

KIV/CPP – Programming in C++

07. Templates II. (variadic templates), static polymorphism, CRTP

Martin Úbl

KIV ZČU

2025/2026

Templates

- recap
- generic programming
- parameterization by type or value
- template instantiation on use (by parameterization)
- there are as many instances as there are different parameter sets

```
template<typename T>
void Print(const T& what)
{
    std::cout << what << std::endl;
}
```

```
Print(20);
```

Templates

- a parameterized class template can be inherited from
 - **Note:** the STL is not designed for inheritance (virtual destructors)
- again, either as a template or as a specialization

```
class CharVector : public std::vector<char>
{
    // ...
};
```

```
template<typename T>
class BetterVector : public std::vector<T>
{
    // ...
};
```

Templates

- but we have to define the constructor(s)

```
class CharVector : public std::vector<char>
{
    public:
        CharVector(std::initializer_list<T> li)
            : std::vector<char>(li)
        {
        }
};
```

```
// ...
```

```
CharVector cv{ 1,2,3,4 };
```

- or inherit all constructors

```
class CharVector : public std::vector<char>
{
    public:
        using std::vector<char>::vector;
};
```

- note: the constructor here is not templated, only the class is

- or inherit all constructors – more generally

```
class B : public A
{
    public:
        using A::A;
};
```

- this always inherits all of them
 - selected ones cannot be inherited individually
- this is not specific to templates, but to any inheritance

Variadic Templates

- variadic templates
- allow defining a variable number of template parameters
- “static analogy” of C `va_args`
- specialization
 - recursive
 - direct (forwarding)

```
template<typename... T>
void DoSomething(T... args)
{
    std::cout << "Doing..." << std::endl;
    Something(args...);
}
```

Variadic Templates

- recursive specialization, e.g., summation with a template

```
template<typename T1>
T1 TemplateSum(T1 fa)
{
    return fa;
}
```

```
template<typename T1, typename... T>
T1 TemplateSum(T1 fa, T... args)
{
    return fa + TemplateSum(args...);
}
```

```
TemplateSum(1,2,3,4,5);
```

Variadic Templates

- recursive specialization
- not always a good idea

```
TemplateSum(1,2,3,4,5);
```

- generates all function specializations

```
TemplateSum<int>
```

```
TemplateSum<int, int>
```

```
TemplateSum<int, int, int>
```

```
TemplateSum<int, int, int, int>
```

```
TemplateSum<int, int, int, int, int>
```

- consequence: the binary size grows, and there are 5 function calls at runtime
 - can be reduced by inlining (typically is)

Variadic Templates

- syntax with three dots is called a *parameter pack*
- represents one or more template parameters
- in the definition it “packs” parameters, in the implementation it “unpacks” them

```
template<typename... Args>  
void DoSomething(Args... args)  
{  
    SomethingElse(args...);  
}
```

- args... unpacks the parameters – args1, args2, args3,
 ...

Variadic Templates

- *fold expressions*
- allow applying a binary operator to a parameter pack when it is expanded
- e.g., a fold expression for output

```
template<typename ...Args>
void Print(Args&&... args) {
    (std::cout << ... << args) << std::endl;
}
```

- `args...` unpacks the parameters – `args1, args2, args3, ...`

Templates

- *perfect forwarding*
- in combination with an *r-value* reference, the *reference-collapsing* principle, and the `std::forward` function, we can forward any template chain further

```
class Student {  
    public:  
        Student(const std::string& name,  
                int birthyear) { ... }  
};  
  
template <typename T1, typename T2>  
Student MakeStudent(T1&& a, T2&& b) {  
    return Student(std::forward<T1>(a),  
                  std::forward<T2>(b));  
}
```

- perfect forwarding with variadic templates

```
class Student {  
    public:  
        Student(const std::string& name,  
                int birthyear) { ... }  
};  
  
template <typename... Args>  
Student MakeStudent_2(Args&&... args) {  
    return Student(std::forward<Args>(args)...);  
}
```

- perfect forwarding in practice
- `std::unique_ptr` and `std::make_unique`
- passes parameters to the constructor

```
auto ptr = std::make_unique<Something>  
          ("Something", 15);
```

- calls, for example, the constructor

```
Something(const std::string& a, int b);
```

- perfect forwarding in practice
- `emplace` for containers
- creates an instance directly in preallocated space in the container
- no copy or move is needed

```
// temporary instance + move  
container.push_back({ "Marco", 1 });
```

```
// in-place creation without move  
container.emplace_back("Polo", 2);
```

- containers based on variadics
- `std::tuple`
- `#include <tuple>`
- a container representing an n-tuple

```
std::tuple<int, std::string, double> ntice;
```

Variadic Templates

- `std::tuple`
- values are retrieved through the template `std::get<n>`, where *n* is the element index
- or by using `std::tie` – assigning the contents to a set of l-values

```
std::tuple<int, std::string, double> ntice;
```

```
int a          = std::get<0>(ntice);  
std::string b  = std::get<1>(ntice);  
double c       = std::get<2>(ntice);
```

