Templates

CS A250 – C++ Programming II

PREDEFINED TEMPLATE CLASSES

- Recall `vector` class
  - It is a **template** class
- Syntax: `vector<Base_Type>`
  - Indicates **template** class
  - Any type can be "plugged in" to **Base_Type**
  - Produces "new" class for vectors with that type
- Example declaration:

```
vector<int> v;
```

where `v` is a vector of type `int`

TEMPLATES

- C++ **templates**
  - Allow very "general" definitions for **functions** and **classes**
    - Can work with many **different** **types** of values
    - Allow the creation of general purpose, re-usable tools
  - Precise definition determined at ***runtime***

Function Templates

- Recall function **swapValues**:

```
void swapValues(int& var1, int& var2)
{
        int temp = var1;
        var1 = var2;
        var2 = temp;
}
```

Applies only to variables of type `int`

Function Templates vs. Overloading

- Could overload function for **char**'s:

```
void swapValues(char& var1, char& var2)
{
        char temp = var1;
        var1 = var2;
        var2 = temp;
}
```

- **But** notice: code is nearly identical!
  - Only difference is **type** used in 3 places

Function Template Syntax

- Allow "swap values" of any type variables:

```
template<typename T>
void swapValues(T& var1, T& var2)
{
        T temp = var1;
        var1 = var2;
        var2 = temp;
}
```

- First line called "**template prefix**"
  - Tells compiler what is coming is "template"
  - And T is a type parameter

Class or Typename?

- These declarations are the same:

```
template <class T>
template <typename T>
```

- `typename` is newer syntax
  - **class** is still used!
  - We will be using **typename**
- `T` is simply an **identifier**

Template Prefix

- **T** can be replaced by ***any*** type
  - Predefined or user-defined (like a C++ class type)
- In function definition body:
  - **T** used like any other type

- **Note:** can use other identifiers instead of "**T**", but **T** is "traditional" usage.

Calling a Function Template

- Consider this function call:

```
swapValues(int1, int2);
```

  - C++ compiler "generates" function definition for two `int` parameters using the template type.

- No need to do anything "special" in function call
  - Required definition automatically generated.

Another Function Template

- **Declaration/prototype:**

```
template<typename T>
void func(int, const T&, const T&);
```

- **Definition:**

```
template<typename T>
void func(int param1, const T& param2,
                const T& param3)
{
        //do something…
}
```

Another Function Template (cont.)

- **Declaration/prototype:**

```
template<typename T>
void func(int, const T&, const T&);
```

Why pass by reference and as a const?

ANOTHER FUNCTION TEMPLATE (CONT.)

- **Declaration/prototype:**

```
template<typename T>
void func(int, const T&, const T&);
```

T could be an object.

CALL TO FUNCTION `func`

- Consider **function call**:

```
func(2, 3.3, 4.4);
```

- **Declaration/prototype:**

```
template<typename T>
void func(int, const T&, const T&);
```

- Compiler generates function definition
  - Replaces **T** with **`double`**

Example 1

- **File:** Function_template

Algorithm Abstraction

- **Algorithm abstraction**:
  - Refers to implementing templates
  - Express algorithms in "general" way:
    - Algorithm applies to variables of any type
    - Ignore incidental detail
    - Concentrate on substantive parts of algorithm
- **Function templates** are one way C++ supports **algorithm abstraction**.

Defining Templates Strategies

- Steps:
  1. Develop function normally
     - Using actual data types
  2. Completely debug "ordinary" function
  3. Then convert to template
     - Replace type names with type parameter as needed

- Advantages:
  - Easier to solve "concrete" case
  - Deal with algorithm, not template syntax

Multiple Type Parameters

- Can have:

```
template<typename T1, typename T2>
```

- Not typical
  - Usually only need one "replaceable" type
  - **Cannot** have "unused" template parameters
    - Each must be "used" in definition
    - Error otherwise!

