CS162 Lecture 17

Templates
A good resource on exception handling

https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm

- Clear explanation of try/throw/catch process
- great graphic of std::exceptions
Building a calculator

```c
int addInts (int a, int b) {
    return a+b;
}

int main () {
    int x=7, y=43, z;
    z = addInt(x, y);
    cout << z << endl;
}
```

- Works great for integers
- But we’re not building an integer adder
- We’re building a calculator
- What about floats, doubles, etc?
- Do we wanna implement an addFloat, addDouble, etc?
- No.
- No we don’t.
- That’s gonna take forever.

What if we could build a function that could handle multiple types?
Function Templates

- We have a general algorithm that doesn’t change even if the input type changes
- Algorithm abstraction: expressing algorithms in a very general way so that we can ignore incidental detail and concentrate on the substantive part
- Make a template function which can take any type

```cpp
template <class T>
void swap(T& x, T& y)
{
    T temp;
    temp = x;
    x = y;
    y = temp;
}
```
Building a calculator

template <class C>
C add (C a, C b) {
    return a+b;
}

int main () {
    int x=7, y=43, z;
    z = add(x, y);
    cout << z << endl;

    double x = 7.5, y = 42.5, z;
    Z = add(x, y);
    cout << z << endl;
}

- C is a generic type of data
- Our template function takes two ‘C’s, adds them, and returns a C!
- When we call it in main(), C is cast to hold whatever data type we pass to it!
- Works for ints and doubles!
- Still can’t add an int to a double, the types of x & y must match.
- Both are dynamically-cast type C
Details

Template prefix tells the compiler that the definition that follows is a template:

```cpp
template <class T>
```

- 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 (.hpp)
**.hpp files**

- C++ **only**.
- **Not compatible with C**
- Otherwise no difference from .h files in this class
- Specifically a C++ Header file
- Can be inserted into a .CPP source code file using the #include directive

```c++
#ifndef MY_HEADER_HPP
#define MY_HEADER_HPP
...
#endif
```
```cpp
#include <iostream>
#include <string>

using namespace std;

template <class T>
void swapVals(T& a, T& b) {
    T temp;
    temp = a;
    a = b;
    b = temp;
}

template <class T>
void print(const T& a, const T& b) {
    cout << "a: " << a << " b: " << b << endl;
}

template <class T>
void print_and_swap(T a, T b) {
    print(a,b);
    swapVals(a,b);
    print(a,b);
}
```
int main() {

    /*int a = 5;
    int b = 6;
    print(a, b);
    swapVals(a, b);
    print(a, b);
    float c = 7.1, d = 8.2;
    print(c,d);
    swapVals(c,d);
    print(c,d);
    string first = "world";
    string second = "hello";
    print(first, second);
    swapVals(first, second);
    print(first, second);*/
    print_and_swap(5, 6);
    print_and_swap(7.1, 8.2);
    print_and_swap("world", "hello");
    return 0;
}
Multiple-Parameter function templates

template <class First, class Second>
First smaller(First a, Second b) {
    return (a<b?a:b);
}

int main () {
    int x = 89;
    double y = 56.78
    cout << smaller(x,y) << endl;
}

- Output: 56
- Anywhere function sees ‘First’, typecasts it as an int
- ‘Second’: double
- a<b?a:b means:
  - if a<b return a
  - Else return b
- So it took x as an int, y as a double, and then returned y as an int!
- This is why the .76 was dropped
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 (template <class T>)
Class Templates (‘template’ == ‘generic data type’)

template <class T>
class Example{
    T first, second;
    public:
        Example(T a, T b) {
            first=a;
            second=b;
        }
        T bigger();
};
template <class T> // need this before every function!
T Example<T>::bigger() {
    return (first>second?first:second);
}

int main () {
    Example <int> ex(3, 7);
    cout << ex.bigger();
}

- Need to pass in what type we are substituting for T when we call the templated function from Example class!
- Output: 7
STL class example: Vector

- Arrays that can grow and shrink in length while 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, still uses [ ] to index
- Use push_back to add an element to the end
- Member variable size tracks number of elements stored
- Member variable capacity tracks currently allocated memory
  - Typically doubles when size >= capacity
  - So capacity always >= size
A Vector of Ants

“../Ants/thrower.h”

vector <Insect*> v

for each insect

i = new Ant

push_back(i)

First iteration:
Size: 1
Capacity: 1

Second:
Size: 2
Capacity: 2

Third:
Size: 3
Capacity: 4
#include <iostream>
#include <vector>

using namespace std;

int main() {

    vector<int> v;
    cout << "PUSH BACK" << endl;
    for(int i=0; i<10; i++) {
        v.push_back(i);
        cout << "Size: " << v.size() << endl;
        cout << "Capacity: " << v.capacity() << endl;
    }
    cout << endl;
    cout << "POP BACK" << endl;
    for(int i=0; i<10; i++) {
        v.pop_back();
        cout << "Size: " << v.size() << endl;
        cout << "Capacity: " << v.capacity() << endl;
    }

    return 0;
}