Reminders/Due Dates

- Summer 2023 Internship Opportunity posted on Canvas.

- Lab 6 makeup due tonight.

- Lab 9 due Wednesday 11/30.

- Lab 10 and Homework 4 will be due next Monday December 5 by 11:59pm.

- Makeup submissions for Homework 3, Lab 7, and Lab 8 will open on Wednesday and will be due Friday December 9.

- Wednesday 11/30 we will have a final exam review.

- Our final is next Wednesday December 7, 2022 from 5:30pm-7:30pm. We will be meeting in JB-358.

Please take 10 minutes to complete the SOTE for this course.

- Responses are completely anonymous and are not released until AFTER grades are posted.
- Feedback really helps us know what is working or what needs improvement in the course.
- You can access on Canvas > Any Course, SOTE (in bottom left menu).

QR Code:

- Landing Page URL:
  https://my.csusb.edu/default/classclimate_survey/index

Chapter 16: Templates part 2

CSE 2010
Week 15

Templates Review

Templates in C++:
A tool that allows a single function or class to work with a variety of data types.

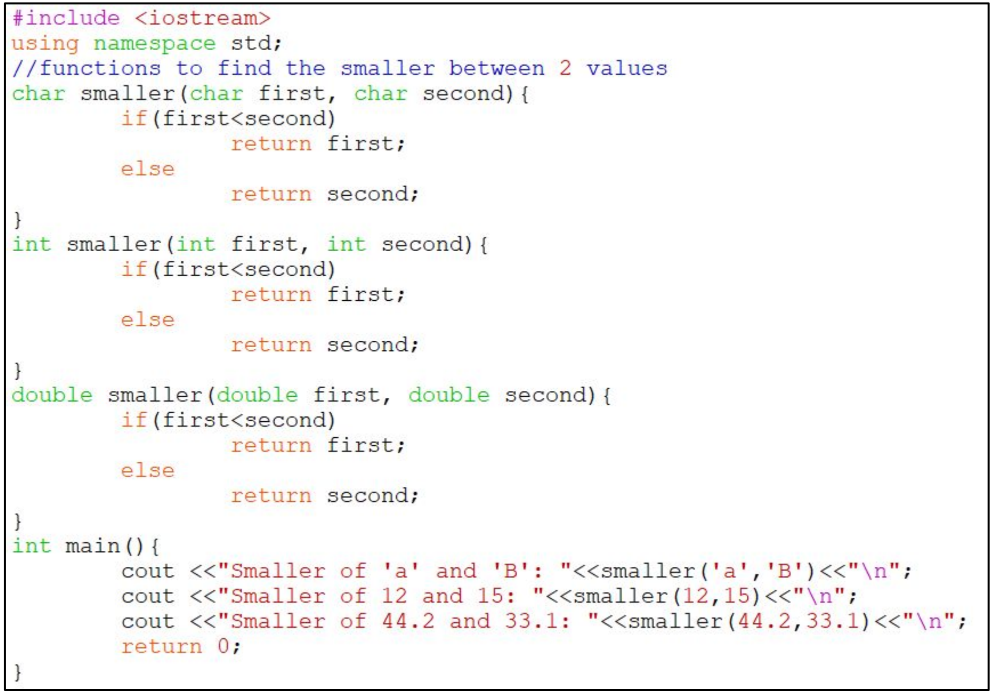
- A template allows a function or a class definition to be *parameterized* by type, instead of values.
- For this chapter we will learn about template functions & template classes.

Syntax of a template function :

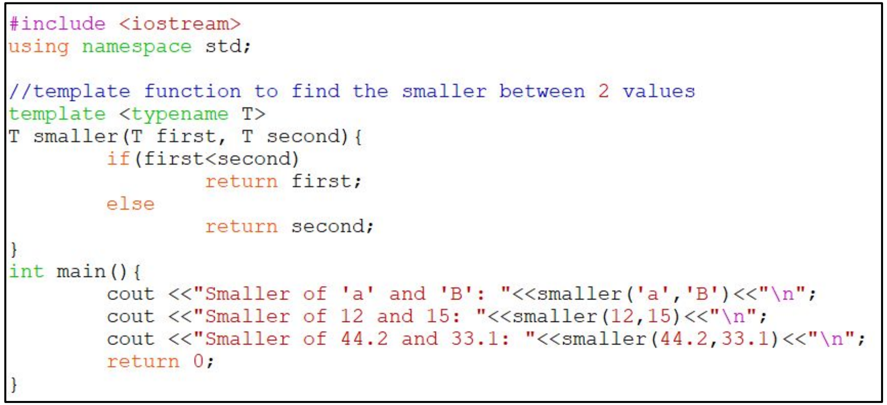
```
template<typename type_var1, …,typename type_varn>

return_type function_name(parameters)

{
    //statements
}
```

- **typename** type_var$_1$, …, **typename** type_var$_n$ is used to list the number of generic types your function will need. Use a different letter for each different type.
- The return type can be a generic type or regular data type.

