



A Tasty Intro to Generic Programming in C++

A Presentation by Cliff Green

Sep 26, 2023, updated Nov 15, 2023

(minor update Apr 25, 2024, adding decimal library)

(Containing many references to food, templates, and an occasional gratuitous pic)

Cover photo by Christopher Holden from Albuquerque, United States - Ristras, CC BY-SA 2.0,
<https://commons.wikimedia.org/w/index.php?curid=95944700>

Theme is Framed Pastel Template by PresentationGO, <https://www.presentationgo.com>

An Overview of the Tasty Contents

- What is a type? Why is understanding types so important in C++?
- Operator function call syntax, why it is foundational in C++
- Ducks, or more specifically, duck typing
- Function templates, from basics to a nibble of concept constraints
- Functors and lambdas and closures, oh my!
- Class templates (just a taste), with examples from the C++ standard library
- Gratuitous photos that may pop up here and there
- Hand off for additional scrumptious morsels from Lou



Types - What Are They?

And why are they important?

Types Are an Essential Component of C++

- Traditional definition: a type has data (but not always) and a set of operations on that data
 - `int` in C or C++ is 32 or 64 (or other size) bits, and `+`, `-`, `*`, `/`, `%`, etc operations
- Types can be created in C++ with `class` or `struct` declaration
- Types also have names
 - Fundamental types (names are language keywords)
 - Name of the `class` or `struct`
 - Sometimes the name is anonymous (more on this later)
- Instantiation of a type creates an object
 - An object may be named (variable) or might be a temporary / unnamed object

A Type is an Important C++ Entity

- Types form the basis of compile time / static checking
- In C++, types are used in many ways besides simple object creation
 - Template / generic programming (focus of this presentation)
 - Function overloading
 - Encoding static / compile time design or constraints - e.g. units libraries
 - Metaprogramming - containers and algorithms on types (vs runtime values)
- Conditional logic, at compile time, based on type
 - `<type_traits>` header in C++ has extensive queryable info
 - e.g. `is_arithmetic`, `is_trivially_copyable`, `is_copy_constructible`
 - `if constexpr` allows enabling / disabling of statements at compile time
- Takeaway - types, in and of themselves, are a powerful design tool

Function Overloading by (Empty) Type - Using Types Never Meant to be Objects

```
struct bidirectional_iterator_tag { }; // simplified, see iterator_tags doc
struct random_access_iterator_tag : public bidirectional_iterator_tag { };
```

```
template <typename RAIter>
void sort_alg (RAIter begin, RAIter end, random_access_iterator_tag) {
    // sort with random access indices, note third parameter is unused
}
template <typename Iter>
void sort_alg (Iter begin, Iter end, bidirectional_iterator_tag) {
    // sort with bidirectional iterator, such as provided by std::list
}
```

```
std::vector v;
sort_alg(v.begin(), v.end(), random_access_iterator_tag{});
std::list ls;
sort_alg(ls.begin(), ls.end(), bidirectional_iterator_tag{});
```

Is C++ a Strongly Typed Language?

- Discussion - state arguments pro and con
- Consider another definition / comparison:
 - Dynamically typed languages - types are attached to values at run time
 - Statically typed languages - types are defined and constrained at compile time
- Both C and C++ have typecasting
- Do C and C++ differ in their use of types?

C++ Type System (From cppreference Page)

- Fundamental types
 - Arithmetic types
 - Integral: `bool`, char types, signed and unsigned `int`
 - Floating point: `float`, `double`, `long double`, fixed width floating (C++ 23)
 - `void`, `std::nullptr_t` (type of null pointer)
- Compound types
 - `class` / `struct`, `union`
 - Reference types (lvalue, rvalue)
 - Pointer types (ptr to obj, ptr to func, ptr to member data or func)
 - Array types
 - Function types
 - Enumeration types (scoped, unscoped)



Function Call Operator Overloading and Ducks

Does a duck quack?

What is Duck Typing?

