1 STL

Some basic, but useful, uses of the STL which I keep forgetting the syntax for.

1.1 Tuples

```
vector<tuple<int, char>> v;
auto t = make_tuple(10, "str");
int num = get<0>(t);
string s = get<1>(t);
get<0>(t)++;
// will print 10 + 1
cout << get<0>(t);
```

1.2 Sets

```
// easy constructor to "convert" vector into set
unordered_set<int> s (vec.begin(), vec.end());

// deleting items while iterating through
for (it = s.begin(); it != s.end(); ) {
    if (want_to_delete)
        // erase returns iterator to element that follows
        // the one that was removed
        it = s.erase(it);
    else
        ++it;
}
```

1.3 Sorting a custom object Obj

```
struct Compare {
    bool operator() (const Obj &a, const Obj &b) {
        // return true iff a is less than b
    }
};
// sorts a vector of Obj in increasing order using a functor
sort(vec.begin(), vec.end(), Compare());
// sorts using lambda
sort(vec.begin(), vec.end(), [](const Obj &a, const Obj &b) {
    // return true iff a is less than b
});
```

There are many functions in the standard library, such as in `<algorithm>`, which take an optional (template) unary predicate, as above.

1.4 Priority queues

```cpp
// create a min priority queue (default without a compare function is a max pq)
priority_queue<int, vector<int>, greater<int>> pq;
// pq of custom objects with custom sort
auto custom_cmp = [](Obj &a, Obj &b) { /* return true iff a is bigger/smaller than b */
    };
priority_queue<Obj, vector<Obj>, decltype(custom_cmp)> pq(custom_cmp);
```

To find the k biggest things in a larger collection of things (eg an array), it may seem like we want a max pq but instead, if we use a min pq, we can remove the min when there are $k + 1$ things in the pq to ensure we only keep track of the top k.

1.4.1 Strings

```cpp
// removes last char from string in O(1)
my_str.pop_back();
// convert char c into a string
string c_as_a_str = string(1, c);
// convert int num into a string
string num_as_a_str = to_string(num);
// convert str into int num
int same_num = stoi(num_as_a_str);
```

1.5 Vectors

```cpp
// vector of size n filled with 1's
vector<int> v (n, 1);
// combines A and B into one vector
vector<int> A_and_B = A;
A_and_B.insert(A_and_B.end(), B.begin(), B.end());
// grid with n rows and m colums, filled with 0's
vector<vector<int>> grid (n, vector<int>(m, 0));
// creates vector from v[0], v[1], ..., v[4]
vector<int> my_new_vector (v.begin(), v.begin() + 5);
// creates empty vector if len is 0, bad alloc if len < 0
vector<int> my_new_vector2 (v.begin(), v.begin() + len);
```

1.6 Threading

```cpp
mutex m;
// unlocked upon destruction, cannot unlock, cannot be moved
lock_guard<mutex> locker(m);
// unlike lock_guard, allows you to unlock and re-lock
unique_lock<mutex> locker(m);
locker.unlock();
// construct with ownership (RAII) but not auto locked
unique_lock<mutex> locker(m, defer_lock);
// unique lock can be moved, lock guard cannot
unique_lock<mutex> locker2 = move(locker);
```

2 General Information

2.1 Lambdas

Introduced in C++11, a lambda constructs a closure, which is an anonymous function object capagble of capturing variables in scope. The general form is,

```
[ captures ]( params ) -> ret { body }
```

The captures is a comma-separated list of zero or more captures, optionally beginning with a capture default, which is either `&` to capture by reference or `=` by value. `*this` can be implicitly captured even if not listed if either capture default is present although it is always captured by reference even if the default is `=`. Lambdas use values at the time the lambda is defined, not when it is invoked. The parameters are similar to those in named functions, except default arguments are only allowed from C++14 onwards. The return type of a lambda can be deduced, but only when there is exactly one statement, and that statement is a return statement that returns an expression (an initializer list is not an expression). If you have a multi-statement lambda, then the return type is assumed to be void, unless otherwise specified. A simple example,

```
int a = 1, b = 1, c = 1;
auto f = [a, &b, &c]() {
    auto f2 = [a, b, &c]() -> int { a = 4; b = 4; c = 4; };
    a = 3; b = 3; c = 3;
    f2();
};
a = 2; b = 2; c = 2;
f(); // after this, a is 2, b is 3, c is 4

int num = [](){ return 3; }(); // immediately invoke the lambda
```

Before C++17, lambdas cannot be `constexpr` since only variable definitions, function/function template declarations, and literal static data member declarations were allowed to be `constexpr`. From C++17 onwards, lambdas can be used in `constexpr` contexts when it makes sense.

Since lambdas are function objects underneath, they can be inherited from.

2.2 Initialization

https://imgur.com/3wlxtI0

This table summarizes all the initialization possibilities in C++17.

	Default init `;`	**Copy init** `= value;`	**Direct init** `(args);`	**Value init** `();`	**Empty braces** `{}; = {};`	**Direct list init** `{args};`	**Copy list init** `= {args};`
Built-in types	Uninitialised. Variables w/ static storage duration: Zero-initialised	Initialised with value (via conversion sequence)	1 arg: Init with arg >1 arg: Doesn't compile	Zero-initialised	Zero-initialised	1 arg: Init with arg >1 arg: Doesn't compile	1 arg: Init with arg >1 arg: Doesn't compile
auto	Doesn't compile	Initialised with value	Initialised with value	Doesn't compile	Doesn't compile	1 arg: Init with arg >1 arg: Doesn't compile	Object of type std::initializer_list
Aggregates	Uninitialised. Variables w/ static storage duration: Zero-initialised***	Doesn't compile	Doesn't compile (but will in C++20)	Zero-initialised***	Aggregate init**	1 arg: implicit copy/move ctor if possible. Otherwise aggregate init**	1 arg: implicit copy/ move ctor if possible. Otherwise aggregate init**
Types with std::initializer_list ctor	Default ctor	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Default ctor	Default ctor if there is one, otherwise std::initializer_list ctor	std::initializer_list ctor if possible, otherwise matching ctor	std::initializer_list ctor if possible, otherwise matching ctor****
Other types with no user-provided* default ctor	Members are default-initialised	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Zero-initialised***	Zero-initialised***	Matching ctor	Matching ctor****
Other types	Default ctor	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Default ctor	Default ctor	Matching ctor	Matching ctor****

*not user-provided = not user-declared, or user-declared as `=default` *inside* the class definition
**Aggregate init copy-inits all elements with given initialiser, or value-inits them if no initialiser given
***Zero initialisation zero-initialises all elements *and initialises all padding to zero bits*
****Copy-list-initialisation considers explicit ctors, too, but doesn't compile if such a ctor is selected

Braces vs parentheses:

There's only a few differences since C++20:

Braces means the arguments are guaranteed to be processed in order from left to right, and the constructor taking `std::initializer_list` will be called whereas with parentheses, you would have to do something like `Foo f ({a, b})` to get this constructor to be called. As well, braces do not allow narrowing, meaning that `unsigned char x{300}` will fail to compile. Lastly, braces avoid the most vexing parse.

