**Template**

Template is used for Generic code

Defined using template<placeholder>

Type is known only when the code is used

Code is instantiated for each specific type

Type binding still applies

You can also use template to compute at compile time

# Function Templates

A function taking a generic type as parameter

We can explicitly specify the type of template parameter when calling the template code

Calling template function without explicitly specifying the template type is known as instantiating the function template

The operations which are used in the template code should have been defined for the type which is instantiating template code

**Example**

template<typename T>

constexpr T get\_max(T val1, T val2) {

return val1 > val2 ? val1 : val2;

}

# Templates in Header Files

Templates are usually defined in header file

* Not only declared
* No inline necessary

When switching to modules we could also put it in modules

## Example – Function Template

// header.hpp

#ifndef HEADER\_HPP

#define HEADER\_HPP

template<typename T>

constexpr T get\_max(T val1, T val2) {

return val1 > val2 ? val1 : val2;

}

template<typename T>

T only\_declare(T val);

#endif

// main.cpp

#include "header.hpp"

#include <iostream>

#include <string>

int main() {

//only\_declare(5); // this will generate linking error

static\_assert(get\_max(5, 3) == 5);

static\_assert(get\_max(5, 5) == 5);

std::cout << "get\_max(5, 3): " << get\_max(5, 3) << '\n';

std::cout << "get\_max(5, 5): " << get\_max(5, 5) << '\n';

static\_assert(get\_max(2.3f, 3.9f) == 3.9f);

static\_assert(get\_max(3.9f, 3.9f) == 3.9f);

std::cout << "get\_max(2.3f, 3.9f): " << get\_max(2.3f, 3.9f) << '\n';

std::cout << "get\_max(3.9f, 3.9f): " << get\_max(3.9f, 3.9f) << '\n';

static\_assert(get\_max(5.8, 5.9) == 5.9);

static\_assert(get\_max(5.8, 5.8) == 5.8);

std::cout << "get\_max(5.8, 5.9): " << get\_max(5.8, 5.9) << '\n';

std::cout << "get\_max(5.8, 5.8): " << get\_max(5.8, 5.8) << '\n';

static\_assert(get\_max(true, false) == true); // ?

std::cout << std::boolalpha << "get\_max(true, false): " << get\_max(true, false) << '\n';

// CE error: non-constant condition for static assertion

//static\_assert(get\_max("Hi", "hi") == "Hi");

std::cout << R"(get\_max("Hi", "hi"): )" << get\_max("Hi", "hi") << '\n';

// explicitly providing temaplte type

std::cout << R"(get\_max<std::string>("Hi", "hi"): )" << get\_max<std::string>("Hi", "hi") << '\n';

// CE error: non-constant condition for static assertion

//static\_assert(get\_max(std::string("Hi"), std::string("hi")) == std::string("Hi"));

std::cout << R"(get\_max(std::string("Hi"), std::string("hi")): )" << get\_max(std::string("Hi"), std::string("hi")) << '\n';

return 0;

}

Output::

get\_max(5, 3): 5

get\_max(5, 5): 5

get\_max(2.3f, 3.9f): 3.9

get\_max(3.9f, 3.9f): 3.9

get\_max(5.8, 5.9): 5.9

get\_max(5.8, 5.8): 5.8

get\_max(true, false): true

get\_max("Hi", "hi"): Hi

get\_max<std::string>("Hi", "hi"): hi

get\_max(std::string("Hi"), std::string("hi")): hi

# Function Template Requirements

Template requires that the type supports all operations

It may also require copy and move operations

#include <iostream>

#include <string>

template<typename T>

T get\_max(T a, T b) {

return (a > b) ? a : b;

}

template<typename T>

T get\_val(T a) {

return a;

}

struct CopyableMoveable { int val; };

struct NonCopyableMoveable {

int val;

NonCopyableMoveable(const NonCopyableMoveable &) = delete;

};

int main() {

CopyableMoveable c1{3};

CopyableMoveable c2{8};

// CE error: no match for 'operator>' (operand types are 'CopyableMoveable' and 'CopyableMoveable')

auto res1 = get\_max(c1, c2);

NonCopyableMoveable nc1{13};

// CE error: use of deleted function 'NonCopyableMoveable::NonCopyableMoveable(const NonCopyableMoveable&)'

auto res2 = get\_val(nc1);

return 0;

}

Compilation Error::

<source>: In function 'int main()':

<source>:30:24: error: use of deleted function 'NonCopyableMoveable::NonCopyableMoveable(const NonCopyableMoveable&)'

