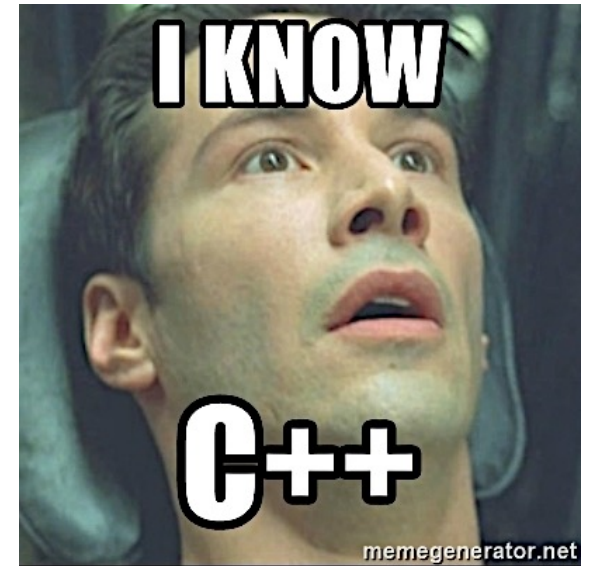
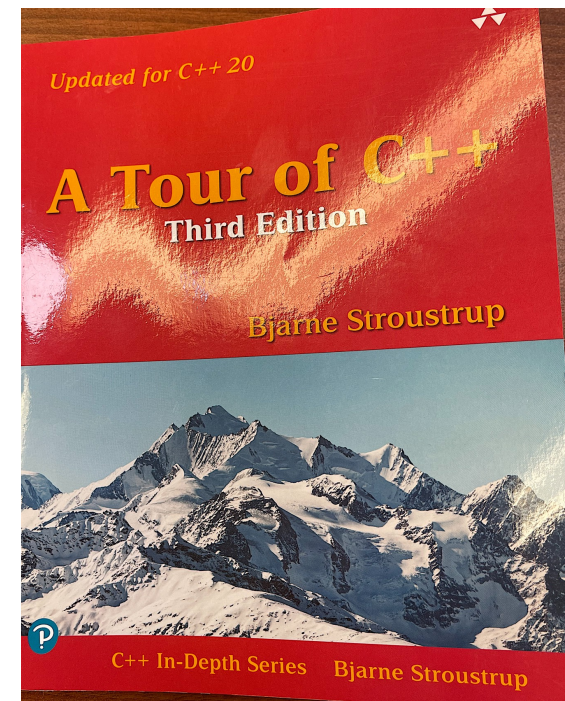


Lecture 26: C++ part XI



Suggested Literature



RVO (Return Value Optimization)

Return Value Optimization

This is such a cool feature!

When a function returns an unnamed object, C++17 standard requires the compiler to avoid the creation of a temporary object (**copy elision**)

```
#include <iostream>

using namespace std;

class Base {
    int x_;
public:
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    return Base(i);
}

int main () {
    Base a = GetBase(50); // no call to
                        // copy constructor
}
```

```
whovian@phy504:~/host/CPP$ g++ ref10.cpp -o ref10 -std=c++20
whovian@phy504:~/host/CPP$ ./ref10
Default Constructor
Destructor
```

```

#include <iostream>

using namespace std;

class Base {
    int x_;
public:
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    return Base(i);
}

int main () {
    Base a = GetBase(50); // no call to
                        // copy constructor
}

```

Return Value Optimization

RVO is NOT using the moving constructor

RVO happens even if there is no moving constructor (in this code, there isn't one because the copy constructor is used defined)

RVO: compiler optimization

```

whovian@phy504:~/host/CPP$ g++ ref10.cpp -o ref10 -std=c++20
whovian@phy504:~/host/CPP$ ./ref10
Default Constructor
Destructor

```

```

#include <iostream>

using namespace std;

class Base {
    int x_;
public:
    Base(int i);
    ~Base();
    Base(const Base& other) = delete;
    Base& operator=(Base other) = delete;
    Base(Base&& other) = delete;
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::~~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    return Base(i);
}

int main () {
    Base a = GetBase(50); // no call to
                        // copy constructor
}

```

Return Value Optimization

RVO is NOT using the
copying constructor
(RVO != copy / move)

Even with deleted copying
and move constructors, RVO
still happens

RVO: compiler optimization

```

whovian@phy504:~/host/Phy504Docker/CPP$ g++ ref17.cpp -o ref17 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./ref17
Default Constructor
Destructor

```

```
#include <iostream>
```

```
using namespace std;
```

```
class Base {
```

```
    int x_;
```

```
public:
```

```
    Base(int i);
```

```
    ~Base();
```

```
    Base(const Base& other);
```

```
};
```

```
Base::Base(int i) :
```

```
    x_(i) {
```

```
        cout << "Default Constructor" << endl;
```

```
}
```

```
Base::Base(const Base& other) {
```

```
    cout << "Copy Constructor" << endl;
```

```
    this->x_ = other.x_;
```

```
}
```

```
Base::~~Base() {
```

```
    cout << "Destructor" << endl;
```

```
}
```

```
Base GetBase(int i) {
```

```
    return Base(i);
```

```
}
```

```
int main () {
```

```
    Base a = GetBase(50); // no ca
```

```
                        // copy
```

```
}
```

```
whovian@phy504:~/host/CPP$ g++ ref10.cpp -o ref10 -std=c++14 -fno-elide-constructors
```

```
whovian@phy504:~/host/CPP$ ./ref10
```

```
Default Constructor
```

```
Copy Constructor
```

```
Destructor
```

```
Copy Constructor
```

```
Destructor
```

```
Destructor
```

Return Value Optimization

In C++<17 **RVO** could have
been disabled by the
compiler (nice, but not a
must feature)

Not possible to disable **RVO**
in C++17 and beyond

```

#include <iostream>

using namespace std;

class Base {
public:
    int x_;
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    Base a(i);
    cout << a.x_ << endl;
    return a;
}

int main () {
    Base b = GetBase(50);
}

```

```

whovian@phy504:~/host/CPP$ g++ ref11.cpp -o ref11 -std=c++20
whovian@phy504:~/host/CPP$ ./ref11
Default Constructor
50
Destructor

```

Return Value Optimization

In **GetBase()**, the var is named **RVO** works on unnamed values

Wouldn't it be nice if the compiler could apply RVO even with named variables?

It did! Somehow, the entire code happened without a copy!

RVO vs NRVO

This is named **NRVO** (C++17):
Named Return Value
Optimizations

RVOs are mandatory to all compilers by the standards and cannot be turned off via compiler flags. **NRVOs** are suggested and can be turned off

```
#include <iostream>

using namespace std;

class Base {
public:
    int x_;
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

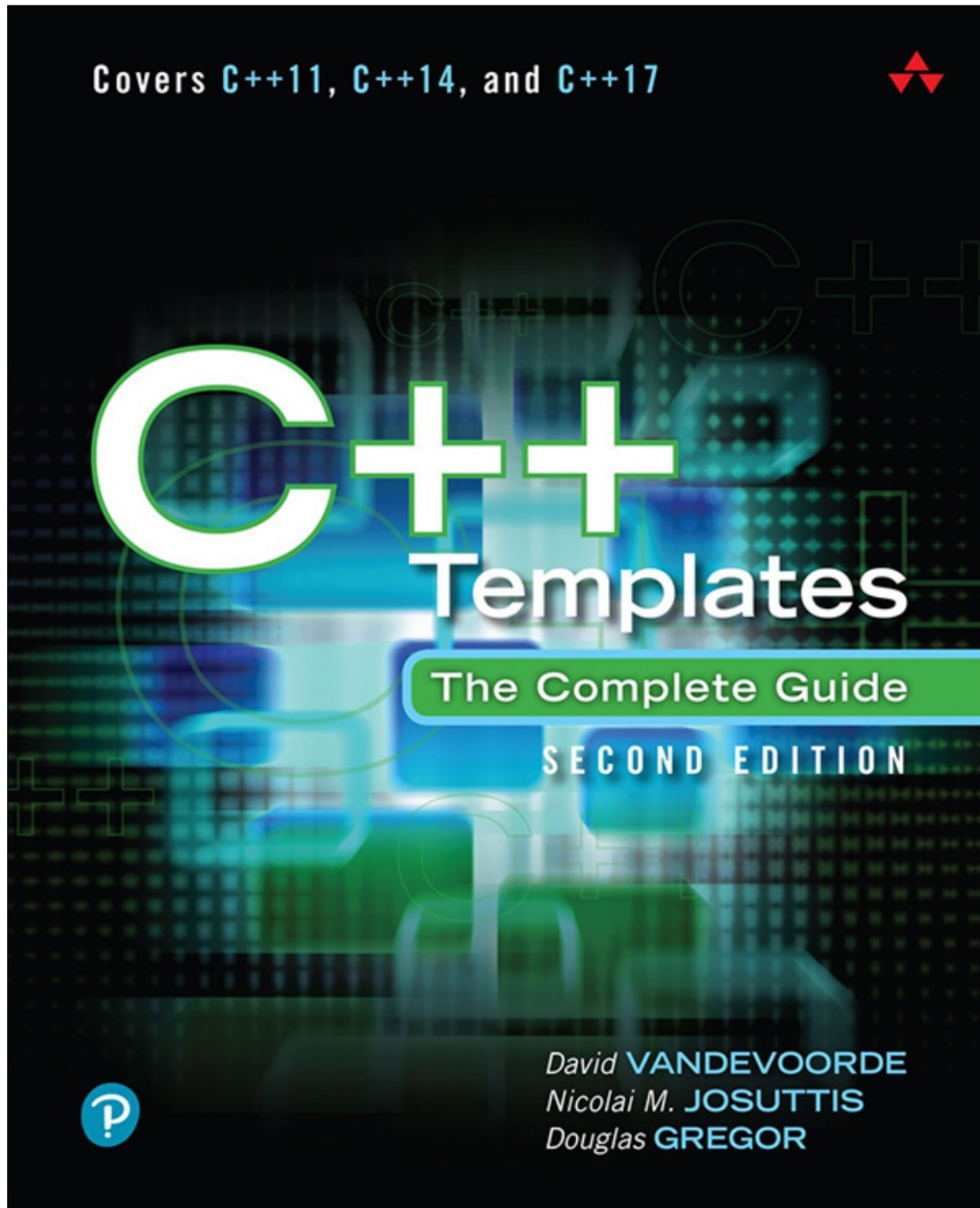
Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    Base a(i);
    cout << a.x_ << endl;
    return a;
}

int main () {
    Base b = GetBase(50); // no call to
                        // copy constructor
}
```

```
whovian@phy504:~/host/CPP$ g++ ref11.cpp -o ref11 -std=c++20
whovian@phy504:~/host/CPP$ ./ref11
Default Constructor
50
Destructor
```