- "If it walks like a duck and it quacks like a duck, then it must be a duck"
- Template / generic functions apply the duck test in a static typing context
 - All templates in C++ do some form of static typing “duck test”
 - Every type passed in to a template must pass the “duck test”
- More on the “duck test” requirements presented later

Function Call Operator Overloading

- Adding `ret-type operator() (parameter-list)` to a class declaration allows an object of its type to be treated like a function
 - Terminology is “function call operator overloading”
 - Can only be a member function (not a stand-alone operator overload)
 - Usual member function qualifiers apply (`const`, `noexcept`, etc)
- Allows an object to be used like a function
 - Functions don’t have local state (static local is global state)
 - An object *does* have state (one of the main purposes of an object)

Function Call Operator Overloading Example

```
struct cnt_cmp {  
    int cnt {0};  
    bool operator() (int a, int b) { ++cnt; return a < b; }  
};  
bool traditional_comp_func(int a, int b) { return a < b; }
```

```
cnt_cmp obj;  
obj(5, 10); obj(1000, 500); obj(0, 0);  
// what is the current value of obj.cnt?
```

```
std::vector<int> v { 5, 3, 1, 1 };
```

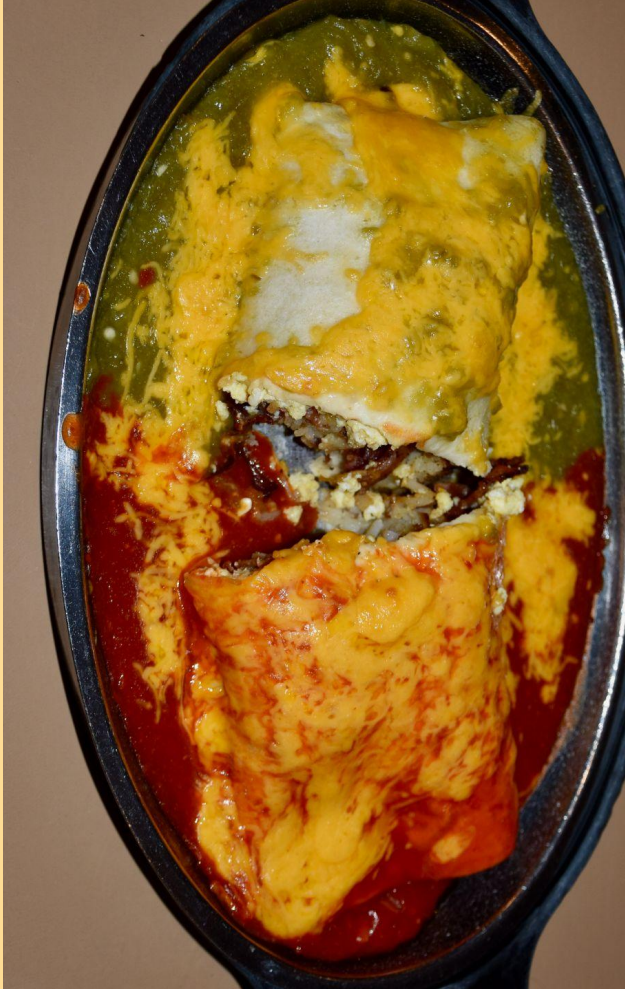
```
std::sort(v.begin(), v.end(), cnt_cmp{});  
std::sort(v.begin(), v.end(), obj);  
std::sort(v.begin(), v.end(), std::ref(obj));  
std::sort(v.begin(), v.end(), traditional_comp_func);
```

- What is the difference between the above four calls to `std::sort`?
- Contrast C++ std lib sort to C std lib sort

Gratuitous Photo 1

My company glider, a Puchacz SZD-50-3 (manufactured in Poland), tail number N503S, contest number 10, at Moriarty, NM





Function Templates

From basics to an advanced topic or two

A Simple Arithmetic Function with Two Parameters and a Return Value

```
int add_div_by_3 (int a, int b) {  
    return (a + b) / 3;  
}  
  
int tmp {20}; int result = add_div_by_3 (tmp, 30);  
// what are the types and what are the type requirements?  
  
// let's change to float and add constexpr  
constexpr float add_div_by_3 (float a, float b) {  
    return (a + b) / 3;  
}  
  
// note auto return type in calls, also types of literals  
float tmp {20.0}; auto result1 = add_div_by_3 (tmp, 30.0);  
auto result2 = add_div_by_3 (20.0, 30.0);
```