2.2.1 Initializer lists

2.2.2 Class member initialization

2.2.3 Aggregate initialization

An aggregate class is one with no: user-defined constructors, private/protected non-static data members, base classes, virtual functions, and in-class initializers. These classes can be initialized in a special way using a brace-enclosed comma-separated list of initializer-clauses.

```
struct A { int v[3]; };
struct B { int n1, n2; };

B b = {10, 20}; // b.n1 is 10, b.n2 is 20
B b2 = {.n2 = 20, .n1 = 10};

// shortcut if the class has only 1 member:
A a = {3}; // equivalent to: A a = {{3}}, exterior braces not needed
```

2.3 References

References are a type of variable that acts as an alias for another variable or value and cannot be re-assigned to after initialization. There are three types of references: reference to non-const values, reference to const values, references to r-values. Note that a reference itself is an lvalue. You cannot have pointers to references but can have references to pointers.

```
int a = 3;
int &b = a;
int c = 4;
b = c; // a is now 4 and &a == &b always
int &d; // invalid, reference must be initialized on construction
int &&r = 3; // r value reference
```

A reference is implemented in assembly as a pointer, which takes up memory on the stack, but when you ask for its address, the address of the variable it is referring to is given, rather than the address that it itself takes up. So, returning by reference is valid (in fact necessary for things such as `ostream &operator<<(ostream & os, T thing_to_print` to allow chaining) but returning a reference to a local, destroyed variable is a problem just like returning the address of a local, destroyed variable is.

2.3.1 Collapsing

Since C++11, the reference collapsing rules dictates type deduction for references to references,

```
A& & becomes A&
A& && becomes A&
A&& & becomes A&
A&& && becomes A&&
```

`A&& &` means an lvalue reference to an rvalue reference. As a programmer, you cannot write code like this, it is just a representation of what the compiler sees. Only an rvalue reference to an rvalue reference results in another rvalue reference, the other cases all deduce to an lvalue reference.

2.3.2 Perfect forwarding

Consider,

```
void foo(MyVec v); // assume MyVec has move and copy constructors

template <typename T> void relay(T arg) { foo(arg); }

int main() {
    MyVec reusable = makeVec();
    relay(reusable);
    relay(makeVec());
}
```

Ideally, we want an rvalue to be forwarded as an rvalue from `relay` to `foo` and lvalue to be forwarded as an lvalue. However in the above code, `arg` in `relay` is itself an lvalue (rvalue references are themselves lvalues), so lvalues will be given in both cases to `foo`. This is a problem because `relay(makeVec())` invokes a copy to copy from `relay` to `foo`, when it should have used move semantics to do two moves. To solve this, consider,

```
template <typename T> void relay(T&& arg) {
    foo(std::forward<T>(arg));
}

// implementation of std::forward, two overloads
```

```cpp
template <typename T> T&& forward(typename remove_reference<T>::type& arg) {
    return static_cast<T&&>(arg);
}

template <typename T> T&& forward(typename remove_reference<T>::type&& arg) {
    return static_cast<T&&>(arg);
}
```

`T&&` is called a **universal reference/forwarding reference** because `T` is a template type and reference collapsing happens to `T`.

Suppose we call `relay(9)`. 9 is an rvalue, so `T` is `int&&` and `T&&` is `int&& &&`. From the rules above, this gives `int&&`, an rvalue reference. If we call `relay(x)`, where `x` is either an lvalue or an lvalue reference (both are themselves lvalues), the deduction rules give `int&`. So, the universal reference allows lvalue references to be kept as lvalue references, and same for rvalues. Combined with `std::forward`, the reference type given to `relay` is now the same as given to `foo`. Perfect forwarding has two steps: receive a universal reference (which is what `relay` does), then forward using `std::forward`. The first overload is called if `std::forward` is called with an lvalue and the second overload is used if called with an rvalue.

In summary, `std::move<T>(arg)` turns `arg` into an rvalue type, `std::forward<T>(arg)` turns `arg` into type `T&&` (which is either an lvalue or rvalue reference, depending on what the value given as `arg` was originally). Perfect forwarding is useful to avoid unnecessary copying and overloads for lvalue/rvalue references.

2.4 Namespaces

2.4.1 Unqualified name

An unqualified name is one which does not appear to the right of a `::` operator. There are many different situations for lookup, but three examples are,

1. If the name is used in the global/top-level scope, outside of any user-declared namespace, then lookup occurs in the global namespace
2. If the name is used in a user-declared namespace outside of any function or class, lookup occurs in this namespace before the use of the name, then enclosing namespaces before the declaration of this namespace, etc. until the global namespace is reached.
3. If the name is used in the definition of a namespace-member variable outside the namespace, lookup is the same as for a name used inside the namespace. So,

```cpp
namespace X {
    extern int x; // declaration
    int n = 1;
};
int n = 2;
int X::x = n; // set to X::n value of 1 rather than 2
```

2.4.2 ADL

Argument-dependent lookup/Koenig lookup is the lookup of an unqualified function name depending on the types of the function arguments. Namespaces which would usually not be considered for lookup might be searched by the compiler and the overall set of declarations discovered during ADL to resolve the function name is the union of the normal lookup and those found by looking in namespaces associated with the function argument types. An example of its use is,

```
namespace ns {
    class test {};
    void func(test t);
}

ns::test obj; // global

int main() {
    func(obj); // ns::func is called even without the ns:: because of ADL
}
```

This provides convenience to the programmer, however may cause unexpected behavior,

```
std::swap(a, b); // correct
using std::swap;
swap(a, b); // surprise, if a and b are in namespace ns, this will call ns::swap
```

2.5 Decay

array to pointer, function to pointer decay

2.6 Other

2.6.1 Translation unit

A translation unit (or, compilation unit) is the final input to a C/C++ compiler from which an object file is generated. It roughly is the source file after the preprocessor does its processing. The `static` keyword indicates that the variable/function it is attached to cannot be used in other translation units and the `extern` keyword indicates that the variable/function may be used in this translation unit, even without a definition. The linker manages these issues.

2.6.2 const vs constexpr

Variables which are `const` and `constexpr` both are constant and cannot be modified but `constexpr` variables are additionally a compile-time constant and must be initialized at compile time (thus can be used in template meta-programming, `static_assert`, etc. However, `const` variables may be initialized at compile or run time.

2.6.3 typedef vs using

`using` was introduced in C++11 and declares a type alias, just like `typedef`. They are basically the same according to the standard: "[alias-declaration] has the same semantics as if it were introduced by the typedef specifier. In particular, it does not define a new type and it shall not appear in the type-id.". One small difference is that `using` is nicer for templates (creating alias templates): `template <typename T> using Vec = std::vector<T>`. This is possible with `typedef` but is complicated.