Templates

C++ template metaprogramming is a large topic that we will only glance

Advanced feature

Takes time to study
/ debug
metaprogramming

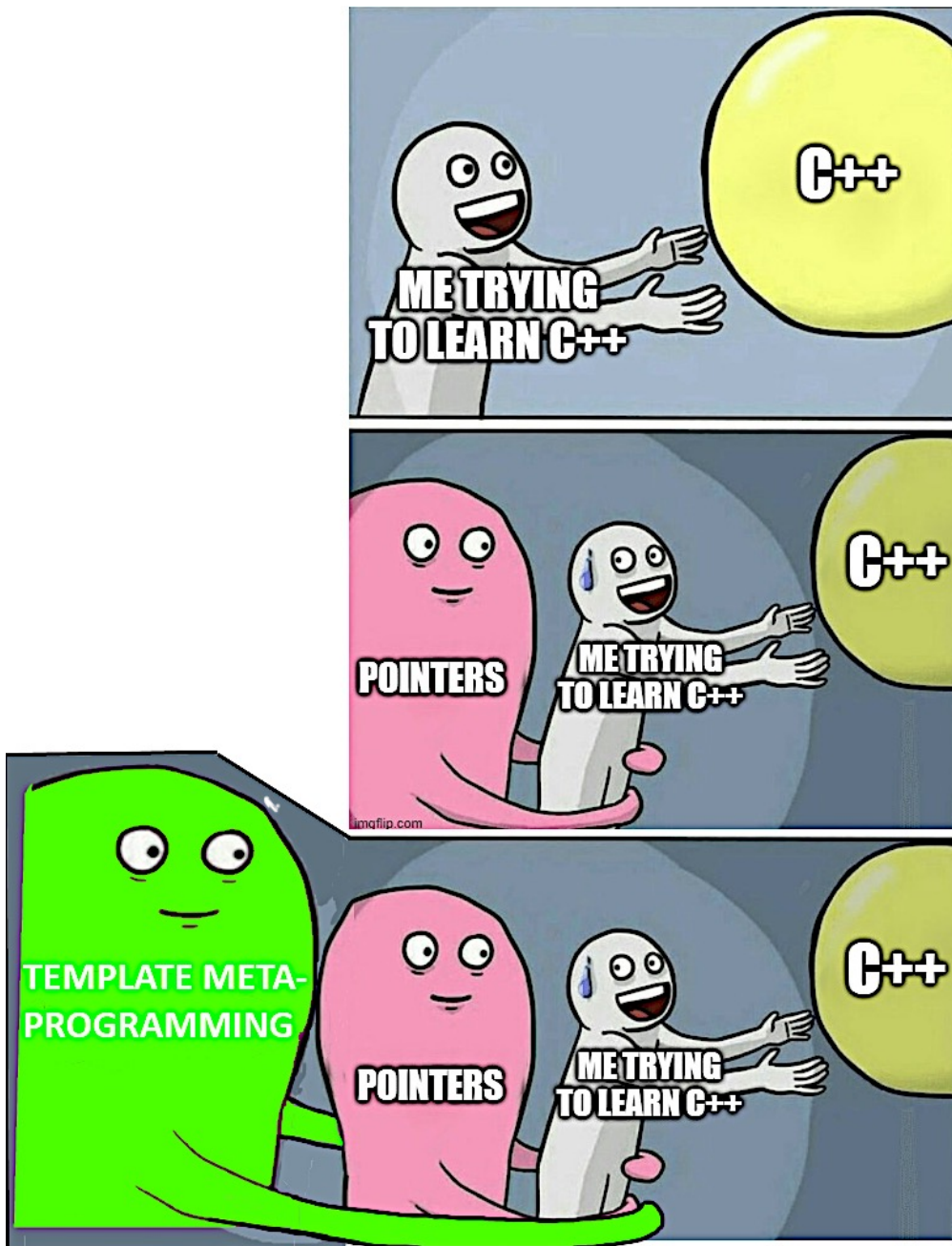
Suggested Literature

Templates

C++ template metaprogramming is hard to master (it takes years)!

Advanced feature

Takes time to study/debug metaprogramming



Templates

```
#include <iostream>
#include <string>
```

```
using std::string;
using std::cout;
using std::endl;
```

```
// We write one single metafunction
// The compiler creates appropriate
// functions for every type
```

```
template<typename T>
T mymax(T a, T b)
{
    return b < a ? a : b;
}
```

```
int main()
{
    cout << "My first program with templates" << endl;
    cout << mymax<double>(2.0, 3.0) << endl;
    cout << mymax<int>(2, 3) << endl;

    std::string s1 = "mathematics";
    std::string s2 = "math";
    cout << mymax<string>(s1, s2) << endl;
}
```

Templates args: <arg1, arg2,...>



```
whovian@phy504:~/host/CPP$ g++ template1.cpp -o template1 -std=c++20
whovian@phy504:~/host/CPP$ ./template1
My first program with templates
3
3
mathematics
```

Template arguments are usually **types** (can be integers)

Templates are converted to normal functions during compilation

Templates

```
#include <iostream>
#include <string>
```

```
using std::string;
using std::cout;
using std::endl;
```

```
template<typename T>
T mymax(T a, T b)
{
    return b < a ? a : b;
}
```

```
int main()
{
    cout << "My second program with templates" << endl;
```

```
    // The second arg being an int is ok. Why?
    // We explicitly made the template arg = double
    // So the argument will be cast to double
    cout << mymax<double>(2.0, 3) << endl;
```

```
    // Same here: the second arg will be cast to int
    cout << mymax<int>(2, 3.0) << endl;
```

```
    std::string s2 = "math";
    // Here: String literals are const char*, but
    // an unnamed string will be created (and std::string
    // accept string literals in its constructor
    cout << mymax<string>("mathematics", s2) << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ template2.cpp -o template2 -std=c++20
whovian@phy504:~/host/CPP$ ./template2
My second program with templates
3
3
mathematics
```

Template arguments can be deduced from arg's types

We need to be a bit careful here. No casting allowed if template args are deduced

