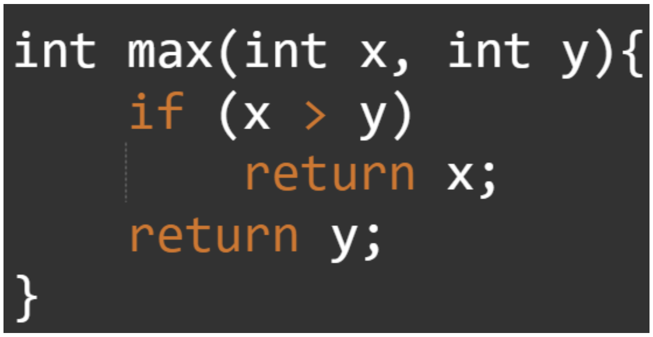
Templates

Template Basics

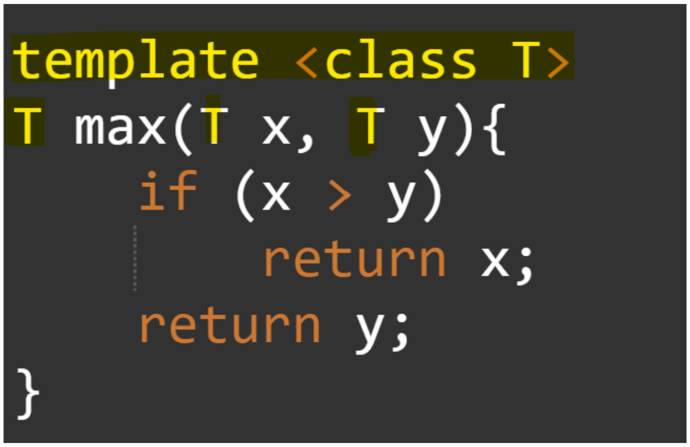
- Templates are used for generic programming
- The generalization is based on the data type so you can write code that works for any type of data



```
int max(int x, int y){
    if (x > y)
        return x;
    return y;
}
```

→ to make this work for any type of data, instead of int, use Template T

→ the same function can work finding max of 2 ints, doubles, floats, and strings when converted into a templated function



```
template <class T>
T max(T x, T y){
    if (x > y)
        return x;
    return y;
}
```

- This function can find the max of 2 ints, doubles, chars
- —> this template function can find the max of any primitive data type

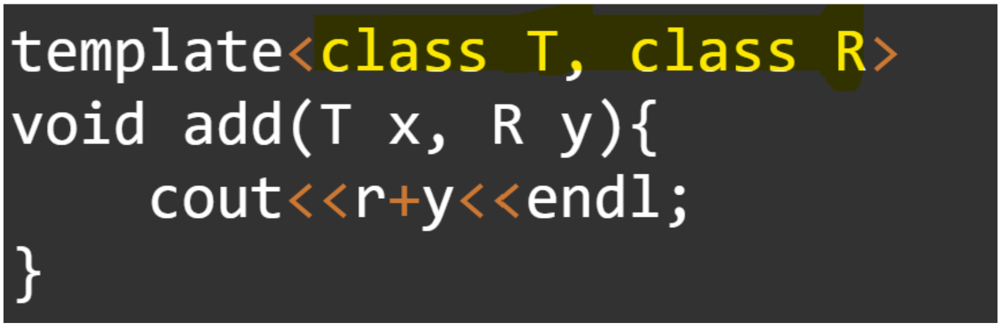
    - **You can pass any primitive data types OR any user-defined classes or structures**

    - **⇒ you can pass in any data type, as long as it can be compared**

    - **If you want your own classes to be passed, and you want to find the max, then in your class, you must overload the < and > operator**

    - ****** YOU SHOULD PROVIDE THE FUNCTIONALITY TO KNOW WHICH OBJECT IS GREATER, THEN THE TEMPLATED FUNCTION WILL WORK FOR YOUR CLASSES ALSO!**

