17 - Templates

March 20, 2023

COMP2404

Darryl Hill

Contents

Overview

Recall some of the principles of OO design:

- Data abstraction
  - Separation of abstract interfaces from concrete implementations
- Code reuse
  - Reuse existing code in your programs
  - Make your classes reusable

Overview

Mechanisms for code reuse

- Use existing libraries
  - generally well made, few bugs, efficient
  - when libraries are widely used, bugs and inefficiencies are found much more quickly
- Generic or Template programming
  - write general algorithms that can be applied to multiple data types
  - "generics" in Java
  - called templates in C++
    - everything is named different in C++, but C++ came first ¯_(ツ)_/¯

Overview

Template : something that establishes or serves as a pattern

- We will write code with generic placeholders for type.
- These placeholders are later filled in with an actual type by the compiler.

Templates goal

- Write an algorithm and reuse it on different data types

Approach

- Data type is a ”parameter”
  - Templates can be function or class templates
- Template code can be used with any data type
  - As long as it implements all needed member functions
  - Compiler will copy and paste the data type in and attempt to compile

Function Templates

Goal - create a function that works with any data type.

Any function can be made into a template.

- Global function or member function.
- If a member function is templated, its class must also be templated
  - When the class is instantiated, the compiler must know the data type in order to reserve the proper amount of memory.

coding example <p1>

Coding p1

- Consider a function `int max(int, int, int)`
- We would like to do the same for `floats`
  - the logic is precisely the same
- We can overload the function and provide:

  `float max(float, float, float)`
- We are copying code. To reuse code (i.e., the compiler do the copying) we instead use a **templated** function:

```
template <typename T>
T max(T v1, T v2, T v3){return v1;}
```

The compiler uses this template to make functions with different data types.

Function Templates

What we do:

- Write a function with a type marker (`T` in this case).
- Call the function using any data type - ***parameterized type***.
  - `T` is a variable for a `type` or `class`

What the compiler does:

- Creates a separate function for each different type used.
  - It copies the function code and does "find-replace" on `T` with the type.
  - If the generated code won't compile it gives an error.
    - For example, if we call a function on `T` that does not exist in the supplied type.
- The compiler generates functions ONLY for types that appear in the code.
- The compiler does all copying and pasting and find/replace for you.
  - Exactly as if we wrote the overloaded function ourselves.

Function Templates

Overloading templated with non-templated functions:

- Templated function can be overloaded
- Non-templated is chosen over templated (that is, the compiler will not create a function with the same signature as one that already exists)

coding example <p2>

Class Templates

Goal

- Create a class that works with any data type.
- This class requires a ***type parameter*** when it is declared.
  - The type parameter will replace the parameterized type (`T` for instance).

Characteristics

- The parameterized type may be used anywhere in the class.
  - Data members (we can have a data member of type `T`)
  - Member functions (local variables or parameters may be of type `T`)

Often used for collection classes.

- Code can be reused for many different types.

coding example <p3>

Class Templates

With templated classes the function implementations went into the header file.

- Templates can cause problems when separated into header and source.
- There are (complex) ways around this.
- In this class we will avoid (even more) compile problems this could create
  - With templated classes, include the source code in the header file.

Coding Example p3 Notes


```
template <class T>
class Array
{
    template <class V>
    friend ostream& operator<<(ostream&, const Array<V>&);
    ...
}
```

In this example, we use `V` instead of `T` because if we (re)used `T` it would hide the class `T`.

class vs typename

Which do we use?

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

- `typename` and `class` are interchangeable for the basic task of naming a template type (see above)
- We will not be looking at the difference in this class in detail, however one example:
- `typename some_template<T>::some_type`
- `some_type` could be a variable name or a type
  - we are telling the compiler that it is a type
- here is a good Stackoverflow post outlining some differences

Class Templates

How we implement it:

- Write the class with the type variable (for example, `T`) as a data type
- Declare an instance of the class using any data type.

What the compiler does:

- Creates a ***new class*** for each data type used.
  - New classes are ***only*** generated for the data types used.
  - Instances of `T` are replaced with the type parameter.
- The newly created class is then compiled with your code.

Class Templates

Special Cases:

- multiple types - **coding <p4>**
- non-type parameter - **coding <p5>**
- default types - **coding <p6>**

Static members

- each specialization gets its own copy of the static members
- after all, they are separate classes

Coding Examples Notes

- Template types really are parameters. You can pass in actual arguments this way.
- Keep in mind the compiler will make ***separate classes*** based on different parameter values.
  - This is an inefficient way to pass in simple values.
  - But sometimes...
- You can even assign them default values.
  - Then instantiate a type with empty angle brackets.

Constraining Template Types

In Java you can restrict generics to a subclass of a class

You CANNOT do this in C++

- at least not explicitly
- it is done implicitly by the functions that you call
- there are other hacks

```
template <class T> void function(){ Baseclass *object = new T(); }
```

This would throw a compiler error if `T` is not a `Baseclass` or a derived class of `Baseclass`, at the expense of a bit of useless code that is never run.

STL vector Class

Standard Template Library (STL)

- This library provides many container classes and algorithms

Arrays are primitive.

- Just a chunk of memory with an associated type.
- No bounds checking
- Doesn't support operators
  - Assignment
  - Relational
  - etc

STL vector class

Alternative to arrays: `vector` class template.

- Stores any data type.
- Supports many operators.
- Has useful member functions:
  - `size()`
  - `at()`

coding example <p7>

Coding Example p7 Notes

A `vector` of pointers is more efficient.

- A `vector` copies and destroys data as it resizes.
- For a `vector` of objects this can be expensive.

Retrieving data:

- `at()` function does bounds checking
  - will throw an exception
    - graceful way to exit or handle the error
- `[ ]` operators work on `vector`s but do not do bounds checking.

Note that `vector` has a copy constructor.

- It will copy your data to a new `vector`.
- If the `vector` stores pointers, **only** the pointers will be copied.
- If you want a "deep" copy, you will have to copy it yourself.

Smart Pointers Redux

We have seen smart pointers, but they were made for `Time` class only.

- If we template `SPointer`, then we can have a smart pointer that works for any class.

coding example <p8>