Templates

We need to be a bit careful here. No casting allowed if template args are deduced

```
#include <iostream>
#include <string>
using std::string; using std::cout; using std::endl;

template<typename T>
T mymax(T a, T b) {
    return b < a ? a : b;
}

int main() {
    cout << "My third program with templates" << endl;

    // compiler error!
    cout << mymax(2, 3.5) << endl;
    // compiler error!
    string s2 = "math";
    cout << mymax("mathematics", s2) << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ template3.cpp -o template3 -std=c++20
```

```
template3.cpp: In function 'int main()':
```

```
template3.cpp:19:16: error: no matching function for call to 'mymax(int, double)'
```

```
19 |     cout << mymax(2, 3.5) << endl;
    |                  ~~~~~^~~~~~
```

```
template3.cpp:9:3: note: candidate: 'template<class T> T mymax(T, T)'
```

```
9 | T mymax(T a, T b)
  |   ~~~~~
```

```
template3.cpp:9:3: note:   template argument deduction/substitution failed:
```

```
template3.cpp:19:16: note:   deduced conflicting types for parameter 'T' ('int' and 'double')
```

```
19 |     cout << mymax(2, 3.5) << endl;
    |                  ~~~~~^~~~~~
```

```
template3.cpp:23:16: error: no matching function for call to 'mymax(const char [12], std::string&)'
```

```
23 |     cout << mymax("mathematics", s2) << endl;
    |                  ~~~~~^~~~~~
```

```
template3.cpp:9:3: note: candidate: 'template<class T> T mymax(T, T)'
```

```
9 | T mymax(T a, T b)
  |   ~~~~~
```

```
template3.cpp:9:3: note:   template argument deduction/substitution failed:
```

```
template3.cpp:23:16: note:   deduced conflicting types for parameter 'T' ('const char*' and 'std::__cxx11::basic_string<char>')
```

```
23 |     cout << mymax("mathematics", s2) << endl;
    |                  ~~~~~^~~~~~
```

```
#include <iostream>
#include <string>
```

```
using std::string;
using std::cout;
using std::endl;
```

```
// Here the type of the first argument
// specifies the type of the return
```

```
template<typename T1, typename T2>
```

```
T1 mymax(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

```
int main()
{
```

```
    cout << "My Fourth program with templates" << endl;
```

```
    // Not a compiler error anymore, but...
```

```
    // the following two lines produce different answers
```

```
    cout << mymax(2, 3.5) << endl; Solving this issue will
```

```
    cout << mymax(3.5, 2) << endl; take a few slides
```

```
}
```

Templates

We can have
multiple **template
arguments**
(even infinite w/
variadic templates)

Here, we still need
to be a bit careful
with arg. deduction

```
whovian@phy504:~/host/CPP$ g++ template4.cpp -o template4 -std=c++20
```

```
whovian@phy504:~/host/CPP$ ./template4
```

```
My Fourth program with templates
```

```
3
```

```
3.5
```

Templates

We can have multiple **template arguments**.

We still need to be a bit careful with arg. deduction

First solution: (a bit ugly): independent return type

```
#include <iostream>
#include <string>
using std::string; using std::cout; using std::endl;

// Here, we did something different. We made the
// return type independent of the types of the
// function arguments. Therefore, we will always
// need to write the return type explicitly as a
// template argument when calling the function.

// We strategically put the return type as the first
// template argument, so T2 and T3 can be deduced
// from the types of the function arguments
// (partially template argument deduction)
template<typename T1, typename T2, typename T3>
T1 mymax(T2 a, T3 b)
{
    return b < a ? a : b;
}

int main()
{
    cout << "My fifth program with templates" << endl;

    // We explicitly included the return type
    // as a template argument. Second and third
    // template arguments will be deduced from
    // the types of the function arguments.
    cout << mymax<double>(2, 3.5) << endl;
    cout << mymax<double>(3.5, 2) << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ template5.cpp -o template5 -std=c++20
whovian@phy504:~/host/CPP$ ./template5
My fifth program with templates
3.5
3.5
```

Templates

STL **type_traits** rescue us

std::common_type_t

determines the common type; the type all Ts can be implicitly converted to.

If such a type exists, ok.

Otherwise, there is no member type (compiler error)

```
#include <iostream>
#include <string>
#include <type_traits>
using std::string; using std::cout; using std::endl;

template<typename T1, typename T2>
std::common_type_t<T1,T2> mymax(T1 a, T2 b)
{
    return b < a ? a : b;
}

int main()
{
    cout << "My sixth program with templates" << endl;

    int x = 2;
    double y = 5.1;
    cout << std::fixed << mymax(x, y) << endl;
    cout << std::fixed << mymax(y, x) << endl;

    int z1 = 5;
    double z2 = 2.2;
    cout << std::fixed << mymax(z1, z2) << endl;
    cout << std::fixed << mymax(z2, z1) << endl;

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ template6.cpp -o template6 -std=c++20
whovian@phy504:~/host/CPP$ ./template6
My sixth program with templates
5.100000
5.100000
5.000000
5.000000
```

Templates

std::common_type_t

Example: no common type
between **string** and **int**.

```
#include <iostream>
#include <string>
#include <type_traits>
using std::string; using std::cout; using std::endl;

template<typename T1, typename T2>
std::common_type_t<T1,T2> mymax(T1 a, T2 b)
{
    return b < a ? a : b;
}

int main() {
    int x = 2;
    std::string y = "Testing";
    cout << std::fixed << mymax(x, y) << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ template7.cpp -o template7 -std=c++20
template7.cpp: In function 'int main()':
template7.cpp:18:30: error: no matching function for call to 'mymax(int&, std::string&)'
   18 |     cout << std::fixed << mymax(x, y) << endl;
      |                               ~~~~~^~~~~~
template7.cpp:7:27: note: candidate: 'template<class T1, class T2> std::common_type_t<Tp1, _Tp2> mymax(T1, T2)'
    7 | std::common_type_t<T1,T2> mymax(T1 a, T2 b)
      |                               ~~~~~
template7.cpp:7:27: note:   template argument deduction/substitution failed:
In file included from /usr/include/c++/12/bits/move.h:57,
                 from /usr/include/c++/12/bits/exception_ptr.h:43,
                 from /usr/include/c++/12/exception:168,
                 from /usr/include/c++/12/ios:39,
                 from /usr/include/c++/12/ostream:38,
                 from /usr/include/c++/12/iostream:39,
                 from template7.cpp:1:
/usr/include/c++/12/type_traits: In substitution of 'template<class ... _Tp> using common_type_t = typename std::common_type::type [with _Tp = {int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >}]':
template7.cpp:7:27:   required by substitution of 'template<class T1, class T2> std::common_type_t<Tp1, _Tp2> mymax(T1, T2) [with T1 = int; T2 = std::__cxx11::basic_string<char>]'
template7.cpp:18:30:   required from here
/usr/include/c++/12/type_traits:2622:11: error: no type named 'type' in 'struct std::common_type<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >'
 2622 |     using common_type_t = typename common_type<Tp...>::type;
      |
```

```

#include <iostream>
#include <string>
#include <type_traits>
using std::string; using std::cout; using std::endl;

template<typename T1, typename T2>
std::common_type_t<T1, T2> mymax(T1 a, T2 b) {
    return b < a ? a : b;
}

// overloading for pointers of the same type
template<typename T>
T* mymax(T* a, T* b) {
    cout << "Template overloading (pointer)" << endl;
    return *b < *a ? a : b;
}

int main() {
    int x = 2;
    int y = 5;
    cout << std::fixed << mymax(x, y) << endl;

    int* z1 = &x;
    int* z2 = &y;
    auto z3 = mymax(z1, z2);
    cout << std::fixed << *z3 << endl;
}

```

Templates

Function
overloading also
works with
template
functions

Ex: expand
template function
to work with
pointers

```

whovian@phy504:~/host/CPP$ g++ template8.cpp -o template8 -std=c++20
whovian@phy504:~/host/CPP$ ./template8
5
Template overloading (pointer)
5

```

Template Classes

Classes can also be
templates

Here, we create a stack
class (which can hold
integers, doubles,
strings, etc.).

```
#include <print>
#include <string>
#include <vector>

template<typename T>
class stack {
private:
    std::vector<T> data_;
public:
    void push(const T& val) { data_.push_back(val); }

    T pop() {
        T top = data_.back();
        data_.pop_back();
        return top;
    }

    T& top() { return data_.back(); }
    bool empty() const { return data_.empty(); }
    int size() const { return data_.size(); }
};

int main() {
    stack<int> istack; // stack of ints
    istack.push(10); istack.push(20); istack.push(30);

    std::println("int stack: ");
    while (!istack.empty()) std::println("{} ", istack.pop());

    stack<std::string> sstack; // stack of strings
    sstack.push("hello"); sstack.push("world"); sstack.push("!");

    std::println("string stack:");
    while (!sstack.empty()) std::println("{} ", sstack.pop());
    return 0;
}
```

```
whovian@phy504:~/host/Lecture26$ g++ template10.cpp -o t10 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t10
int stack:
30
20
10
string stack:
!
world
hello
```

Integer as Template

Integers can also be
template arguments

With recursion, we can
do compile-time math
(be careful with large
numbers)

```
#include <iostream>
#include <cmath>

template<typename T, int N>
struct FixedArray { // fixed-size array
    T data[N];

    constexpr int size() const { return N; }
    T& operator[](int i) { return data[i]; }
    const T& operator[](int i) const { return data[i]; }