30 | auto res2 = get\_val(nc1);

| ~~~~~~~^~~~~

<source>:17:5: note: declared here

17 | NonCopyableMoveable(const NonCopyableMoveable &) = delete;

| ^~~~~~~~~~~~~~~~~~~

<source>:10:13: note: initializing argument 1 of 'T get\_val(T) [with T = NonCopyableMoveable]'

10 | T get\_val(T a) {

| ~~^

<source>: In instantiation of 'T get\_max(T, T) [with T = CopyableMoveable]':

<source>:26:24: required from here

<source>:6:15: error: no match for 'operator>' (operand types are 'CopyableMoveable' and 'CopyableMoveable')

6 | return (a > b) ? a : b;

| ~~~^~~~

Compiler returned: 1

# Multiple Template Parameter

We can have multiple different template parameter

We can pass different type of parameter to this type of template code

We can explicitly specify one or multiple parameter

#include <iostream>

template<typename T1, typename T2, typename T3>

void display(T1 t1\_val, T2 t2\_val, T3 t3\_val) {

std::cout << t1\_val << ", " << t2\_val << ", " << t3\_val << '\n';

}

int main() {

display(1, 2, 3);

display(1, 2.2, "char pointer data");

display(1, 2.2, std::string("sample string"));

display<std::string>("sample string", 2.5f, true);

display<std::string, float>("sample string", 2.5, 3);

display<std::string, float, double>("sample string", 2.5, 3);

// CE error: parse error in template argument list

//display<, , std::string>(2.5f, true, "can not skip a type");

return 0;

}

Output::

1, 2, 3

1, 2.2, char pointer data

1, 2.2, sample string

sample string, 2.5, 1

sample string, 2.5, 3

sample string, 2.5, 3

# auto Return Type in Case of Multiple Parameter (since C++14)

Consider following code

template<typename T1, typename T2>

<return\_type ???> get\_max(T1 t1\_val, T2 t2\_val) {

return (t1\_val > t2\_val) ? t1\_val : t2\_val;

}

is called in following two ways

<return\_type ???> res1 = get\_max(53, 45.68); // 1 int

<return\_type ???> res1 = get\_max(75.89, 45); // 2 double

What should be its return type?

* for 1 it should be int
* for 2 it should be double

But there is no way we could do this using template type

**Use 'auto'**

In this scenario we can use 'auto' as return type

When using 'auto' compiler will decide which type to return and will return appropriate type

It may do implicit cast to another type if necessary

#include <iostream>

template<typename T1, typename T2>

auto get\_max(T1 t1\_val, T2 t2\_val) {

return (t1\_val > t2\_val) ? t1\_val : t2\_val;

}

int main() {

std::cout << "get\_max(11, 9.3): " << get\_max(11, 9.3) << '\n';

std::cout << "get\_max(8, 9.3): " << get\_max(8, 9.3) << '\n';

std::cout << "get\_max(11.3, 9): " << get\_max(11.3, 9) << '\n';

std::cout << "get\_max(8.5, 12): " << get\_max(8.5, 12) << '\n';

return 0;

}

Output::

get\_max(11, 9.3): 11

get\_max(8, 9.3): 9.3

get\_max(11.3, 9): 11.3

get\_max(8.5, 12): 12

# auto parameters (since C++20)

Since C++20 we can use 'auto' for generic parameters

This is still same thing as template

We do not have the type info directly now

We need to detect type using parameter

You may even explicitly specify the parameter type like template

// header.hpp

#ifndef HEADER\_HPP

#define HEADER\_HPP

constexpr auto get\_max(auto val1, auto val2) {

return val1 > val2 ? val1 : val2;

}

auto only\_declare(auto val);

#endif

// main.cpp

#include "header.hpp"

#include <iostream>

#include <string>

int main() {

//only\_declare(5); // this will now result in compilation error

static\_assert(get\_max(5, 3) == 5);

static\_assert(get\_max(5, 5) == 5);

std::cout << "get\_max(5, 3): " << get\_max(5, 3) << '\n';

std::cout << "get\_max(5, 5): " << get\_max(5, 5) << '\n';

static\_assert(get\_max(2.3f, 3.9f) == 3.9f);

static\_assert(get\_max(3.9f, 3.9f) == 3.9f);

std::cout << "get\_max(2.3f, 3.9f): " << get\_max(2.3f, 3.9f) << '\n';

std::cout << "get\_max(3.9f, 3.9f): " << get\_max(3.9f, 3.9f) << '\n';

static\_assert(get\_max(5.8, 5.9) == 5.9);

static\_assert(get\_max(5.8, 5.8) == 5.8);

std::cout << "get\_max(5.8, 5.9): " << get\_max(5.8, 5.9) << '\n';

std::cout << "get\_max(5.8, 5.8): " << get\_max(5.8, 5.8) << '\n';

static\_assert(get\_max(true, false) == true); // ?