Example: Finding the smaller of two values (without/with templates)

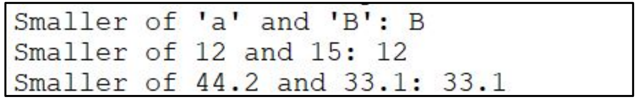


```
#include <iostream>
using namespace std;
//functions to find the smaller between 2 values
char smaller(char first, char second){
        if(first<second)
                return first;
        else
                return second;
}
int smaller(int first, int second){
        if(first<second)
                return first;
        else
                return second;
}
double smaller(double first, double second){
        if(first<second)
                return first;
        else
                return second;
}
int main(){
        cout <<"Smaller of 'a' and 'B': "<<smaller('a','B')<<"\n";
        cout <<"Smaller of 12 and 15: "<<smaller(12,15)<<"\n";
        cout <<"Smaller of 44.2 and 33.1: "<<smaller(44.2,33.1)<<"\n";
        return 0;
}
```



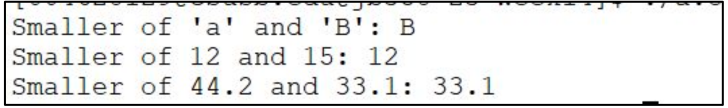
```
#include <iostream>
using namespace std;

//template function to find the smaller between 2 values
template <typename T>
T smaller(T first, T second){
        if(first<second)
                return first;
        else
                return second;
}
int main(){
        cout <<"Smaller of 'a' and 'B': "<<smaller('a','B')<<"\n";
        cout <<"Smaller of 12 and 15: "<<smaller(12,15)<<"\n";
        cout <<"Smaller of 44.2 and 33.1: "<<smaller(44.2,33.1)<<"\n";
        return 0;
}
```

Output:



```
Smaller of 'a' and 'B': B
Smaller of 12 and 15: 12
Smaller of 44.2 and 33.1: 33.1
```

Output:



```
Smaller of 'a' and 'B': B
Smaller of 12 and 15: 12
Smaller of 44.2 and 33.1: 33.1
```

Class Templates

We have learned in previous chapters that a class is a combination of data members and member functions.

Now consider that we need a class with the same data members and overall functionality, but with different data types.

We can accomplish this with a class template!

Syntax:

```
template <typename T>//you can have multiple generic types
class className
{
      private:
            T data;

      public:
            className(T init); //overload constructor
            T get() const;//accessor
            void set(T d);//mutator

};
```

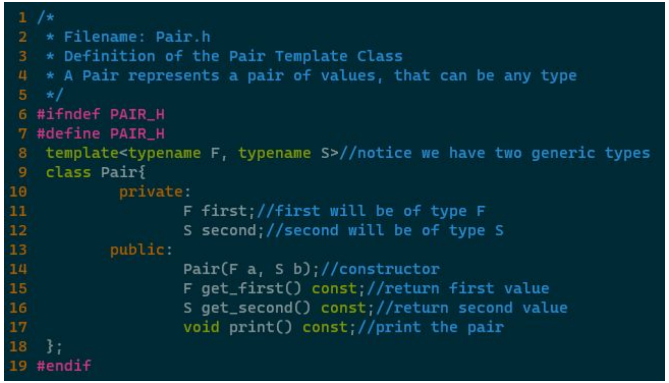
Syntax for function definitions:

```
template <typename T>
className<T>::className(T init):data(init)
{
}

template <typename T>
T className <T>::get()const
{
      return data;
}

template <typename T>
void className <T>::set(T d)
      data = d;
}
```

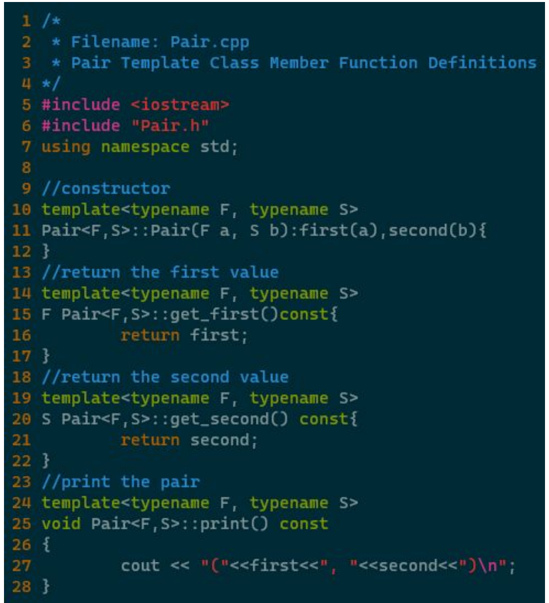
Compilation of Class Templates

- Templates are not like ordinary classes in the sense that the compiler doesn't generate object code for a template or any of its members until the template is instantiated with concrete types.
- Acceptable methods of compiling class templates vary depending on the C++ compiler you use, but the following method should work across all versions.
- **The inclusion method**
  - Define your template class in a .h file
  - Define your template class functions in a .cpp file
  - Include the .cpp file in whatever program file you are using the class in.
  - When you go to compile, you only need to compile the program file, not the class template.

