C++ Software Engineering

for engineers of other disciplines

Module 3

"C++ Templates"

1st Lecture: **`std::`**

ALTEN

Spring 2022

Gothenburg, Sweden

std::

- C++ Standard Template Library (STL) is a collection of headers providing *"basic necessary"* functionalities.
- The library is mostly template based.
- The library is implemented in **std** namespace.
- Each new C++ version, expands the library.

Dynamic memory management

`<new>`	Low-level memory management utilities
`<memory>`	High-level memory management utilities
`<scoped_allocator>` (C++11)	Nested allocator class
`<memory_resource>` (C++17)	Polymorphic allocators and memory resources

Numeric limits

`<climits>`	Limits of integral types
`<cfloat>`	Limits of floating-point types
`<cstdint>` (C++11)	Fixed-width integer types and limits of other types
`<cinttypes>` (C++11)	Formatting macros, `intmax_t` and `uintmax_t` math and conversions
`<limits>`	Uniform way to query properties of arithmetic types

Error handling

`<exception>`	Exception handling utilities
`<stdexcept>`	Standard exception objects
`<cassert>`	Conditionally compiled macro that compares its argument to zero
`<system_error>` (C++11)	Defines `std::error_code`, a platform-dependent error code
`<cerrno>`	Macro containing the last error number

Utilities library

`<cstdlib>`	General purpose utilities: program control, dynamic memory allocation, random numbers, sort and search
`<csignal>`	Functions and macro constants for signal management
`<csetjmp>`	Macro (and function) that saves (and jumps) to an execution context
`<cstdarg>`	Handling of variable length argument lists
`<typeinfo>`	Runtime type information utilities
`<typeindex>` (C++11)	`std::type_index`
`<type_traits>` (C++11)	Compile-time type information
`<bitset>`	`std::bitset` class template

Input/output library

`<iosfwd>`	Forward declarations of all classes in the input/output library
`<ios>`	`std::ios_base` class, `std::basic_ios` class template and several typedefs
`<istream>`	`std::basic_istream` class template and several typedefs
`<ostream>`	`std::basic_ostream`, `std::basic_iostream` class templates and several typedefs
`<iostream>`	Several standard stream objects
`<fstream>`	`std::basic_fstream`, `std::basic_ifstream`, `std::basic_ofstream` class templates and several typedefs
`<sstream>`	`std::basic_stringstream`, `std::basic_istringstream`, `std::basic_ostringstream` class templates and several typedefs
`<syncstream>` (C++20)	`std::basic_osyncstream`, `std::basic_syncbuf`, and typedefs
`<strstream>` (deprecated in C++98)	`std::strstream`, `std::istrstream`, `std::ostrstream`
`<iomanip>`	Helper functions to control the format of input and output
`<streambuf>`	`std::basic_streambuf` class template
`<cstdio>`	C-style input-output functions

- List of all the headers in STL could be found here: https://en.cppreference.com/w/cpp/header
- **`<iostream>`** is one of many headers of STL's Input/Output library, providing console output through **`cout`**.

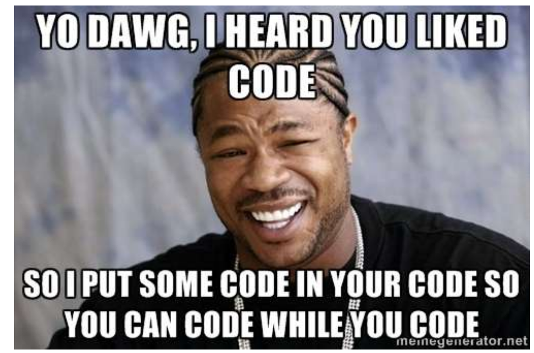
template<>

- In C++ **`template`** **<>** keyword is used for both Metaprograming and Generic Programing.

- *Programming: Writing a program that creates, transforms, filters, aggregates and otherwise manipulates data.*

- *Metaprogramming: Writing a program that creates, transforms, filters, aggregates and otherwise manipulates programs.*

- *Generic Programming: Writing a program that creates, transforms, filters, aggregates and otherwise manipulates data, but makes only the minimum assumptions about the structure of the data, thus maximizing reuse across a wide range of datatypes.* https://stackoverflow.com/a/3937852



- In C++, *Metaprogramming* could happen both at compile time and run time, while *Generic Programing* is a compile time procedure.

