C++ Advanced Topics: Templates (Session 1)

Anupama Chandrasekhar

Jayashree Venkatesh

Welcome to Advanced C++ Series

- Kicking off with a 3 session class on C++ templates
- Thank you to Nvidia for letting hosting us and allowing us access the learning and development framework. Special shout out to **Chen Luo** and **Sneha Koka**.
- Thank you to WWC for providing a platform where like-minded professionals can come together and exchange ideas.
- We'll have more to come in the future .. stay tuned ☺

Introductions

- **Anupama Chandrasekhar**: C++ programmer for over 10 years at Intel and Nvidia. Worked on system level graphics software and compilers for Intel and Nvidia GPUs
- **Jayashree Venkatesh:** Compiler engineer and C++ programmer with 8 years of experience on Intel and Nvidia GPU software. Application software engineer at Intel-Movidius with VPU end-to-end integration experience.

Overview of the Series

Session 1	Session 2	Session 3
Motivating Templates	Type deduction and Overload resolution	Tag dispatch, SFINAE
Introduction to Template syntax	Deep dive into Class Templates,	Common compiler/linker errors and debugging tips
Deep dive into Function Templates	Variable Templates, Alias Templates	Questions and Open Discussion

Housekeeping notes

- In a templates class, we are interested in what is generated at compile time in addition to the runtime behavior. For that we find this tool invaluable : https://godbolt.org/
- Interrupt at us at any point to ask questions

Problem Statement

- If I want to ”do the same thing” for different datatypes. How do I express this intent?
- Example:
  - Find the smaller one of two objects
  - Sort a collection of objects
  - Create a stack of objects

OOP Concept: Polymorphism

- C++ is an object oriented language
- Amongst other important properties of object oriented languages like: classes, encapsulation etc. … polymorphism is an important property.
- In computer science we take it to mean, code in which the variables can be of different types at different times” and the same code can figure out the correct behavior.

Kinds of Polymorphism

- **Inheritance** -> dynamic, sub-type, run-time

```
class Concatinate : public class Sum { };
```

- **Templates** -> Generic, parametric, compile-time

```
template <typename T> T Sum(T a, Tb)
```

- **Overloading** -> ad hoc polymorphism

```
int Sum(int A, int B);
double Sum (double A, double B);
```

- **Casting** -> Coercion Polymorphism

```
int acc = Sum(static_cast<int>(7.0), static_cast<int>(4.0));
```

Casting

- One way to achieve ”Polymorphism” is casting, in fact some like to call this coercion Polymorphism.

```
int min (int a, int b);

//in main()
int i = 10, j = 20;
auto k = min(i,j);

float l = 1.f, m = 2.f;
auto n = static_cast<float>(min(static_cast<int>(l)
                              , static_cast<int>(m)));
// same as
auto n = static_cast<float>(min(l,m));
```

Casting : Drawbacks

- Not every type can be cast into another. So Very Limited

```
int min (int a, int b);
min(static_cast<int>("hello"), static_cast<int>("World")); // compiler error
```

- Gets around the type system, so subtle bugs can creep in. Buggy
- And quite frankly, quite ugly. Ugly
- In general the use of casting is discouraged.

Function Overloading

- Consider the following

```
int min(int a, int b); // (A)
float min(float a, float b) // (B)
```

- When you call min, the compiler decides based on the type which is the best match for the argument types (Overload resolution)

```
int a, b;
min(a,b); // calls min(int a, int b)

float c,d;
min(c,d); //calls min(float a, float b)
```

Function Overloading : Drawbacks

- Code duplication. If the algorithm for different types is the same, you still have to duplicate the function for every single type.

```
int min (int a, int b) { return a < b ? a : b; }

float min (float a, float b) { return a < b ? a : b; }
```

- All the code gets compiled, not only what you are using, so code bloat.
- What if you have a new type for which you want the functionality, need to add a new function, not always easy. Not scalable.

