C++ Course 13 : Templates.

By Oleksiy Grechnyev

Comparison function

Compare two **int** numbers (returns 0, +-1) :

```
int compareInt(int a, int b) {
	if (a < b)  return -1;
	else if (b < a)  return 1;
	else  return 0;
}
```

Same for **double** numbers:

```
int compareDouble(double a, double b) {
	if (a < b)  return -1;
	else if (b < a)  return 1;
	else  return 0;
}
```

Same for **unsigned long long, float, short, char, string** ...

Lots and lots of types !

Comparison function (overloaded)

Overloading : Use the same name **compare()** for all types.

Compare two **int** numbers (returns 0, +-1) :

```
int compare(int a, int b) {
        if (a < b)  return -1;
        else if (b < a)  return 1;
        else  return 0;
}
```

Same for **double** numbers:

```
int compare(double a, double b) {
        if (a < b)  return -1;
        else if (b < a)  return 1;
        else  return 0;
}
```

Same for **unsigned long long, float, short, char, string** ...

Lots and lots of types !

Solution : Templates (Шаблоны)

```
template <typename T>
int compare(const T & a, const T & b) {
        if (a < b)
                return -1;
        else if (b < a)
                return 1;
        else
                return 0;
}
...
compare(3, 4)                   // int
compare(3, 2.5)                 // ERROR !!!
compare<double>(3, 2.5)         // double
compare(string("ABC"), string("AB"))      // string
compare<string>("ABC", "ABCD")            // string
```

Containers

```
ContainerInt ic(3);
ContainerString sc{"John", "Jim", "Jane"};
```

Lots and lots of types ! Solution : *class templates* :

```
template <typename T>
class Container{
...
private:
	T * data;
};
```

All C++ containers are templates.

Most other standard library classes are templates:

basic_string, basic_istream, shared_ptr, function, future ...

These are synonyms (**class** was used before C++ 11) :

<typename T> and **<class T>**

How do C++ templates work?

- Every time we use **compare()** in our code, it's *instantiated*
  Каждый раз, когда мы используем **compare()** в коде, происходит *инстанциация*
- *Instantiation* means creating function with a concrete type, e.g. **compare<int>()**
  Это означает создание функции конкретного типа, например **compare<int>()**
- Compiler generates different binary code for **compare<int>()** and **compare<char>()**
  Компилятор создает разный двоичный код для **compare<int>()** и **compare<char>()**
- Instantiation at *compile-time*. Инстанциация во время компиляции.
- Compiler errors are common at instantiation (type without operator **<**).
  Ошибки компиляции при инстанциации (тип без операции **<**).
- Library templates are always in .h files, never in .cpp (like class definitions) !
- This includes functions and definition of all methods!
- Local templates in a .cpp file can only be used in the same file.
- Library templates are in *.h files, not in .a or .so/.dll !
- We cannot pre-compile **compare()** before we know argument type !
  Мы не можем пре-компилировать **compare()** пока мы не знаем тип аргументов !

Once again, how does it all work?

1. The compiler reads the template definition. No code generated. Minimal syntax checks.

```
template <typename T>
int compare(const T & a, const T & b) { ... }
```

2. The compiler sees :

```
compare(3, 4)
```

3. (*Type deduction*) The type **T** is *deduced* to be **int**.

4. (*Instantiation*) The instance **compare<int>()** is generated and compiled. Compile errors can happen!

5. The code to call **compare<int>()** is generated

6. The code for **compare<int>()** is reused if used more than once (at least in the given .cpp file).

3 language tiers of C++

3 "этажа" языка в C++

4. Bonus tier: CMake, make if used.

3. Preprocessor: processes **#include** , **#ifdef** , **#define** etc.

2. Compile time: Templates, **auto** , **decltype()**, **constexpr**.
Template metaprogramming is a Turing-complete language!