- **We can have multiple parameters in template declaration when required**



```
template<class T, class R>
void add(T x, R y){
    cout<<r+y<<endl;
}
```

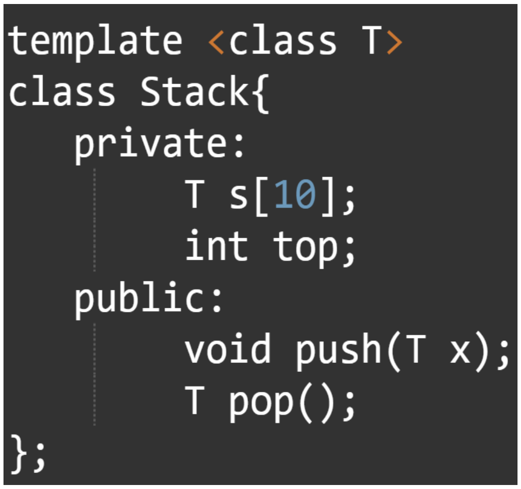
    - **Adding multiple parameters in the template declaration allows for the function to accept multiple data types**

⇒ ex) add(10, 12.9) = add(int, double);

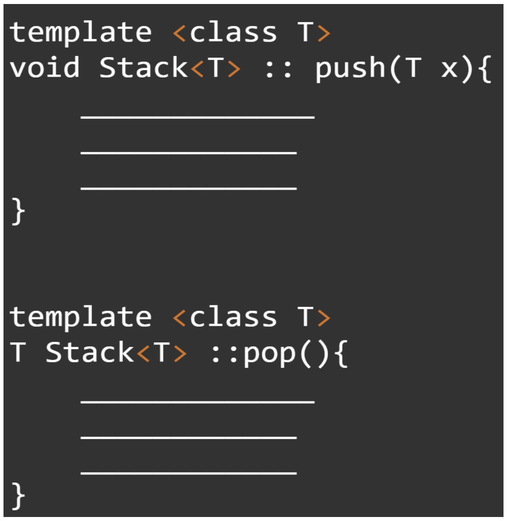
Template Classes

- I am writing a stack class and implementing a stack using an int array
- This is a stack class that can only store integers
- This stack class will not work for float, char, or any other data type
- If I want a float stack, I have to write a separate class

*** Instead of writing many classes for different data types, you can write a single class for all data types by making the class a template



```
template <class T>
class Stack{
   private:
        T s[10];
        int top;
   public:
        void push(T x);
        T pop();
};
```



```
template <class T>
void Stack<T> :: push(T x){
     _____________
     ____________
     ____________
}

template <class T>
T Stack<T> ::pop(){
     _____________
     ____________
     ____________
}
```

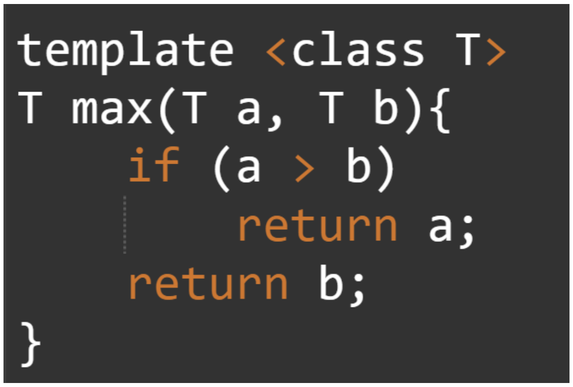
Summary for Template Basics

- Templates are a powerful feature
- Templates reduce the work of a programmer and make programming a lot easy
- In C++, we can define our own classes and function of type template