Interfaces

- **Inheritance** or dynamic polymorphism allows for common interfaces for different types that are related through a class hierarchy. Can we use this to solve the type abstraction problem we have been trying to solve.
- To illustrate how we might approach this, let us consider two problems where we would like treat different types in the same way

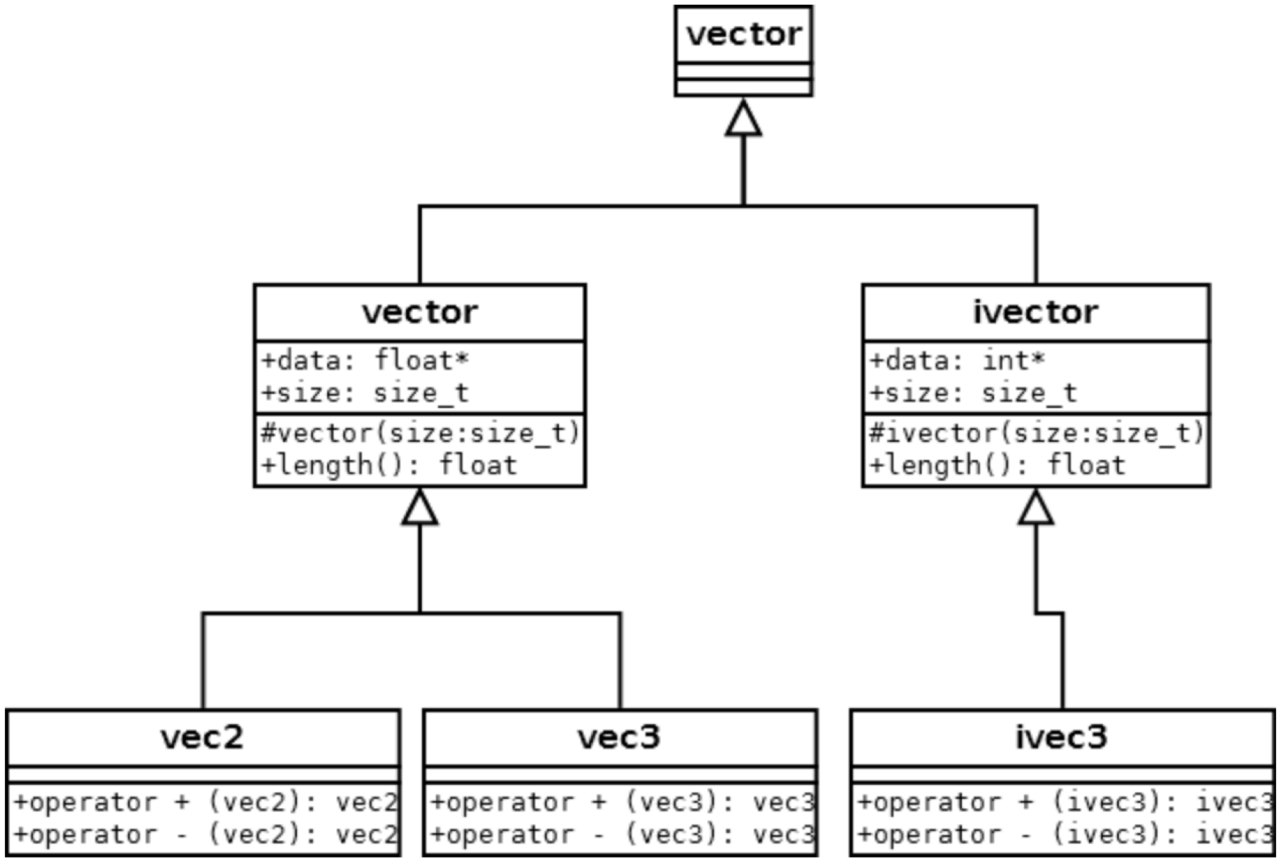
```
vector<int> vector<float> vector<MyClass>
sort(Range, predicate);
```

Non-template solutions for **`vector`** class

- Consider Collection e.g. vector
  - What we would like is `vector<int>, vector<float>`
  - So what are our choices:
    - We could have `vectorInt`, `vectorFloat` inherit from vector. Vector will expose an interface and the subclass implements those for specific types

Non-template solutions for **`vector`** class

- Since the code needs to accommodate multiple types like `float, int, double` etc, the base case is virtually useless
- Or you need to use type erasure and store `void` pointers in the base class and this gets around the type system which we would like to avoid… because all the bookkeeping involved in maintaining the size of elements etc. makes the implementation more prone to bugs.



