CIS 190: C/C++ Programming

Lecture 13

Templates

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

Overloading

- used to create multiple definitions for functions in various settings:
  - constructors (in a class)
  - operators (in a class)
  - functions

- let's look at a simple swap function

Swap Function

- this is a function to swap two integers:

```
void SwapVals(int &v1, int &v2) {
   int temp;

   temp = v1;
   v1 = v2;
   v2 = temp;
}
```

Swap Function

- this is a function to swap two integers:

```
void SwapVals(int &v1, int &v2) {
  int temp;

  temp = v1;
  v1 = v2;
  v2 = temp;
}
```

what if we want to swap two floats?

Swap Function

- this is a function to swap two floats:

```
void SwapVals(float &v1, float &v2) {
   float temp;

   temp = v1;
   v1 = v2;
   v2 = temp;
}
```

Swap Function

- this is a function to swap two floats:

```
void SwapVals(float &v1, float &v2) {
   float temp;

   temp = v1;
   v1 = v2;
   v2 = temp;
}
```

what if we want to swap two chars?

Swap Function

- this is a function to swap two chars:

```
void SwapVals(char &v1, char &v2) {
   char temp;

   temp = v1;
   v1 = v2;
   v2 = temp;
}
```

Swap Function

- this is a function to swap two chars:

```
void SwapVals(char &v1, char &v2) {
    char temp;

    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

what if we want to swap two strings?

Swap Function

- okay, this is getting ridiculous

Swap Function

- okay, this is getting ridiculous
- should be able to write just one function that can handle all of these things
  - and it is!
  - using templates

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

What Are Templates?

- ***templates*** let us create functions and classes that can use "generic" input and types

- this means that functions like **`SwapVals()`** only need to be written once
  - and then it can be used for almost anything

Indicating Templates

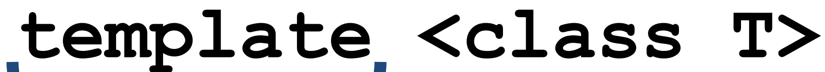
- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

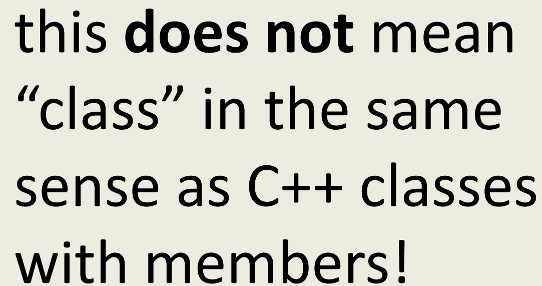


this keyword tells the compiler that what follows this will be a template

Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```



Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

this **does not** mean "class" in the same sense as C++ classes with members!

in fact, the (more) correct keyword to use is actually "**`typename`**", because we are defining a new type

Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

this **does not** mean "class" in the same sense as C++ classes with members!

in fact, the (more) correct keyword to use is actually "**`typename`**", because we are defining a new type

but "**`class`**" is more common by far, and so we will use class to avoid confusion

Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

"**T**" is the name of our new type

Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

"**T**" is the name of our new type

we can call it anything we want, but using "**T**" is the traditional way

Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

"**`T`**" is the name of our new type

we can call it anything we want, but using "**`T`**" is the traditional way

(of course, we can't use "**`int`**" or "**`for`**" or any other types or keywords as a name for our type)

Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

- what this line means overall is that we plan to use "**T**" in place of types (int, char, etc.)

Indicating Templates

- to let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

- what this line means overall is that we plan to use "**`T`**" in place of types (int, char, etc.)
- this ***template prefix*** needs to be used before both function declarations and definitions

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

Using Templates

```
template <class T>
```

- in order to create a function that uses templates, we first prefix it with this

Using Templates

```
template <class T>
void SwapVals(T &v1, T &v2) {



}
```

- next, we use "**T**" in place of the types that we want to be "generic" for our function

Using Templates

```
template <class T>
void SwapVals(T &v1, T &v2) {
  T temp;



}
```

- if we need these “generic” types inside our function, we declare them as being type “**T**”

Using Templates

```
template <class T>
void SwapVals(T &v1, T &v2) {
  T temp;
  temp = v1;
  v1   = v2;
  v2   = temp;
}
```

- everything else about our function can remain the same

Using Templates

- when we call these templated functions, nothing looks different:

```
SwapVals(intOne,  intTwo);
SwapVals(charOne, charTwo);
SwapVals(strOne,  strTwo);
```

(In)valid Use of Templates

- which of the following will work?