Pair Class

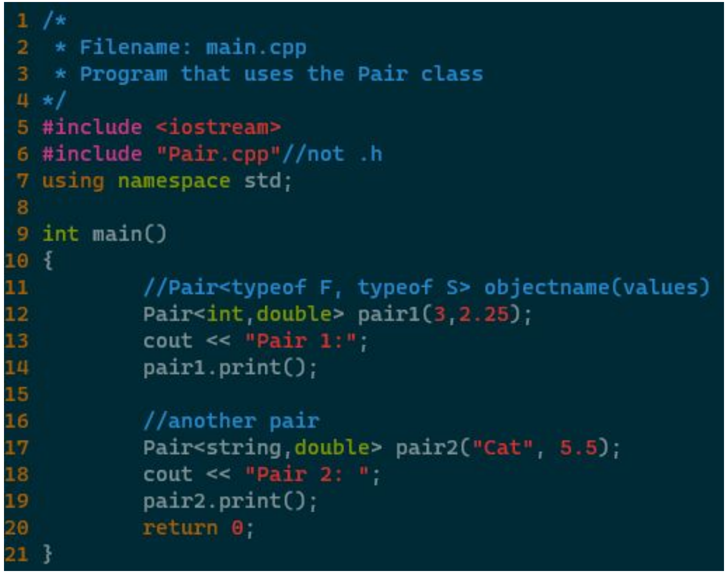


```cpp
/*
 * Filename: Pair.h
 * Definition of the Pair Template Class
 * A Pair represents a pair of values, that can be any type
 */
#ifndef PAIR_H
#define PAIR_H
 template<typename F, typename S>//notice we have two generic types
 class Pair{
         private:
                 F first;//first will be of type F
                 S second;//second will be of type S
        public:
                 Pair(F a, S b);//constructor
                 F get_first() const;//return first value
                 S get_second() const;//return second value
                 void print() const;//print the pair
 };
#endif
```



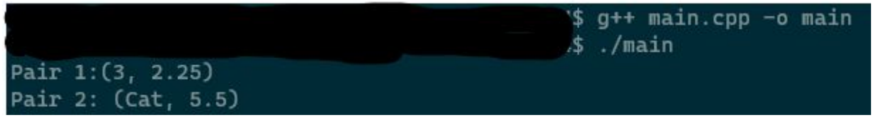
```cpp
/*
 * Filename: Pair.cpp
 * Pair Template Class Member Function Definitions
*/
#include <iostream>
#include "Pair.h"
using namespace std;

//constructor
template<typename F, typename S>
Pair<F,S>::Pair(F a, S b):first(a),second(b){
}
//return the first value
template<typename F, typename S>
F Pair<F,S>::get_first()const{
        return first;
}
//return the second value
template<typename F, typename S>
S Pair<F,S>::get_second() const{
        return second;
}
//print the pair
template<typename F, typename S>
void Pair<F,S>::print() const
{
        cout << "("<<first<<", "<<second<<")\n";
}
```

Program that uses the Pair class template



```cpp
/*
 * Filename: main.cpp
 * Program that uses the Pair class
*/
#include <iostream>
#include "Pair.cpp"//not .h
using namespace std;

int main()
{
        //Pair<typeof F, typeof S> objectname(values)
        Pair<int,double> pair1(3,2.25);
        cout << "Pair 1:";
        pair1.print();

        //another pair
        Pair<string,double> pair2("Cat", 5.5);
        cout << "Pair 2: ";
        pair2.print();
        return 0;
}
```

Compilation and output



```
[redacted]$ g++ main.cpp -o main
$ ./main
Pair 1:(3, 2.25)
Pair 2: (Cat, 5.5)
```

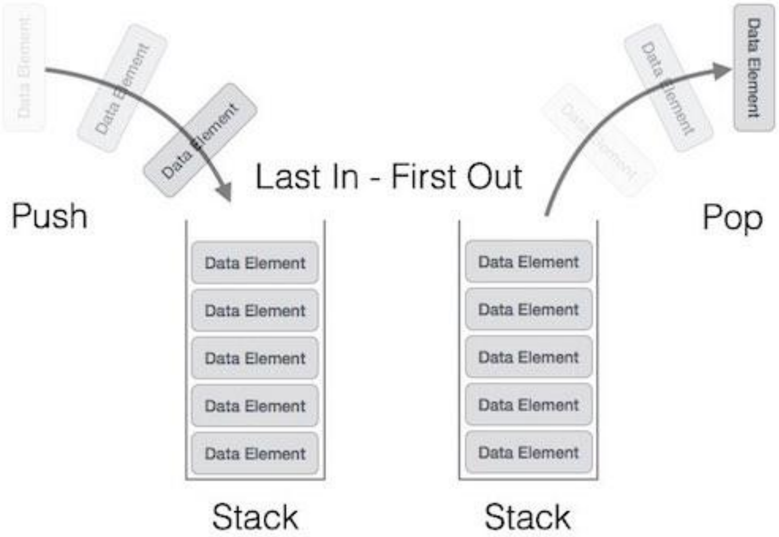
Data Structures

- Data structures in programming are methods in which to store data in efficient and easy to access ways.
- Depending on how you need to store and access data, you have different data structures to choose from.
- A popular data structure that we have been using are vectors, which is part of the C++ Standard Template Library (STL).
- The C++ STL is a powerful set of **template classes** to provide general-purpose classes and functions with templates that implement commonly used data structures and algorithms.
- Let’s try out a simple approach at implementing one of these data structures.