std::cout << std::boolalpha << "get\_max(true, false): " << get\_max(true, false) << '\n';

// CE error: non-constant condition for static assertion

//static\_assert(get\_max("Hi", "hi") == "Hi");

std::cout << R"(get\_max("Hi", "hi"): )" << get\_max("Hi", "hi") << '\n';

// explicitly providing temaplte type

std::cout << R"(get\_max<std::string>("Hi", "hi"): )" << get\_max<std::string>("Hi", "hi") << '\n';

// CE error: non-constant condition for static assertion

//static\_assert(get\_max(std::string("Hi"), std::string("hi")) == std::string("Hi"));

std::cout << R"(get\_max(std::string("Hi"), std::string("hi")): )" << get\_max(std::string("Hi"), std::string("hi")) << '\n';

return 0;

}

Output::

get\_max(5, 3): 5

get\_max(5, 5): 5

get\_max(2.3f, 3.9f): 3.9

get\_max(3.9f, 3.9f): 3.9

get\_max(5.8, 5.9): 5.9

get\_max(5.8, 5.8): 5.8

get\_max(true, false): true

get\_max("Hi", "hi"): Hi

get\_max<std::string>("Hi", "hi"): hi

get\_max(std::string("Hi"), std::string("hi")): hi

# Concepts - To Constraints Template Parameter (since C++20)

To formulate formal constraints for generic code

Instead of detecting the error inside template code function body we can detect it early in the template type itself

To disable bad behaviour

Use can write their own concepts

#include <iostream>

struct CopyableMoveable { int val; };

struct NonCopyableMoveable {

int val;

NonCopyableMoveable(int val) : val(val) { }

NonCopyableMoveable(const NonCopyableMoveable &) = delete;

};

template<typename T>

concept support\_greater\_than = requires(T val) { val > val; };

template<typename T>

requires std::copyable<T> && support\_greater\_than<T>

auto get\_max(T t1\_val, T t2\_val) {

return (t1\_val > t2\_val) ? t1\_val : t2\_val;

}

int main() {

std::cout << "get\_max(11, 9): " << get\_max(11, 9) << '\n';

CopyableMoveable c1{8}, c2{9};

// CE because concept support\_greater\_than fails ... the required expression '(val > val)' is invalid

// std::cout << "get\_max(c1, c2): " << get\_max(c1, c2) << '\n';

NonCopyableMoveable nc1{3}, nc2{13};

// CE because concept std::copyable fails

// ... the expression 'is\_constructible\_v<\_Tp, \_Args ...> [with \_Tp = NonCopyableMoveable; \_Args = {NonCopyableMoveable}]' evaluated to 'false'

std::cout << "get\_max(nc1, nc2): " << get\_max(nc1, nc2) << '\n';

return 0;

}

# Class Templates

Class code for generic types

## Implementation of Class Template

### Inside class

We can define template class methods inside the class like inline functions

#include <iostream>

#include <vector>

#include <stdexcept>

template<typename T>

class my\_stack {

std::vector<T> m\_elems;

public:

void push(const T & elem) {

m\_elems.push\_back(elem);

}

T pop() {

if(empty()) {

throw std::runtime\_error("Stack is empty");

}

T val = m\_elems.back();

m\_elems.pop\_back();

return val;

}

bool empty() const {

return m\_elems.empty();

}

};

int main() {

my\_stack<int> st;

st.push(1);

st.push(2);

std::cout << "st.pop() " << st.pop() << '\n';

st.push(3);

std::cout << "st.pop() " << st.pop() << '\n';

std::cout << "st.pop() " << st.pop() << '\n';

std::cout << std::boolalpha << "st.empty() " << st.empty() << '\n';

try {

std::cout << "st.pop() " << st.pop() << '\n';

} catch (std::runtime\_error & exp) {

std::cout << "Run time error: " << exp.what() << '\n';

}

return 0;

}

Ouptut:

st.pop() 2

st.pop() 3

st.pop() 1

st.empty() true

st.pop() Run time error: Stack is empty

### Outside class

We could also define class template methods outside the class

#include <iostream>

#include <vector>

#include <stdexcept>

template<typename T>

class my\_stack {

std::vector<T> m\_elems;

public:

void push(const T & elem);

T pop();

bool empty() const;