1. Actual code compilation. Machine code is generated.

What happens if we use a wrong type?

```
template <typename T>
int compare(const T & a, const T & b) {
	if (a < b)
		return -1;
	else if (b < a)
		return 1;
	else
		return 0;
}
...
compare(3, 4)                    // T == int, OK, int has operator<
```

Now let us try:

compare(cout, cerr)

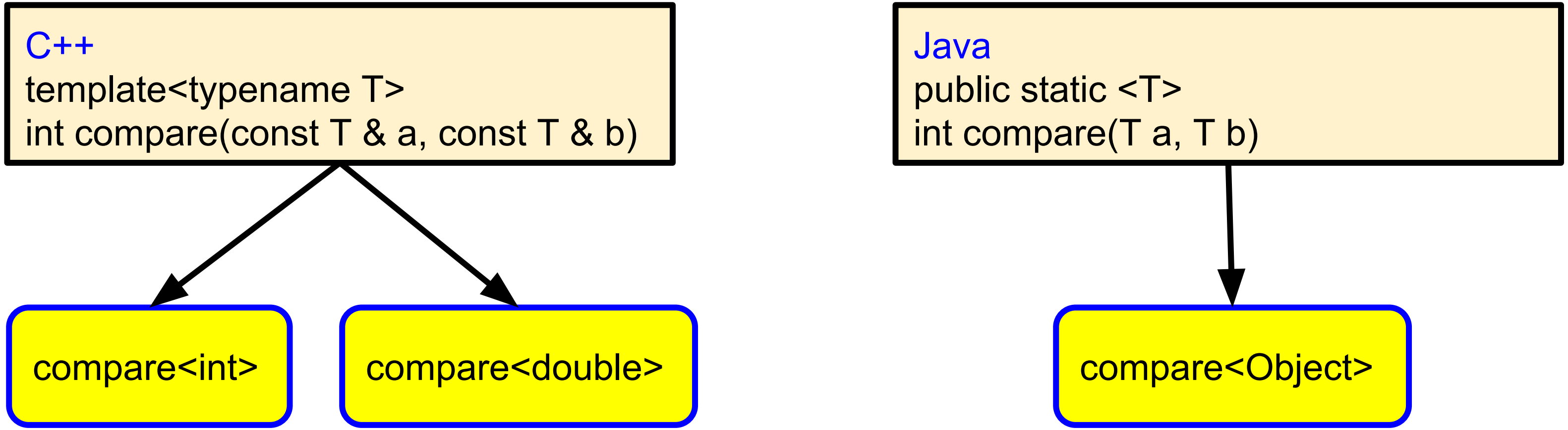
Instantiation: **T == ofstream**. Trying to compile **compare<ofstream>()** .

Compile Error ! No operator< !

C++ templates vs Java Generics

1. C++ templates are instantiated at *compile time*. Separate binary code is generated for each instance !

2. Java Generics are instantiated at *run time*. A common binary code is generated for a common parent, usually **Object**. Based on *class polymorphism* !