Stacks (Data Structure)

Stacks: A data structure in which the last item pushed into the stack is the first item that will be popped from the stack.

- Last in, first out (LIFO)



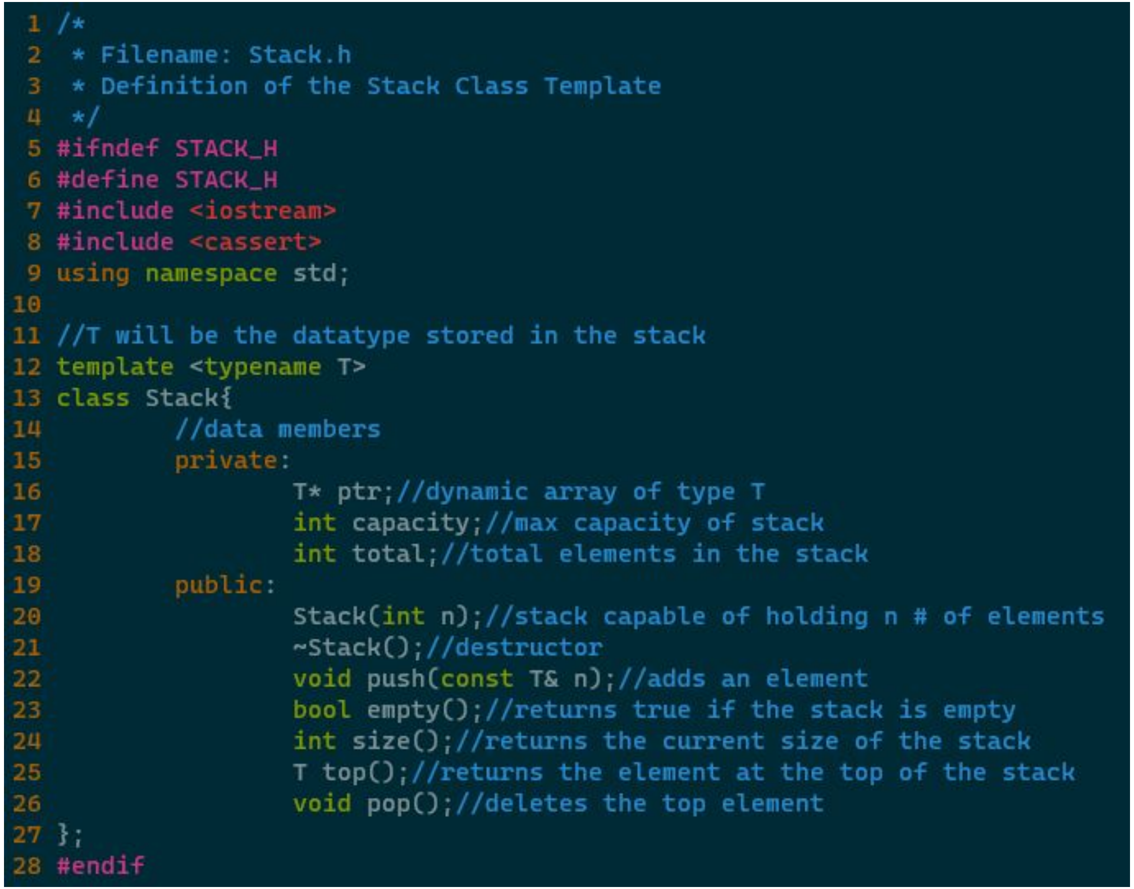
Typical operations for stacks:

- Push (add an item to top)
- Pop (delete item from top)
- Peek/Top (See item at top without deleting)

Class Template for a Stack (Stack.h)

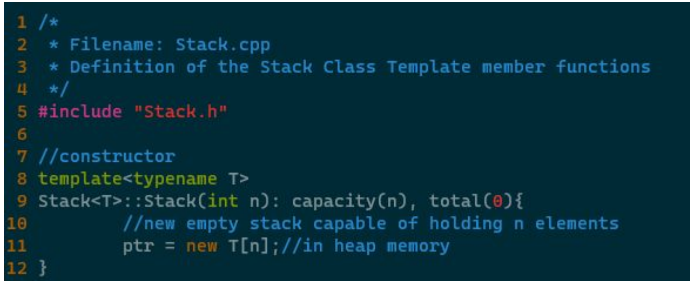
- We will only need one generic type here, since a data structure should only hold one type.
- But by using templates it can hold any type!
- To simulate a stack, we use an array of type T, but as far as accessing/manipulating it, we limit the member functions to `pop()`, `push()`, and `top()`.
- `empty()` and `size()` are just extra accessor functions.

Let’s implement the member functions!