Comparing the two implementations

```
vector operator + (const vector& a,
                   const vector& b)
{
    size_t new_size = max(a.size, b.size);
    // new memory is allocated on the heap
    vector result(new_size);

    for (unsinged int i = 0; i < new_size; i++)
    {
        // guard against out of bounds access
        float ai = i < a.size ? a[i] : 0.0f;
        float bi = i < b.size ? b[i] : 0.0f;
        result[i] = ai + bi;
    }
    return result;
}
```

```
template <typename T, size_t N>
vector<T, N> operator + (const vector<T, N>& a,
                         const vector<T, N>& b)
{
    vector<T, N> result;

    for (unsinged int i = 0; i < N; i++)
    {
        // you may use v[i] directly, we know i < N
        result[i] = a[i] + b[i];
    }
    return result;
}
```

Non-template solutions for **`vector`** class

- Another technique often used for implementing generics is macros. Not covered in this presentation. Look here for a comprehensive example: https://accu.org/index.php/journals/1473

```
#define vector( TYPE ) \
class VectorOf##TYPE                    \
{                                       \
    .                                   \
    .                                   \
    .                                   \
}
```

Type Erasure

- **Type Erasure** : Type erasure is a one of the commonly used techniques to implement generics. Here we make the code generic by casting the different data types to void* pointers and storing other meta information about the types like size of element, stride etc.
- Though common and used extensive, especially in C libraries, this technique is bug prone and misses lot of safe guards provided by the type system.

Possible Solutions for `sort`

- Consider Algorithms e.g sorting
  - Look at the Standard Library sort, accepts iterators and predicate. And we see that it is templated.
  - How would be implement sorting of different types without templates?
    - Look at C standard library -> C does not have templates
    - `qsort` (**type erasure**: pointer, num or elements, size of elements, function pointer)
      - More prone to error (could get `numElem` or `size` wrong), off by one errors.
      - Elements need to be contiguous
    - Sortable Interface:
      - Everyone who want to be sortable can inherit from this interface
      - Primitive types cannot inherit from a base class … define a wrapper for the primitive data types?

Summary

Casting	Function Overloading	Inheritance	Macros
Limited to a handful of cases.	Code Duplication	Code Duplication	Hard to Debug
Bug Prone (goes around the type system)	Code Bloat: Compile functions that are not used	Resorts internally to type erasure type techniques to handle the underlying type differences	Bug Prone (goes around the type system)
	Not Scalable : Handling new types means adding new functions	Imposes type relations artificially	Hard to maintain

Templates

- So, what we really want is a way by which we can pass type as a parameter and then the compiler, using the function/class template definition, manufactures the specific function/class for that type.