2.6.4 lvalue, xvalue, glvalue, rvalue, prvalue

An lvalue designates a function or object and historically could appear on the left side of an assignment expression. An xvalue (expiring value) also refers to an object, usually near the end of its lifetime, so its resources may be moved. For example, the result from calling a function which returns an rvalue reference is an xvalue. A glvalue (generalized lvalue) is an lvalue or xvalue. An rvalue is an xvalue, a temporary object, or a value not associated with an object and historically could appear on the right side of an assignment expression. A prvalue (pure rvalue) is an rvalue that is not an xvalue. For example, the result from calling a function which returns something that is not a reference is a prvalue.

2.6.5 void parameters

In C++, `int func(void)` means the same thing as `int func()` – a declaration for a function which takes mo arguments. However, in C K&R, the latter means a function which takes an unspecified number of arguments of unspecified type and parameter checking is turned off.

2.6.6 inline

This keyword is a hint to the compiler to inline the function but does not guarantee anything. Recall from CS 241 that the compiler will place the inline function at the point of call and does not need to generate code to pass parameters. However, if the function is called many times, this can bloat the compiled binary because the function needs to be placed at every call location. Recursive functions may be inlined by unrolling to a certain depth and a separate function is called if the depth is exceeded to finish the computation.

2.6.7 auto

2.6.8 nullptr

This keyword is a pointer literal which specifies a null pointer value. It was introduced to replace `NULL` from C to improve readability/teachability and because `NULL` has the value 0 and this causes issue with operator overloading (ambiguity between int and pointer, for example).

3 Object Oriented

3.1 Basics

Every class comes with a default, zero-argument constructor, and default copy ctor, copy assignment operator, dtor, move ctor, and move assignment operator.

Constructor: If a custom constructor is made, then we can no longer use the default constructor. When a custom ctor is called, space is allocated, fields are constructed (without any values), then object fields call their own ctors, and lastly, the ctor body runs. Thus, `const` and references need to be constructed in the member initialization list (MIL) since they are immutable. The order of initialization is in the order of declaration in the class, not in the order in the MIL. MIL is also more efficient since instead of calling ctors for each field then copying the right values in, we just construct with these values in the first place.

When the ctor of a subclass runs: space is allocated, superclass fields are constructed (invoking the default ctor of the superclass), the subclass's own fields are constructed, then finally the ctor body runs. So, if the superclass has private fields and no zero-argument/default ctor, it must be constructed in the MIL because the superclass ctor is needed to initialize superclass fields, both because they are private and because there is no default superclass ctor.

Copy constructor: The copy ctor is called in three different occasions: an object is initialized by another object (eg `Car my_car = other_car`), an object is passed by value, or an object is returned by value. For a node class making up a linked list, looks like,

```
Node(const Node &other) {
    data = other.data;
    next = other.next ? new Node(*(other.next)) : nullptr;
}
```

Note it must take the parameter as a const ref because passing by value will cause infinite recursion (copying the parameter invokes the copy ctor which invokes the copy ctor, etc).

Copy assignment operator: Must support self assignment. Typically looks like,

```
// version 1: simplest
Node &operator=(const Node &other) { // pass by const ref to prevent copy in parameter
    if (&other == this) return *this; // self-assignment
    data = other.data;
    delete next; // clean up old fields before creating a copy of new data
    next = other.next ? new Node(*(other.next)) : nullptr;
    return *this; // returning *this allows for chaining of assignment, eg a = b = c
}

// version 2: using copy constructor and swap
Node &operator=(const Node &other) {
    Node tmp = other; // deep copy with copy ctor
    std::swap(data, tmp.data);
    std::swap(next, tmp.next);
    return *this;
    // dtor will destroy tmp when out of scope
}
```

Destructor: Default behavior is shallow, just like the copy constructor. When an object is destroyed, the steps are: dtor body runs, dtors are invoked for fields that are objects, in reverse declaration order (every object comes with its own dtor), and lastly, space is deallocated.

```
~Node() { delete next; }
```

Move constructor: Is invoked when constructing from an rvalue (eg `Node n = give_node()`), the result of `give_node` is an rvalue, which is somewhere in memory and has an address which is unknown to us. To construct the current object, we can just steal its values (shallow copy), then invalidate the fields in the rvalue so that when the dtor is called on the temporary result at the end of the move ctor, it can be safely destroyed without affecting the current object.

```
Node(Node &&other) : data{other.data}, next{other.next} {
    other.next = nullptr;
}
```

Move assignment operator: Similar to copy assignment operator but we don't need to care about messing with `other`'s data since it is a temporary that will be destroyed immediately anyway. So a swap (shallow copy) is sufficient.

```
Node &operator=(Node &&other) {
    swap(data, other.data);
    swap(next, other.next);
    return *this;
    // other is now filled with garbage values but that's fine because it will
    // be destroyed now since it is out of scope
}
```

If move ctor/move assignment operators aren't defined, then the copying versions are used instead. This is fine since a const lvalue ref can take in an rvalue ref. Move operations are more efficient and useful when the argument is a temporary value. Modern compiles usually also have copy elision which will construct in place (eg instead of calling a copy constructor on a return by value then moving into the right place, just construct immediately).

These custom operators and constructors are typically used when shallow copying is insufficient. Thus, the rule of five says that if you define one of these, you probably need all five.

3.2 Virtual functions

Virtual functions are functions in a base class which derived classes can override to get appropriate behavior of an object through a pointer. The goal is to get "late binding" (also called dynamic binding or dynamic dispatch), which is when the method which is used gets decided at runtime based on the type of the pointed-to object (what it was originally constructed as) compared to early/static binding, which is when the method is chosen at compile time based on the type of the pointer you call through. Virtual calls are more expensive than regular function calls which is why early binding is the default since C++ philosophy is fast by default.

```
class Super {
public:
    virtual void func() { cout << "in super" << endl; }
};
class Sub : public Super {
public:
    // override keyword not necessary but good style
```

```
    void func() override { cout << "in sub" << endl; }
};

Sub s;
s.func(); // prints "in sub"
Super *p = &s;
p->func(); // prints "in sub", without the virtual, would have printed "in super"

// assigning derived to a base (other way not possible)
Super super = s;
super.func(); // prints "in super", because of object slicing
Super &super_ref = s;
super_ref.func(); // again prints "in sub", object slicing does not occur with refs or
    ptrs
```

As well, generally any class with virtual functions should also have a virtual destructor, so that when an object is deleted through a superclass pointer, its destructor can be called to clean up the appropriate fields which are not present in the superclass.

Implementation of virtual functions: For every class that contains virtual functions, the compiler constructs a virtual table (vtable), which contains an entry for each virtual function accessible by the class and stores a pointer to the definition of this function. Only the most specific function definition callable by the class is stored in the vtable. In the example above, the vtable for `Sub` would point to the override version, not the superclass version. So, entries in the vtable point to either functions declared in the class itself or virtual functions inherited from a base class that it did not override. There is a single vtable for each class and it is shared by all instances.