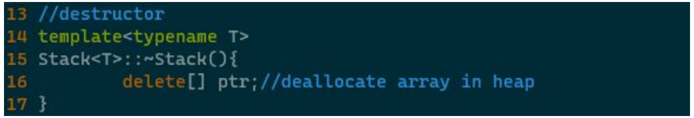


```cpp
/*
 * Filename: Stack.h
 * Definition of the Stack Class Template
 */
#ifndef STACK_H
#define STACK_H
#include <iostream>
#include <cassert>
using namespace std;

//T will be the datatype stored in the stack
template <typename T>
class Stack{
        //data members
        private:
                T* ptr;//dynamic array of type T
                int capacity;//max capacity of stack
                int total;//total elements in the stack
        public:
                Stack(int n);//stack capable of holding n # of elements
                ~Stack();//destructor
                void push(const T& n);//adds an element
                bool empty();//returns true if the stack is empty
                int size();//returns the current size of the stack
                T top();//returns the element at the top of the stack
                void pop();//deletes the top element
};
#endif
```

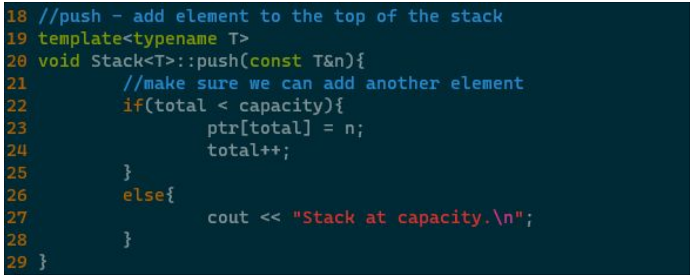
Class Template for a stack - Member Functions (Stack.cpp)



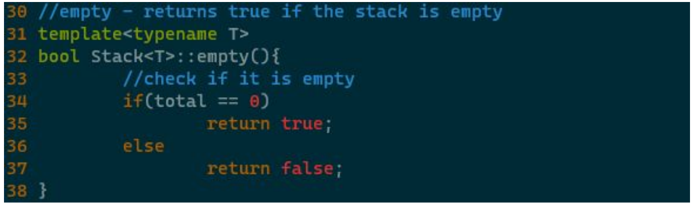
```cpp
/*
 * Filename: Stack.cpp
 * Definition of the Stack Class Template member functions
 */
#include "Stack.h"

//constructor
template<typename T>
Stack<T>::Stack(int n): capacity(n), total(0){
        //new empty stack capable of holding n elements
        ptr = new T[n];//in heap memory
}
```



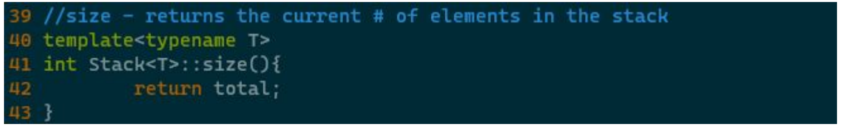
```cpp
//destructor
template<typename T>
Stack<T>::~Stack(){
        delete[] ptr;//deallocate array in heap
}
```



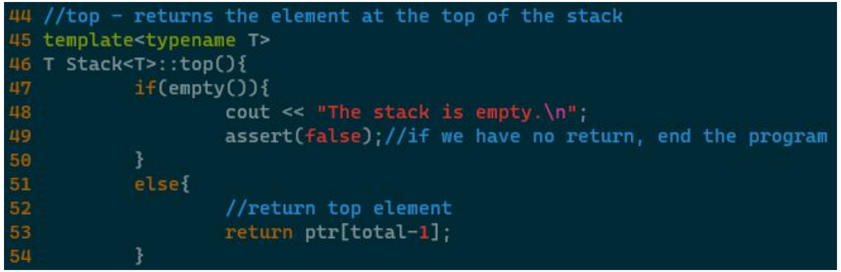
```cpp
//push - add element to the top of the stack
template<typename T>
void Stack<T>::push(const T&n){
        //make sure we can add another element
        if(total < capacity){
                ptr[total] = n;
                total++;
        }
        else{
                cout << "Stack at capacity.\n";
        }
}
```



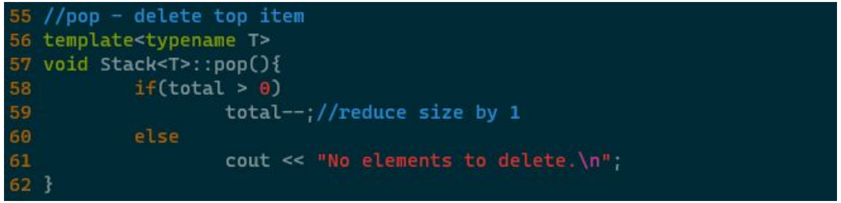
```cpp
//empty - returns true if the stack is empty
template<typename T>
bool Stack<T>::empty(){
        //check if it is empty
        if(total == 0)
                return true;
        else
                return false;
}
```



```cpp
//size - returns the current # of elements in the stack
template<typename T>
int Stack<T>::size(){
        return total;
}
```