- Compare the assembly between the two calls in Compiler Explorer

Change the Code to a Function Template

```
template <typename T>
constexpr T add_div_by_3 (T a, T b) {
    return (a + b) / 3;
}
```

```
add_div_by_3 (20, 30); add_div_by_3(20u, 30u);
add_div_by_3 (20.0, 30.0); add_div_by_3(20.0f, 30.0f);
```

```
// in C++ 20, template syntax can be simplified (more on next slide)
constexpr auto add_div_by_3 (auto a, auto b) {
    return (a + b) / 3;
}
```

- What are the types in use now?
- How many functions are being generated?

Auto Template Parameters in C++ 20

```
constexpr auto add_sub_div (auto a, auto b) {  
    return (a + b) / (a - b);  
}
```

```
#include <complex>
```

```
auto res1 = add_sub_div (15, 44);  
auto res2 = add_sub_div(3.3f, 22.1f);  
auto res3 = add_sub_div(std::complex<double>{5.0, 2.0}, std::complex<double>{3.0, 4.0});  
// values of res1, res2, res3 are -2  -1.35106  (0.5,3.5)
```

- In C++ 20, template syntax can be simplified (called “abbreviated function template syntax”); `auto` for generic parameters came from lambdas, but added to templates
- Previous example does not compile: `<complex>` type defined for `operator /` with scalar (i.e. `3` should be of type `double`, or `3.0` instead of `3`)

Notes and Gotchas

- Compiler must have all template code available when instantiating a function template (i.e. when it is called) or a class template
 - Varying styles to achieve this (I put everything in same header file)
 - No longer `.hpp` / `.cpp` separation of declaration and definition
- ODR (one-definition-rule) can cause differences between functions and function templates
 - `inline` typically needed for (regular) functions fully defined in a header
 - Templates are not functions (or classes) so ODR does not apply
- Remember that a function template is a function generator, not a function (similar thought process for class template)

Tasty Subtleties

```
template <typename T>
constexpr T add_div_by_3 (T a, T b) {
    return (a + b) / 3;
}
```

```
auto res1 = add_div_by_3 (20, 30.0); // Does not compile!
// what would be the types of a, b, res1 if it did compile?
```

```
auto res2 = add_div_by_3<double>(20, 30u); // explicitly specify the type
// what are the types of a, b, res2 now? what happens with 20 and 30u?
```

- 3 (in the func impl) is `int` literal; what does that affect? What if it was 3.3 (`double` literal)?
- Parameter types are “deduced types” (i.e. based on the type of the argument when calling), unless explicitly specified as shown above

More Tasty Subtleties

```
template <typename T, int SZ>
constexpr T* gen_array () {
    return new T[SZ];
}
// what is this "int SZ" thingie in the template declaration?
auto* my_arr1 = gen_array<double, 20>(); // why can't sz be passed in as normal parm?
auto* my_arr2 = gen_array<std::string, 66>(); // what is the type of the pointer?
auto* my_arr3 = gen_array<double, 44>(); // different than the first func call? in
// what way?
```

- There are no “deduced types” for a return type, must be explicit
- Parameter types are “deduced types” (i.e. based on the type of the argument when calling), unless explicitly specified as shown above
- `SZ` is a non-type template parameter (more on this later)

Gratuitous Photo 2

Santa Fe 2926, a beautifully restored steam engine, on a short outing at Albuquerque, NM



Expand the Palate Beyond Fundamental Types

```
#include "math_utils/big_honkin_nums.hpp" // “bhn” namespace
```

```
template <typename T>  
constexpr T add_div_by_3 (T a, T b) {  
    return (a + b) / 3;  
}
```

```
bhn::big_num big_honkin_num1 { "77666555444333222111000888999" };  
bhn::big_num big_honkin_num2 { "555666777888999000555777888" };  
auto res1 = add_div_by_3 (big_honkin_num1, big_honkin_num2);
```