- `std::tuple`
- `std::tie` – assigning the contents to a set of l-values

```
std::tuple<int, std::string, double> ntice;
```

```
int a;  
std::string b;  
double c;
```

```
std::tie(a,b,c) = ntice;
```

- note: `std::tie` creates a temporary tuple of l-value references

- `std::tie` – assigning the contents to a set of l-values
- `std::ignore` can be used to “discard” a value we are not interested in

```
std::tuple<int, std::string, double> ntice;
```

```
int a;
```

```
std::string b;
```

```
std::tie(a,b,std::ignore) = ntice;
```

- a somewhat nicer construct than `std::tie` – structured binding (in the following weeks)

- other uses of variadic templates
 - database drivers and value binding
 - sometimes in combination with the SFINAE idiom
 - generalization of object creation
 - a generic “pass-through” scheme, e.g., caching results
 - and more...

- static polymorphism
- polymorphism without virtual methods
- evaluated at compile time
- inheritance from a specialized template with itself as the parameter
- does not replace dynamic polymorphism, it only complements it for certain tasks
- idiom CRTP
 - *Curiously Recurring Template Pattern*

Static Polymorphism and CRTP

```
template <typename Child>
class Base {
public:
    void interface() {
        static_cast<Child*>(this)->impl();
    }
};

class Derived : public Base<Derived> {
public:
    void impl() {
        std::cout << "Derived\n";
    }
};
```

Static Polymorphism and CRTP

```
using Vec = std::vector<int>;

template<typename Child>
class Deletor {
public:
    void remove_items(Vec& cont) {
        static_cast<Child*>(this)->remove_impl(cont);
    }
};

class OddDeletor : public Deletor<OddDeletor> {
public:
    void remove_impl(Vec& cont) {
        ...
    }
};

OddDeletor deleter;
deleter.remove_items();
```

Static Polymorphism and CRTP

- static polymorphism
- the generic parent defines the interface
- the specific child defines the implementation
- repeated casts are often replaced by a small method

```
Derived& derived() {  
    return *static_cast<Derived*>(this);  
}
```

- `static_cast` instead of `dynamic_cast`
- we are sure that the cast result will be correct
 - the child inherits from the parent as a template
 - it parameterizes the template with itself

Static Polymorphism and CRTP

- *polymorphic chaining*
- a principle also known as *fluent interface*
- a method call returns a reference to itself
 - this lets us chain method calls on the object

```
class Printer {  
    public:  
        Printer& println(const std::string& ln) {  
            std::cout << ln << std::endl;  
            return *this;  
        }  
}
```

```
Printer pr;
```

```
pr.println("a").println("Hello");
```

- problem: polymorphism

Static Polymorphism and CRTP

- *polymorphic chaining* - problem

```
class PrinterIface {
public:
    virtual PrinterIface& println(const std::string& ln) = 0;
};

class ConsolePrinter : public PrinterIface {
public:
    virtual PrinterIface& println(const std::string& ln) overr
        std::cout << ln << std::endl;
        return *this;
    }

    ConsolePrinter& set_console_size(int w, int h) {
        // ...
        return *this;
    }
};

ConsolePrinter pr;

pr.println("a").println("Hello").set_console_size(10, 30);
// println returns Printer&, does not compile ^^^
```

Static Polymorphism and CRTP

- *polymorphic chaining* - solution using CRTP

```
template<typename T>
class PrinterIface {
public:
    T& println(const std::string& ln) {
        return static_cast<T*>(this)->println(ln);
    }
};

class ConsolePrinter : public PrinterIface<ConsolePrinter> {
public:
    ConsolePrinter& println(const std::string& ln) {
        std::cout << ln << std::endl;
        return *this;
    }

    ConsolePrinter& set_console_size(int w, int h) {
        // ...
        return *this;
    }
};

ConsolePrinter pr;

pr.println("a").println("Hello").set_console_size(10, 30);
```

- many other uses
- *polymorphic clone* idiom
- container filtering
- and more...

- `assert`
 - runtime assertion
 - during program debugging, it can, for example, filter out inputs for which the function is not defined
 - runtime evaluation
 - allows, for example, stopping program execution (breakpoint)
- `static_assert`
 - evaluation at compile time
 - verification of semantic rules
 - verification of size, types, object properties, ...
 - typically in combination with some standard template declaration
 - we usually try to replace it with SFINAE or concepts

- `static_assert`

```
template<typename T>
void RequiresCopy(const T& obj)
{
    static_assert(
        std::is_copy_constructible_v<T>,
        "T must be copy-constructible");

    // ...
}
```

- T can be `std::string`, `int`, ...
- T cannot be, for example, `std::unique_ptr`

- a set of standard static predicates
 - `std::is_integral`
 - `std::is_floating_point`
 - `std::is_copy_constructible`
 - `std::is_copy_assignable`
 - `std::is_class`
 - `std::is_pod`
 - `std::is_final`
 - `std::is_unsigned`
 - `std::is_base_of`
 - and more...
 - or other statically evaluable conditions
 - `sizeof(T) == ...`

- stripping
 - `std::remove_reference`
 - `std::remove_const`
 - `std::remove_volatile`
 - `std::remove_cv`
 - and more...