



# CS 11 C++ track: lecture 7

---

- Today:
  - Templates!



# Templates: motivation (1)

---

- Lots of code is generic over some type
- Container data types:
  - List of integers, doubles, strings
  - Sparse matrix of ints, doubles, complex
- How do we handle this?



## Templates: motivation (2)

---

- Could write multiple similar classes:
  - IntList, FloatList, DoubleList
- But would be almost identical
  - except for types
- Adding new method means modifying many files
- There has to be a better way...



# Templates: motivation (3)

---

- Instead can write **template classes**:
  - `List<int>`, `List<double>`, `List<string>`
  - `SparseVector<int>`,  
`SparseVector<double>`
- Re-use same code for different types



# Templates: example (1)

---

```
template <typename T>
class Array {
private:
    int len_;
    T* data_;
    int check(int i) const {
        if(i < 0 || i >= len_) {
            throw std::string("out of range!");
            return i;
        }
    }
}
// continued on next slide...
```



# Templates: example (2)

---

public:

```
Array(int len=0) : len_(len), data_(new T[len]) {}  
~Array() { delete [] data_; }  
int len() const { return len_; }  
T& operator[](int i) const { return data_[check(i)]; }  
Array(const Array&);  
Array& operator=(const Array&);  
};
```



# Templates: example (3)

---

```
// still in Array.hh:  
// Copy constructor:  
template <typename T> inline  
Array<T>::Array(const Array<T>& other)  
    : len_(other.len_), data_(new T[other.len_]) {  
    for (int i = 0; i < len_; i++) {  
        data_[i] = other.data_[i];  
    }  
}  
// operator= left as exercise...
```



# Templates: example (4)

---

```
// testArray.cc:
#include <iostream>
#include <string>
#include "Array.hh"
int main() {
    Array<int> a(10);
    try {
        a[3] = 3;
        a[10] = 5; // oops!
    }
    catch (std::string s) {
        std::cout << s << std::endl;
    }
    return 0;
}
```



# Templates: issues

---

- *All* the Array code is in the header file!
- Template is like a big macro that gets expanded to a real type when given a type argument
- In theory, can define template member functions in a separate class and compile separately; in practice, few compilers support this (**g++** doesn't)



# Templates: wrapup

---

- Much, much more to templates!
  - Whole books have been written on templates
  - Can write templated functions
  - Can use integer template parameters
  - Default template parameters
  - Multiple template parameters
  - STL (Standard Template Library)
- See the advanced C++ track for more