C++ Templates: Session 2

Anupama Chandrasekhar

Jayashree Venkatesh

Instantiating function templates (Recap)

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

- **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*
```

Overloading Function Templates (Recap)

- **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.

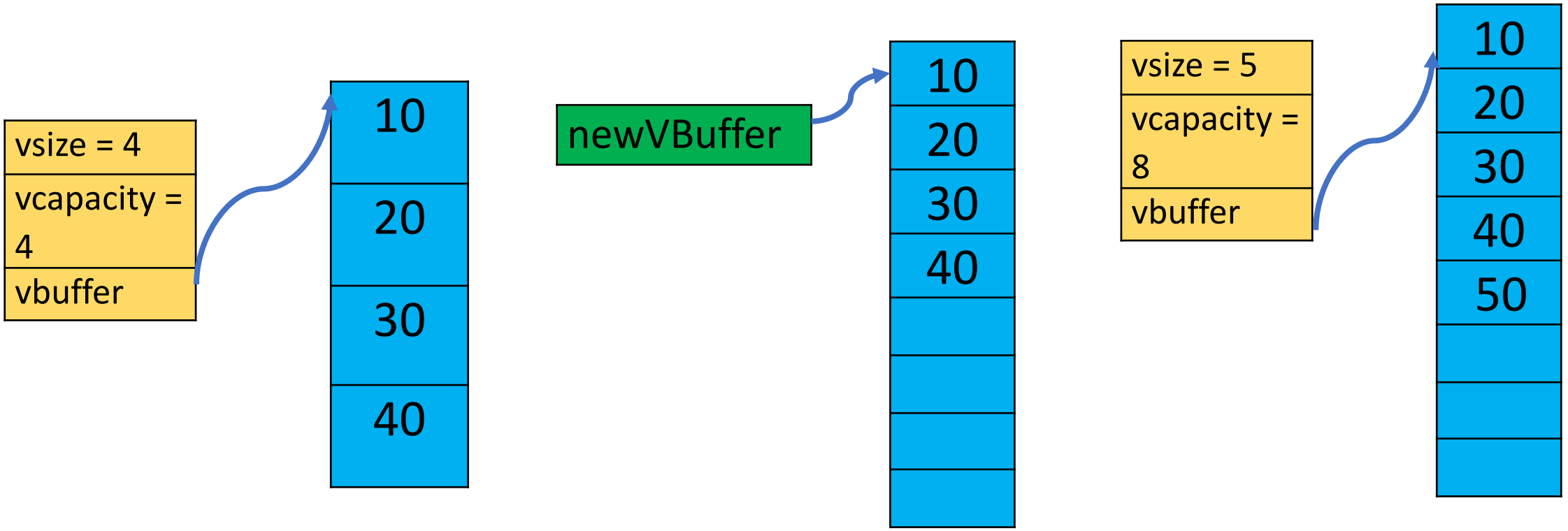
What are class templates?

- Provides specification for generating classes based on parameters.
- Not a type, object or any entity by itself. No code is generated till instantiated.
- Used to implement containers (generic classes)
- Full definition of the template should be available for separate compilation to work. (define in header file)

```
template <typename type> class class-name {

}
```

Std::Vector Class Template



Vector Class Template

```
template<typename T>
class _Vector{
    T *vbuffer;
    size_t vsize = 0;
    size_t vcapacity = 0;
public:
    void push_back(T val);
    size_t size() const;
    void pop_back();
    T back();
    T operator[](size_t index);
    bool is_empty();

    _Vector::_Iter begin() const;
    _Vector::_Iter end() const;
};
```

```
template<typename T>
class _Iter {
public:
    _Iter(const _Vector<T> *vec, size_t nIndex);
    bool operator!=(const _Iter &other) const;
    const T &operator*() const;
    _Iter &operator++();

private:
    const _Vector<T> *vIterVec;
    size_t vIterIndex = -1;
};
```

Class Templates

- Inside a class template : Template argument is in scope. Use the class name not followed by template arguments
- Outside the class declaration: To define a member function of a class template you have to specify that it is a template, and you have to use the full type qualification of the class template.

```
template<typename T> void _Vector<T>::push_back(T val) {};
```

- Unlike nontemplate classes, you can’t declare or define class templates inside functions or block scope. Templates can only be defined in global/namespace scope or inside class declarations

Instantiating class templates

- **Implicit Instantiation**

```
_Vector<float>* vec3; // nothing is instantiated here
_Vector<float> vec2;  // implicit instantiation of Vector<float>
vec3 = &vec2;
vec2.push_back(true); // instantiation of Vector<float>::push_back()
bool emptyCont =  vec3->is_empty(); // of Vector<float>::is_empty()
std::cout << std::boolalpha << emptyCont << std::endl;
```

- **Partial Usage of Class Templates**

Template arguments of Class Templates only have to provide all necessary operations that *are* needed (instead of that *could* be needed)

Instantiating Class Templates

- **Explicit Instantiation**

An explicit instantiation definition forces instantiation of the class, struct, or union they refer to.