In addition, there is also a vpointer for each class which points to the vtable of that class. This is a class member which the compiler adds (thus increasing the size of every object that has a vtable by `sizeof(vpointer)`. Now, when a virtual call is performed, the vpointer of the object is used to find the corresponding vtable of the class, and the entry in this table will point to the appropriate function to call.

Pure virtual: A virtual method with optional implementation (usually no implementation except for dtors). A class with at least one pure virtual method is abstract, meaning it cannot be instantiated. All inheriting subclasses will also be abstract until they override (implement) all the inherited pure virtual functions. According to good OO design, the superclass should always be abstract.

```
class AbstractSuper {
public:
    virtual ~AbstractSuper() = 0; // this makes AbstractSuper an abstract class
};
class Derived : AbstractSuper {
public:
    ~Derived() override { cout << "in derived dtor" << endl; }
};
// must implement pure virtual dtor
AbstractSuper::~AbstractSuper() {
    cout << "in abstract dtor" << endl;
}
```

4 Bites and Bytes

The C++ standard does not specify the exact sizes of the various fundamental types but does guarantee:

`1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`

There is a macro `CHAR_BIT` which defines the number of bits in a byte. In almost all cases, there are 8 bits in a byte. To count the number of bits in a byte,

```
// easiest way to see why this works is to draw out an example, eg 8 bits: 11111111
unsigned int count_bits(void) {
    unsigned char c = ~0u;
    unsigned int count = 0u;
    while (c) {
        ++count;
        c &= c - 1u;
    }
    return count;
}
```

A byte is the smallest addressable unit of memory. A word is a machine-specific grouping of bytes. In 32-bit architecture, that is 4 bytes (32 bits) and in 64-bit, that is 8 bytes. The base-16 representation is hexadecimal, consisting of numbers from 0 to 9 and letters a to f to represent numbers 10 to 15 in decimal. Each hex digit is 4 bits. For example, 0x1 is 0001 and 0xa is 1010. The most significant bit is the left-most bit (highest value/sign bit), the least significant bit is the right-most.

Endianness defines the byte orders in memory. Little endian (eg Intel x86) is where the least significant bytes is stored at the smallest address. Big endian (eg System/161) is the opposite and is probably what you are logically used to. Little endianness is a legacy of older processors, since it was slightly more efficient for some math operations and also required fewer instructions for casting values. To determine whether the system we are on is little or big endian,

```
bool is_little_endian(void) {
    // if little endian, smallest byte should be 1 and rest 0's, opposite for big endian
    unsigned int num = 1;
    // cast to char * so that when we dereference we get the values in first byte
    char *p = (char *)&num;
    return *p == 1;
}
```

`uintptr_t` is an unsigned integer type that is capable of storing a data pointer (same size as a pointer). A common reason to want an integer type that can hold an architecture's pointer type is to perform integer-specific operations on a pointer, or to obscure the type of a pointer by providing it as an integer "handle". A variable `a` is contained in memory addresses `[uintptr_t(&a) ... uintptr_t(&a) + sizeof(a) - 1]`.

5 Templates

5.1 Basics

Templates are primarily used for generic programming and template meta-programming. When the name of a template is used where a function/type/variable is expected, the compiler will instantiate (create) the expected entity from that template. So, function templates cannot be called, only instantiations of a function template can be called. This is why if a file is to be compiled (and potentially linked against by other binaries), and it includes a template definition, it must also have at least one instantiation, otherwise the template will never be created by the compiler.

As a result, a common structure is to have template class definitions in libraries were in `.H` files and function implementations in `.hxx` files. User header files then included the `.H` and user `.cpp` files included the `.hxx` and also instantiate the templates. However, this exposes the `.hxx` files so to prevent this, the library provider can have a "dummy" `.cpp` file that includes all the appropriate `.hxx` and instantiations and is compiled as part of the library, which users link against. Overall, either way, the `.hxx` cannot themselves be compiled and linked against, because this would not force any instantiations.

A tricky situation is when a class template has a function template which has different template parameters from the class. At the point of class instantiation, the function templates cannot be deduced; they are only known when the function template is itself instantiated (even if there are default, concrete types). To prevent linker errors, the function templates need to be instantiated themselves, in the same way that class templates are.

A **dependent name** is one that depends on a template argument. For templates, there is a distinction between the point of definition of the template and point of instantiation. Dependent names are not bound until instantiation whereas nondependent names are bound at the point of definition. For example,

```
template <typename T> int add( T x ) { return num + x.to_int(); }
```

`num` is a nondependent name, `x` is dependent. Thus, the definition of `to_int` must appear before `add` is used but not necessarily before this definition of `add`. Also,

```
template<typename T>
struct X : B<T> // B<T> is dependent on T
{
    typename T::A* pa; // T::A is dependent on T
    void f(B<T>* pb) {
        static int i = B<T>::i; // B<T>::i is dependent on T
        pb->j++; // pb->j is dependent on T
    }
};
```

Deduction is the process of determining the type of a template parameter from a given argument and applies to function templates, auto, partial specialization, etc. For example, if the function template `template <typename T> void f(std::vector<T>)` is called with a `std::vector<int>`, then `T` is deduced as `int` and `f` is specialized as `f<int>`. In order for this to work, the template parameter type that is to be deduced must be in a **deducible context**. Here, the parameter to `f` is one such deducible context. This means that an argument in the function call expression allows us to determine what `T` should be in order for the call expression to be valid.

Non-deduced contexts are ones where no deduction is possible. An example is a template parameter that appears to the left of a `::`. For example,

```
template <typename S>
struct Outer { using inner_alias = void; };

template <typename S>
struct is_inner_alias : std::false_type {};

template <typename S>
struct is_inner_alias<typename Outer<S>::inner_alias> : std::true_type {};
```

This will not compile because the template parameter `S` is not deducible in the specialization. It can be anything. There is no "backwards correspondance" between arbitrary types and `typename Outer<S>::inner_alias` and generally no relationship between class template parameters and class members, so no sensible argument deduction can be made. More on this in the `Other` section.

A **non-type parameter** is one that is substituted by a value rather than a type and can be any of the following: a value that has an integral type or enumeration, a pointer/reference to a class object/function/class member function, or `std::nullptr_t`. Others (such as `std::string`) are not allowed because these non-constant expressions cannot be parsed and substituted during compile-time since they can change during runtime and would require a new template during runtime, which is not possible since templates are a compile time concept. Non-type parameters are very useful to do computation for meta-programming.

5.1.1 decltype (since C++11)

`decltype` takes an expression (a sequence of operators and operands, possibly containing comma operators) and inspects its type. With comma operators, the expression is evaluated left to right, where the left items is discarded and the type and value of the result are the type and value of the right-most item. So, `std::is_same<decltype(42, 3.14), double>::value` is true.