- How are `a` and `b` passed in to the function template?
- What operations and / or methods must the `big_num` type support? (4 used above - 2 inside function, 2 elsewhere)

A Savory Decimal Type

```
#include "decimal.h"

// same add_div_by_3 function template
auto a = decimal::decimal<2>{7.55}; // two decimal places
auto b = decimal::decimal<2>{14.44};
auto res1 = add_div_by_3 (a, b); // res == decimal::decimal<2>{7.33}

auto c = decimal::decimal<3>{5.111}; // three decimal places
auto d = decimal::decimal<3>{19.222};
auto res2 = add_div_by_3 (c, d); // res == decimal::decimal<3>{8.111}
```

- The `decimal` type supports addition and division (in addition to construction and copying)

Flexibility is Good (Two Template Params!)

```
template <typename N1, typename N2>
constexpr N1 add_div_by_3 (N1 a, N2 b) {
    return (a + b) / 3;
}

auto res1 = add_div_by_3 (20, 30.0); // now compiles! what are the types?
auto res2 = add_div_by_3<double, float>(20, 30);
auto res3 = add_div_by_3 (big_honkin_num1, 30.0);
```

- What is the value of `res1`? is it 16 or 16.667? what about `res2`?
- What does the big number type now need to support (in the way of operations)?
- Should the return type be `N2` instead of `N1`? Why?

Let's Add Even More Flexibility

```
template <typename N1, typename N2>
constexpr auto add_div_by_3 (N1 a, N2 b) -> decltype((a+b)/3) {
    return (a + b) / 3;
}

auto res1 = add_div_by_3 (20, 30.0); // what is the return type?
auto res2 = add_div_by_3<double, float>(20, 30);
auto res3 = add_div_by_3 (big_honkin_num1, 30.0);
```

- Allow the compiler to decide the return type based on the result type of the expression (note trailing return type syntax)
- Does this affect the operations that must be supported by big num type?

A Quick Taste of Requires

```
#include <complex>
#include <type_traits>

template <typename T>
constexpr std::complex<T> some_complex_math(std::complex<T> a, T b) {
    return a + b;
}

template <typename C, typename T>
requires (std::is_same_v<C, std::complex<T>>)
constexpr C similar_complex_math(C a, T b) {
    return a + b;
}

std::complex<float> x {3.0f, 4.0f};
some_complex_math(x, 5.0f); similar_complex_math(x, 5.0f);
```

- Both function template interfaces are similar, requiring a `std::complex` and a scalar (but compiler error messages may differ)

A Quick Taste of Concepts

```
#include <type_traits>

template <typename T>
concept big_math_capable = std::is_copy_constructible_v<T> &&
                           requires (T x) {
    x + x;
    x / x;
};

void math_func (big_math_capable auto a) {
    // do addition and division with a
}

// the above is equivalent to (other than naming T):
template <big_math_capable T>
void math_func(T a) {
}
```

- The concept requires a trait (copyable) and two valid expressions (add, div)

A Pause For Digestion

- Write one algorithm that can be used with multiple types! Type based decisions allow writing generic functions that can be used by multiple users (generic libraries)
- Many details and subtleties have not been covered, this presentation is for whetting the appetite
- Concepts, traits, requires, etc are all ways of constraining and making decisions at compile time based on the type
- Note the concept that requires two valid expressions (addition, division) - this is one form of “duck typing” (“if it adds like a duck and divides like a duck, then I’m going to use it like a duck”)

Functors and Lambdas and Closures

Oh My!



Remember Function Call Overloading?

```
template <typename Ctr, typename F>
void traverse(Ctr& container, F func) {
    for (auto& elem : container) {
        func(elem);
    }
}
```

```
void square_val(int& x) {
    x = x*x;
}
void incr_char(char& c) {
    c += 1;;
}
```

- Note template param `F` - func must a function *or a function object* that takes one argument (of container element type) and returns `void` (i.e. no return)

Similar to std library `std::for_each`

```
std::vector<int> v { 1, 3, 5, 7 };  
std::list<int> lst { 2, 4, 6, 8 };
```

```
traverse(v, square_val);  
traverse(lst, square_val);
```

```
std::string str { "Howdy!" };  
traverse(str, incr_char);
```

- Simple enough, one function template used with any container and multiple simple functions; but what if state is wanted?
- Iterators or C++ 20 ranges preferable to passing in whole container (why?)