```
template class _Vector<int>;  // explicit instantiation, all members
template void _Vector<double>::push_back(double elem); // only push_back()
```

Likewise, extern keyword prohibits instantiation of class templates.

```
extern template class _Vector<double>;
```

Class templates: Explicit Full Specialization

- Specialize a class template for all template arguments.

```
template<>
class _Vector<bool> {

}
```

- Optimize implementations for certain types or to fix a misbehavior of certain types for an instantiation of the class template.
- If you specialize a class template, you must also specialize all member functions. Although it is possible to specialize a single member function of a class template, once you have done so, you can no longer specialize the whole class template instance that the specialized member belongs to.

Class templates: Full Specialization

```
template<>
class _Vector<bool> {
    size_t *vbuffer = nullptr;
    size_t vsize = 0;
    size_t vcapacity = 0;
public:
    void push_back(bool val);
    size_t size() const;
    void pop_back();
    bool back();
    bool operator[](size_t index);
    bool is_empty();
    _Vector::_Iter begin() const;
    _Vector::_Iter end() const;
};
```

```
class _Iter {
public:
    _Iter(const _Vector<bool> *vec, size_t nIndex);
    bool operator!=(const _Iter &other) const;
    const bool operator*() const;
    _Iter &operator++();

private:
    const _Vector<bool> *vIterVec;
    size_t vIterIndex = -1;
};
```

Class templates: Partial Specialization

- Special implementations for certain types, some template parameters must still be defined by the user.

```
template<typename _Alloc>
class _Vector<bool, _Alloc> {

}
```

- Class templates might also specialize the relationship between multiple template parameters. For example, for the following class template:

```
template<typename T1, typename T2>
class SomeClass {
};
```

Class templates Partial Specialization

```
// both template parameters have same type
template<typename T>
class SomeClass<T,T> {
};
```

```
// second type is int
template<typename T>
class SomeClass<T,int> {
};
```

```
// both template parameters are pointer types
template<typename T1, typename T2>
class SomeClass<T1*,T2*> {
};
```

Class templates: Partial Specialization

```cpp
template<typename T, class _Alloc = Allocator<T>>
class _Vector {
    T *vbuffer;
    size_t vsize = 0;
    size_t vcapacity = 0;
public:
    void push_back(T val);
    size_t size() const;
    void pop_back();
    T back();
    T operator[](size_t index);
    bool is_empty();

    _Vector::_Iter begin() const;
    _Vector::_Iter end() const;
};
```

```cpp
template<typename _Alloc>
class _Vector<bool, _Alloc> {
    size_t *vbuffer = nullptr;
    size_t vsize = 0;
    size_t vcapacity = 0;
public:
    void push_back(bool val);
    size_t size() const;
    void pop_back();
    bool back();
    bool operator[](size_t index);
    bool is_empty();
    _Vector::_Iter begin() const;
    _Vector::_Iter end() const;
};
```

Default Class Template Arguments

- As for function templates, you can define default values for class template parameters. For example, in Class Vector<> you can define the container that is used to allocate/deallocate memory as a second template parameter, using std::allocator<> as the default

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.

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

Overload Resolution

- Like ordinary functions templated function can be overloaded too. They can be overloaded by by either a template function or a non template function.

```
#include <iostream>
using namespace std;

template<class T>
void f(T x, T y) { cout << "Template" << endl; }

void f(int w, int z) { cout << "Non-template" << endl; }

int main()
{
    f( 1 ,  2 );
    f('a', 'b');
    f( 1 , 'b');
}
```

Steps for overload resolution

1. Find candidate functions through name lookup
2. Filter this set to a set of *viable* functions
3. For ordinary function overload resolution, pick the function which has minimal cost of implicit conversion.
4. If overload involved templated and ordinary function, pick ordinary if no implicit conversions are required, else pick the template that is more specialized. Note that we are not talking about explicit specialization here.