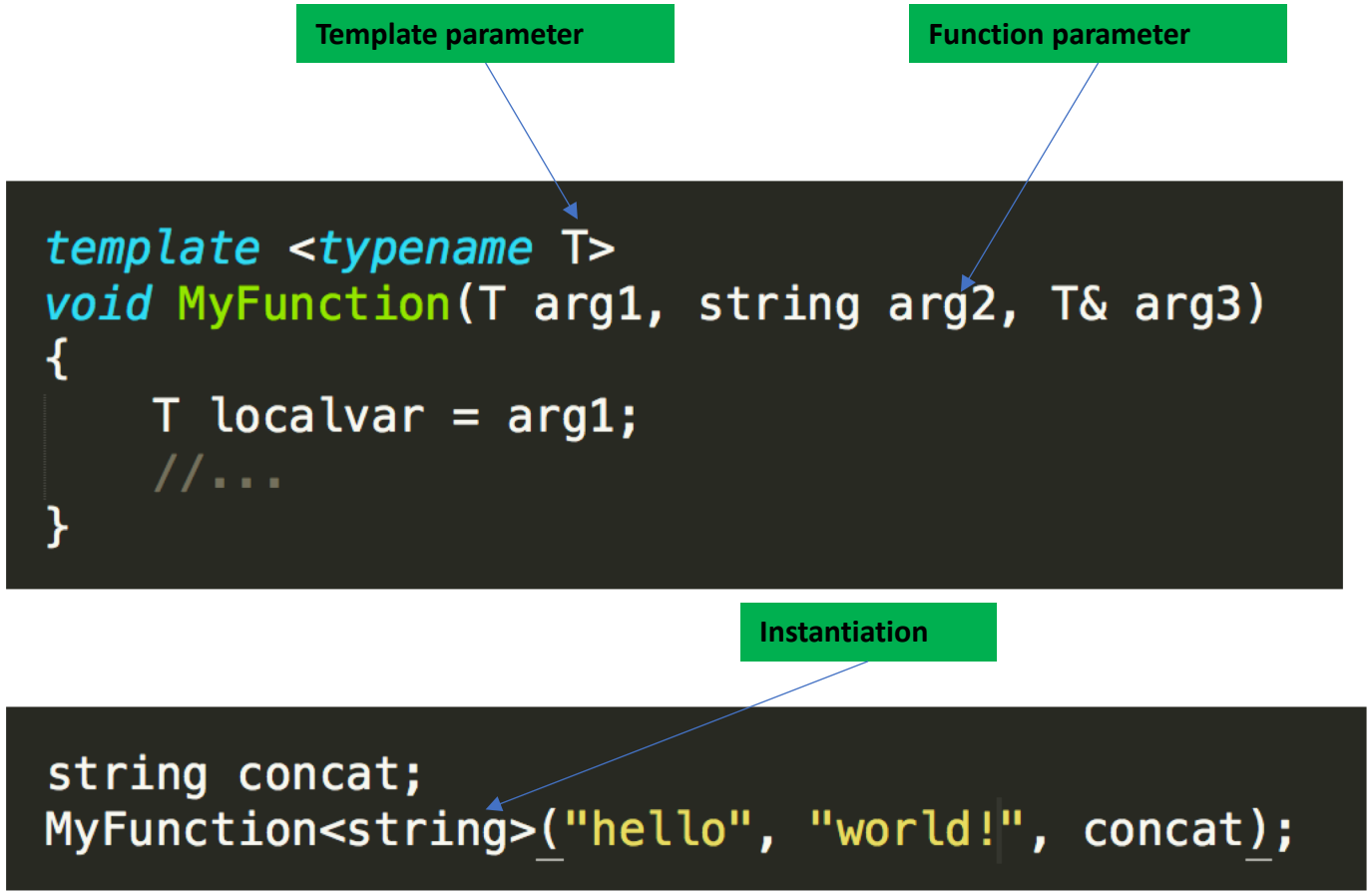
```
vector<int> intVec = {1,2,3,5};
vector<Apple> appleVec = {Apple{green}, Apple{red}}
```

- Templates provide a way to represent a wide variety of general concepts dealing in a more general way about the types of the input.
- The resulting classes and functions can match handwritten, less general code in run-time and space efficiency.

Templates

- Built-in types can be template parameters.
- The arguments to a template are unconstrained, only their instantiations are type checked. This is “Duck typing” (if it walks like a duck and quacks like a duck, it is a duck). Sometimes this leads to late type checking and poor error messages which is an ongoing area of research.

Anatomy of a Template function



Types of Templates

- **Function Template** - defines a family of functions

```
template <typename T>
void Hello(T arg);
```

- **Class Template** - defines a family of classes

```
template <typename T>
class Hello;
```

- **Variable templates** (C++ 14) – variables that are templated

```
template<class T>
constexpr T pi = T(3.14);
```

- **Alias Templates** (C++11) – alias to a family of types

```
template<class T>
using ptr = T*;
```

What are function templates?

- Represent a family of functions
- Cannot be called nor can its address be taken.
- No code generated till instantiated.
- Generic algorithm that is usable with any type that meets the algorithm constraints
- Looks like a regular function, some elements are left undetermined
- Function templates have parametrized elements that can take arguments of many different types

A Visual Analogy

Defining Function Template

- `template< class identifier> function_declaration;`
- `template <typename identifier> function_declaration;`

```
template<typename T> // typename/class introduces type parameter T
T min (T a, T b)     // a, b : template type parameters
{
    // if a < b then return a else return b
    return a < b ? a : b;
}
```

Two Phase Translation

- **Template Definition time:** checked for correctness syntax errors, non-dependent names that do not depend on template parameters, static assertions that do not depend on template parameters. Does not generate code of its own.
- **Instantiation Time:** template code is checked to ensure all code is valid (all parts that depend on template parameters). When the template is instantiated, the compiler looks up any "dependent" names, now that it has the full set of template arguments to perform lookup.