Generic Programming

- C++ offers generic programming with template parameters

```
template <GenericParameterList> Declaration
```

- template provides developers with the opportunity of implementing algorithms or defining objects in classes independent of data types.
- Compilers, depending on whether the function is used and what actual data types has substituted the generic parameters, will generate the necessary code, this procedure is called instantiation of a template and has nothing to do with objects. Each instantiation is called a specializing of that template for that specific type.
- Although, autogenerated code by the compiler raises some concerns, yet templates are very popular specially for implementing libraries, frameworks, and/or SDKs. It is a favorite choice for huge code bases as well due to the reusability and flexibility it provides. Specialization of a template could happen explicitly by the developer as well, to avoid code generation by the compiler.

"Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters." https://en.wikipedia.org/wiki/Generic_programming

Templates

- There are two main types of templates:
  - Function templates:
    - Declared as below:
      ```
      template <class GenericType,…> FunctionSignature;
      template <typename GenericType,…> FunctionSignature;
      ```
    - Invoked using the actual (concrete) type(s):
      ```
      FunctionName <ActualTypes> (InputParameters);
      ```
  - Class templates:
    - Declared as below:
      ```
      template <class GenericType,…> ClassDeclaration;
      template <typename GenericType,…> ClassDeclaration;
      ```
    - Instantiated using the actual (concrete) type(s):
      ```
      ClassName <ActualTypes> VariableName(InputParameters);
      ```

- There are other uses of templates which are out of the scope of this course. Those are: Variable templates, Variadic template, and template aliases.
- Both `class` and `typename` provide the exact same functionalities in template-parameter. There are some special scenarios where these keywords have specific usages, such as dependent types: https://stackoverflow.com/questions/2023977/difference-of-keywords-typename-and-class-in-templates

Function Templates

- Function templates:
  - Declared as below:

    ```
    template <class GenericType,…>
    FunctionSignature;
    template <typename GenericType,…>
    FunctionSignature;
    ```

  - Invoked using the actual (concrete) type(s):

    ```
    FunctionName <ActualTypes> (InputParameters);
    ```

```cpp
template <typename T1, class T2>
bool isBigger (T1 const& a, T2 const& b) {
    return a > b ? true : false;
}
```

- The syntax for conditional ternary operator is as follows `condition ? Result1 : Result2;`
- Conditional Ternary Operator evaluates a `condition` and returns `Result1` if the condition holds; otherwise, it returns `Result2`.

Function Templates

- Function templates:
  - Declared as below:
    ```
    template <class GenericType,…>
    FunctionSignature;
    template <typename GenericType,…>
    FunctionSignature;
    ```
  - Invoked using the actual (concrete) type(s):
    ```
    FunctionName <ActualTypes> (InputParameters);
    ```

```
template <typename T1, class T2>
bool isBigger (T1 const& a, T2 const& b) {
    return a > b ? true : false;
}
```

- Compiler will try to deduce the types automatically wherever needed, as good as it could.

```
int main() {
    std::cout << std::boolalpha;
    std::cout << isBigger<float, int>(100.22,100) << std::endl;
    std::cout << isBigger<int>(100.22,100) << std::endl;
    std::cout << isBigger<int,char>(100.22,'a') << std::endl;
    std::cout << isBigger("HelloWord","GoodByeWorld") << std::endl;
    std::cout << isBigger<char[10],char[10]>("abcdeasdf","aasdbcdef") << std::endl;
    std::cout << isBigger<std::string,std::string>("abcdeasdf","absdbcdef") << std::endl;
    std::cout << isBigger(100,"GoodByeWorld") << std::endl;
    return 0;
}
```

```
true
false
true
true
true
false
```

Function Templates

- Function templates:
  - Declared as below:

```
template <class GenericType,…>
FunctionSignature;
template <typename GenericType,…>
FunctionSignature;
```

  - Invoked using the actual (concrete) type(s):

```
FunctionName <ActualTypes> (InputParameters);
```

```
template <typename T1, class T2>
bool isBigger (T1 const& a, T2 const& b) {
    return a > b ? true : false;
}
```

- If the type does not provide the implantation specified in the template function, the compiler will generate compilation error and terminates.

```
int main() {
    std::cout << std::boolalpha;
    std::cout << isBigger<float, int>(100.22,100) << std::endl;
    std::cout << isBigger<int>(100.22,100) << std::endl;
test.cpp: In instantiation of 'bool isBigger(const T1&, const T2&) [with T1 = int; T2 = char [13]]':
test.cpp:79:45:   required from here
test.cpp:68:13: error: ISO C++ forbids comparison between pointer and integer [-fpermissive]
   68 |     return a > b ? true : false;
      |            ~~^~~
    std::cout << isBigger(100,"GoodByeWorld") << std::endl;
    return 0;
}
```