};

template<typename T>

void my\_stack<T>::push(const T & elem) {

m\_elems.push\_back(elem);

}

template<typename T>

T my\_stack<T>::pop() {

if(empty()) {

throw std::runtime\_error("Stack is empty");

}

T val = m\_elems.back();

m\_elems.pop\_back();

return val;

}

template<typename T>

bool my\_stack<T>::empty() const {

return m\_elems.empty();

}

int main() {

my\_stack<int> st;

st.push(1);

st.push(2);

std::cout << "st.pop() " << st.pop() << '\n';

st.push(3);

std::cout << "st.pop() " << st.pop() << '\n';

std::cout << "st.pop() " << st.pop() << '\n';

std::cout << std::boolalpha << "st.empty() " << st.empty() << '\n';

try {

std::cout << "st.pop() " << st.pop() << '\n';

} catch (std::runtime\_error & exp) {

std::cout << "Run time error: " << exp.what() << '\n';

}

return 0;

}

Output:

st.pop() 2

st.pop() 3

st.pop() 1

st.empty() true

st.pop() Run time error: Stack is empty

# Generic Member Functions

Generic member functions are only instantiated if used

**Class Template Arguments**

Have to support all operations of member templates that are used

Don’t have to support all the operations that could be used

#include <iostream>

#include <vector>

#include <stdexcept>

template<typename T>

class my\_stack {

std::vector<T> m\_elems;

public:

void push(const T & elem) {

m\_elems.push\_back(elem);

}

T pop() {

if(empty()) {

throw std::runtime\_error("Stack is empty");

}

T val = m\_elems.back();

m\_elems.pop\_back();

return val;

}

bool empty() const {

return m\_elems.empty();

}

void print() {

std::cout << "Stack elements ";

for(const T & elem : m\_elems) {

std::cout << elem << ' ';

}

std::cout << '\n';

}

};

struct S { int m\_val; };

int main() {

my\_stack<int> sti;

sti.push(1);

sti.push(2);

std::cout << "sti.pop() " << sti.pop() << '\n';

sti.push(3);

sti.print();

std::cout << "sti.pop() " << sti.pop() << '\n';

std::cout << "sti.pop() " << sti.pop() << '\n';

std::cout << std::boolalpha << "sti.empty() " << sti.empty() << '\n';

try {

std::cout << "sti.pop() " << sti.pop() << '\n';

} catch (std::runtime\_error & exp) {

std::cout << "Run time error: " << exp.what() << '\n';

}

std::cout << '\n';

S s1{1}, s2{2}, s3{3};

my\_stack<S> sts;

sts.push(s1);

sts.push(s2);

sts.push(s3);

std::cout << std::boolalpha << "sts.empty() " << sts.empty() << '\n';

S sval = sts.pop();

std::cout << "sval.m\_val: " << sval.m\_val << '\n';

//sts.print();

return 0;

}

Output:

sti.pop() 2

Stack elements 1 3

sti.pop() 3

sti.pop() 1

sti.empty() true

sti.pop() Run time error: Stack is empty

sts.empty() false

sval.m\_val: 3

Check this for more <https://cppinsights.io/s/f7c2c076>

Or compile this code on <https://cppinsights.io/>

# Class Template Argument Deduction (CTAD) (since C++17)

No need to specify template parameter explicitly for a class template if we can find it out from constructor

Do not use CTAD unless it is obvious

#include <iostream>

#include <complex>

#include <vector>

template<typename T>

void display\_coll(const T & coll) {

for(const auto & elem: coll) {

std::cout << elem << ' ';

}

std::cout << '\n';

}

int main() {

{ std::complex<int> cplx(2, 3); }

{ std::complex<int> cplx{2, 3}; }

{ std::complex<int> cplx(2, 3);

std::complex<int> cplx2 = cplx; }

{ std::complex<int> cplx = 2; }

{ std::vector<int> vec{1, 2}; }

// CTAD

{ std::complex cplx(2, 3); } // deduces std::complex<int>

{ std::complex cplx{2, 3}; } // deduces std::complex<int>

{ std::complex cplx(2, 3); // deduces std::complex<int>

std::complex cplx2 = cplx; } // deduces std::complex<int>

{ std::complex cplx = 2; }

{ std::vector vec{1, 2}; } // deduces std::vector<int>

// Do not use CTAD unless it is obvious

{

std::vector<int> vec{1, 2, 3, 4, 5};

display\_coll(vec);

std::vector vec2(vec.begin(), vec.end()); // deduces std::vector<int>

display\_coll(vec2);

std::vector vec3{vec.begin(), vec.end()}; // deduces std::vector<std::vector<int>::iterator>