```
template <class T> void f(T );  // #1
template <class T> void f(T* ); // #2

int* p;
f(p); // calls #2, more specialized
```

Explicit Specialization

- Explicit Specializations do not take part in overload resolution. Another case for using ordinary function overloads instead of explicit specialization

```
template<class T> // (a) a base template
void f( T );

template<class T> // (b) a second base template, overloads (a)
void f( T* );     //     (function templates can't be partially
                  //     specialized; they overload instead)

template<>        // (c) explicit specialization of (b)
void f<>(int*);

// ...

int *p;
f( p );           // calls (c)
```

```
template<class T> // (a) same old base template as before
void f( T );

template<>        // (c) explicit specialization, this time of (a)
void f<>(int*);

template<class T> // (b) a second base template, overloads (a)
void f( T* );

// ...

int *p;
f( p );           // calls (b)! overload resolution ignores
                  // specializations and operates on the base
                  // function templates only
```

Solution to Explicit Specialization

- Use a wrapper class

```
template<class T>
struct FImpl;

template<class T>
void f( T t ) { FImpl<T>::f( t ); } // users, don't touch this!

template<class T>
struct FImpl
{
  static void f( T t ); // users, go ahead and specialize this
};
```

Argument Dependent Lookup(ADL)

- During argument-dependent lookup, other namespaces not considered during normal lookup may be searched where the set of namespaces to be searched depends on the types of the function arguments. Specifically, set of declarations is the **union** of the declarations found by **normal lookup** with the declarations found by looking in the set of **namespaces associated with the types of the function arguments**.

Argument Dependent Lookup

```
namespace NS {

class A {};

void f(A& a, int i) {}

}  // namespace NS

int main() {
   NS::A a;
   f(a, 0);  // Calls NS::f.
}
```

- In the example, function f is called even though `main()` is not in `namespace NS` because, the argument `NS::A a` brings NS into the scope of main.

Tag Dispatch

- Problem Statement
  - How can we select between template class partial specializations or template function overloads?
- Solution 1: Tag Dispatch
  - “tag” in tag dispatch refers to a type that has no behavior and no data.
  - The point of the tag is that be creating several tags , we can use them to route execution through various function overloads

```
struct Mytag {};
```

Tag Dispatch

std::is_floating_point

Defined in header <type_traits>

```
template< class T >
struct is_floating_point;
```
(since C++11)

Checks whether T is a floating-point type. Provides the member constant value which is equal to `true`, if T is the type `float`, `double`, `long double`, including any cv-qualified variants. Otherwise, value is equal to `false`.

std::conditional

Defined in header <type_traits>

```
template< bool B, class T, class F >
struct conditional;
```
(since C++11)

Provides member typedef type, which is defined as T if B is `true` at compile time, or as F if B is `false`.

SFINAE: Substitution Failure Is Not An Error

- This rule applies during overload resolution of function templates: When substituting the explicitly specified or deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error. This feature is used in template programming.

std::enable_if

Defined in header <type_traits>

```
template< bool B, class T = void >
struct enable_if;
```

(since C++11)

If B is `true`, **`std::enable_if`** has a public member typedef type, equal to T; otherwise, there is no member typedef.

SFINAE : Example

```cpp
struct Test {
  typedef int foo;
};

template <typename T>
void f(typename T::foo) {}  // Definition #1

template <typename T>
void f(T) {}  // Definition #2

int main() {
  f<Test>(10);  // Call #1.
  f<int>(10);   // Call #2. Without error (even though there is no int::foo)
                // thanks to SFINAE.
}
```

Type Deduction Basics

- How does a template obtain the template argument “T”:
  - It may be supplied directly

```
g<float>(12);
g<const float>(12);
```

  - Infer it from the function argument, if possible **(TYPE INFERENCE)**

```
template <typename T>
void g2(size_t k = sizeof(T));
```

```
template <typename T>
void g3(remove_reference_t<T> x); //Is T &?
```

Type Deduction Basics

- Infer it from the function argument, if possible **(TYPE INFERENCE)**

```
template <typename T>
void g4(T x, T y);

g4(1, 0.1);
```

- However if a default type argument is provided, that will be inferred.

```
template <typename T = double>
void g5(T x, T y);

g5(1, 0.1);
```

Type Inference

- Type inference depends on the following:
  - Type of the function parameter, contrast it with type of the function argument.
  - Conversions are not a part of type inference
- Problem Statement: Given the function below, use the function argument and relate it to the function parameter to determine T.