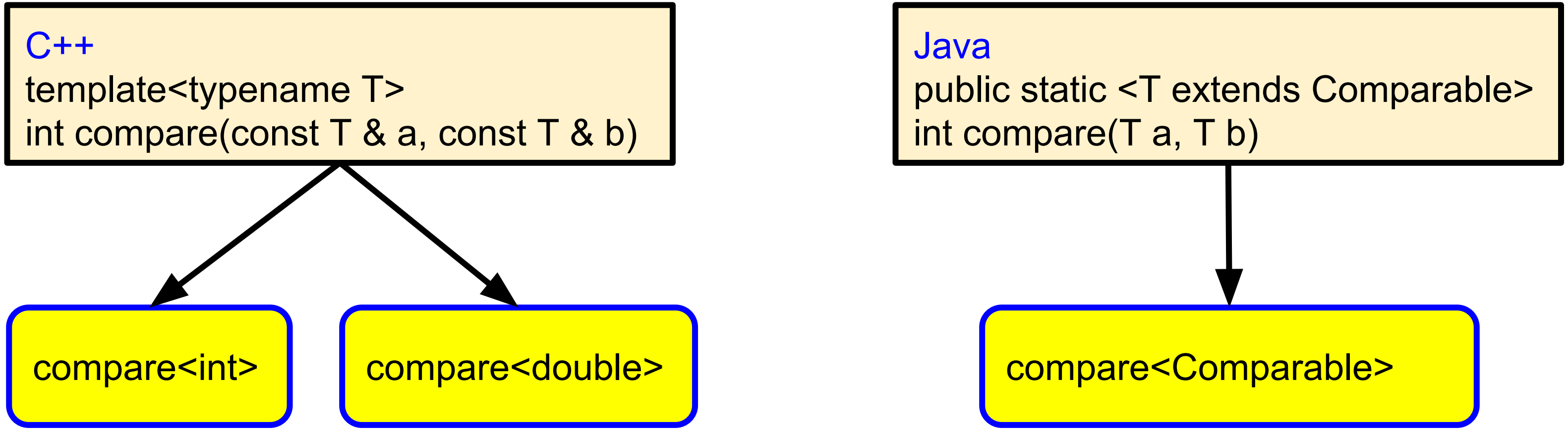


But Java **Object** cannot be compared !!!

C++ templates vs Java Generics (Comparable interface)

1. C++ templates are instantiated at *compile time*. Separate binary code is generated for each instance !

2. Java Generics are instantiated at *run time*. A common binary code is generated for a common parent, usually **Object**, but here **Comparable** (an interface).



With **Comparable** everything should work.

C++ templates vs Java Generics (table)

C++	Java
Instantiated at compile time	Instantiated at run time
Functions, classes, methods	Classes, methods
Templates in .h files, not precompiled	Generics are precompiled
No class polymorphism	Based on a common parent (or Object)
Behavior can depend on type	Behavior is identical for all types
Can be used with primitive types	Only for class types
No simple way to limit type	Limit the type: extends, super, ?
Rich template metaprogramming	No metaprogramming
Turing complete language	No Turing complete language

Template specialization

```
template <typename T>
int compare(const T & a, const T & b) {
     if (a < b) return -1;
     else if (b < a) return 1;
     else return 0;
}
```

Suppose we want a different behavior for **float** :

```
template <>
int compare(const float & a, const float & b) {
     if (a < b) return -7;
     else if (b < a) return 7;
     else return 0;
}
```

Note : this is different from an overload

int compare(const float & a, const float & b) {...}

The two approaches are different in details.

Non-type arguments: Multiply by N

Multiply something by a compile-time **int** number.

```
template <int N, typename T>  // N is first, T can be deduced
T mul(const T & t) {
    return t*N;
}
```

N is an **int** argument (compile-time number).

T is a regular type argument.

N must be a literal or **constexpr**.

```
cout << "mul<3>(2.1) = " << mul<3>(2.1) << endl;                    // double, deduced
cout << "mul<3, float>(2.1) = " << mul<3, float>(2.1) << endl;    // float

int i = 17;
cout << mul<i, int>(2) << endl;                 // Error, i is not constexpr !
```

Container operations : range for

Print a container (or array) using range **for** :

```
template <typename C>
void print1(const C & c){
    for (const auto & e : c)
        cout << e << " ";
    cout << endl;
}
```

The same without **auto** (does not work with built-in arrays) :

```
template <typename C>
void print2(const C & c){
    for (const typename C::value_type & e : c)
        cout << e << " ";
    cout << endl;
}
```

What on Earth is **typename** ???

typename keyword, the real reason for it

Suppose **C** is a type parameter of a template.
We know absolutely nothing of it before instantiation (**int** ?, **vector<string>** ?, **ofstream** ?).
What is **C::value_type** ? There are 2 options:

1. It can be a *type* defined in the class **C**.
C::value_type * b;   // Pointer definition

2. It can be a *static member* of the class **C**.
C::value_type * b;   // Multiplication

The compiler always assumes option 2 (static member) by default.

Use **typename** (NOT **class**) for option 1:
typename C::value_type * b;

Secondary use **typename** as a synonym to **class** in template arguments.

Container operations : iterators

Print a container (or array) using iterators :

```
template <typename C>
void print3(const C & c){
    for (auto it = cbegin(c); it != cend(c); ++it)
        cout << *it << " ";
    cout << endl;
}
```

Possible implementation of **std::find()** :

```
template <typename I, typename T>
I myFind(const I & begin, const I & end, const T & val){
    for (I it = begin; it != end; ++it)
        if (val == *it)
            return it;
    return end;
}
```

In this example we return the type **I**.