Class Templates

- Class templates:
  - Declared as below:
    ```
    template <class GenericType,…> ClassDeclaration;
    template <typename GenericType,…> ClassDeclaration;
    ```
  - Instantiated using the actual (concrete) type(s):
    ```
    ClassName <ActualType> VariableName(InputParameters);
    ```

> - Fundamental datatypes could be used as template parameters as well; the actual value of the fundamental datatypes should be provided upon declaring the template type (class).
> - Type deduction cannot happen for classes and the actual types should be explicitly defined. In some very rare cases, the constructor of the class could receive the type of the template argument and then some compiler could deduce the types.

```
template <typename T, size_t SIZE>
class Container {
    public:
        bool add (const T &_element, size_t _i) {
            if (_i > SIZE) return false;
            else {
                Data[_i] = _element;
                return true;
            }
        }
        T fetch(size_t _i) {
            T ret;
            if (_i < SIZE) ret = Data[_i];
            return ret;
        }
        ~Container() {
            delete [] Data;
        }
    private:
        T *Data = new T[SIZE];

};
```

Class Templates

- Class templates:
  - Declared as below:
    `template <class GenericType,…> ClassDeclaration;`
    `template <typename GenericType,…> ClassDeclaration;`
  - Instantiated using the actual (concrete) type(s):
    `ClassName <ActualType> VariableName(InputParameters);`

```
template <typename T, size_t SIZE>
class Container {
    public:
        bool add (const T &_element, size_t _i) {
            if (_i > SIZE) return false;
            else {
                Data[_i] = _element;
                return true;
            }
        }
        T fetch(size_t _i) {
            T ret;
            if (_i < SIZE) ret = Data[_i];
            return ret;
        }
        ~Container() {
            delete [] Data;
        }
    private:
        T *Data = new T[SIZE];

};
```

```
int main() {
    std::cout << std::boolalpha;
    Container<int,10> intContaier_10;
    for (size_t i = 0; i < 10; i++)
        intContaier_10.add(i,(i+3)*100);
    std::cout << intContaier_10.add(100,100) << ndl;
    std::cout << intContaier_10.fetch(2) << std
    std::cout << intContaier_10.fetch(20) << st ;
    std::cout << intContaier_10.add("TEXT",1) << std::endl;
    return 0;
}
```

```
false
500
32652
```

Class Templates

- Class templates:
  - Declared as below:

    ```
    template <class GenericType,…>
    ClassDeclaration;
    template <typename GenericType,…>
    ClassDeclaration;
    ```

  - Instantiated using the actual (concrete) type(s):

    ```
    ClassName <ActualType>
    VariableName(InputParameters);
    ```

```
int main() {
    std::cout << std::boolalpha;
    Container<int,10> intContaier_10;
    for (size_t i = 0; i < 10; i++)
        intContaier_10.add(i,(i+3)*100);
    std::cout << intContaier_10.add(100,100) << ndl;
    std::cout << intContaier_10.fetch(2) << std
    std::cout << intContaier_10.fetch(20) << st ;
    std::cout << intContaier_10.add("TEXT",1) << std::endl;
    return 0;
}
```

```
false
500
0
```

```
template <typename T, size_t SIZE>
class Container {
    public:
        bool add (const T &_element, size_t _i) {
            if (_i > SIZE) return false;
            else {
                Data[_i] = _element;
                return true;
            }
        }
        T fetch(size_t _i) {
            T ret{};
            if (_i < SIZE) ret = Data[_i];
            return ret;
        }
        ~Container() {
            delete [] Data;
        }
    private:
        T *Data = new T[SIZE];

};
```

- Uniform initialization ensures invocation of the constructor.

Class Templates

- Class templates:
  - Declared as below:
    ```
    template <class GenericType,…>
    ClassDeclaration;
    template <typename GenericType,…>
    ClassDeclaration;
    ```
  - Instantiated using the actual (concrete) type(s):
    ```
    ClassName <ActualType>
    VariableName(InputParameters);
    ```

```cpp
int main() {
    std::cout << std::boolalpha;
    Container<int,10> intContaier_10;
    std::cout << intContaier_10.add("TEXT",1) << std::endl;
    return 0;
}
```

```
test.cpp: In function ‘int main()’:
test.cpp:102:37: error: invalid conversion from ‘const char*’ to ‘int’ [-fpermissive]
  102 |     std::cout << intContaier_10.add("TEXT",1) << std::endl;
      |                                     ^~~~~~
      |                                     |
      |                                     const char*
test.cpp:75:28: note:   initializing argument 1 of ‘bool Container<T, SIZE>::add(const T&, size_t) [with T = int; long unsigned int SIZE = 10; size_t = long unsigned int]’
   75 |         bool add (const T &_element, size_t _i) {
      |                   ~~~~~~~~~^~~~~~~~
```

```cpp
template <typename T, size_t SIZE>
class Container {
    public:
        bool add (const T &_element, size_t _i) {
            if (_i > SIZE) return false;
            else {
                Data[_i] = _element;
                return true;
            }
        }
        T fetch(size_t _i) {
            T ret{};
            if (_i < SIZE) ret = Data[_i];
            return ret;
        }
        ~Container() {
            delete [] Data;
        }
    private:
        T *Data = new T[SIZE];
};
```