    // Dot product – only works with same-sized arrays
    T dot(const FixedArray<T, N>& other) const {
        T result = T{};
        for (int i = 0; i < N; ++i) {
            result += data[i] * other.data[i];
        }
        return result;
    }
    T magnitude() const {
        return std::sqrt(dot(*this));
    }
};

// NTT for compile-time exponentiation
template<int Exp> constexpr double power(double base) {
    if constexpr (Exp == 0) {
        return 1.0;
    } else if constexpr (Exp > 0) {
        return base * power<Exp - 1>(base);
    } else {
        return 1.0 / power<-Exp>(base);
    }
}

int main() {
    FixedArray<double, 3> a = {1.0, 2.0, 3.0};
    FixedArray<double, 3> b = {4.0, 5.0, 6.0};

    std::cout << "a.size() = " << a.size() << "\n";
    std::cout << "a dot b = " << a.dot(b) << "\n"; // 1*4 + 2*5 + 3*6 = 32
    std::cout << "|a| = " << a.magnitude() << "\n"; // sqrt(14)

    // FixedArray<double, 4> c = {1, 2, 3, 4}; // compile time error
    // a.dot(c); // error: FixedArray<double,3> vs FixedArray<double,4>

    std::cout << "2^10 = " << power<10>(2.0) << "\n"; // 1024
}
```

```
whovian@phy504:~/host/Lecture26$ nano template11.cpp
whovian@phy504:~/host/Lecture26$ g++ template11.cpp -o t11
whovian@phy504:~/host/Lecture26$ ./t11
a.size() = 3
a dot b = 32
|a| = 3.74166
2^10 = 1024
```

CTAD

CTAD (C++17): Class Template argument deduction The compiler deduces class template arguments from constructor arguments

```
#include <iostream>
#include <string>
#include <vector>
#include <tuple>
#include <array>
#include <optional>

// Example 1: CTAD with standard library types
// Before C++17: std::pair<int, double> p(1, 3.14);
// After C++17: std::pair p(1, 3.14); // deduced!

// Example 2: Custom class with CTAD
template<typename T>
class Wrapper {
    T value_;
public:
    Wrapper(T val) : value_(val) {}

    const T& get() const { return value_; }

    void print() const {
        std::cout << " Wrapper<" << typeid(T).name()
            << ">: " << value_ << std::endl;
    }
};

// Example 3: Custom deduction guide
// Without this, Wrapper("hello") would deduce Wrapper<const char*>
// The deduction guide forces it to Wrapper<std::string>

Wrapper(const char*) -> Wrapper<std::string>;

int main() {
    std::cout << "STL CTAD" << std::endl;

    std::pair p(42, 3.14); // deduces pair<int, double>
    std::cout << " pair: " << p.first << ", " << p.second << "\n";

    std::tuple t(1, 2.0, "three"); // deduces tuple<int, double, const char*>
    std::cout << " tuple: " << std::get<0>(t) << ", "
        << std::get<1>(t) << ", " << std::get<2>(t) << "\n";

    std::vector v = {1, 2, 3, 4, 5}; // deduces vector<int>
    std::cout << " vector size: " << v.size() << "\n";

    std::cout << "\nCustom Wrapper CTAD" << std::endl;

    Wrapper w1(42); // deduces Wrapper<int>
    Wrapper w2(3.14); // deduces Wrapper<double>
    w1.print();
    w2.print();

    std::cout << "\nDeduction guide (const char* -> string)" << std::endl;
    Wrapper w4("hello"); // deduces Wrapper<std::string> (not const char*)!
    w4.print();

    Wrapper w5(std::string("world")); // also Wrapper<std::string>
    w5.print();
}
```

```
whovian@phy504:~/host/Lecture26$ nano template28.cpp
whovian@phy504:~/host/Lecture26$ g++ template28.cpp -o t28 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t28
STL CTAD
pair: 42, 3.14
tuple: 1, 2, three
vector size: 5

Custom Wrapper CTAD
Wrapper<i>: 42
Wrapper<d>: 3.14

Deduction guide (const char* -> string)
Wrapper<NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE>: hello
Wrapper<NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE>: world
```

Variadic templates

We can have an infinite
number of template
parameters

We use template
specialization to end the
recursion

```
#include <iostream>
#include <string>

// Example 1: Recursive sum
// Base case: single argument
template<typename T>
T sum(T val) {
    return val;
}
// peel off first arg, recurse of the rest
template<typename T, typename... Args>
auto sum(T first, Args... rest) {
    return first + sum(rest...);
}

// Example 2: Print all arguments
// Base case
void print() {
    std::cout << "\n";
}
// Recursive case
template<typename T, typename... Args>
void print(T first, Args... rest) {
    std::cout << first;
    if constexpr (sizeof...(rest) > 0) {
        std::cout << ", ";
    }
    print(rest...);
}

int main() {
    // Sum of mixed types
    std::cout << "sum(1, 2, 3, 4, 5)    = " <<
        sum(1, 2, 3, 4, 5) << "\n";
    std::cout << "sum(1.5, 2.5, 3.0)    = " <<
        sum(1.5, 2.5, 3.0) << "\n";
    std::cout << "sum(string + string) = " <<
        sum(std::string("hel"), std::string("lo")) << "\n";

    // Print any number of any types
    std::cout << "print: ";
    print(42, 3.14, "hello", std::string("world"), true);
    return 0;
}
```

```
whovian@phy504:~/host/Lecture26$ nano template12.cpp
whovian@phy504:~/host/Lecture26$ g++ template12.cpp -o t12
whovian@phy504:~/host/Lecture26$ ./t12
sum(1, 2, 3, 4, 5)    = 15
sum(1.5, 2.5, 3.0)    = 7
sum(string + string)  = hello
print: 42, 3.14, hello, world, 1
```

Variadic templates

We can have an infinite
number of template
parameters

We use template
specialization to end the
recursion

```
#include <iostream>
#include <string>

// Type-safe printf (simplified)
void my_printf(const char* fmt) {
    // base case: no more args, just print remaining format string
    while (*fmt) {
        if (*fmt == '%') {
            throw std::runtime_error("Too few arguments for format string");
        }
        std::cout << *fmt++;
    }
}

template<typename T, typename... Args>
void my_printf(const char* fmt, T first, Args... rest) {
    while (*fmt) {
        if (*fmt == '%') {
            std::cout << first;
            my_printf(fmt + 1, rest...); // recurse with remaining args
            return;
        }
        std::cout << *fmt++;
    }
}

// Count arguments at compile time
template<typename... Args>
constexpr int count_args(Args... args) {
    return sizeof...(args); // sizeof... gives the pack size
}

int main() {
    // Type-safe printf
    std::cout << "printf: ";
    my_printf("Name: %, Age: %, GPA: %\n", "Alice", 22, 3.95);

    // Count args
    std::cout << "count_args(a,b,c,d) = " << count_args(1, 2, 3, 4) << "\n";
    std::cout << "count_args() = " << count_args() << "\n";

    return 0;
}
```

```
whovian@phy504:~/host/Lecture26$ nano template13.cpp
whovian@phy504:~/host/Lecture26$ g++ template13.cpp -o t13
whovian@phy504:~/host/Lecture26$ ./t13
printf: Name: Alice, Age: 22, GPA: 3.95
count_args(a,b,c,d) = 4
count_args() = 0
```

Template- Template params

Template parameters
can themselves be
templates

```
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <string>
#include <numeric>

// Example 1: Generic fill-and-print for any container
// Container is a template that takes a type + allocator
template<template<typename, typename> class Container, typename T>
void fill_and_print(int n, const T& val) {
    Container<T, std::allocator<T>> c(n, val);
    std::cout << " [";
    bool first = true;
    for (const auto& x : c) {
        if (!first) std::cout << ", ";
        std::cout << x;
        first = false;
    }
    std::cout << "]" (size=" << c.size() << "\n";
}

// Example 2: Convert between containers
template<template<typename, typename> class To,
        template<typename, typename> class From,
        typename T>
To<T, std::allocator<T>> convert_container(const From<T, std::allocator<T>>& source) {
    return To<T, std::allocator<T>>(source.begin(), source.end());
}

int main() {
    std::cout << "fill_and_print" << std::endl;
    std::cout << "vector: ";
    fill_and_print<std::vector>(5, 3.14);
    std::cout << "list: ";
    fill_and_print<std::list>(3, 42);

    std::cout << "\nconvert_container" << std::endl;
    std::vector<int> v = {1, 2, 3, 4, 5};
    auto l = convert_container<std::list>(v);

    std::cout << "vector -> list: ";
    for (int x : l) std::cout << x << " ";
    std::cout << "\n";
}
```