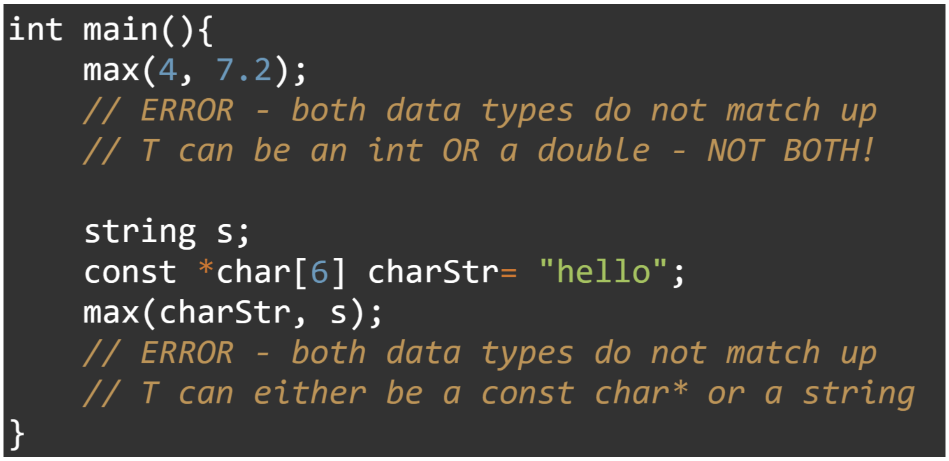
Beginning Advanced Templates

A) What's template argument deduction?

- Templates only accept the input that matches both parts.
- T can NOT take on multiple data types, because both data types do NOT match
- This creates a problem with template argument deduction, as the template is not able to determine/deduce which argument it should take.



```
template <class T>
T max(T a, T b){
    if (a > b)
        return a;
    return b;
}
```

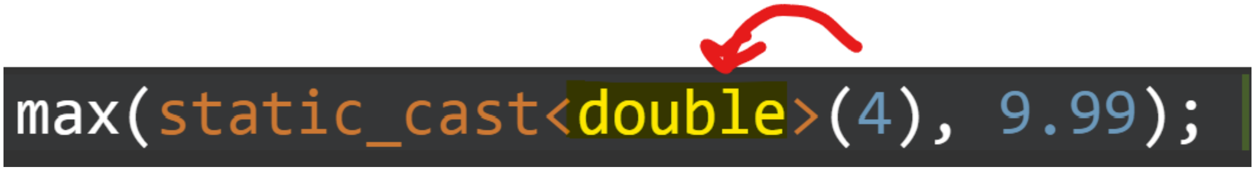


```
int main(){
    max(4, 7.2);
    // ERROR - both data types do not match up
    // T can be an int OR a double - NOT BOTH!

    string s;
    const *char[6] charStr= "hello";
    max(charStr, s);
    // ERROR - both data types do not match up
    // T can either be a const char* or a string
}
```

Question: How do we deal with template argument deduction? AKA how do we allow the template to accept multiple data types

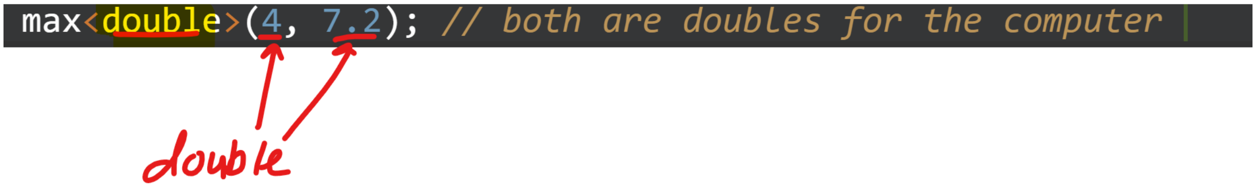
Way1 → cast the arguments so that they both match



```
max(static_cast<double>(4), 9.99);
```