But what if we don't know the return type ?

Find first negative number in a container

Stupid version : return type **T** :

cout << findNeg<int>(vi.cbegin(), vi.cend()) << endl;

```
template <typename R, typename I>
const R & findNeg1(const I & begin, const I & end){
    for (I it = begin; it != end; ++it)
    if (*it < 0)
        return *it;
    throw runtime_error("NOT FOUND !");
}
```

Clever version : use **decltype** :

```
template <typename I>
auto findNeg1(const I & begin, const I & end) -> const decltype(*end) & {
    for (I it = begin; it != end; ++it)
    if (*it < 0)
        return *it;
    throw runtime_error("NOT FOUND !");
}
```

Templates and built-in arrays

Size of a built-in array (NOT pointer !) compile time :

```
template <typename T, size_t N>
constexpr size_t arraySize( T (&a)[N]){
    return N;
}
```

Possible implementation of **begin()**, **end()** for an array :

```
template <typename T, size_t N>
T * myBegin(T (&a)[N]){
    return & a[0];
}

template <typename T, size_t N>
T * myEnd(T (&a)[N]){
    return & a[N];
}
```

Class templates : MyArray example

MyArray : my own implementation of **std::array** (more or less) :

```
template <typename T, size_t N>
class MyArray{
..
public:
     T dat[N];
}
```

MyArray : is a thin wrapper to a fixed-size built-in array.
It can be copied and moved with automatically generated ctors.
I has no explicit ctors.
The data is in the field **dat** (not separately on heap like **std::vector** does !)
dat is public, so we can do the list initialization :

```
MyArray<string, 6> names{"Maria", "Nel", "Sophia", "Mirage", "Peppita",
"Clair"};
```

Defining types

MyArray, like standard C++ containers, defines a number of types

```
template <typename T, size_t N>
class MyArray{
public: //=== Type definitions
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using reference = T &;
    using const_reference = const T &;
    using pointer = T *;
    using const_pointer = const T *;
    using iterator = T *;
    using const_iterator = const T *;
    ...
}
```

operator[], at()

```
template <typename T, size_t N>
class MyArray{
     ...
public:      //===== Methods
     T & operator[] (std::size_t i){ return dat[i];}
     const T & operator[] (std::size_t i) const { return dat[i];}
     T & at(std::size_t i){
          checkRange(i);
          return dat[i];
     }
     const T & at(std::size_t i) const {
          checkRange(i);
          return dat[i];
     }
private:      //===== Methods
     void checkRange(std::size_t i) {
          if (i >= N) throw std::out_of_range("MyArray");
     }
     ...
}
```

friends, non-member operator==, one-to-one friendship

```
template <typename T, size_t N>  // Forward declaration
class MyArray;
template <typename T1, size_t N1>
bool operator==(const MyArray<T1, N1> & lhs, const MyArray<T1, N1> & rhs);

template <typename T, size_t N>
class MyArray{
    friend bool operator==<T, N>(const MyArray<T, N> & lhs,
     const MyArray<T, N> & rhs);
}

template <typename T1, size_t N1>
bool operator==(const MyArray<T1, N1> & lhs, const MyArray<T1, N1> & rhs) {
    for (size_t i = 0; i < N1; ++i)
        if (lhs.dat[i] != rhs.dat[i])
            return false;
    return true;
}
```

Template method in a template class

assign() : assign the array from a pair of iterators (NOT in **std::array**)

It's a template method of template class with its own template parameter **I**

It is defined outside class

```
template <typename T, size_t N>
class MyArray{
public:      //===== Methods
    template <typename I>
    void assign(I begin, I end);
}

template<typename T, size_t N>
template<typename I>
void MyArray<T, N>::assign(I begin, I end)
{
    int i = 0;
    for (I it = begin; it != end; ++it, ++i)
        at(i) = *it;
}
```

Iterators

```
template <typename T, size_t N>
class MyArray{
    ...
public:      //===== Iterators are pointers
    T* begin(){  return dat;}

    const T* begin() const {  return dat;}

    const T* cbegin() const {return dat;}

    T* end(){ return & dat[N];}

    const T* end() const { return & dat[N];}

    const T* cend() const { return & dat[N];}
    ...
}
```