//display\_coll(vec3);

std::vector<int> vec4{vec.begin(), vec.end()}; // deduces std::vector<int>

display\_coll(vec4);

}

}

Output:

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

# Non-Type Template Parameter (NTTP) Types

**Supported types:**

* Types for constant integral values (int, long, enum, …)
* Std::nullptr\_t (the type of nullptr)
* Pointers to globally visible objects/functions/members
* Lvalue references to objects or functions

**Not supported are:**

* String literals
* Classes

**Since C++20 supported are:**

* Floating-point types (float, double, …)
* Data structure with public members
* Lambdas

#include <iostream>

struct S {

int member;

};

template<S st>

auto get\_member() {

return st.member;

}

int main() {

constexpr S s1{13};

std::cout << get\_member<s1>() << '\n';

return 0;

}

Output:

13

# Variadic Templates

Templates for a variable number of template arguments

* Type-safe varargs interface

For functions and classes

Names parameter pack

* Represent multiple arguments (types/objects)
* Can be passed together to somewhere else

## Handling no argument

**In template code, all the code needs to be valid at compile time even if they are not called at run time**

In instantiation of variadic template, the function call will once be called with no argument

To handle this

1. Use a function with no argument

Provide definition of a function with no argument

1. Use ‘if constexpr’ and ‘sizeof…()’

Use ‘if constexpr’ and ‘sizeof…()’ and call function with variadic args only if its size is greater than 0

## Use a function with no argument

#include <iostream>

void display() { }

template<typename T, typename ... Tv>

void display(T val, Tv ... vals) {

std::cout << val << ", ";

display(vals ...);

}

int main() {

display(1, 2.5, 'A', "const char \*");

return 0;

}

Output:

1, 2.5, A, const char \*,

## Use ‘if constexpr’ and ‘sizeof…()’

#include <iostream>

template<typename T, typename ... Tv>

void display(T val, Tv ... vals) {

std::cout << val << ", ";

if constexpr(sizeof...(vals) > 0)

display(vals ...);

}

int main() {

display(1, 2.5, 'A', "const char \*");

return 0;

}

Output:

1, 2.5, A, const char \*,

# Concepts as Type Constraints

We can use concepts as type constraints to constraint the template type

## Concept and template type parameter

#include <set>

#include <vector>

template <typename Coll>

concept has\_push\_back =

requires(Coll coll, Coll::value\_type val) { coll.push\_back(val); };

template <typename Coll, typename T>

requires has\_push\_back<Coll>

void add(Coll coll, T val) {

coll.push\_back(val);

}

template <typename Coll, typename T>

void add(Coll coll, T val) {

coll.insert(val);

}

int main() {

std::vector vec{1, 2, 3, 4, 5};

add(vec, 6);

std::set st{11, 12, 13, 14, 15};

add(st, 16);

return 0;

}

## Concept and auto parameter

#include <set>

#include <vector>

template <typename Coll>

concept has\_push\_back =

requires(Coll coll, Coll::value\_type val) { coll.push\_back(val); };

void add(has\_push\_back auto & coll, const auto & val) {

coll.push\_back(val);

}

void add(auto & coll, const auto & val) {

coll.insert(val);

}

int main() {

std::vector vec{1, 2, 3, 4, 5};

add(vec, 6);

std::set st{11, 12, 13, 14, 15};

add(st, 16);

return 0;

}

# requires and Compile-Time if

requirement and concept are boolean expression

so they can be used in if-constexpr for conditional compilation at compile time

#include <set>

#include <vector>

template <typename Coll, typename T>

void add(Coll coll, T val) {

if constexpr ( requires { coll.push\_back(val); } ) {

coll.push\_back(val);

}

else if constexpr ( requires { coll.push\_back(val); } ) {

coll.insert(val);

}

}

int main() {

std::vector vec{1, 2, 3, 4, 5};

add(vec, 6);

std::set st{11, 12, 13, 14, 15};

add(st, 16);

return 0;

}

#include <set>

#include <vector>

void add(auto & coll, const auto & val) {

if constexpr ( requires { coll.push\_back(val); }) {

coll.push\_back(val);

}

else if constexpr ( requires { coll.insert(val); }) {

coll.insert(val);

}

}

int main() {

std::vector vec{1, 2, 3, 4, 5};

add(vec, 6);

std::set st{11, 12, 13, 14, 15};

add(st, 16);

return 0;

}

# References

[Back to Basics: Templates in C++ - Nicolai Josuttis - CppCon 2022](https://www.youtube.com/watch?v=HqsEHG0QJXU)