5.1.2 std::declval (since C++11)

This function converts any type to a reference type. Specifically,

```
template<class T>
typename std::add_rvalue_reference<T>::type declval() noexcept;
```

The reason an rvalue reference is added rather than an lvalue reference is because of reference collapsing. By adding an rvalue reference, we have the following result,

```
std::declval<Foo>() is of type Foo&&
std::declval<Foo&>() is of type Foo& // Foo& && collapses to Foo&
std::declval<Foo&&>() is of type Foo&& // Foo&& && collapses to Foo&&
```

With adding an lvalue reference,

```
std::declval<Foo>() would be of type Foo&
std::declval<Foo&>() would be of type Foo& // Foo& & collapses to Foo&
std::declval<Foo&&>() would be of type Foo& // Foo&& & collapses to Foo&
```

This is problematic because type `Foo&&` would never be possible. `std::declval` is often used with `decltype` to access member functions without going through constructors,

```
struct Default { int foo() const { return 1; } };
struct NonDefault
{
    NonDefault() = delete;
    int foo() const { return 1; }
};

decltype(Default().foo()) n1 = 1; // type of n1 is int
decltype(NonDefault().foo()) n2 = n1; // error: no default constructor
decltype(std::declval<NonDefault>().foo()) n2 = n1; // type of n2 is int
```

5.1.3 Arrow operator

Since C++11, there are two equivalent ways to declare a function,

1. `int name(int arg);`
2. `auto name(int arg) -> int;`

An example of its use if the return type depends on the type of template arguments,

```
template <typename T>
decltype(a) func(T a); // error because a is not known until the argument list

template <typename T>
decltype(std::declval<T>()) func(T a); // works but is messy

template <typename T>
auto func(T a) -> decltype(a); // works using the arrow operator
```

Since C++14, it is also possible to specify an `auto` return type without explicitly giving the return type using the arrow operator, so long as the function is fully defined before use and all return statements deduce to the same type.

5.1.4 typename

The `typename` keyword is used to tell the compiler that what follows is a type. This is needed because there is sometimes ambiguity during name lookup (when the compiler tries to associate a name with its declaration). For example, `t*f;` can either be multiplying `t` with `f`, or declaring a `t` pointer named `f`, depending on whether `t` is a type or not. Or similarly, what about `t::x*f`, where `t` is a template type? Then `t::x` can either be a static data member or a nested class. `t::x` is a dependent name (see earlier sections for more on this) and its meaning will not be discovered until the template `t` is instantiated. The standard says that `t::x` will be assumed to not be a type unless the `typename` keyword is used.

This keyword can also only be used for qualified names (one which appears to the right of the `::` operator). Unqualified names are assumed to be types.

5.1.5 name::template

Consider `boost::function<int()> f;` and,

```
namespace boost { int function = 9; }
int main() {
    int f = 5;
```

```
    boost::function<int()> f;
}
```

Then, `boost::function<int()> f` will use the `<` and `>` operators to compare `boost::function`, which is 9, against `int()` against `f`, which is 5. This is unlikely what you want, but the compiler cannot know, so the standard says that if the name is a template name, the `<` is taken to be the beginning of a template argument list rather than as the `<` operator but also that the name of a member template specialization appearing in this situation is assumed to be a non-template. So, the `template` keyword must be used to indicate that it is a template name.

```
template <typename T>
struct Outer {
    template <typename U> struct Inner { static constexpr int num = 30; };
    template <typename U> static void print() {
        std::cout << "PRINTING\n";
    }
};

template <typename T>
void func() {
    T::template print<double>(); // note the use of template keyword
    std::cout << T::template Inner<double>::num; // not just for functions
}

int main() {
    func<Outer<int>>();
    Outer<int>::print<double>(); // not dependent so template keyword is not needed here
}
```

This keyword is also sometimes useful after the `this` keyword,

```
template <typename T>
struct Super {
    template <typename U> void print() { std::cout << "PRINTING\n"; }
};

template <typename T>
struct Outer : Super<T> {
    template <typename U> void print2() { this->template print<U>(); }
};

int main() {
    Outer<int> t; t.print2<double>();
}
```

It is needed here because `Super<T>` is a dependent name and `print` is a member of `Super<T>`. The reason `this->` is even necessary in the first place is discussed later in the section about derived class templates.

5.1.6 typename vs class

`template <typename T>` is the same as `template <class T>`. General advice is to use `class` if `T` is always expected to be a class, and `typename` for other types (such as int, pointers, etc). `typename` also has other uses (see above).

5.1.7 Nested templates

```
template <typename T>
struct test {
    template <typename A> // function template in class template
    void func(A a);

    template <typename B> // class template in class template
    struct inner {};
};

template <typename T>
template <typename A>
void test<T>::func(A a) { /* defn */ } // invoked as: test<int> t; t.func<double>(0);

template test<int>; // declaration, specific instantiation
template void test<int>::func<double>(void); // specific instantiation
```

5.2 Specialization

Template specialization is a way to get specific behavior for a specific data type.

```
template <typename T>
struct test : std::true_type {};

template <>
struct test<int> : std::false_type {};

static_assert( !test<int>::value ); // instantiates specialization
static_assert( test<double>::value ); // primary template
```

TODO: partial specialization is not allowed for functions, something to do with messing up overloading

5.2.1 Specialization of nested class template

It is not possible to specialize a nested class template without specializing the outer class. The following does not compile,

```
template <typename T>
struct outer {
    template <typename S>
    struct inner;
};

template <typename T>
template <typename S>
struct outer<T>::inner { // primary class template definition
    S data;
};
```

```cpp
template <typename T>
template <>
struct outer<T>::inner<int> { // specialization does not compile
    int my_other_data;
};
```

There is no particular technical reason why it cannot be done but it can lead to unexpected behavior, depending on how the outer class is specialized and defined. To get around this, introduce a dummy template parameter with a default value then do partial specialization (which is allowed),

```cpp
template <typename T>
struct outer {
    template <typename S, char dummy = 'a'>
    struct inner;
};

template <typename T>
template <typename S, char dummy>
struct outer<T>::inner {
    S data;
};

template <typename T>
template <char dummy>
struct outer<T>::inner<int, dummy> {
    int my_other_data;
};
```

5.2.2 Template template specialization

```cpp
template <typename T>
struct X {};

template<typename T>
struct Z : std::false_type {};

// specialization on the template template parameter with arbitrary T
template<typename T>
struct Z<X<T>> : std::true_type {};

// template template parameter required
template <template<class> class C>
struct Z<C<int>> : std::true_type {};

static_assert(!Z<Z<double>>::value ); // first Z
static_assert( Z<X<double>>::value ); // second Z
static_assert( Z<Z<int >>::value ); // third Z
static_assert( Z<X<int >>::value ); // error: ambiguous between 2nd and 3rd Z
```

