Templates

Last updated 5/4/20

Function Templates

- Motivation
  - Many times a function performs the same operations on various types of data

```
int square(int val);
double square(double val);
```

  - Instead of creating a version of the function that works with each type of data – use a function template

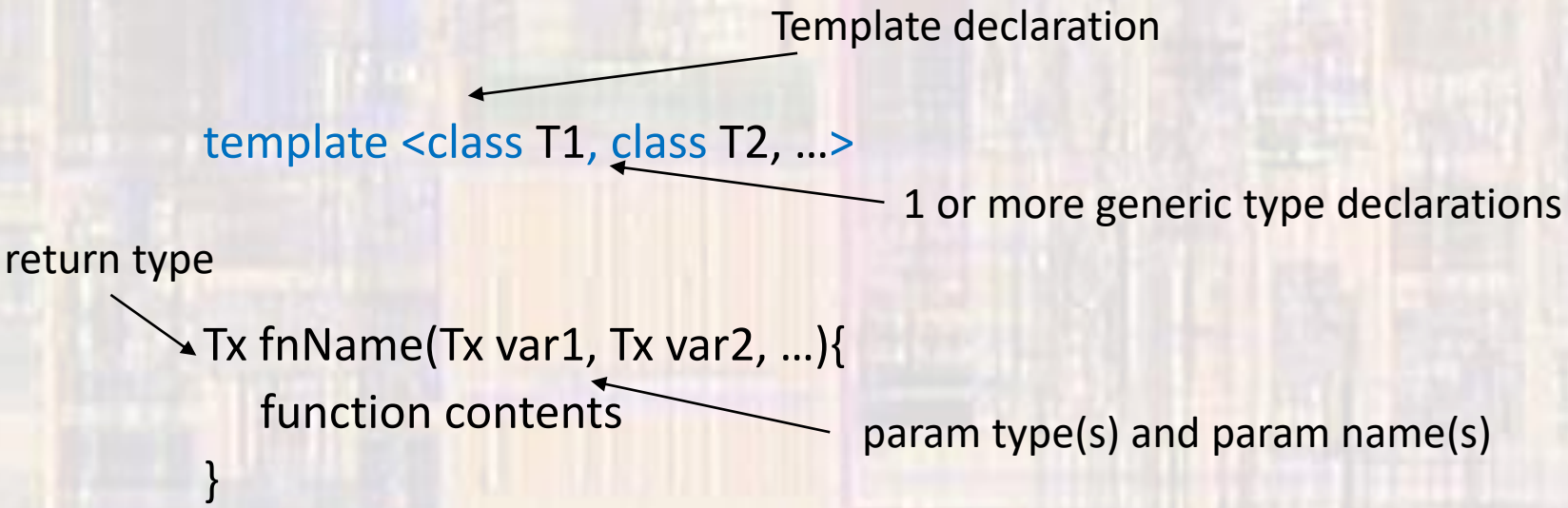
```
T square(T val);
```

Function Templates

- Template
  - The compiler creates the required version of the function (appropriate types)
    - call the function with int – creates an int version of the function
    - call the function with double – creates a double version of the function
  - Note: all operations used in the template must be supported by every type that may be called with the template
    - May require operator overloading