```
template <typename T>
void g6(<func_parameter> x);
```

- `func_parameter` may be:
  - Passed by value, Pointer or reference type, forwarding reference

Type Inference: Case 1 (Pass by Value)

```
template <typename T>
void g6(<func_parameter> x);
```

```
template <typename T>
void g6(T x);
```

```
int k = 12;
int& kl = k;
int const k = 12;
int const & kcl = 12;
```

g(x)	T
42	int
k	int
kl	int
kc	int
kcl	int

Rules for Type Inference
Rule 1: Ignore the reference part
Rule 2: Ignore cv qualifiers (like `const`)

Type Inference: Case 2 (Reference or Pointer)

```
template <typename T>
void g6(T& x);

template <typename T>
void g6(const T& x);
```

```
template <typename T>
void g6(<func_parameter> x);
```

```
int k = 12;
int& kl = k;
int const k = 12;
int const & kcl = 12;
```

g(x)	T &	T const &
42	~~int~~	int
k	int	int
kl	int	int
kc	int const	int
kcl	int const	int

Rules for Type Inference
Rule 1: Ignore the reference part
Rule 2: Pattern match function argument with function parameter to find T

Introduction to Rvalues and Lvalues

- One of the most pervasive (and confusing) features of C++11 is *move semantics* and to understand these we need to distinguish between expressions that are lvalues and rvalues, because rvalues indicate objects that are eligible for move operation.
- Lvalues expression : yes you can take its address. **`t&`**
- Rvalues are typically temporaries and you can't take their values. **`t&&`**

Type Inference: Case 3 (Forwarding Reference)

```
template <typename T>
void g6(T&& x)
```

g(x)	T &&
42	int
k	int &
kl	int &
kc	int const &
kcl	int const &

```
template <typename T>
void g6(<func_parameter> x);
```

```
int k = 12;
int& kl = k;
int const k = 12;
int const & kcl = 12;
```

- **Rule 1:** If the expr is an lvalue, both T and func_parameter are deduced to be lvalue references. This is the only case where T is deduced to be a reference.
- **Rule 2:** If expression is an rvalue, Case 2 rules apply.

Perfect Forwarding

- We want to write a wrapper function that passes the arguments correctly:

```
template <typename T1, typename T2>
void wrapper(T1 e1, T2 e2) {
    func(e1, e2);
}
```

- If func accepts arguments by reference, the wrapper will pass by value, so the changes made by func will not be seen by the caller.

Perfect Forwarding

- If wrapper accepts arguments by reference

```
template <typename T1, typename T2>
void wrapper(T1& e1, T2& e2) {
    func(e1, e2);
}
```

- Different Problem: If func accepts parameters by value, r-values cannot be bound

```
wrapper(42, 3.14f);
```

- And making the parameters const wont help, because we might want to pass non-const parameters.

Brute Force Solution

```
void wrapper(T1& e1, T2& e2)                    { func(e1, e2); }

template <typename T1, typename T2>
void wrapper(const T1& e1, T2& e2)              { func(e1, e2); }

template <typename T1, typename T2>
void wrapper(T1& e1, const T2& e2)              { func(e1, e2); }

template <typename T1, typename T2>
void wrapper(const T1& e1, const T2& e2)        { func(e1, e2); }
```

Better Solution: Forwarding References

- Use forwarding reference

```
template <typename T1, typename T2>
void wrapper(T1&& e1, T2&& e2)
{
    func(std_forward<T1>(e1), std_forward<T2>(e2));
}
```

Back Up Slides

Type Deduction Ambiguity

- Compiler can deduce types but not your intent

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

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

int main()
{
    int a, int b;
    min(a,b);
    int* c = &a, d = &b;
    min(c, d);
}
```

```
int f(int a, int b) { return a; }

int& f(int& a, int& b){ return a; }

int main()
{
    f(10,20);
    int a = 10, b = 20;
    int& ra = a, rb = b;
    f(ra,rb);
}
```

Type Inference: Case 1 (Reference or Pointer)

```
template <typename T>
void g6(<func_parameter> x);
```

```
T, T&, T const&, T const &&, T&&
```

```
int k = 12;
int& kl = k;
int&& kr = move(k);
int const k = 12;
int const & kcl = 12;
int const && kcr = 12;
```

Forwarding Reference

g(x)	T	T &	T const &	T const &&	T &&
42	int	~~int~~	int		int
k	int	int	int		int &
kl	int	int	int	**Not Useful**	int &
kr	int	int	int		int &
kc	int	int const	int		int const &
kcl	int	int const	int		int const &
kcr	int	int const	int		int const &

Partial Ordering

- Function templates can be overloaded. So candidate functions for overload resolution can come from different function templates.
- Partial ordering is the algorithm the compiler uses to determine which function is preferred, the ordering is partial because some candidates can be considered equally specialized.
- The compiler is looking for most specialized of viable candidates.

https://docs.microsoft.com/en-us/cpp/cpp/partial-ordering-of-function-templates-cpp?view=vs-2019