5.3 Variadic templates

Parameter packs, variable template

5.4 Meta-programming

Template meta-programming uses template instantiation to drive compile-time evaluation. The goal is to improve source code flexibility and runtime performance. Unlike runtime programming, meta-programming does not allow for mutability, virtual functions, other runtime type information, etc. A basic example for computing factorials,

```
template <unsigned N>
struct fac { static constexpr int value = N * fac<N - 1>::value; }; // initializing
template <>
struct fac<1> { static constexpr int value = 1; }; // specialization used as base case
// usage: static constexpr result = fac<10>::value;
```

The result is wrapped in a struct. Alternatively, functions marked `constexpr` are evaluated at compile time when all its arguments are constant expressions and the result is used in a constant expression as well. Otherwise, it might be evaluated at runtime despite being marked `constexpr`. However, metafunctions are more powerful than `constexpr` functions because they can take types as an argument. For example, `sizeof` operates on types. An example of computing the rank ("dimension") of an array type,

```
template <class T>
struct rank { static constexpr size_t value = 0u; };
template <class U, size_t N> // partial specialization for array types
struct rank<U[N]> { static constexpr size_t value = 1u + rank<U>::value; };
using array_t = int[10][20][30];
// usage: static constexpr result = rank<array_t>::value; // 3u
```

Specializations are very powerful. Another example for removing `const`,

```
template <class T>
struct remove_const { using type = T; };
template <class U>
struct remove_const<const U> { using type = U; }; // specialization for const types
// usage: remove_const<T>::type t;
```

As convention and similar to the examples above, metafunctions with a type result should be aliased to `type` and values aliased to `value`.

During template instantiation, the compiler will:

1. Figure out the template arguments. Take verbatim if explicitly supplied, else (for function templates), deduce from function arguments at point of call, else taken from the declaration's default template arguments.
2. Replace each template parameter by its corresponding template argument.

If the second step produces well-formed code, the instantiation succeeds but otherwise, it is considered not viable and silently discarded but is not an error; **SFINAE** (Substitution Failure Is Not An Error). For example,

```
struct test {
    int a = 10;
```

```
    //int b = 20;
};

template <typename T>
void func(std::true_type tt, T &t) {
    std::cout << t.a << std::endl;
}

template <typename T>
void func(std::false_type ft, T &t) {
    std::cout << t.b << std::endl; // test does not have a b member
}

int main() {
    test t;
    func<test>( std::integral_constant<bool, true>{}, t );
}
```

Another example,

```
struct Test { using foo = int; };

template <typename T>
void f(typename T::foo) {}

template <typename T>
void f(T) {}

int main() {
  f<Test>(10); // calls the first one
  f<int>(10);  // calls the second, without error
}
```

5.4.1 STL examples

`std::enable_if` gives: "if true, use the given type, if false, use no type at all". One possible implementation,

```
template <bool B, class T = void>
struct enable_if {};

template <class T>
struct enable_if<true, T> { using type = T; }; // partial specialization for true
```

Then, `std::enable_if<false, T>::type`, or, `std::enable_if_t<false, T>` is not an error because of SFINAE. Values can be wrapped in a type using `std::integral_constant`. The rank example from above now looks like,

```
template <class T>
struct rank : std::integral_constant<size_t, 0u> {};

template <class U, size_t N>
struct rank<U[N]> : std::integral_constant<size_t, 1u + rank<U>::value> {};
```

Convenient aliases for bools,

```
using true_type = std::integral_constant<bool, true>;
using false_type = std::integral_constant<bool, false>;
```

Determining whether two types are the same, using `std::is_same` which is possibly implemented as,

```
template <class T, class U> struct is_same : std::false_type {};
template <class T> struct is_same<T, T> : std::true_type {};
```

Determining whether a type is one of a variable amount of times (parameter pack of types), not in STL,

```
template <class T, class... T0toN>
struct is_one_of;

template <class T>
struct is_one_of<T> : false_type {}; // specialization for empty pack

template <class T, class... T1toN>
struct is_one_of<T, T, T1toN...> : true_type {}; // T is at the front

template <class T, class T0, class... T1toN>
struct is_one_of<T, T0, T1toN...> : is_one_of<T, T1toN...> {}; // mismatch at the front
```

Specializations are very useful for "recursive" metafunctions.

There are four functions (from C++11) whose operands are never evaluated, not even at compile time: `sizeof`, `decltype`, `typeid`, `noexcept`. This means that only a declaration not a definition is needed in these contexts and no code is generated for operand expressions. For example, `decltype(foo(std::declval<T>()))` gives the return type of `foo`, if it were called with a `T` rvalue, without actually instantiating a `T`. An example of its usage for testing copy-assignability is `std::is_copy_assignable`, and possibly implemented as,

```
template <class T>
struct is_copy_assignable {
private:
    template <class U, class = decltype(std::declval<U&>() = std::declval<const U&>())>
    static std::true_type try_assignment(U&&); // SFINAE to check for copy-assignable

    static std::false_type try_assignment(...); // overload for anything else
public:
    using type = decltype(try_assignment(std::declval<T>()));
};
```

This works because copy-assignment operators takes an lvalue const reference and assigns it to an lvalue. However, this does not verify that the return type of the operator is an lvalue reference. `std::void_t` (from C++17), which is implemented as, `template <class...> using void_t = void;` can help make a better version,

```
template <class T>
using copy_assignment_t = decltype(std::declval<T&>() = std::declval<const T&>());
```

```cpp
template <class T, class = void> // default void argument is essential for void_t
struct is_copy_assignable : std::false_type {};

template <class T>
struct is_copy_assignable<T, std::void_t<copy_assignment_t<T>>>
    : std::is_same<copy_assignment_t<T>, T&> {};
```

This works because `std::void_t` acts as a metafunction which maps a variable amount of types to an arbitrarily chosen but predictable type (`void`). If all of the given types are well-formed, `std::void_t` is simply an alias for `void`. Otherwise, SFINAE will discard it. The above now also checks that the return type is a `T&` using `std::is_same`. `std::is_move_assignable` can be similarly implemented by changing the `const T&` to `T&&` because move-assignment operators take in an rvalue and return an lvalue reference. All operators can be tested like this by replacing the `using` part appropriately.