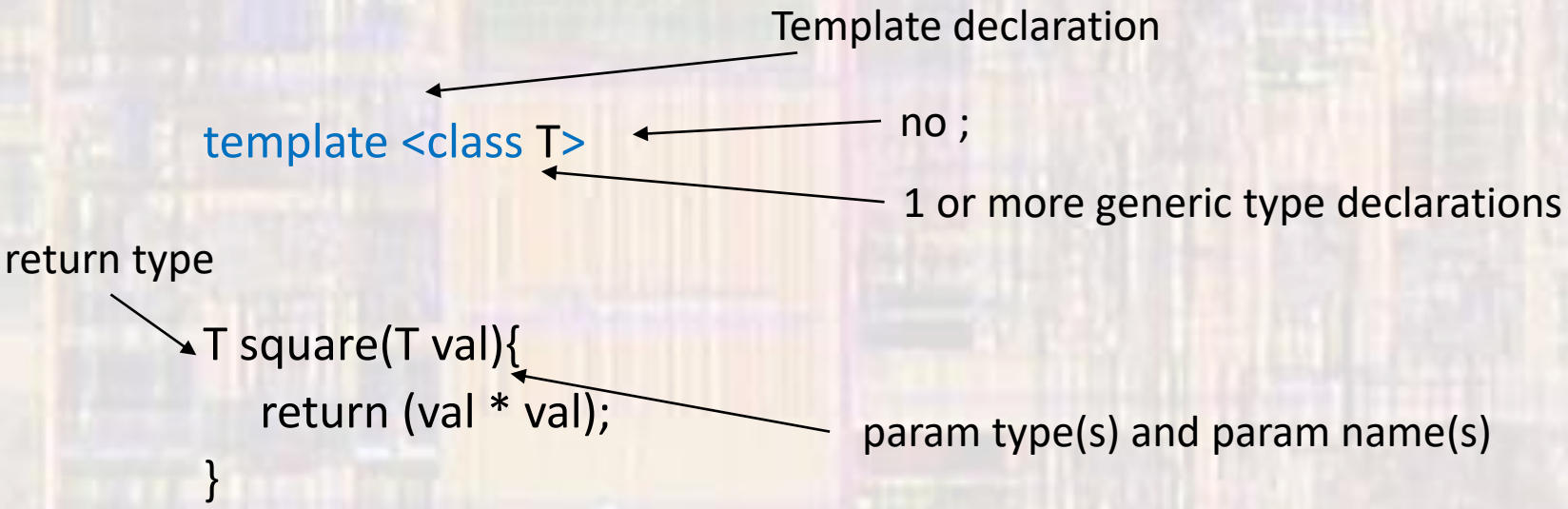
Function Templates

- Template – syntax



Function Templates

- Template – example
  - Calculate the square of given value (type independent)



Function Templates

- Template – example
  - Calculate the square of given value (type independent)
  - called with int parameters
    - Compiler creates:

```
int square(int val){
    return (val * val);
}
```

  - called with double parameters
    - Compiler creates:

```
double square(double val){
    return (val * val);
}
```

Function Templates

- Template – example
  - Calculate the square of given value (type independent)

```
/*
 * square.cpp
 *
 *  Created on: Apr 26, 2019
 *      Author: johnsontimoj
 */
////////////////////////////////
//
// Template example file - square a number
//
////////////////////////////////
#include <iostream>
using namespace std;

/////////////////
// template
/////////////////
template <class T>
T square(T val){
  return (val * val);
}
/////////////////

int main(void){
int foo;
double boo;
char soo;
Cube loo(1,5);
Cube doo;

foo = square(5);
boo = square(5.1);
soo = square('G');
doo = square(loo);

cout << foo << " " << boo << " " << soo << " " << loo.getEdge() << " " << doo.getEdge()
<< endl;

return 0;
}
```

```
Cube Cube::operator*(const Cube & rhs){
    Cube tmpcube;
    tmpcube.edge = edge * rhs.edge;
    return tmpcube;
}
bool Cube::operator<(const Cube & rhs){
    if(edge < rhs.edge)
        return true;
    else
        return false;
}
```

```
<terminated> (exit value
25 26.01 ± 5 25
```