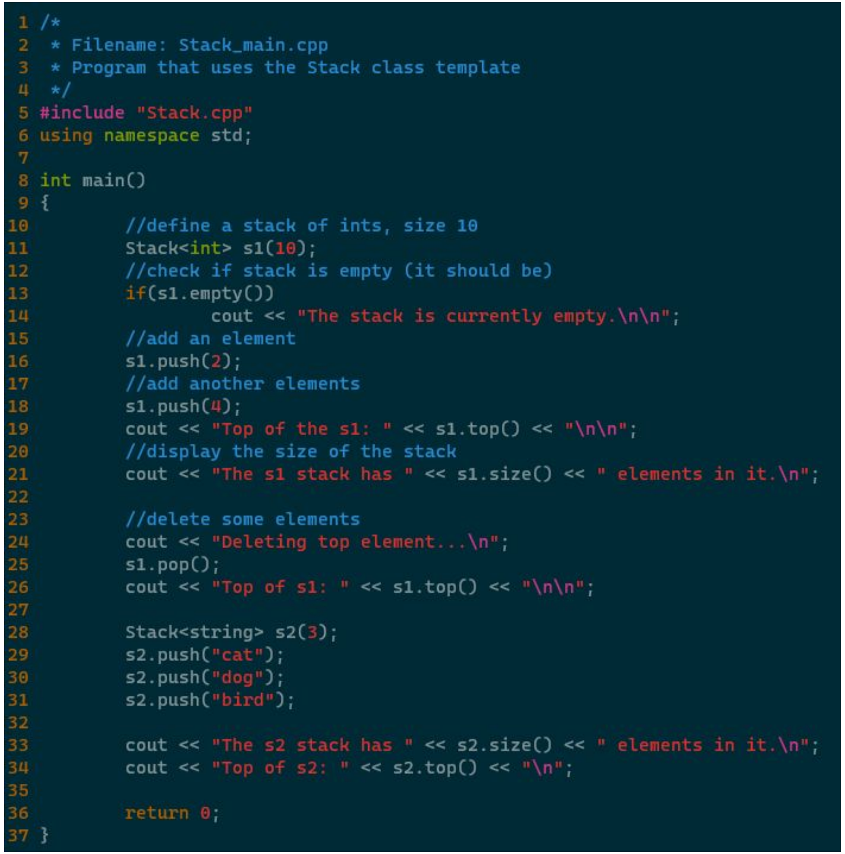


```cpp
//top - returns the element at the top of the stack
template<typename T>
T Stack<T>::top(){
        if(empty()){
                cout << "The stack is empty.\n";
                assert(false);//if we have no return, end the program
        }
        else{
                //return top element
                return ptr[total-1];
        }
```



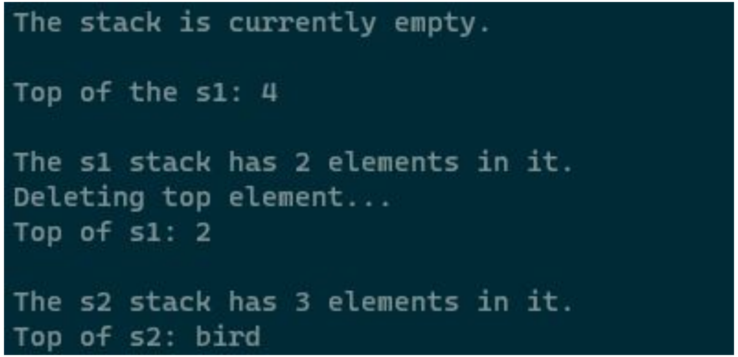
```cpp
//pop - delete top item
template<typename T>
void Stack<T>::pop(){
        if(total > 0)
                total--;//reduce size by 1
        else
                cout << "No elements to delete.\n";
}
```

Stack_main.cpp



```cpp
/*
 * Filename: Stack_main.cpp
 * Program that uses the Stack class template
 */
#include "Stack.cpp"
using namespace std;

int main()
{
        //define a stack of ints, size 10
        Stack<int> s1(10);
        //check if stack is empty (it should be)
        if(s1.empty())
                cout << "The stack is currently empty.\n\n";
        //add an element
        s1.push(2);
        //add another elements
        s1.push(4);
        cout << "Top of the s1: " << s1.top() << "\n\n";
        //display the size of the stack
        cout << "The s1 stack has " << s1.size() << " elements in it.\n";

        //delete some elements
        cout << "Deleting top element...\n";
        s1.pop();
        cout << "Top of s1: " << s1.top() << "\n\n";

        Stack<string> s2(3);
        s2.push("cat");
        s2.push("dog");
        s2.push("bird");

        cout << "The s2 stack has " << s2.size() << " elements in it.\n";
        cout << "Top of s2: " << s2.top() << "\n";

        return 0;
}
```