Instantiating function templates

- Instantiation: process of replacing template parameters by types.
- Different functions generated for every type the template is used for.
- Type should support operations used within a template (else compile time error)
- Type conversion does not happen during template instantiation

Instantiating function templates

- Implicit Instantiation
- Explicit Instantiation
- Address-of Instantiation
- Explicit Specialization

Instantiating function templates

- **Implicit Instantiation**

```
template <typename T>
T min(T a, T b)
{
    return a < b ?  a : b;
}

min(3, 5);                  // instantiated as int
min(4.2, 6.2);              // instantiated as float
min<double> (4.2, 6.2); // argument explicitly specified as double
min(3, 5.5f);               // ambiguous type could not be deduced. Fix?
```

Instantiating function templates

- **Explicit Instantiation**

```
template <typename T>
T min(T a, T b)
{
    return a < b ?  a : b;
}

template int min<int>(int a, int b);         // instantiate here
extern template int min<int>(int a, int b); // instantiated elsewhere
```

Instantiating function templates

- **Taking the address of a function template**

```
template <typename T>
T min(T a, T b)
{
    return a < b ?  a : b;
}
int (*pfn) (int, int) = min;
```

Instantiating function templates

- **Explicit specialization**
- Provides correct semantics for some datatype
- Or Implement algorithm optimally for a specific type
- Defined in .cpp file (violation of the ODR is specified in header)
- Primary template definition should appear before template specialization

Instantiating function templates

- **Explicit Specialization**

```
template <typename T>
T min(T a, T b)
{
    return a < b ?  a : b;
}

const char* b{ “B”}
const char* a{“A”}
auto s = min(a, b)          // returns?
```

Instantiating function templates

- **Explicit Specialization**

In .h file

```
template <typename T>
T min(T a, T b)
{
    return a < b ?  a : b;
}
```

In .cpp file

```
template<>
const char * min(const char *a, const char *b)
{
    return strcmp(a,b) < 0 ? a : b;
}
```

Instantiating function templates

- **Implicit Instantiation**

```
min(3, 5);                  // instantiated as int
min<double> (4.2, 6.2)      // instantiated as double, explicit argument
double
```

- **Explicit Instantiation**

```
template int min<int>(int a, int b);            // instantiate here
extern template int min<int>(int a, int b);     // instantiated elsewhere
```

- **Address of Instantiation**

```
int (*pfn) (int, int) = min;                    // instantiated as int
```

- **Explicit Specialization**

```
template<>
const char *
min(const char *a, const char *b);       //instantiated as const char*
```

Non-type parameters

- Templates do not have to have parameters that are types, they can have parameters that are values (non-type arguments)
- E.g. `template<unsigned N, class T>` // N is non-type parameter. Has to be known at compile time.

Default Template Arguments

- Default template arguments are specified in the parameter lists after the = sign.

```
template<typename T = std::string>
void func(T = "");

func();      // OK
func(1);     // OK: deduced T to be int, calls func<int>(1)
```

Template Parameters for Return Types

- Template argument for return type

```
template<typename T1, typename RT>
RT Truncate(T1 a);
Truncate<int64_t, char>(400);   // OK, but tedious

template<typename RT, typename T1>
RT Truncate(T1 a);
Truncate<char>(400)             //OK: return type is char, T1 is deduced

template<typename T1>
auto Truncate(T1 a);
auto retValue = Truncate(400)   //OK: return type deduced (since C++14)
```

Overloading Function Templates

- **Overloaded functions:** Different function definitions with the same function name
- **Overloaded Function Templates:** Different function template definitions with the same function name
- **Automatic type conversion** is not considered for deduced template parameters but is considered for ordinary function parameters
- Ensure that all overloaded versions of a function are declared before the function is called.

Templates & One Definition Rule (ODR)

- In the entire program, an object or non-inline function cannot have more than one definition;
- As long as each definition is the same:
  - Inline functions, can be defined in more than one translation unit.
  - In any translation unit, a template type, template function, or template object can have no more than one definition.

Questions?

Source Code Organization

- **Inclusion Model** – definition in header file
  - Simplest and most common way to make template definitions visible.
  - Every .cpp file that `#includes` the header will get its own copy of the function templates and all the definitions
  - Any namespace, classes, functions, variables that the header file includes is also available for the .cpp file that #includes header to use.
- **Separation Model** – declaration in header file and definition in cpp file
  - Solves problems listed in the inclusion model
  - Can be used in conjunction with explicit instantiation