Function Templates

- Template – example
  - Calculate the max of 2 values (type independent)

```
/*
 * my_max.cpp
 *
 *  Created on: Apr 26, 2019
 *      Author: johnsontimoj
 */
////////////////////////////////
// Template example file - find max
////////////////////////////////
#include <iostream>
#include <string>
using namespace std;

/////////////////
// template
///////////////
template <class T>
T Max (T a, T b) {
   return a < b ? b:a;
}

int main () {
   int i1 = 5;
   int i2 = 7;
   cout << "Max(i1, i2): " << Max(i1, i2) << endl;

   double f1 = 12.5;
   double f2 = 9.2;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   Cube c1(1,5);
   Cube c2(2,7);
   Cube tmpcube;
   tmpcube = Max(c1, c2);
   cout << "Max(c1, c2): ID " << tmpcube.getIdentity() << endl;

   return 0;
}
```

```
Cube Cube::operator*(const Cube & rhs){
    Cube tmpcube;
    tmpcube.edge = edge * rhs.edge;
    return tmpcube;
}
bool Cube::operator<(const Cube & rhs){
    if(edge < rhs.edge)
        return true;
    else
        return false;
}
```

```
<terminated> (exit value: 0)
Max(i1, i2): 7
Max(f1, f2): 12.5
Max(s1, s2): World
Max(c1,c2): 2
```

Function Templates

- Template – example
  - modulo (type independent)

```
/*
 * my_mod.cpp
 *
 *  Created on: Apr 26, 2019
 *      Author: johnsontimoj
 */
////////////////////////////////
//
// Template example file - modulo any 2 types
//
////////////////////////////////
#include <iostream>
#include <string>
using namespace std;
/////////////////
// template
/////////////////
template <class T1, class T2>
int mod_tmp(T1 a, T2 b) {
   return (static_cast<int>(a) % (static_cast<int>(b)));
}
/////////////////
int main () {
   int i1 = 5;
   int i2 = 7;
   cout << "mod_tmp(i1, i2): " << mod_tmp(i1, i2) << endl;

   double f1 = 12.5;
   double f2 = 9.2;
   cout << "mod_tmp(f1, f2): " << mod_tmp(f1, f2) << endl;

   char c1 = 'h';
   char c2 = 'w';
   cout << "mod_tmp(c1, c2): " << mod_tmp(c1, c2) << endl;

   return 0;
}
```

```
mod_tmp(i1, i2): 5
mod_tmp(f1, f2): 3
mod_tmp(c1, c2): 104
```

Class Templates

- Motivation
  - Many times a Class may store similar data but of different types and perform the same operations on various types of data

    vector is an example of this

  - Instead of creating a version of the class that works with each type of data – use a class template

    template <class T>
    class myClass

This is a common question in C++ programming. There are two valid answers to this. There are advantages and disadvantages to both answers and your choice will depend on context. The common answer is to put all the implementation in the header file, but there's another approach will will be suitable in some cases. The choice is yours.

The code in a template is merely a 'pattern' known to the compiler. The compiler won't compile the constructors cola<float>::cola(...) and cola<string>::cola(...) until it is forced to do so. And we must ensure that this compilation happens for the constructors at least once in the entire compilation process, or we will get the 'undefined reference' error. (This applies to the other methods of cola<T> also.)

Understanding the problem
The problem is caused by the fact that main.cpp and cola.cpp will be compiled separately first. In main.cpp, the compiler will implicitly instantiate the template classes cola<float> and cola<string> because those particular instantiations are used in main.cpp. The bad news is that the implementations of those member functions are not in main.cpp, nor in any header file included in main.cpp, and therefore the compiler can't include complete versions of those functions in main.o. When compiling cola.cpp, the compiler won't compile those instantiations either, because there are no implicit or explicit instantiations of cola<float> or cola<string>. Remember, when compiling cola.cpp, the compiler has no clue which instantiations will be needed; and we can't expect it to compile for every type in order to ensure this problem never happens! (cola<int>, cola<char>, cola<ostream>, cola< cola<int> > ... and so on ...)

The two answers are:

Tell the compiler, at the end of cola.cpp, which particular template classes will be required, forcing it to compile cola<float> and cola<string>.
Put the implementation of the member functions in a header file that will be included every time any other 'translation unit' (such as main.cpp) uses the template class.
Answer 1: Explicitly instantiate the template, and its member definitions
At the end of cola.cpp, you should add lines explicitly instantiating all the relevant templates, such as

```
template class cola<float>;
template class cola<string>;
```

and you add the following two lines at the end of nodo_colaypila.cpp:

```
template class nodo_colaypila<float>;
template class nodo_colaypila<std :: string>;
```

This will ensure that, when the compiler is compiling cola.cpp that it will explicitly compile all the code for the cola<float> and cola<string> classes. Similarly, nodo_colaypila.cpp contains the implementations of the nodo_colaypila<...> classes.