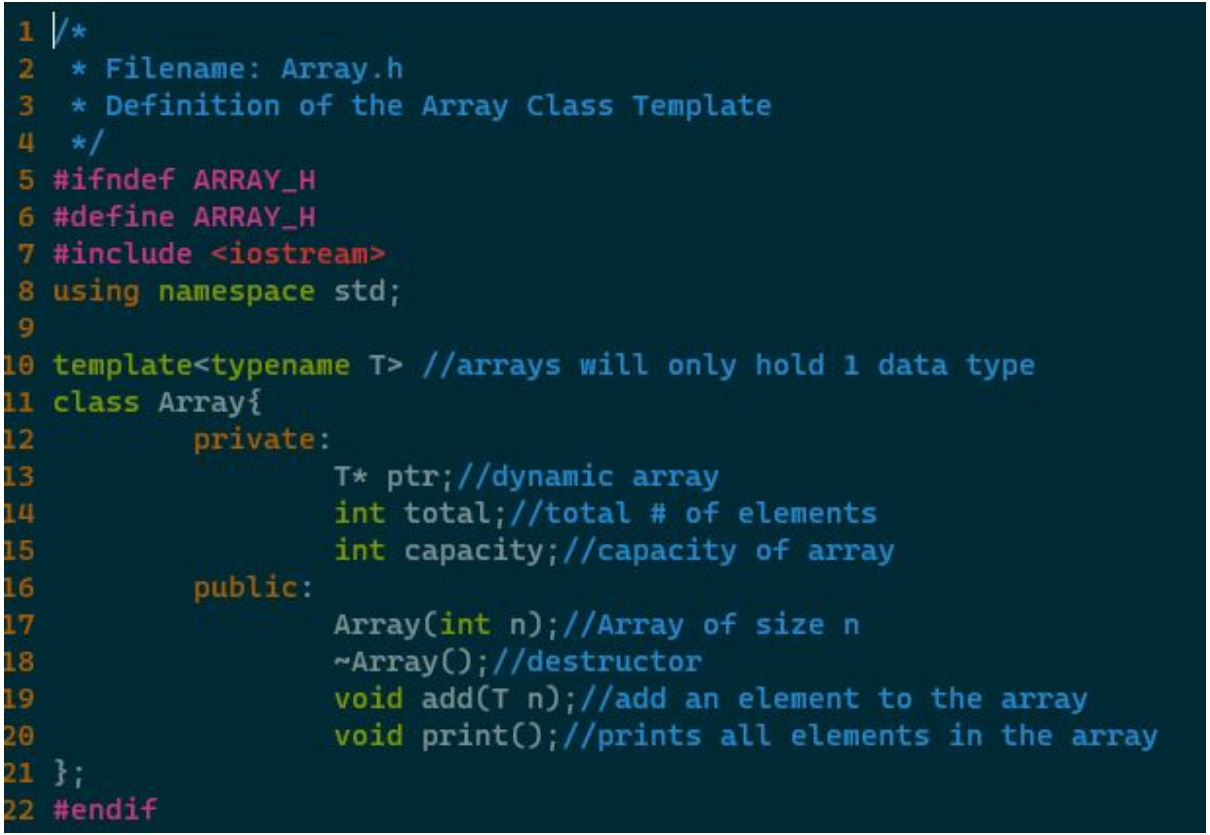
Output:



```
The stack is currently empty.

Top of the s1: 4

The s1 stack has 2 elements in it.
Deleting top element...
Top of s1: 2

The s2 stack has 3 elements in it.
Top of s2: bird
```

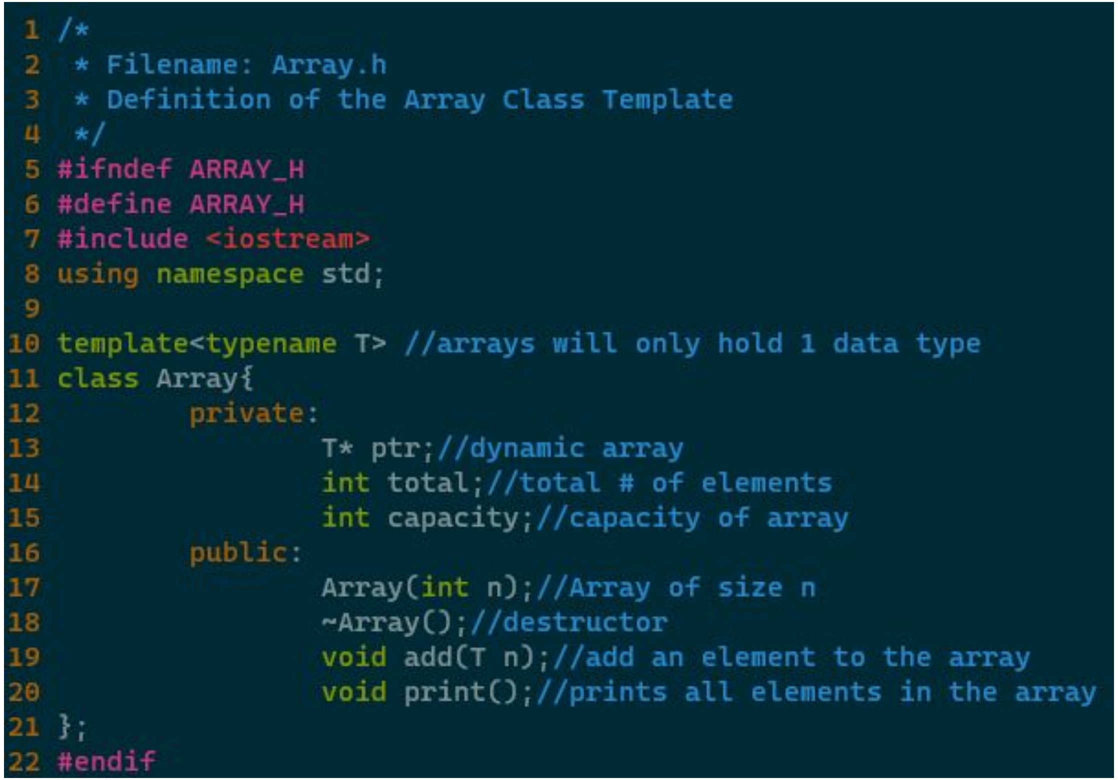
Class Templates for an Array (Array.h)