```
SwapVals(int, int);
SwapVals(char, string);
SwapVals(TRAIN_CAR, TRAIN_CAR);
SwapVals(double, float);
SwapVals(Shape, Shape);
SwapVals("hello", "world");
```

(In)valid Use of Templates

- templated functions can handle any input types that “makes sense”
  - i.e., any type where the behavior that occurs in the function is defined
- even user-defined types!
  - as long as the behavior is defined

Question from Class

- Q: What will happen if we overload SwapVals() manually for a specific type (like int)?
- A: The compiler accepts it, and a call to SwapVals() with integers will default to our manual overload of the function.
  - It makes sense that if an int version of SwapVals() exists, the compiler will not create one of its own.

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

Compiler Handling

- the compiler can create code to handle any (valid) call of **`SwapVals()`** we can create

- it creates separate (overloaded) functions called **`SwapVals()`** that take in ints, or chars, or floats, or TRAIN_CAR structs, or anything else that we could give

Compiler Handling

- exactly what versions of **`SwapVals()`** are created is determined at ________ time

Compiler Handling

- exactly what versions of **`SwapVals()`** are created is determined at compile time
- if we call **`SwapVals()`** with integers and strings, the compiler will create versions of the function that take in integers and strings

Separate Compilation

- which versions of templated function to create are determined at compile time

- how does this affect our use of separate compilation?
  - function declaration in .h file
  - function definition in .cpp file
  - function call in separate .cpp file

Separate Compilation

- here's an illustrative example:

```
#include "swap.h"

int main()
{
  int a = 3, b = 8;
  SwapVals(a, b);
}
```

main.cpp

```
template <class T>
void SwapVals(T &v1, T &v2);
```

swap.h

```
#include "swap.h"

template <class T>
void SwapVals(T &v1, T &v2)
{
  T temp;
  temp = v1;
  v1   = v2;
  v2   = temp;
}
```

swap.cpp

Separate Compilation

- most compilers (including eniac's) cannot handle separate compilation with templates
- when **`swap.cpp`** is compiled…

Separate Compilation

- most compilers (including eniac's) cannot handle separate compilation with templates

- when **`swap.cpp`** is compiled…
  - there are no calls to **`SwapVals()`**

Separate Compilation

- most compilers (including eniac's) cannot handle separate compilation with templates

- when **`swap.cpp`** is compiled…
  - there are no calls to **`SwapVals()`**
  - **`swap.o`** has no **`SwapVals()`** definitions made

Separate Compilation

- most compilers (including eniac’s) cannot handle separate compilation with templates
- when **`main.cpp`** is compiled…

Separate Compilation

- most compilers (including eniac’s) cannot handle separate compilation with templates
- when **`main.cpp`** is compiled…
  - it assumes everything is fine
  - since **`swap.h`** has the appropriate declaration

Separate Compilation

- most compilers (including eniac’s) cannot handle separate compilation with templates
- when **`main.o`** and **`swap.o`** are linked…

Separate Compilation

- most compilers (including eniac's) cannot handle separate compilation with templates
- when **`main.o`** and **`swap.o`** are linked…
  - everything goes wrong

Separate Compilation

- most compilers (including eniac's) cannot handle separate compilation with templates

- when **`main.o`** and **`swap.o`** are linked…
  - everything goes wrong
  - **`error: undefined reference to 'void SwapVals<int>(int&, int&)'`**

Solutions

- the template function definition code must be in the same file as the function call code

- two ways to do this:
  - place function definition in **`main.c`**
  - place function definition in **`swap.h`**, which is #included in **`main.c`**

Solutions

- second option keeps some sense of separate compilation, and better allows code reuse

```
#include "swap.h"

int main()
{
  int a = 3, b = 8;
  SwapVals(a, b);
}
```

main.cpp

```
// declaration
template <class T>
void SwapVals(T &v1, T &v2);

// definition
template <class T>
void SwapVals(T &v1, T &v2)
{
  T temp;
  temp = v1;
  v1   = v2;
  v2   = temp;
}
```

swap.h

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

Class Templates

- syntax for class declaration is very similar:

```
template <class T>
class Pair {
private:
  T    GetFirst();
  void SetFirst(T first);
private:
  T m_first;
  T m_second;
};
```

Class Templates

- syntax for class declaration is very similar:

```
template <class T>
class Pair {
private:
   T     GetFirst();
   void SetFirst(T first);
private:
   T m_first;
   T m_second;
};
```

Class Templates

- syntax for class declaration is very similar:

```
template <class T>
class Pair {
private:
    T     GetFirst();
    void SetFirst(T first);
private:
   T m_first;
   T m_second;
};
```

Class Templates

- syntax for class declaration is very similar:

```
template <class T>
class Pair {
private:
  T     GetFirst();
   void SetFirst(T first);
private:
  T m_first;
  T m_second;
};
```

Templated Classes

- most common use for templated classes is containers (like our Pair example)

- in fact, many of the C++ STL containers actually use templates behind the scenes!
  - like vectors!

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

Class Constructors

- normally, we create just one constructor, by using default parameters:

```
Date (int m = 10, int d = 15,
      int y = 2014);
```

- this allows us to create a Date object with no arguments, all arguments, and everything in between

Templated Class Constructors

- can we do the same with our **`Pair`** class?

Templated Class Constructors

- can we do the same with our **`Pair`** class?

```
Pair (T first = 0, T second = 0);
```

Templated Class Constructors

- can we do the same with our **Pair** class?

```
Pair (T first = 0, T second = 0);
```

- this works fine if we're creating a **Pair** object of a number type (int, float, double), but what about strings, or TRAIN_CAR?

Templated Class Constructors

- can we do the same with our **`Pair`** class?

```
Pair (T first = 0, T second = 0);
```

- this works fine if we're creating a **`Pair`** object of a number type (int, float, double), but what about strings, or TRAIN_CAR?

- the nature of templates means we can't use default parameters for templated classes

Templated Class Constructors

- need to create two constructors for the class
- ???
  - and
- ???

Templated Class Constructors

- need to create two constructors for the class

- empty constructor (no arguments)

  ```
  Pair ();
  ```

  – and

- ???

Templated Class Constructors

- need to create two constructors for the class

- empty constructor (no arguments)

  ```
  Pair ();
  ```

  – and

- complete constructor (all arguments)

  ```
  Pair (T first, T second);
  ```

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

Templated Class Definitions

- just like with regular functions, member function definitions need the ***template prefix***

```
template <class T>
```

Templated Class Definitions

- just like with regular functions, member function definitions need the ***template prefix***

```
template <class T>
```

- in addition, they need a template indicator before the scope resolution operator:

```
Pair<T>::Pair(T first, T second);
```

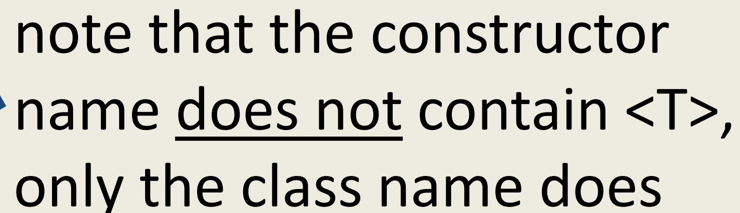
Templated Class Definitions

- just like with regular functions, member function definitions need the ***template prefix***

```
template <class T>
```

- in addition, they need a template indicator before the scope resolution operator:

```
Pair<T>::Pair(T first, T second);
```



Templated Class Definitions

- everything else about the function behaves as with non-member templated functions

```
Pair<T>::Pair(T first, T second)
{
    SetFirst(first);
    SetSecond(second);
}
```

Templated Class Definitions

- everything else about the function behaves as with non-member templated functions

```
Pair<T>::Pair(T first, T second)
{
  m_first = first;
  m_second = second;
}
```

since most error checking is not feasible with templated classes, it is fine to directly set variables in the constructor, as above

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

Using Templated Classes

- identical to the way you use templated classes provided by the STL (like vectors)

Using Templated Classes

- identical to the way you use templated classes provided by the STL (like vectors)

```
vector <int>  myVector(10);
vector <char> aVector;
```

Using Templated Classes

- identical to the way you use templated classes provided by the STL (like vectors)

```
vector <int>  myVector(10);
vector <char> aVector;

Pair <string> hi("hello", "world");
Pair <int> coordinates;
```

Outline

- Overloading Functions
- Templates
  - Function Templates
  - Compiler Handling & Separate Compilation
- Class Templates
  - Declaring
  - Constructors
  - Defining
  - Using
- Project

Project

- alpha due this Sunday (23rd) @ midnight
- presentation days will be Tuesday, Dec 2nd (6-7:30) and Wednesday, Dec 3rd (1:30-3)
- **attendance at both presentation days is mandatory! you will lose points for skipping!**
- final code turn-in is Wed, Dec 3rd @ midnight