Inappropriate Types in Templates

- Can use any type in template for which code makes "sense"
  - Code must behave in appropriate way
  - For example, **`swapValues()`** template function
    - Cannot use type for which assignment operator is ***not*** defined
    - Example: an array:

```
int a[10], b[10];
swapValues(a, b);
```

    - Arrays **cannot** be "assigned" ➔ $a \neq b$

Class Templates

- Can also "generalize" classes
  - `template<typename T>` can be applied to **class definition**
  - All instances of **T** in class definition replaced by type parameter
  - Just as seen on function templates
- Once template is defined, you can declare objects of the class.

Example Class Template

- Assume you have a class **Pair**
  - Creates objects that contain two member variables
    - A "**pair**"
    - Can be any type of pairs (**int**, **double**, etc.)

```
class Pair
{
public:
      Pair();
      Pair(int firstVal, int secondVal);
      void setFirst(int newVal);
      void setSecond(int newVal);
      int getFirst() const;
      int getSecond() const;
private:
      int first, second;
};
```

Example Class Template (cont.)

- We can generalize it by making it a **template class**

```
template<typename T>
class Pair
{
public:
      Pair();
      Pair(const T& firstVal, const T& secondVal);
      void setFirst(const T& newVal);
      void setSecond(const T& newVal);
      T getFirst() const;
      T getSecond() const;
private:
      T first, second;
};
```

Example Class Template (cont.)

- Here we have two member functions implemented:

```
template<typename T>
Pair<T>::Pair(const T& firstVal,
                   const T& secondVal)
{
      first = firstVal;
      second = secondVal;
}

template<typename T>
void Pair<T>::setFirst(const T& newVal)
{
      first = newVal;
}
```

Example Class Template (cont.)

- Now we can create objects of the class Pair using any type <u>that fits</u>:

```
Pair<int> score;
Pair<char> seats;
```

- And use any of the functions:

```
score.setFirst(3);
score.setSecond(5);

seats.setFirst('a');
seats.setSecond('b');
```

Pair Member Function Definitions

- Notice in **member function definitions**:
  - Each definition is itself a "template"
  - Requires **template prefix** before each definition
  - **Class qualifier** before **`scope resolution`** is "**`Pair<T>`**"
    - Not just "Pair"

```
template<typename T>
void Pair<T>::setPair(const T& firstItem,
                                        const T& secondItem)
{ … }

template<typename T>
T Pair<T>::getFirstItem() const
{ … }
```

Compiler Complications

- Function declarations and definitions
  - Typically we have them separate (***separate compilation***)
  - For **templates** → ***not*** supported on most compilers!

  - **Solution:**
    - Use **template<typename T>**

      before the **function definition** **AND**

      before the **function declaration**.

Compiler Complications (cont.)

- Check your compiler's specific requirements
  - Some need to set special options
  - Some require special order of arrangement of template definitions vs. other file items
  - **MS Visual Studio:** Need to **include the .cpp** template file in all files where you are including the **.h** template file.

Class Templates as Parameters

- Consider:

```
int addUp(const Pair<int>& thePair) const;
```

  - The type (**int**) is supplied to be used for **T** in defining this class type parameter
  - It "happens" to be call-by-reference here

- Again: template types can be used anywhere standard **types** can.

COMMON ERRORS

- You **cannot** use mixed types of parameters with the same identifier

```
template <typename T>
void swapValues (T& var1, T& var2){...}
```

Cannot have a function call like this:

```
swapValues (int var1, double var2);
```

Common Errors (cont.)

- Forgetting to include the **.cpp** file in the file where the template class is used.

```
#include "TemplateClass.h"
#include "TemplateClass.cpp"
…
int main()
{
    TemplateClass<int> obj;
    ...
    return 0;
}
```

Common Errors (cont.)

- Forgetting that the **member function definitions** are themselves templates and need to have `template<typename T>`

```
template<typename T>
Pair<T>::Pair(const T& firstValue,
                          const T& secondValue)
{
    first = firstValue;
    second = secondValue;
}
```

Templates and Inheritance

- Nothing new here

- Derived template classes
  - Can derive from template or non-template class
  - Derived class is then naturally a template class

- Syntax same as ordinary class derived from ordinary class.

Example 2

- **Project:** Pair Class

(Templates)