Define a Functor Type With `operator()`

```
struct add_x {  
    int x { 0 };  
    void operator() (int& elem) { elem += x; x += 1; }  
};
```

```
traverse(v, add_x(42)); // add 42, then 43, then 44, etc to each element  
traverse(1st, add_x(11)); // add 11, then 12, then 13, etc to each element
```

- Remember the “duck rule” - `traverse` takes a container (1st arg), 2nd arg looks like a function, but isn't (in above example) - it's a function object
- The function object is first constructed, passed in (by value), then invoked through the `operator()()` overload on each element of the container

An Example With `std::sort`

```
#include <functional> // for std::ref or std::cref
struct cnt_cmp { // count number of comparisons
    int cmp { 0 };
    template <typename T>
    bool operator() (T a, T b) { ++cmp; return a < b; }
};

std::vector<int> v1 { 3, 5, 1, 7, -4, 55, 44 };
std::vector<double> v2 { 26.0, -2.0, -1.4, 0.5, 8.0 };
std::string str { "Howdy!" };

std::sort(v1.begin(), v1.end(), cnt_cmp{});
cnt_cmp cntr{};
std::sort(v2.begin(), v2.end(), std::ref(cntr)); // what is purpose of this line and line above?
std::sort(str.begin(), str.end(), cnt_cmp{});
```

- The `operator()()` overload is a member function template, can take any type as long as requirements are met (what are the two requirements?)

Member Function Template can be Written With `auto` in C++ 20

```
struct cnt_cmp { // count number of comparisons
    int cmp { 0 };
    bool operator() (auto a, auto b) { ++cmp; return a < b; }
}; // behaves exactly as before, valid only in C++ 20 and later
```

- `auto` works well (in C++ 20) as long as a type parameter name *is not* needed; e.g. if a type trait needs to be queried, `template <typename T>` (or similar) would be needed for `T` (or other name)
- How to access the function object state may require thinking or design (note the use of `std::ref` in the previous example)

What is a Lambda?

```
struct person {  
    std::string name;  
    unsigned int age;  
};  
  
std::vector<person> v { { "Cliff", 35u }, { "Lou", 77u }, { "Nathan", 23u } };  
std::sort(v.begin(), v.end(), // let's sort by age  
    [] (auto a, auto b) { return a.age < b.age; } );  
std::sort(v.begin(), v.end(), // now sort by name  
    [] (auto a, auto b) { return a.name < b.name; } );
```

- A functor (type with `operator()()`) or regular function can't always be defined close to where it is used; what if it is needed in only one place?
- A lambda is a nameless / anonymous functor, complete with parameters and implementation written as an expression (capture list coming up)

What About a Closure?

```
int cnt { 0 };
std::vector<person> v { { "Cliff", 35u }, { "Lou", 77u }, { "Nathan", 23u } };
std::sort(v.begin(), v.end(), // sort by name, but also count comparisons
          [&cnt] (auto a, auto b) { ++cnt; return a.name < b.name; } );
std::cout << "Count of name comparisons: " << cnt;
```

- Square brackets on a lambda define the capture list; the variables become “member data” of the lambda, this is sometimes called a closure
- Variables can be captured by value or reference; syntax allows capturing “all variables” in the current scope, or selecting specific ones
- C++ 20 allows specifying template parameters instead of only using `auto`
- Be careful capturing all variables, as well as scoping of the variables

Can a Lambda be Reused? What Are Other Cool Aspects of Lambdas?

```
auto lam = [] (auto a, auto b) { return a.age < b.age; };  
std::sort(v.begin(), v.end(), lam); // sort by age  
other_alg(lam);
```

- `lam` is of anonymous type known only to the compiler, so `auto` must be used
- `lam` can be used repeatedly as needed
- A lambda with no captures can be used as a regular function pointer
- Lambdas are `const` by default, `mutable` must be used in certain contexts (opposite of typical C++ syntax)
- The `this` pointer of an object can be passed in to a lambda, convenient in some contexts



Class Templates, Just a Nibble or Two

And please don't eat the cute Roadrunner!