- **Casting allows us to treat the 4 like a double**

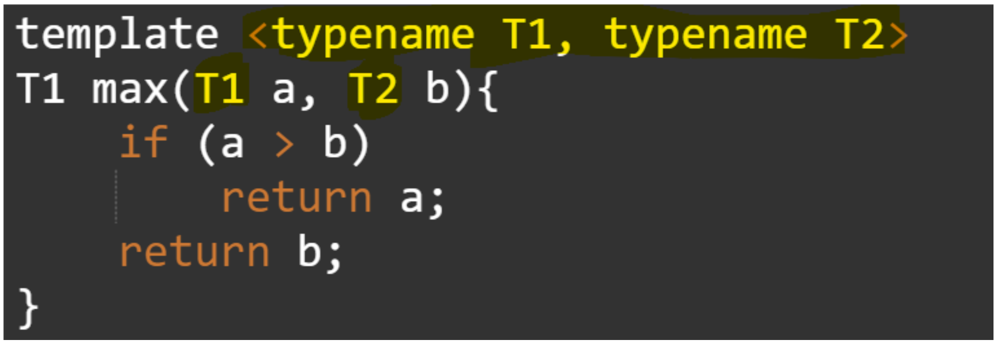
Way2 → explicitly specify the data type of T to prevent the compiler from attempting type deduction



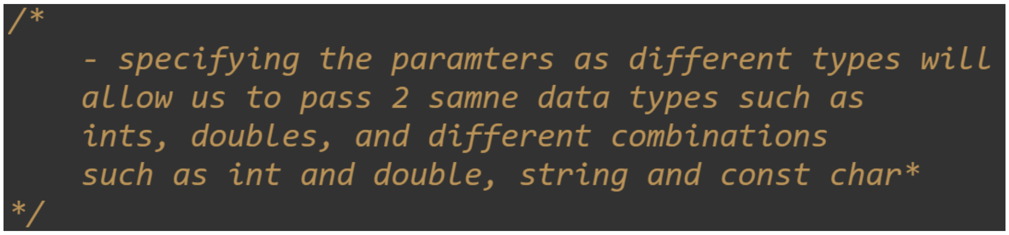
```
max<double>(4, 7.2); // both are doubles for the computer
```

double

Way3 - specify the parameters may have different data types



```
template <typename T1, typename T2>
T1 max(T1 a, T2 b){
    if (a > b)
        return a;
    return b;
}
```



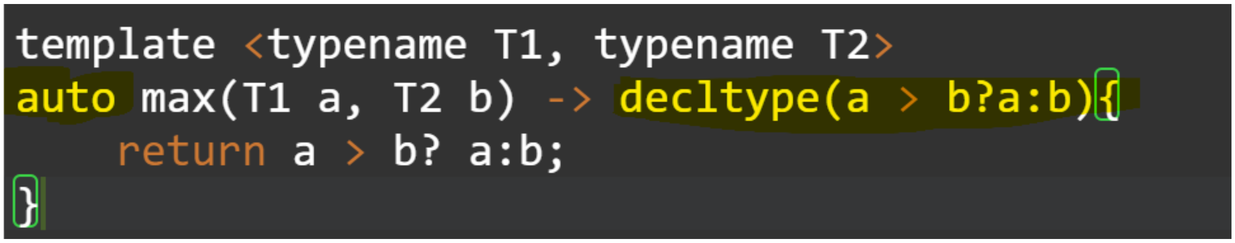
```
/*
    - specifying the paramters as different types will
    allow us to pass 2 samne data types such as
    ints, doubles, and different combinations
    such as int and double, string and const char*
*/
```

PROBLEM!!

→ what about my return type??

→ what if I don't want it to be something other than T1 ie T2?

