10 - C++ Templates

It is the parametric polymorphism explicit bounded (invariant) of C++.

A template is a C++ entity that defines one of the following:

- A family of functions (function template);
- A family of classes (class template);
- An alias to a family of types (alias template);
- A family of variables (variable template);
- A concept (constraints and concepts);

Templates are parameterized by >= 1 template parameters, of 3 kinds:

- Type template parameters;
- Non-type template parameters;
- Template template parameters.

When template arguments are provided/deduced, they are substituted for the template parameters to obtain a specialization of the template.

When a specialization is referenced in some context, the template is instantiated (the code for it is actually compiled). In this process, monomorphization happens: for each unique instantiation, their respective specializations are replaced by monomorphic entities, producing copies of the generic code. If a template is never instantiated, the specific code is never generated. It is a concept opposed to type erasure in Java.

Function Templates in C++

Example:

```language-cpp
// T is a class variable, then T can be used as a type variable
template <class T>
T sqr { return x * x; }
```

Parameter types are inferred or indicated explicitly (necessary in case of ambiguity):

```language-cpp
int a = 3;
double b = 3.14;

// generates: int sqr(int x){ return x * x; }
// The type is inferred
sqr(a);

// generates: double sqr(double x){ return x * x; }
// The type is explicit
double bb = sqr(b);
```

Works for user-defined types as well:

```language-cpp
class Complex{
public:
        double real; double imag;
        Complex(double r, double im): real(r), imag(im){};
```

```
        //overloading of *
        Complex operator*(Complex y){
                return Complex(
                        real * y.real - imag * y.imag,
                        real * y.imag + imag * y.real
                );
        }
};
// ------------------------------------------------------- //

{ // ...
        Complex c(2, 2);
        Complex cc = sqr(c);
        cout << cc.real << " " << c.imag << endl;
  // ...
}
```

Non-Type Template Parameter

It is a template parameter where the type of the parameter is predefined and is substituted for a constant value passed in as an argument.

Example:

```language-cpp
// 2 template parameters: type variable T
// and primitive type int N (non-type parameter)
template <Class T, int N>
T fixed_multiply(T val){
        return val * N;
}

int main(){
        //Class T: <int, N>
        //int N: (10)
        std::cout << fixed_multiply<int, 2>(10) << '\n'; // 20
}
```

Partial/Full template specialization

They allows customizing class and variable templates for a given category of template arguments. *As intuition: it is similar to overriding (without the concepts of inheritance).*

They work by defining a new template over an existing one with:

- Same name;
- More specific parameters (partial specialization);
- No parameters (full specialization);

Using it, we can use better implementation for specific kinds of types.

Examples:

```language-cpp
template<class T1, class T2, int I>
class A {};              // primary template

template<class T, int I>
class A<T, T*, I> {};    // #1: partial specialization where T2 is a pointer to T1
```

```
template<class T>
class A<int, T*, 5> {};  // #2: partial specialization where
                         //     T1 is int, I is 5, and T2 is a pointer

template<>
class A<int, int, 5> {}; // #3: Full specialization
```

Template Metaprogramming

Templates can be used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled.

- Only constant expressions: they have to be computable at compile time.
- No mutable variables: same reason;
- no support by IDE's, compilers and other tools.

Computing at Compile Time

C++ function that compute sum of first n integers

```language-cpp
#include <iostream>

int triangular(int n){
        return (n == 1) ? 1 : triangular(n - 1) + n;
}

int main(){
        int result = triangular(20);
        std::cout << result << '\n';
}
```

C++ template with specialization computing sum of first n integers

```language-cpp
#include <iostream>

template <int t>
constexpr int triangular(){
        return triangular<t - 1>() + t;
}
//base case
template <>
constexpr int triangular<1>(){
        return 1;
}

int main(){
        int result = triangular<20>();
        std::cout << result << 'n';
}
```

- `constexpr` invites the compiler to evaluate the expression

With the template specialization in the compiled code we actually find the result of `triangular<20>()`, which is 210.
Since templates generates code, when we execute the first snippet we have a lot of code generation (the compiler generates the code for `tirangular(20)`, `triangular(19)`, ...).
Instead with the second snippet we have a long compilation time (the compiler has to compute the triangular number of 20) but the size of the compiled code do not explode.

Templates vs Macros

Macros can be used for polymorphism in simple cases:

```language-cpp
#define SQR(T) T SQR(T x) { return x * x; }
SQR(int); // int sqr(int x) { return x * x; }
SQR(double); // double sqr(double x) { return x + x; }
```

Remember that macros are executed by the pre-processor, templates by the compiler. Pre-processor makes only (possibly parametric) textual substitutions: no parsing or static analysis checks are performed.

Compared to templates, side-effects are replicated onto every substitution performed. For example:

```language-cpp
#define sqr(x) ((x) * (x))
int a = 2;
int aa = sqr(a++);
```

In this example `sqr(a++)` is expanded as `(a++) * (a++)`:

- The first `a++` returns 2 and then `a` is incremented to 3;
- The second `a++` is evaluated as 3 and then `a` is incremented to 4.
- With a function we would have that `aa` is equal to 4, while with macro expansion we have that it is equal to 6. Side effects are duplicated with macros. For the same reason now `a` is 4 while it should be 3.

Finally, compared to templates recursion is not possible with macros:

```language-cpp
#define fact(n) (n == 0 ? 1 : fact(n - 1) * n)
```

The compilation fails because `fact` is not defined: `fact(3-1)` is an undefined function since the macro expansion is not recursive.

SIDE NOTE: Macros expansion is visible when compiling with option `-E`.