```
whovian@phy504:~/host/Lecture26$ nano template25.cpp
whovian@phy504:~/host/Lecture26$ g++ template25.cpp -o t25 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t25
fill_and_print
vector:  [3.14, 3.14, 3.14, 3.14, 3.14] (size=5)
list:    [42, 42, 42] (size=3)

convert_container
vector -> list:  1 2 3 4 5
```

Folding expressions

Fold expressions (C++17)
are a new way to expand
parameter packs that
previously required
recursive template
specialization

```
#include <iostream>
#include <string>
#include <vector>
// Unary right fold:
// (args + ...) means arg1 + (arg2 + (arg3 + arg4))
template<typename... Args>
auto sum(Args... args) {
    return (args + ...);
}
// Unary left fold:
// (... + args) means ((arg1 + arg2) + arg3) + arg4
template<typename... Args>
auto sum_left(Args... args) {
    return (... + args);
}
// Binary fold with init: (init + ... + args)
// Handles the empty pack case by returning init
template<typename... Args>
auto sum_safe(Args... args) {
    return (0 + ... + args); // if no args, returns 0
}
// Print all fold
template<typename... Args>
void print_all(Args... args) {
    ((std::cout << args << " "), ...);
    std::cout << "\n";
}
// max fold
template<typename T, typename... Args>
T max_of(T first, Args... rest) {
    T result = first;
    ((result = (rest > result ? rest : result)), ...);
    return result;
}

int main() {
    // Sum
    std::cout << "sum(1,2,3,4,5) = " << sum(1, 2, 3, 4, 5) << "\n";
    std::cout << "sum_safe() = " << sum_safe() << "\n"; // empty: returns 0
    // Print
    std::cout << "print_all: ";
    print_all("hello", 42, 3.14, true, 'x');
    // Max
    std::cout << "max_of(3,7,2,9,1) = " << max_of(3, 7, 2, 9, 1) << "\n";
}
```

```
whovian@phy504:~/host/Lecture26$ nano template14.cpp
whovian@phy504:~/host/Lecture26$ g++ template14.cpp -o t14
whovian@phy504:~/host/Lecture26$ ./t14
sum(1,2,3,4,5) = 15
sum_safe() = 0
print_all: hello 42 3.14 1 x
max_of(3,7,2,9,1) = 9
```

```

#include <iostream>
#include <string>
#include <vector>

// Logical AND fold: are all values positive?
template<typename... Args>
bool all_positive(Args... args) {
    return ((args > 0) && ...);
}

// Logical OR fold: is any value negative?
template<typename... Args>
bool any_negative(Args... args) {
    return ((args < 0) || ...);
}

// Push multiple items into a vector
template<typename T, typename... Args>
void push_many(std::vector<T>& vec, Args&&... args) {
    (vec.push_back(args), ...); // executes push_back for each arg
}

int main() {

    // Logical folds
    std::cout << "all_positive(1,2,3) = " <<
        all_positive(1, 2, 3) << "\n";
    std::cout << "all_positive(1,-2,3) = " <<
        all_positive(1, -2, 3) << "\n";
    std::cout << "any_negative(1,2,3) = " <<
        any_negative(1, 2, 3) << "\n";
    std::cout << "any_negative(1,-2,3) = " <<
        any_negative(1, -2, 3) << "\n";

    // Push many
    std::vector<int> v;
    push_many(v, 10, 20, 30, 40, 50);
    std::cout << "vector after push_many: ";
    for (int x : v) std::cout << x << " ";
    std::cout << std::endl;

}

```

Folding expressions

Fold expressions (C++17)
are a new way to expand
parameter packs that
previously required
recursive template
specialization

```

whovian@phy504:~/host/Lecture26$ nano template15.cpp
whovian@phy504:~/host/Lecture26$ g++ template15.cpp -o t15
whovian@phy504:~/host/Lecture26$ ./t15
all_positive(1,2,3) = 1
all_positive(1,-2,3) = 0
any_negative(1,2,3) = 0
any_negative(1,-2,3) = 1
vector after push_many: 10 20 30 40 50

```

SFINAE

SFINAE means
Substitution Failure is
not an error.

When template
substitution fails, the
compiler removes the
overload instead of
giving a hard error

```
#include <iostream>
#include <string>
#include <type_traits>
#include <vector>

// Example 1: enable_if to constrain by type category

// Only enabled for integral types (int, long, char, etc.)
template<typename T>
typename std::enable_if<std::is_integral<T>::value, std::string>::type
classify(T val) {
    return std::to_string(val) + " is an integer type";
}

// Only enabled for floating point types (float, double)
template<typename T>
typename std::enable_if<std::is_floating_point<T>::value, std::string>::type
classify(T val) {
    return std::to_string(val) + " is a floating point type";
}

// Example 2: enable_if in return type (C++14)

template<typename T>
auto double_it(T val) -> std::enable_if_t<std::is_arithmetic_v<T>, T> {
    return val * 2;
}

// For strings: "double" means repeat
template<typename T>
auto double_it(T val) -> std::enable_if_t<std::is_same_v<T, std::string>, T> {
    return val + val;
}

int main() {
    // classify: picks the right overload by type
    std::cout << classify(42) << "\n";
    std::cout << classify(3.14) << "\n";
    // classify("hello"); // won't compile: no matching overload

    // double_it: works for numbers and strings
    std::cout << "double_it(21) = " << double_it(21) << "\n";
    std::cout << "double_it(\"abc\") = " << double_it(std::string("abc")) << "\n";
    std::cout << std::endl;
    return 0;
}
```

```
whovian@phy504:~/host/Lecture26$ nano template16.cpp
whovian@phy504:~/host/Lecture26$ g++ template16.cpp -o t16
whovian@phy504:~/host/Lecture26$ ./t16
42 is an integer type
3.140000 is a floating point type
double_it(21) = 42
double_it("abc") = abcabc
```

SFINAE

SFINAE means

Substitution Failure is not
an error.

When template substitution
fails, the compiler removes
the overload instead of
giving a hard error

```
#include <iostream>
#include <string>
#include <type_traits>
#include <vector>
// EXAMPLE 1: Detecting if a type has a .size() method

// Primary: assume no .size()
template<typename T, typename = void>
struct has_size : std::false_type {};

// Specialization: if decltype(T.size()) is valid, this is chosen
template<typename T>
struct has_size<T, std::void_t<decltype(std::declval<T>().size())>>
    : std::true_type {};

template<typename T>
void print_info(const T& val) {
    if constexpr (has_size<T>::value) {
        std::cout << " Has .size() = " << val.size() << "\n";
    } else {
        std::cout << " No .size() method" << std::endl;
    }
}

// EXAMPLE 2: enable_if as a default template parameter
template<typename T,
        typename = std::enable_if_t<std::is_arithmetic_v<T>>>
T safe_abs(T val) {
    return val < 0 ? -val : val;
}

int main() {
    std::cout << "has_size<vector<int>> = " <<
        has_size<std::vector<int>>::value << "\n";
    std::cout << "has_size<string> = " <<
        has_size<std::string>::value << "\n";
    std::cout << "has_size<int> = " <<
        has_size<int>::value << "\n";

    // print_info uses the trait
    std::vector<int> v = {1, 2, 3};
    std::cout << "vector: ";
    print_info(v);
    int x = 42;
    std::cout << "int:    ";
    print_info(x);

    // safe_abs
    std::cout << "safe_abs(-7)  = " << safe_abs(-7) << "\n";
    std::cout << "safe_abs(-3.5) = " << safe_abs(-3.5) << "\n";
    // safe_abs(std::string("x")); // compile error
    std::endl;
}
```

```
whovian@phy504:~/host/Lecture26$ g++ template17.cpp -o t17
whovian@phy504:~/host/Lecture26$ ./t17
has_size<vector<int>> = 1
has_size<string> = 1
has_size<int> = 0
vector:  Has .size() = 3
int:     No .size() method
safe_abs(-7)  = 7
safe_abs(-3.5) = 3.5
```

Concepts

Concepts aims to be
the C++26 replacement
of **SFINAE**

The idea is to restrict
template behavior
without need for failed
overload

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <concepts>
#include <type_traits>

// Example 1: function only accepts integral types
template<std::integral T>
T gcd(T a, T b) {
    while (b != 0) {
        T temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main() {
    // gcd: only works with integers
    std::cout << "gcd(48, 18) = " << gcd(48, 18) << std::endl;
    // gcd(3.14, 2.0);
    // compile error: double doesn't satisfy std::integral
}
```

```
whovian@phy504:~/host/Lecture26$ nano template18.cpp
whovian@phy504:~/host/Lecture26$ g++ template18.cpp -o t18 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t18
gcd(48, 18) = 6
```