5.4.2 Member detection

These are three ways to detect whether a type member exists,

```cpp
/* -- method 1 -- */
#define DETECTOR(detector_name, detector) \
    template <class T>                   \
    using BOOST_PP_CAT( _detect_, detector_name ) = detector; \
    template <typename T>                                \
    using detector_name =                                \
        std::experimental::is_detected<BOOST_PP_CAT( _detect_, detector_name ), T>
#define MEMBER_DETECTOR(detector_name, member_name) \
    DETECTOR(detector_name, decltype(&T::member_name))
#define HAS_MEMBER(name, input_type, detector) \
    MEMBER_DETECTOR(BOOST_PP_CAT( name, _detector ), detector); \
    static constexpr bool name = BOOST_PP_CAT( name, _detector )<input_type>::value

/* usage */
HAS_MEMBER(has_member, TestClass, TestMember);
/* has_member is true iff TestClass has TestMember */

/* -- method 2 -- */
struct _test_has_foo {
    template<class T>
    static auto test(T* p) -> decltype(p->foo(), std::true_type()); // SFINAE on foo
    template<class>
    static auto test(...) -> std::false_type;
};

template<class T>
struct has_foo : decltype(_test_has_foo::test<T>(0)) {};

/* has_foo<T> is true_type iff T has member foo */

/* -- method 3 -- */
template <class, class = void> // default argument must be void to match void_t
struct has_member : false_type {};
```

```
template <class T>
struct has_member<T, void_t<typename T::foo>> : true_type {}; // "more" specialized
```

5.4.3 Type selection

`std::conditional` and `std::conditional_t` can be used to do type selection at compile time.

```
template <bool B, class T, class F>
struct conditional { using type = T; };

template <class T, class F>
struct conditional<false, T, F> { using type = F; };

/* usage */
using TYPE = std::conditional_t<true, std::string, int>;
TYPE my_var = "a string";
```

There might be cases where one of the types `T, F` is invalid depending on the value of the boolean. `std::conditional<true, types::A, types::B>` does not work in the following example because `types::B` does not exist, even though the boolean is true.

```
struct types {
    using A = int;
    //using B = std::string;
};
```

To fix this, an additional level of template indirection is needed,

```
struct types {
    using A = int;
    // using B = std::string;
};

template <typename T>
struct A_type { using tt = typename T::A; };

template <typename T>
struct B_type { using tt = typename T::B; };

int main()
{
    using TYPE = std::conditional_t<true, A_type<types>, B_type<types>>::tt;
    TYPE t = 10;
}
```

This is required because the instantiation of `std::conditional` will go into the specialization for `true`, so the template in the false branch is never instantiated. It just has to be a valid type-expression at the top level (which it is, because `B_type<types>` exists as an identifier). However, if the false template were to be instantiated, this would fail because `B_type<types>` is not a valid type because of the `T::B` substitution failure. Adding a level of template indirection gives the effect of "delaying" template instantiation of a type that is invalid because of SFINAE.

5.4.4 Overloader

5.5 CRTP

5.6 Static dispatch

5.7 Other

5.7.1 Conditional inheritance

```
struct A { int data = 40; };
struct B : A { int data = A::data + 10; };
struct C : std::conditional_t<false, A, B> { /* inherited data has value 50 */ };
```

5.7.2 Derived class templates

In cases where a class template derives from another class template,

```
template <typename T>
struct B { void f() {} };

template <typename T>
struct D : public B<T> { void g() { f(); } };
```

Within `D<T>::g()`, the `f` is a nondependent name since it does not depend on `T`. This causes an error because `B<T>` is a dependent name and the compiler does not look in dependent base classes when looking up nondependent names. Three ways to get around this,

1. Change `f()` to `this->f()`. This works because `this` is always implicitly dependent in a template, so `this->f()` becomes dependent, so the lookup is deferred until the template is actually instantiated, at which point `B<T>` is also considered.
2. Insert `using B<T>::f` before `f()`.
3. Change `f()` to `B<T>::f()`. However, this inhibits virtual dispatch so might not work as expected if `f()` is virtual.

5.7.3 Tricky specialization and non-deduced contexts

Note to self: the following is still confusing to me and my explanation is possibly incorrect.

```
template <typename T> struct InnerT {};
struct Inner {};

template <typename S>
struct Outer {
    template <typename T> using InnerT_Alias = InnerT<T>;
    using Inner_Alias = Inner;
```

```
};

template <typename Out, typename T>
struct test {
    static void where(void) { std::cout << "primary\n"; }
};

template <typename Out, typename T>
struct test<Out, typename Out::template InnerT_Alias<T>> {
    static void where(void) { std::cout << "specialization InnerT\n"; }
};

template <typename Out>
struct test<Out, typename Out::Inner_Alias> {
    static void where(void) { std::cout << "specialization Inner\n"; }
};

int main() {
    test<Outer<int>, Outer<int>::InnerT_Alias<double>>::where(); // <- prints primary
    test<Outer<int>, Outer<int>::Inner_Alias>::where(); // <- prints specialization Inner
}
```

Why does the second instantiation go into the expected Inner specialization but the first does not, even though the only difference is that `InnerT_Alias` is a template class and `Inner_Alias` is a concrete type? Actually, this code is an error. It compiles in GCC but with clang, has the error: the `T` in the `specialization InnerT` cannot be deduced so it will never be used. This is because `typename Out::template InnerT_Alias<T>` is a dependent type (dependent on `Out` and `T`), thus is a non-deduced context, so plays no part in figuring out which template/specialization to match. It only gets expanded after `T` is deduced some other way, which is not possible all the time. For example, what if in the above code, it was `template<typename T> using InnerT = int`? Then all `T` would result in the same type – `int`. We must have some way of unambiguously deducing `T` for this specialization to work. Perhaps in this specific example, the compiler should be able to figure it out but in general, it is not the case and if the compiler allows this, could cause weird/unexpected errors.

The reason dependent types are in a non-deduced context is because there is generally not a one-to-one-mapping from a type to a dependent type. For example, given `T::U`, you cannot figure out `T` unambiguously (imagine `using U = int;` – how do you map an `int` to a `T` which encloses it). In some cases, eg `using U = T*`, there is a one-to-one, so in theory it could work.

Next, we look at an alternate piece of code which is not quite the same,

```
// template <typename T> using inner = int; // doesn't work, ambiguous T
template <typename T> struct inner {};

template <typename T>
struct test { static void where() { std::cout << "primary\n"; } };

template <typename T>
struct test <inner<T>> { static void where() { std::cout << "spec\n"; } };

int main() {
    test<inner<int>>::where(); // prints spec
    test<inner<double>>::where(); // prints spec
```

```
    test<void>::where(); // prints primary
}
```

This works because the compiler can see the definition of `inner`. In this case, since `inner` is a class template, then `inner<T>` has a one-to-one mapping with `T`. However, if it were a typedef/alias template (e.g. `template <typename T> using inner = int`), then it is ambiguous and will not compile.