In this approach, you should ensure that all the of the implementation is placed into one .cpp file (i.e. one translation unit) and that the explicit instantation is placed after the definition of all the functions (i.e. at the end of the file).

Answer 2: Copy the code into the relevant header file
The common answer is to move all the code from the implementation files cola.cpp and nodo_colaypila.cpp into cola.h and nodo_colaypila.h. In the long run, this is more flexible as it means you can use extra instantiations (e.g. cola<char>) without any more work. But it could mean the same functions are compiled many times, once in each translation unit. This is not a big problem, as the linker will correctly ignore the duplicate implementations. But it might slow down the compilation a little.

Summary
The default answer, used by the STL for example and in most of the code that any of us will write, is to put all the implementations in the header files. But in a more private project, you will have more knowledge and control of which particular template classes will be instantiated. In fact, this 'bug' might be seen as a feature, as it stops users of your code from accidentally using instantiations you have not tested for or planned for ("I know this works for cola<float> and cola<string>, if you want to use something else, tell me first and will can verify it works before enabling it.").

Class Templates

- Example - vector

```
/*
 * myVect.h
 *
 *  Created on: Apr 29, 2019
 *      Author: johnsontimoj
 */

#ifndef MYVECT_H_
#define MYVECT_H_

template <class T>
  class myVect {
    private:
      T * array_ptr;
      int array_size;

    public:
      myVect();
      myVect(int);
      myVect(const myVect &);
      ~myVect();
      T getValAt(int);
      T & operator[](const int &);
};
```

```
//////////////////////////////
//
// Special case for templates
//
// Member functions must be part of file with main or in .h file
//
//////////////////////////////////
template <class T>
myVect<T>::myVect(){
  array_ptr = 0;
  array_size = 0;
}

template <class T>
myVect<T>::myVect(int s){
  array_size = s;
  array_ptr = new T[s];
  for(int i=0; i<array_size; i++)
    *(array_ptr + i) = 0;
}

template <class T>
myVect<T>::myVect(const myVect & vect){
  array_size = vect.array_size;
  array_ptr = new T[array_size];
  for(int i=0; i<array_size; i++)
    *(array_ptr + i) = *(vect.array_ptr + i);
}

template <class T>
myVect<T>::~myVect(){
  if(array_size >0)
  delete[] array_ptr;
}

template <class T>
  T myVect<T>::getValAt(int position){
  return array_ptr[position];
}

template <class T>
  T & myVect<T>::operator[](const int & position){
  return array_ptr[position];
}

#endif /* MYVECT_H_ */
```

Class Templates

- Example - vector

```
/*
 * myVect_test.cpp
 *
 *  Created on: Apr 29, 2019
 *      Author: johnsontimoj
 */

#include "myVect.h"

#include <iostream>
using namespace std;

int main(void){
	// create a few vectors
	myVect<int> int_vect(5);
	myVect<double> double_vect(5);
	myVect<char> char_vect(5);

	// enter some values
	for(int i=0; i<5; i++){
		int_vect[i] = 2*i;
		double_vect[i] = 2.1*i;
	}
	char_vect[0] = 's';
	char_vect[2] = 't';
	char_vect[4] = 'u';

	// print values
	for(int i=0; i<5; i++)
		cout << i << " : " << int_vect[i] << " : "
		     << double_vect[i] << " : " << char_vect[i] <<endl;

	return 0;
}
```

```
0 : 0 : 0 : s
1 : 2 : 2.1 :
2 : 4 : 4.2 : t
3 : 6 : 6.3 :
4 : 8 : 8.4 : u
```