Concepts

Concepts aims to be
the C++26 replacement
of **SFINAE**

The idea is to restrict
template behavior
without need for failed
overload

You can create custom
concepts

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <concepts>
#include <type_traits>
template<typename T>
concept Printable = requires(T val) {
    { std::cout << val } -> std::same_as<std::ostream&>;
};

template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;

template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

template<typename T>
concept Container = requires(T c) {
    { c.size() } -> std::convertible_to<std::size_t>;
    { c.begin() };
    { c.end() };
    typename T::value_type;
};
```

Concepts

Concepts aims to be the C++26 replacement of **SFINAE**

The idea is to restrict template behavior without need for failed overload

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <concepts>
#include <type_traits>
template<typename T>
concept Printable = requires(T val) {
    { std::cout << val } -> std::same_as<std::ostream&>;
};

template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;

template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

template<typename T>
concept Container = requires(T c) {
    { c.size() } -> std::convertible_to<std::size_t>;
    { c.begin() };
    { c.end() };
    typename T::value_type;
};

// Using concepts in different syntaxes
template<Numeric T> T square(T x) { return x * x; }

template<typename T>
requires Addable<T> && Printable<T> T add_and_print(T a, T b) {
    T result = a + b;
    std::cout << " " << a << " + " << b << " = " << result << "\n";
    return result;
}

// Syntax 3: abbreviated function template (auto + concept)
void show_size(Container auto const& c) {
    std::cout << " Container size: " << c.size() << std::endl;
}

// Syntax 4: trailing requires
template<typename T>
T half(T val) requires Numeric<T> { return val / 2; }

int main() {
    std::cout << "square(5)  = " << square(5) << std::endl;
    // square("hi"); // error: const char* doesn't satisfy Numeric

    add_and_print(10, 20);
    add_and_print(std::string("hello "), std::string("world"));

    std::cout << "show_size:\n";
    std::vector<int> v = {1, 2, 3, 4, 5};
    show_size(v);
    std::string s = "hello";
    show_size(s); // string is also a container!
    // show_size(42); // error: int doesn't satisfy Container

    // half
    std::cout << "half(10)  = " << half(10) << "\n";
    std::cout << "half(7.5) = " << half(7.5) << "\n";
}
```

```
whovian@phy504:~/host/Lecture26$ nano template19.cpp
whovian@phy504:~/host/Lecture26$ g++ template19.cpp -o t19 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t19
square(5)  = 25
10 + 20 = 30
hello + world = hello world
show_size:
Container size: 5
Container size: 5
half(10)  = 5
half(7.5) = 3.75
```

Compile Time if/else

if constexpr (C++17):

Branches that are not taken are completely discarded at compile time.

They don't even need to be valid code for the given type.

```
#include <iostream>
#include <string>
#include <type_traits>
#include <vector>
#include <array>

// Example 1: Type-dependent behavior in a single function
template<typename T>
std::string to_debug_string(const T& val) {
    if constexpr (std::is_integral_v<T>) {
        return "int(" + std::to_string(val) + ")";
    } else if constexpr (std::is_floating_point_v<T>) {
        return "float(" + std::to_string(val) + ")";
    } else if constexpr (std::is_same_v<T, std::string>) {
        return "string(\"" + val + "\")";
    } else if constexpr (std::is_pointer_v<T>) {
        if (val == nullptr) return "nullptr";
        return "ptr -> " + to_debug_string(*val);
    } else {
        return "[unknown type]";
    }
}

// Example 2: Replacing recursive variadic templates
template<typename T, typename... Rest>
void print_types(T first, Rest... rest) {
    std::cout << to_debug_string(first);

    if constexpr (sizeof...(rest) > 0) {
        std::cout << ", ";
        print_types(rest...); // only instantiated if there are more args
    } else {
        std::cout << "\n";
    }
}

int main() {
    // Debug strings
    std::cout << to_debug_string(42) << "\n";
    std::cout << to_debug_string(3.14) << "\n";
    std::cout << to_debug_string(std::string("hello")) << "\n";
    int x = 99;
    int* px = &x;
    std::cout << to_debug_string(px) << "\n" << std::endl;

    // Print types
    print_types(1, 2.5, std::string("hi"), true);
}
```

```
whovian@phy504:~/host/Lecture26$ nano template20.cpp
whovian@phy504:~/host/Lecture26$ g++ template20.cpp -o t20 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t20
int(42)
float(3.14)
string("hello")
ptr -> int(99)

int(1), float(2.5), string("hi"), int(1)
```

Compile Time if/else

if constexpr (C++17):

Branches that are not taken are completely discarded at compile time.

They don't even need to be valid code for the given type.

```
#include <iostream>
#include <string>
#include <type_traits>
#include <vector>
#include <array>

// Example 3: Different algorithms based on container type
template<typename Container>
auto get_element(const Container& c, size_t index) {
    // std::array and std::vector have operator[], but other containers might not
    if constexpr (requires { c[index]; }) {
        return c[index]; // random access
    } else {
        // fallback: iterate
        auto it = c.begin();
        std::advance(it, index);
        return *it;
    }
}

// Example 4: Compile-time factorial using if constexpr
template<int N>
constexpr int factorial() {
    if constexpr (N <= 1) {
        return 1;
    } else {
        return N * factorial<N - 1>();
    }
}

// Example 5: Safe division based on type
template<typename T>
auto safe_divide(T a, T b) {
    if constexpr (std::is_integral_v<T>) {
        // Integer division: check for zero
        if (b == 0) {
            std::cerr << " Warning: integer division by zero!\n";
            return T{0};
        }
        return a / b;
    } else if constexpr (std::is_floating_point_v<T>) {
        // Floating point: division by zero gives inf, which is fine
        return a / b;
    }
}

int main() {
    // Get element from different containers
    std::vector<int> v = {10, 20, 30, 40, 50};
    std::array<int, 3> a = {100, 200, 300};
    std::cout << "vector[2] = " << get_element(v, 2) << "\n";
    std::cout << "array[1] = " << get_element(a, 1) << "\n";

    // Compile-time factorial
    std::cout << "5! = " << factorial<5>() << "\n";
    std::cout << "10! = " << factorial<10>() << "\n";
    static_assert(factorial<6>() == 720, "6! must be 720");

    // Safe division
    std::cout << "10 / 3 = " << safe_divide(10, 3) << "\n";
    std::cout << "10 / 0 = " << safe_divide(10, 0) << "\n";
    std::cout << "10.0/3.0 = " << safe_divide(10.0, 3.0) << "\n";
    std::cout << "10.0/0.0 = " << safe_divide(10.0, 0.0) << "\n"; // inf
    return 0;
}
```

```
whovian@phy504:~/host/Lecture26$ nano template21.cpp
whovian@phy504:~/host/Lecture26$ g++ template21.cpp -o t21 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t21
vector[2] = 30
array[1] = 200
5! = 120
10! = 3628800
10 / 3 = 3
10 / 0 = Warning: integer division by zero!
0
10.0/3.0 = 3.33333
10.0/0.0 = inf
```

C RTP

C RTP: Curiously Recurring Template Pattern

Emulate what virtual
functions do in base class
but without runtime cost

It is also called **static
polymorphism**

```
#include <iostream>
#include <string>
#include <cmath>

template<typename Derived>
class Shape {
public:
    double area() const {
        // static_cast to Derived - we *know* the type at compile time
        return static_cast<const Derived*>(this)->area_impl();
    }

    double perimeter() const {
        return static_cast<const Derived*>(this)->perimeter_impl();
    }

    void describe() const {
        std::cout << "  Shape: area=" << area()
                  << ", perimeter=" << perimeter() << "\n";
    }
};

class Circle : public Shape<Circle> {
    double r_;
public:
    Circle(double r) : r_(r) {}
    double area_impl() const { return M_PI * r_ * r_; }
    double perimeter_impl() const { return 2.0 * M_PI * r_; }
};

class Rectangle : public Shape<Rectangle> {
    double w_, h_;
public:
    Rectangle(double w, double h) : w_(w), h_(h) {}
    double area_impl() const { return w_ * h_; }
    double perimeter_impl() const { return 2.0 * (w_ + h_); }
};

// This function works with ANY Shape<Derived> - no vtable needed
template<typename Derived>
void print_shape(const Shape<Derived>& s) {
    s.describe();
}

int main() {
    Circle c(5.0);
    Rectangle r(4.0, 6.0);

    print_shape(c); // no virtual dispatch
    print_shape(r); // no virtual dispatch
}
```

```
whovian@phy504:~/host/Lecture26$ nano template22.cpp
whovian@phy504:~/host/Lecture26$ g++ template22.cpp -o t22 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t22
  Shape: area=78.5398, perimeter=31.4159
  Shape: area=24, perimeter=20
```