- Create a template class Array that can handle an array of objects of any type and any size in the heap.
- Define an add member function to add elements to the end of the array.
- Define a print function to print all elements in the array.



```
/*
 * Filename: Array.h
 * Definition of the Array Class Template
 */
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>
using namespace std;

template<typename T> //arrays will only hold 1 data type
class Array{
        private:
                T* ptr;//dynamic array
                int total;//total # of elements
                int capacity;//capacity of array
        public:
                Array(int n);//Array of size n
                ~Array();//destructor
                void add(T n);//add an element to the array
                void print();//prints all elements in the array
};
#endif
```

Array.cpp



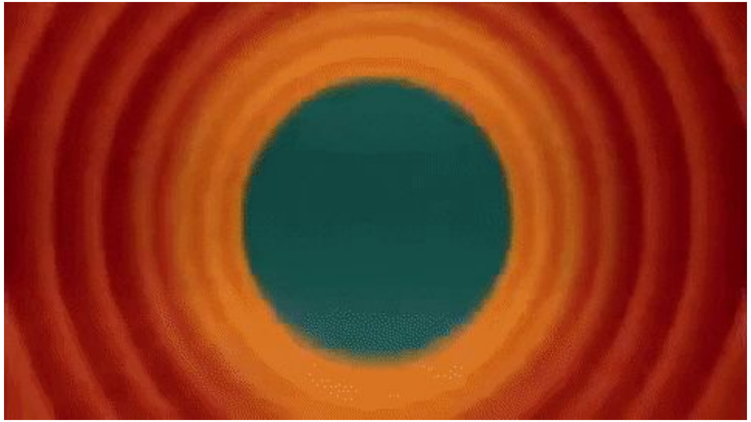
```
/*
 * Filename: Array.h
 * Definition of the Array Class Template
 */
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>
using namespace std;

template<typename T> //arrays will only hold 1 data type
class Array{
        private:
                T* ptr;//dynamic array
                int total;//total # of elements
                int capacity;//capacity of array
        public:
                Array(int n);//Array of size n
                ~Array();//destructor
                void add(T n);//add an element to the array
                void print();//prints all elements in the array
};
#endif
```

Let’s write a program that uses this class!

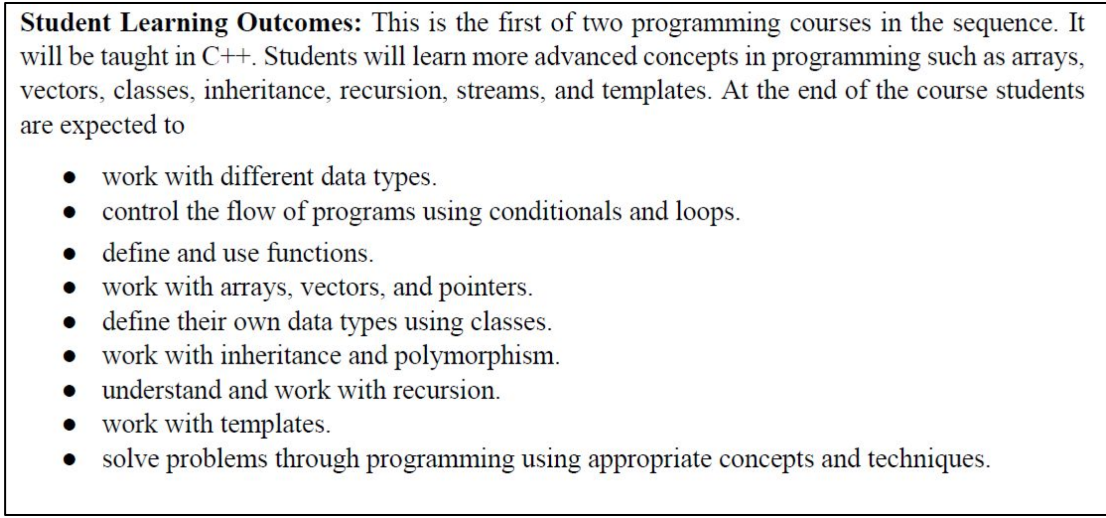
Templates Review

- Templates in C++: A tool that allows a single function or class to work with a variety of data types.
    - This saves us from having to overload our functions with different datatypes.
- A template allows a function or a class definition to be *parameterized* by type, instead of values.
- We can overload template functions.
- Class templates allow us to design generic classes that accept different data types and objects.

And with that…we have covered everything you need to know for CSE 2010!



Let's look at the Student Learning Outcomes we looked at in Week 1 in the syllabus



Student Learning Outcomes: This is the first of two programming courses in the sequence. It will be taught in C++. Students will learn more advanced concepts in programming such as arrays, vectors, classes, inheritance, recursion, streams, and templates. At the end of the course students are expected to

- work with different data types.
- control the flow of programs using conditionals and loops.
- define and use functions.
- work with arrays, vectors, and pointers.
- define their own data types using classes.
- work with inheritance and polymorphism.
- understand and work with recursion.
- work with templates.
- solve problems through programming using appropriate concepts and techniques.