = max_size;
  m_size = 0;
}
void Vector::push_back(your_type value) {
  if (m_size < m_max_size) {
    m_array[m_size] = value;
```

```
    m_size++;
  }}
```

Source Code Organization

Template code in header file only

The way templates work is that for each type, such as `vector<int>` or `vector<bool>`, the entire header file is copied and pasted by the compiler for the new type. Because of this reason, classes that use templates must be in header file only. This means that you cannot have `vector.hh` and also `vector.cc` because the entire code has to exist in the header file only.

Standard code

Standard code that doesn't use templates can be in `file.hh` and also `file.cc` and doesn't need to be in a header. Although not that many libraries sometimes tend to be header only and the code split to `file.cc` is for cosmetic reasons only.

```
// Header file:
// File: adder.hh
#pragma once
// Example of header file and source file
// Class should only declare functions, but not define them
class adder {
  public:
    int x;
    int y;
    // Only declare functions, do not "define" functions
    int get_sum();};
```

And then there should be a separate *.cc file or *.cpp file:

```
// File: adder.cc
#include "adder.hh"
int adder::get_sum() {
    return x + y;
}
```

Revision #5
Created 7 months ago by Preet Kang
Updated 6 months ago by Preet Kang