C RTP

C RTP: Curiously Recurring Template Pattern

static
polymorphism is
not the only
application of
C RTP

```
#include <iostream>
#include <string>
#include <cmath>

// C RTP Mixin for operator!= from operator==

template<typename Derived>
class EqualityComparable {
public:
    friend bool operator!=(const Derived& a, const Derived& b) {
        return !(a == b); // automatically generates != from ==
    }
};

class Point : public EqualityComparable<Point> {
    int x_, y_;
public:
    Point(int x, int y) : x_(x), y_(y) {}

    // User only needs to define ==
    friend bool operator==(const Point& a, const Point& b) {
        return a.x_ == b.x_ && a.y_ == b.y_;
    }

    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        return os << "(" << p.x_ << ", " << p.y_ << ")";
    }
};

int main() {

    Point p1(1, 2), p2(1, 2), p3(3, 4);
    std::cout << p1 << " == " << p2 << " ? " << (p1 == p2) << "\n";
    std::cout << p1 << " != " << p3 << " ? " << (p1 != p3) << "\n";
}
```

```
whovian@phy504:~/host/Lecture26$ nano template23.cpp
whovian@phy504:~/host/Lecture26$ g++ template23.cpp -o t23 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t23
(1, 2) == (1, 2) ? 1
(1, 2) != (3, 4) ? 1
```

C RTP

C RTP: Curiously Recurring Template Pattern

static polymorphism is
not the only application
of **C RTP**

```
#include <iostream>
#include <string>
#include <cmath>

// Example: C RTP for method chaining (Builder pattern)

template<typename Derived>
class Builder {
public:
    Derived& set_name(std::string name) {
        static_cast<Derived*>(this)->name_ = std::move(name);
        return static_cast<Derived*>(*this);
    }
};

class PersonBuilder : public Builder<PersonBuilder> {
public:
    std::string name_;
    int age_ = 0;

    PersonBuilder& set_age(int age) {
        age_ = age;
        return *this;
    }

    void print() const {
        std::cout << " Person: " << name_ << ", age " <<
            age_ << "\n";
    }
};

int main() {
    PersonBuilder pb;
    pb.set_name("Alice").set_age(30); // chaining works across
                                     // base and derived
    pb.print();
}
```

```
whovian@phy504:~/host/Lecture26$ nano template24.cpp
whovian@phy504:~/host/Lecture26$ g++ template24.cpp -o t24 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t24
Person: Alice, age 30
```

Type-traits

```
#include <iostream>
#include <string>
#include <type_traits>
#include <vector>
#include <memory>

template<typename T>
void inspect_type(const std::string& name) {
    std::cout << "--- " << name << " ---\n";
    std::cout << "    is_integral:      " << std::is_integral_v<T> << "\n";
    std::cout << "    is_floating_point:  " << std::is_floating_point_v<T> << "\n";
    std::cout << "    is_arithmetic:     " << std::is_arithmetic_v<T> << "\n";
    std::cout << "    is_pointer:        " << std::is_pointer_v<T> << "\n";
    std::cout << "    is_reference:      " << std::is_reference_v<T> << "\n";
    std::cout << "    is_const:          " << std::is_const_v<T> << "\n";
    std::cout << "    is_signed:         " << std::is_signed_v<T> << "\n";
    std::cout << "    is_class:          " << std::is_class_v<T> << "\n";
    std::cout << "    is_trivially_copyable: " << std::is_trivially_copyable_v<T> << "\n";
    std::cout << "    sizeof:           " << sizeof(T) << "\n\n";
}

int main() {
    inspect_type<int>("int");
    inspect_type<double>("double");
    inspect_type<std::string>("std::string");
    inspect_type<int*>("int*");
    inspect_type<const int>("const int");
}
```

compile-time templates used to
inspect or modify the properties of
a type.

```
whovian@phy504:~/host/Lecture26$ g++ template29.cpp -o t29 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t29
--- int ---
is_integral:      1
is_floating_point: 0
is_arithmetic:    1
is_pointer:       0
is_reference:     0
is_const:         0
is_signed:        1
is_class:         0
is_trivially_copyable: 1
sizeof:          4

--- double ---
is_integral:      0
is_floating_point: 1
is_arithmetic:    1
is_pointer:       0
is_reference:     0
is_const:         0
is_signed:        1
is_class:         0
is_trivially_copyable: 1
sizeof:          8

--- std::string ---
is_integral:      0
is_floating_point: 0
is_arithmetic:    0
is_pointer:       0
is_reference:     0
is_const:         0
is_signed:        0
is_class:         1
is_trivially_copyable: 0
sizeof:          32

--- int* ---
is_integral:      0
is_floating_point: 0
is_arithmetic:    0
is_pointer:       1
is_reference:     0
is_const:         0
is_signed:        0
is_class:         0
is_trivially_copyable: 1
sizeof:          8

--- const int ---
is_integral:      1
is_floating_point: 0
is_arithmetic:    1
is_pointer:       0
is_reference:     0
is_const:        1
is_signed:        1
is_class:         0
is_trivially_copyable: 1
sizeof:          4
```

Type-traits

```
#include <iostream>
#include <string>
#include <type_traits>
#include <vector>
#include <memory>

template<typename T>
void inspect_type(const std::string& name) {
    std::cout << "--- " << name << " ---\n";
    std::cout << "  is_integral:      " << std::is_integral_v<T> << "\n";
    std::cout << "  is_floating_point:  " << std::is_floating_point_v<T> << "\n";
    std::cout << "  is_arithmetic:     " << std::is_arithmetic_v<T> << "\n";
    std::cout << "  is_pointer:        " << std::is_pointer_v<T> << "\n";
    std::cout << "  is_reference:      " << std::is_reference_v<T> << "\n";
    std::cout << "  is_const:          " << std::is_const_v<T> << "\n";
    std::cout << "  is_signed:         " << std::is_signed_v<T> << "\n";
    std::cout << "  is_class:          " << std::is_class_v<T> << "\n";
    std::cout << "  is_trivially_copyable: " << std::is_trivially_copyable_v<T> << "\n";
    std::cout << "  sizeof:            " << sizeof(T) << "\n\n";
}

int main() {
    inspect_type<int>("int");
    inspect_type<double>("double");
    inspect_type<std::string>("std::string");
    inspect_type<int*>("int*");
    inspect_type<const int>("const int");
}
```

They act as a way to "query" information about a type ("is this an integer?") or transform it (e.g., "remove const") before the program even runs.

```
whovian@phy504:~/host/Lecture26$ g++ template29.cpp -o t29 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t29

--- int ---
is_integral:      1
is_floating_point: 0
is_arithmetic:    1
is_pointer:       0
is_reference:     0
is_const:         0
is_signed:        1
is_class:         0
is_trivially_copyable: 1
sizeof:          4

--- double ---
is_integral:      0
is_floating_point: 1
is_arithmetic:    1
is_pointer:       0
is_reference:     0
is_const:         0
is_signed:        1
is_class:         0
is_trivially_copyable: 1
sizeof:          8

--- std::string ---
is_integral:      0
is_floating_point: 0
is_arithmetic:    0
is_pointer:       0
is_reference:     0
is_const:         0
is_signed:        0
is_class:         1
is_trivially_copyable: 0
sizeof:          32

--- int* ---
is_integral:      0
is_floating_point: 0
is_arithmetic:    0
is_pointer:       1
is_reference:     0
is_const:         0
is_signed:        0
is_class:         0
is_trivially_copyable: 1
sizeof:          8

--- const int ---
is_integral:      1
is_floating_point: 0
is_arithmetic:    1
is_pointer:       0
is_reference:     0
is_const:         1
is_signed:        1
is_class:         0
is_trivially_copyable: 1
sizeof:          4
```

Type-traits

They act as a way to "query" information about a type ("is this an integer?") or transform it (e.g., "remove const") before the program even runs.

```
#include <iostream>
#include <string>
#include <type_traits>
#include <vector>
#include <memory>

// Part 2: Type transformations
template<typename T>
void show_transformations(const std::string& name) {
    // remove_const: const int -> int
    using no_const = std::remove_const_t<T>;
    std::cout << "    remove_const is same as original: "
               << std::is_same_v<T, no_const> << "\n";

    // remove_reference: int& -> int
    using no_ref = std::remove_reference_t<T>;
    std::cout << "    remove_reference is same: "
               << std::is_same_v<T, no_ref> << "\n";

    // remove_pointer: int* -> int
    using no_ptr = std::remove_pointer_t<T>;
    std::cout << "    remove_pointer is same: "
               << std::is_same_v<T, no_ptr> << "\n";

    // decay: simulates pass-by-value (removes ref, const, array->pointer)
    using decayed = std::decay_t<T>;
    std::cout << "    decay is same: "
               << std::is_same_v<T, decayed> << "\n";

    // add_pointer
    using with_ptr = std::add_pointer_t<T>;
    std::cout << "    add_pointer makes pointer: "
               << std::is_pointer_v<with_ptr> << "\n\n";
}

int main() {
    show_transformations<const int>("const int");
    show_transformations<int&>("int&");
    show_transformations<int*>("int*");
}
```

```
whovian@phy504:~/host/Lecture26$ g++ template30.cpp -o t30 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t30
remove_const is same as original: 0
remove_reference is same: 1
remove_pointer is same: 1
decay is same: 0
add_pointer makes pointer: 1

remove_const is same as original: 1
remove_reference is same: 0
remove_pointer is same: 1
decay is same: 0
add_pointer makes pointer: 1

remove_const is same as original: 1
remove_reference is same: 1
remove_pointer is same: 0
decay is same: 1
add_pointer makes pointer: 1
```