DEMO!

Function Template


```cpp
template <typename T>
void sum(const T &_a, const T &_b) {
    T c = _a + _b;
    std::cout << c << std::endl;
}
struct A {
    A() = default;
    A(const int &_a, const int &_b):a(_a),b(_b){}
    int a,b;
};
A operator +(const A  &_o, const A  &_f) {
    return A(_f.a + _o.a, _f.b + _o.b);
}
std::ostream& operator<<(std::ostream &_os,const A &_m) {
    return _os << _m.a << " " << _m.b;
}
```

```cpp
struct C {
    int a;
    std::string s;
    C():a(-2),s("Initialized!"){}
};

template<>
void sum<C>(const C &_a, const C &_b) {
    std::cout << "Nothing to see here!" << std::endl;
}

int main() {
    sum<>(2.02,1.89);
    sum<int> (12,12);
    sum<std::string> ("Hello ", "World!");
    sum<A>(A(2,3),A(1,4));
    sum<C>(C(),C());
    return 0;
}
```

Class Template

ALTEN

```cpp
template <typename T, size_t SIZE = 2>
class Container {
    public:
        bool add (const T &_element, size_t _i) {
            bool ret = false;
            if (_i >= SIZE);
            else {
                Data[_i] = _element;
                ret = true;
            }
            return ret;
        }
        T fetch(size_t _i) {
            T ret{};
            if (_i < SIZE) ret = Data[_i];
            return ret;
        }
        ~Container() {delete [] Data;}

    private:
        T *Data = new T[SIZE];

};
```

```cpp
if (_i > SIZE) return false;
else {
```

```cpp
template <>
class Container<char> {
    public:
        bool add (const char * _e) {
            Data+=_e;
            return true;
        }
        bool add (const char &_e) {
            Data+=_e;
            return true;
        }
        const char *fetch(size_t _from, size_t _to) {
            std::string ret = "";
            if (_from > Data.length() || _to > Data.length() || _from > _to);
            else {
                ret = Data.substr(_from,_to);
            }
            return ret.c_str();
        }
        char fetch(size_t _i) {
            char ret = 0x00;
            if (_i < Data.length()) ret = Data[_i];
            return ret;
        }
    private:
        std::string Data;

};
```

```cpp
Container<int>   intPair;
Container<char>  charContainer;
Container<int,5> intContainer;

charContainer.add("Hello World!");

for (size_t i = 0; intPair.add( i+((i+1)*10), i); i++) {
```

std::vector


```cpp
int main() {
    std::vector< std::vector<int> > v;
    std::vector<int> a,b = {-5,-4,-3,-2,-1,0,1};

    a.insert(a.begin(),b.cbegin(),b.cbegin()+4);
    b.pop_back();

    v.push_back(a);
    v.push_back(b);

    v[0].push_back(11);
    v[1][2] = 13;

    for (std::vector<int> e: v) {
        std::cout << ">>> ";
        for (auto i = e.cbegin(); i < e.cend();i ++)
            std::cout << *i << " ";
        std::cout << std::endl;
    }

    return 0;
}
```

placeholder type specifiers (since C++11)

For variables, specifies that the type of the variable that is being declared will be automatically deduced from its initializer.

For functions, specifies that the return type will be deduced from its return statements. (since C++14)

For non-type template parameters, specifies that the type will be deduced from the argument. (since C++17)

https://en.cppreference.com/w/cpp/language/auto

```cpp
v.push_back(a);
v.push_back(b);

for (std::vector<int
    std::cout << ">>
    for (auto i = e.cbegin(); i < e.cend();i ++)
```

```
std::vector<int>::const_iterator std::vector<int>::cbegin() const
```

Returns a read-only (constant) iterator that points to the first element in the %vector. Iteration is done in ordinary element order.

Assignment 1

- Write a function template which receives two of any *shapes* from last week's assignment, compares them, and prints out the result.