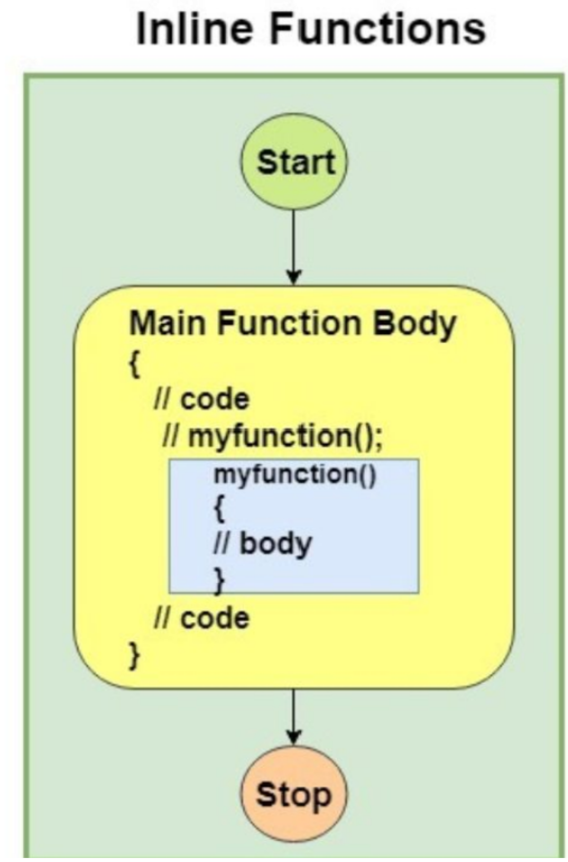
- **GOAL: The return type should be discovered by the compiler**



```
template <typename T1, typename T2>
auto max(T1 a, T2 b) -> decltype(a > b?a:b){
    return a > b? a:b;
}
```

Lambda Expressions

- **Not common, but useful**
- **Lambda expressions are unmanned functions that are inline**
- **Inline is function is defined and called INSIDE THE MAIN**



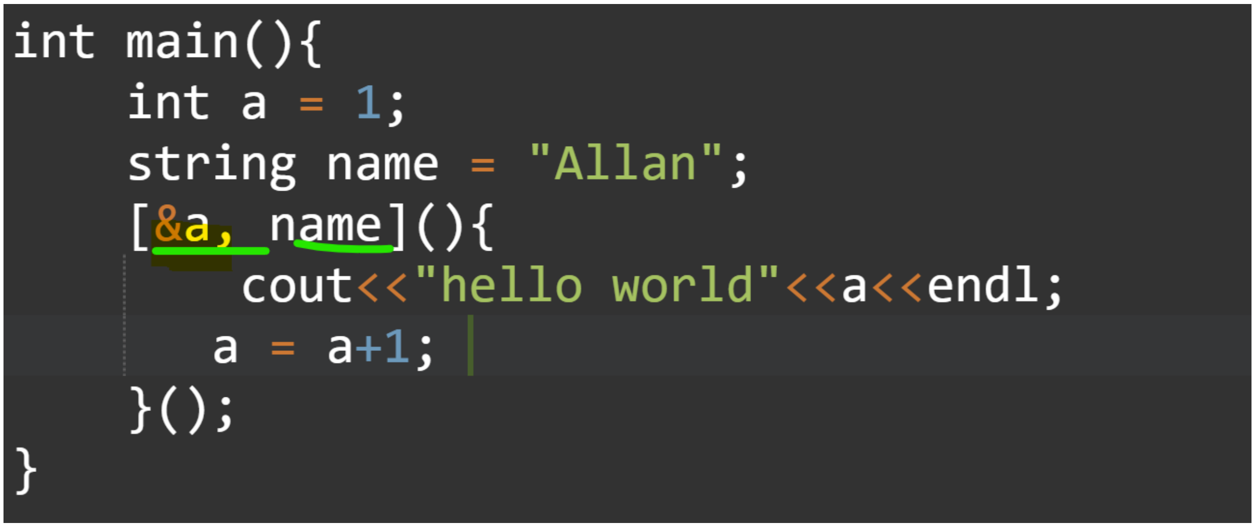
-

LAMBDA EXPRESSION = UNNAMED FUNCTION

Format of lambda expression
[capture_list](parameter list) -> returnType{...BODY..};

- **Capture list extends the scope of the lambda expression**
- **→ capture list gives access to external variables**
- **Different ways to capture**
- **1) capture by reference**
- **2) capture by values**
- **3) capture by both [MIXED!]**

// example of mixed capture



```
int main(){
    int a = 1;
    string name = "Allan";
    [&a, name](){
        cout<<"hello world"<<a<<endl;
        a = a+1;
    }();
}
```