Variable Templates

C++14: Variables can be templated

This is how the **_v** suffix traits work in **<type_traits>**

```
#include <iostream>
#include <type_traits>
#include <cmath>
#include <string>
#include <iomanip> // setw, setfill, setprecision
using std::setprecision;

// Example 1: Mathematical constants with correct precision
template<typename T>
constexpr T pi = T(3.14159265358979323846264338327950288L);

template<typename T>
constexpr T e = T(2.71828182845904523536028747135266250L);

template<typename T>
T circle_area(T radius) {
    return pi<T> * radius * radius;
}

// Example 2: Type traits as variable templates
// (This is how the standard library implements _v helpers)
template<typename T>
struct is_string_like_impl : std::false_type {};

template<>
struct is_string_like_impl<std::string> : std::true_type {};

template<>
struct is_string_like_impl<const char*> : std::true_type {};

template<size_t N>
struct is_string_like_impl<char[N]> : std::true_type {};

// Variable template version (_v suffix, just like the standard library)
template<typename T>
constexpr bool is_string_like_v = is_string_like_impl<std::decay_t<T>>::value;

int main() {
    // Mathematical constants with correct precision
    std::cout << "Math Constants" << std::endl;
    std::cout << "pi<float>: " << setprecision(15) << pi<float> << "\n";
    std::cout << "pi<double>: " << setprecision(15) << pi<double> << "\n";
    std::cout << "e<double>: " << setprecision(15) << e<double> << "\n";

    std::cout << "Circle area (r=5, float): " <<
        setprecision(15) << circle_area(5.0f) << "\n";
    std::cout << "Circle area (r=5, double): " <<
        setprecision(15) << circle_area(5.0) << "\n";

    // Type trait variable template
    std::cout << "\nis_string_like_v" << std::endl;
    std::cout << "string: " << is_string_like_v<std::string> << "\n";
    std::cout << "const char*:" << is_string_like_v<const char*> << "\n";
    std::cout << "char[5]: " << is_string_like_v<char[5]> << "\n";
    std::cout << "int: " << is_string_like_v<int> << "\n";
    std::cout << "double: " << is_string_like_v<double> << "\n";
}
```

```
whovian@phy504:~/host/Lecture26$ nano template26.cpp
whovian@phy504:~/host/Lecture26$ g++ template26.cpp -o t26 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t26
Math Constants
pi<float>: 3.14159274101257
pi<double>: 3.14159265358979
e<double>: 2.71828182845905
Circle area (r=5, float): 78.5398178100586
Circle area (r=5, double): 78.5398163397448

is_string_like_v
string: 1
const char*:1
char[5]: 0
int: 0
double: 0
```

constexpr / consteval compile-time functions

Constexpr (C++11): MAY
be evaluated at compile time
(if inputs are **constexpr**)

consteval (C++20): MUST
be evaluated at compile time

```
#include <iostream>
#include <array>
#include <cmath>

// Example 1: old style (pre-C++11)
template<int N>
struct FactorialTMP {
    static constexpr long long value = N * FactorialTMP<N-1>::value;
};
template<>
struct FactorialTMP<0> { static constexpr long long value = 1; };

// Example 2: Modern constexpr function (C++14+)
// reads like normal code, but runs at compile time
constexpr long long factorial(int n) {
    long long result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}

// Example 2: consteval - guaranteed compile-time
consteval int square_ct(int x) { return x * x; }

// Example 3: constexpr if to validate at compile time
consteval int safe_divide_ct(int a, int b) {
    if (b == 0) {
        throw "Division by zero!"; // this makes it a compile error!
    }
    return a / b;
}

int main() {
    std::cout << "Old style" << std::endl;
    std::cout << "5! = " << FactorialTMP<5>::value << "\n";
    std::cout << "10! = " << FactorialTMP<10>::value << "\n";

    std::cout << "\nModern constexpr" << std::endl;
    constexpr auto f5 = factorial(5); // computed at compile time
    std::cout << "5! = " << f5 << "\n";
    std::cout << "15! = " << factorial(15) << "\n";

    // Can also be used at runtime
    int n;
    std::cout << "Enter n for n!: ";
    n = 7; // simulating input
    std::cout << n << "! = " << factorial(n) << "\n"; // runtime evaluation

    // consteval - must be compile-time
    std::cout << "\nconsteval" << std::endl;
    constexpr int sq = square_ct(7); // ok: compile-time
    std::cout << "square_ct(7) = " << sq << "\n";

    // int runtime_val = 7;
    // int bad = square_ct(runtime_val); // ERROR: consteval requires compile-time args

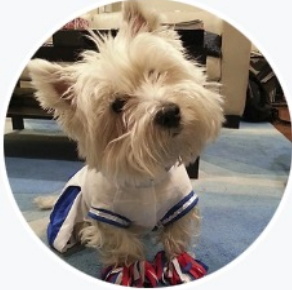
    constexpr int div_result = safe_divide_ct(10, 3);
    std::cout << "safe_divide_ct(10, 3) = " << div_result << "\n";
    // constexpr int bad = safe_divide_ct(10, 0); // COMPILER ERROR
}
```

```
whovian@phy504:~/host/Lecture26$ nano template27.cpp
whovian@phy504:~/host/Lecture26$ g++ template27.cpp -o t27 -std=c++26
whovian@phy504:~/host/Lecture26$ ./t27
Old style
5! = 120
10! = 3628800

Modern constexpr
5! = 120
15! = 1307674368000
Enter n for n!: 7! = 5040

consteval
square_ct(7) = 49
safe_divide_ct(10, 3) = 3
```

Additional Reading: Where to read more?



Barry's C++ Blog

Just a blog about C++



HOME



TAGS



ARCHIVES



ABOUT

There are several expert blogs about C++ at a level way higher than I will teach. If you want more – check, for instance <https://brevzin.github.io>

Exceptions

```

#include <fmt/core.h>
#include <tuple>
#include <string>

// example from en.cppreference.com/w/cpp/utility/tuple

std::tuple<double, char, std::string>
get_student(const int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
        case 1:
            return {2.9, 'C', "Milhouse Van Houten"};
    }

    throw std::invalid_argument("id");
}

int main()
{
    const auto [ gpa, grade, name ] = get_student(3);

    // this part of the program won't run
    fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
        1, gpa, grade, name);

    return 0;
}

whovian@phy504:~/host$ g++ tuples6.cpp -o tuple6 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple6
terminate called after throwing an instance of 'std::invalid_argument'
what(): id
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted

```

Exceptions

What happens if we type
the wrong student **id**?

C++ allows the use of
exceptions to perform
error handling (~Python)

Exception safety is an
advanced topic. Rule for
newbies: dangerous to use

Why? It disrupts the code
flow. What if we have an
array in the heap?

```

#include <fmt/core.h>
#include <tuple>
#include <string>

// example from en.cppreference.com/w/cpp/utility/tuple
std::tuple<double, char, std::string> get_student(int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
    }
    std::string emsg =
        fmt::format("id error on function {0}", __FUNCTION__);
    throw std::invalid_argument(emsg);

    fmt::print("This won't run!!\n");
}

int main()
{
    // catching exceptions: try/catch makes code clucky
    // usually I only try to catch exceptions in the main()
    // function, so I can write some extra debugging msg

    try {
        const auto [ gpa, grade, name ] = get_student(3);
        // code below won't run
        fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
            1, gpa, grade, name);
    }
    catch (std::exception& ex) {
        fmt::print("Error (file: {0}, line: {1})\n", __FILE__, __LINE__);
        fmt::print("{0}\n", ex.what());
        // problem - fmt::print itself may throw an exception!
        // (yes - exception is difficult to get 100% right)
        exit(1);
    }
    catch (...) {
        fmt::print("Error (file: {0}, line: {1})\n",
            __FILE__, __LINE__);
        throw std::runtime_error("Unknown Exception\n");
    }
    return 0;
}

```

```

whovian@phy504:~/host$ g++ tuples7.cpp -o tuple7 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple7
Error (file: tuples7.cpp, line: 32)
id error on function get_student

```

Exceptions

Exception: Dangerous

Why? It disrupts normal flow of the program

If you adopt **malloc** / **free** for mem allocation (C++ ver. **new** / **delete**), then C++ exceptions can create memory leaks.

Congratulations

You now have an intermediate understanding of the C++ language, which can be incredibly helpful in scientific research