A `struct` Containing Two Arbitrary Type Elements is Useful

```
template <typename T1, typename T2>
struct two_items {
    T1 first;
    T2 second;
};

using person = two_items<std::string, unsigned int>;
person cliff { "Cliff", 36u };
person lou { "Lou", 66u };
lou.second += 1u; // Lou just aged a year
```

- The example is essentially the `std::pair` class template in the std library

Generalize to an Arbitrary Number of Elements

```
template <typename... Types>
class many_items {
};
```

```
#include <tuple>
std::tuple<int, std::string, double> foo1 { 42, "Howdy!", 44.0 };
auto foo2 = std::make_tuple(42, std::string{"Howdy!"}, 44.0);

std::get<int>(foo1); std::get<std::string>(foo1); std::get<double>(foo1);
// same as:
std::get<0>(foo1); std::get<1>(foo1); std::get<2>(foo1);
```

- The `std::tuple` class template in the std library allows an arbitrary number of elements
- The `...` is called a “template parameter pack” and specifies 0 to N template types

Let's Define a Class to Provide “Maybe I Have a Value, Maybe I Don't” Functionality

```
template <typename T>
class my_opt {
    T val;
    bool is_present;
public:
    my_opt() : val(), is_present(false) { }
    my_opt(const T& v) : val(v), is_present(true) { }
    T& operator*() { return val; }
    const T& operator*() const { return val; }
    T* operator->() { return &val; }
    const T* operator->() const { return &val; }
    operator bool() const { return is_present; }
    // ...
};
```

- The `std::optional` class template provides this functionality

Function Returns an “Optional” That May or May Not Contain a Value

```
using my_opt_str = my_opt<std::string>;  
my_opt_str my_func () {  
    // ...  
    return some_cond ? my_opt_str("Howdy!") : my_opt_str();  
}
```

```
#include <optional>  
using opt_str = std::optional<std::string>;  
// exact same code as above except opt_str instead of my_opt_str
```

```
auto res = my_func();  
if (res) { // a std::string value is present  
    std::cout << "Result is " << *res;
```

- The `std::optional` class template more flexible and efficient - e.g. default construction (for “no value”) of contained type is not performed or required

Non-Type Template Parameters

```
template <typename T, std::size_t N> // inside std namespace
struct array {
    // implementation details
};
#include <array>
#include <algorithm>

std::array<int, 4> my_array { 46, 20, 44, 77 };
std::sort(my_array.begin(), my_array.end());
for (auto i : my_array) { // ... do something }
my_array[2] = 10; // access 3rd element of array, same syntax as built-in arrays
```

- `std::array` is a class template that encapsulates fixed size arrays
- Each declaration of `std::array` with a different `T` and / or `N` is a different type

C++ 20 Expanded What Can be Used as a Non-Type Template Parameter

```
template <double Percentage>
constexpr double calc (double val) {
    return val * Percentage;
}
```

```
calc<0.10> (100.0); // returns 10.0, calculated at compile time
double x { 300.0 };
calc<0.50> (x); // returns 150.0, calculated at runtime
calc<0.90> (100.0); // returns 90.0, calculated at compile time
```

- Non-Type TP can be a pointer type, function type, integer type, lvalue reference type, double type, and others

Gratuitous Photo 3

My son's cat named Stoney; my wife and I are proud grandparents of our grandcat.



Thank You and Final Thoughts!

- Two Essential Online Tools:
 - CPP Reference page: <https://en.cppreference.com/w/> - have it ready, learn to read / use it
 - Compiler Explorer: <https://godbolt.org/> - compile and run your code on multiple compilers, analyze the assembler output
- Decimal type from Tim Quelch, <https://github.com/TimQuelch/decimal>
- C++ templates define a turing complete “language within a language”
- Templates, just like all of C++, is a large and complex subject
- Templates provide a powerful ability to design libraries for multiple users / clients