Revisiting the original problem, we could explicitly provide the "T", so the compiler no longer has to deduce it. This is one workaround.

```
template <typename T> struct Inner {};
template <typename T> struct Inner2 {};
template <typename T> struct Inner3 {};

template <typename S>
struct Outer {
    template <typename T> using Inner_Alias = Inner<T>;
    template <typename T> using Inner_Alias2 = Inner2<T>;
    template <typename T> using Inner_Alias3 = Inner3<T>;
};

template <typename Out, typename FullT, typename InnerT, typename = void>
struct test {
    static void where(void) { std::cout << "primary\n"; }
};

template <typename Out, typename FullT, typename InnerT>
struct test<Out, FullT, InnerT,
            std::enable_if_t<
                std::is_same_v<
                    typename Out::template Inner_Alias<InnerT>, FullT>>> {
    static void where(void) { std::cout << "spec alias\n"; }
};

template <typename Out, typename FullT, typename InnerT>
struct test<Out, FullT, InnerT,
            std::enable_if_t<
                std::is_same_v<
                    typename Out::template Inner_Alias2<InnerT>, FullT>>> {
    static void where(void) { std::cout << "spec alias2\n"; }
};

int main() {
    test<Outer<int>, Outer<int>::Inner_Alias<double>, double>::where(); // prints "spec
        alias"
    test<Outer<int>, Outer<int>::Inner_Alias2<double>, double>::where(); // "spec alias
        2"
    test<Outer<int>, Outer<int>::Inner_Alias2<double>, int>::where(); // "primary"
    test<Outer<int>, void, void>::where(); // "primary"
    test<Outer<int>, Outer<int>::Inner_Alias3<double>, double>::where(); // "primary"
}
```

The third instantiation goes into the primary template because `Outer<int>::InnerAlias2<double>` does not match with `Outer<int>::InnerAlias2<int>`. Here, we are giving the type for the inner alias class template explicitly as `InnerT`.

However, we actually do not need to explicitly give `InnerT`. We can use template template specialization and the compiler can figure out `InnerT` for us,

```
--- *this first part is the same as above* ---

template <typename Out, typename FullT, typename = void>
struct test {
    static void where(void) { std::cout << "primary\n"; }
};

template <typename Out, template<typename> typename FullT, typename InnerT>
struct test<Out, FullT<InnerT>,
            std::enable_if_t<
                std::is_same_v<
                    typename Out::template Inner_Alias<InnerT>, FullT<InnerT>>>> {
    static void where(void) { std::cout << "spec alias\n"; }
};

template <typename Out, template<typename> typename FullT, typename InnerT>
struct test<Out, FullT<InnerT>,
            std::enable_if_t<
                std::is_same_v<
                    typename Out::template Inner_Alias2<InnerT>, FullT<InnerT>>>> {
    static void where(void) { std::cout << "spec alias2\n"; }
};

int main() {
    test<Outer<int>, Outer<int>::Inner_Alias<double>>::where(); // prints "spec alias"
    test<Outer<int>, Outer<int>::Inner_Alias2<double>>::where(); // "spec alias 2"
    test<Outer<int>, void>::where(); // "primary"
    test<Outer<int>, Outer<int>::Inner_Alias3<double>>::where(); // "primary"
}
```

Similar to above, if the `Inner_Alias<T>` could not unambiguously map to `T` (for example, `template <typename T> using Inner_Alias = int`), this would not work either because the template template specialization needs to be able to deduce `T`.

5.7.4 Type selection on sometimes-invalid parameter types

What if we want to call one function depending on if a bool is true and another function if false? And to further complicate it, the parameter type is only valid if the bool is true. That is what the following code does, using the bool `Wrap::b`. If it is true, we want to call a function which takes in a type `const &test`. If false, we want to throw a runtime error.

```
template <typename T>
struct identity { using type = T; };

struct test {};
```

```
struct Wrap {
    static constexpr bool b = false;
    using TYPE = std::conditional_t<b, test, void>;
};

template <typename W = Wrap>
std::enable_if_t<W::b, void> func(
        typename std::conditional_t<
            W::b,
            std::add_lvalue_reference<typename std::add_const<typename W::TYPE>::type>,
            identity<void *>
        >::type p) {
    static_assert(std::is_same_v<decltype(p), const typename W::TYPE &>);
    std::cout << "p is const ref TYPE\n";
}

template <typename W = Wrap>
std::enable_if_t<!W::b, void> func(
        typename std::conditional_t<
            W::b,
            std::add_lvalue_reference<typename std::add_const<typename W::TYPE>::type>,
            identity<void *>
        >::type p) {
    static_assert(std::is_same_v<decltype(p), void *>);
    std::cout << "p is void ptr, in dummy function\n";
    throw std::runtime_error("");
}

int main() {
    func(std::conditional_t<Wrap::b, test(), std::nullptr_t>());
}
```

There are several interesting things here which work together,

1. We require `func` to be a template function (which has a default template parameter for convenience, since we always want to use `Wrap` as `W` anyway) for SFINAE from the `enable_if`.
2. Depending on whether `W::b` is true, the `conditional` in `main` and `enable_if` on `W::b` will ensure that the correct `func` is called.
3. To turn `W::TYPE` into `const &W::TYPE`, we needed to add an lvalue reference and const. This results in an extra layer on top of the original type which must be "peeled back" later using `::type`. To mirror this for the false case, we use a dummy `identity` to add a type around the plain `void *`. Otherwise, we would be doing `T::type`, where `T` is `void *`, which is invalid.
4. We use `void *` rather than `void` because you cannot have a parameter of type void.

5.7.5 Specialization

Here are the steps which are run by the compiler to determine whether to use a specialization over a primary template, using `void_t` in member detection as an example,

```
template <class, class = void>
struct has_member : std::false_type {};
```

```cpp
template <class T >
struct has_member<T, void_t<decltype(T::member)>> : std::true_type {};

/* usage: has_member<A>::value */
```

First, the compiler looks up the name `has_member` and finds the primary template and since it only has one parameter, the second is taken to be the default and is equivalent to if `has_member<A, void>::value` were used.

Next, the template parameter list is compared against any specializations. Template argument deduction is matched because the template parameters of the partial specialization needs to be "filled" by the given arguments. The pattern `T` is trivially deduced to `A`, but this is not always the case (eg, if it were `T const &`). In the second pattern, `T` appears in a context where it cannot be deduced from any template argument because of two reasons:

1. Expressions inside `decltype` are explicitly excluded from template argument deduction since they can be arbitrarily complex
2. Even if it were `void_t<T>`, then we are trying to deduce `T` from a resolved alias template (we resolve the alias template and later deduce `T` from the resulting pattern). However, the alias template resolves to `void`, so `T` cannot be deduced because `void` is not dependent on `T`, so we cannot possibly find a unique `T`. In other words, `void_t<int>` and `void_t<double>` both resolve to `void` so how can we know if it was `int` or `double`?

So, only the first template argument can be deduced and template argument deduction is now finished and the deduced arguments are substituted, creating a specialization which looks like,

```cpp
template <>
struct has_member<A, void_t<decltype(A::member)>> : true_type {};
```

Now, `void_t<decltype(A::member)>` can be evaluated and it is well-formed if and only if `A` has `member` (or else SFINAE would take place). Supposing `A` had `member`, the template parameter list of the specialization is found to match the template arguments given to `has_member<A>::value` (specifically, `A` and `void`), so the specialization is chosen.

This process also highlights why the default second parameter must be `void` and `template <class, class = int>` would not work. If it were like this, then the given `has_member<A>::value` would be found to be equivalent to `has_member<A, int>::value` due to the default parameter in the primary template but this does not match the `A, void` list deduced in the specialization. Since they do not match, the primary template is chosen.

Overall, an important takeaway is that template usage goes through the primary template first and deductions of specializations are considered afterwards.