Class template specialization

Class template specializations:

A single method.
The entire class (**std::vector<bool>**).
Partial specialization: a possible implementation of **std::remove_reference** :
(From Lippman, C++ Primer)

```
// original, most general template (used for non-references)
template <class T> struct remove_reference {
        typedef T type;
};
// partial specializations that will be used for lvalue and rvalue references
template <class T> struct remove_reference <T&>  // lvalue references
{ typedef T type; };
template <class T> struct remove_reference <T&&> // rvalue references
{ typedef T type; };
```

Variadic templates

Variadic functions are functions with variable number of arguments.
C variadic function (runtime) : **printf("%s = %d\n", name, value);**
C variadic functions DO NOT WORK with C++ classes and references !

C++ variadic templates, ellipsis (**...**) is part of the C++ syntax!
The list of all arguments is called *variadic pack*.
Example: return the pack size (number of arguments):

```
size_t packSize(const Args & ... pack){
        return sizeof...(pack);        // or
//      return sizeof...(Args);        // Same result
}
```

Arguments can be of *different types*.
For arguments of one type, use **std::initializer_list** or **std::vector**.
Args : types of arguments.
Ellipsis (**...**) is called *pack expansion*.

Example: print all arguments to cout

Non-variadic **print()** with 1 argument to break the recursion.

```
template <typename T>
void print(const T & t){
    cout << t << endl;
}
```

Variadic **print()** with recursion.

```
template <typename T, typename... Args>
void print(const T & t, const Args & ... rest){
    cout << t << endl;
    print(rest...);
}
```

A simple C++ stream implementation of printf()

Use C-string **snprintf()** to write a fixed-size buffer.

```
template <typename... Params>
void printfCPP(std::ostream & os, const std::string & fmt, Params... p) {
     constexpr size_t SIZE = 1000;
     static char buffer[SIZE];  // Hidden global buffer = ugly
     std::snprintf(buffer, SIZE, fmt.c_str(), p...);
     os << buffer;
}
```

Usage example:

printfCPP(cout, "%s : %d * %d = %d\n", "Hello", 3, 7, 3*7);

Note : this function cannot print any classes or **std::string** !

There are better versions, like **print()** in Boost.

Advanced template (meta)programming:

Compile-time factorial (from Wikipedia):

```
// Induction
template <int N>
struct Factorial {
  static const int value = N * Factorial<N - 1>::value;
};
// Base case via template specialization:
template <>
struct Factorial<0> {
  static const int value = 1;
};
```

Factorial is a class template with a single *static* field **value**.

This is a standard trick for *template metaprogramming*.

Usage:

```
cout << "Factorial<5>::value = " << Factorial<5>::value << endl;
```

Type traits

is_pointer
remove_pointer
add_pointer
is_reference
remove_reference
add_rvalue_reference
is_class
is_integral
is_base_of
min
...

For example:

is_pointer<int*>::value is **true** (of type **bool**)
is_pointer<int*>() is an object of **true_type** (compile-time version of **bool**)
true_type::value is **true** (of type **bool**)

is_pointer() example

Print either value or pointer : Naive approach (WRONG !) :

```
template<typename T>
void katana(const T & val){
	if (is_pointer<T>::value)
		cout << "Pointer : " << val << " , *val = "<< *val << endl;
	else
		cout << "Value : " << val << endl;
}
```

Problem: ***val** would not compile for a non-pointer !

is_pointer() example 2. Correct code.

```
template<typename  T>
void katanaImpl(const T & val, true_type){    // Pointer
     cout << "Pointer : " << val << " , *val = "<< *val << endl;
}

template<typename  T>
void katanaImpl(const T & val, false_type){    // Not Pointer
     cout << "Value : " << val << endl;
}

template<typename  T>                           // Print value or pointer
void katana(const T & val){
     katanaImpl(val, is_pointer<T>());   // true_type or false_type
}
```

is_pointer<T>() returns an object of either **true_type** or **true_type**.

One of the two overloaded templates **katanaImpl()** is selected.

This is called *tag dispatch* (second argument = tag).

Thank you for your attention !

title

text