COMP6771
Advanced C++ Programming

8.1 Static Polymorphism

In This Lecture

Why?

- C uses #define or void* (a.k.a, is weakly-typed).
- C++ is strongly-typed.
- Understanding compile time polymorphism in the form of templates helps understand the workings of C++ on generic types.

What?

- All the different templates in C++
- Template parameters (type, non-type, template template)
- Mechanics of Templates

Recommended Reference:

- C++ Templates: The Complete guide (David Vandevoorde, et al. 2018)

Motivation for Templates

- Question: how to remove code duplication for functions with the same logic but that operate on different types?
  - This branch of programming is called *Generic Programming*.
- **Generic Programming:** Generalising software components to be independent of a particular type
  - STL is a great example of generic programming
- Without generic programming, to create two logically identical functions that behave independently of types, we have to rely on function overloading.
- Templates is how Generic Programming is implemented in C++ to be *fast* and *flexible*.

```
// in C.
// only works because of literal cut & paste!
// there is no type checking here at all!
#define min(x, y) ((x) < (y) : (x) : (y))


// in C++ before templates
int &min(int &x, int &y) {
  return x < y ? x : y;
}

const int &min(const int &x, const int &y) {
  return x < y ? x : y;
}
// repeat for the other types you care about...


// in C++ WITH templates
// one function template to rule them all!
T &&min(T &&x, T &&y) {
  return x < y ? x : y;
}
```

Templates Overview

- C++ has **five** kinds of templates:
  - Function templates
  - Class templates
  - Alias templates (since C++11)
  - Variable templates (since C++14)
  - Variadic Templates (since C++11)
- Each template is a recipe for the compiler to generate a usable entity of that kind.
  - E.g., a function template can be used to *synthesise* a function.
  - It itself is **not** a function!
- Templates can also be used for metaprogramming (next week).

Template Parameters

Three kinds of template parameters.

- **Type** template parameters:
  - Each template parameter holds a type name.
- **Non-type** template parameters:
  - Each template parameter holds a compile-time knowable value.
- **Template template** parameters:
  - Each template parameter holds the name of a template.
- All template parameters can be given default values.
- Template parameters can also be anonymous (no name).

```
template < // template parameter list
  // a type parameter
  // defaults to int
  typename T = int
>
auto min(T &&x, T &&y) -> T&& {
  return x < y ? x : y;
}

// each occurrence of `T` above is
// replaced by the concrete type
// at compile time.
```

Type Template Parameters

- A **type variable** that holds a type.
- Created by either using `typename` or `class` in the template parameter list.
  - Before C++11, only `class` could be used.
  - Now, prefer using `typename`.
  - No difference between the two, just a name change.
- Name style: PascalCase.

```cpp
// below is the template signature for
// std::vector

namespace std {
  template <
    // element type, T
    // using `typename`
    typename T,

    // allocator, for dynamic memory
    // allocation
    // using pre-C++11 `class` syntax
    class Allocator
  >
  class vector { /* ... */ };
}
```

Non-type Template Parameters

- An implicit `constexpr` **variable** that holds a value at compile-time.
- Restrictions on what can be a non-type template parameter (as of C++20):
  - Integral types (`int`, `char`, `long`, etc.)
  - Floating point types (`float`, `double`)
  - Pointers and references (including function pointers)
  - Literal structural types
    - Essentially, C-style structs optionally with base classes
- As of C++20, it is possible to let the compiler deduce the type of the non-type template parameter with `auto`.

```cpp
template <int I>
int sub(int j) { return I - j };

template <float F>
float identity() { return F; }

template <void(*Fn)(int)>
auto invoke(int i) { return Fn(i); }

struct base { int i; };
struct derived : base { int j; };

template <derived D>
void sum() { std::cout << D.i + D.j << std::endl; }

auto course = sub<6771>(0);
auto pi = identity<3.14f>();
auto result = invoke<sub<6771>>(42);
constexpr derived d = {};
auto summed = sum<d>();
```

Template Template Parameters

- A template parameter that holds the name of a **template**.
- The number of template parameters the template itself takes must be known.
  - The number must be exact; no default arguments can be used.
- It is not necessary to name the template template parameter's template parameters.

```cpp
// the definition of std::stack
// probably looks something like this

namespace std {
  template <
    // element type of the stack
    typename T,

    // the underlying sequential container
    // no need to give a name to Container's
    // template parameter – it cannot be used.
    template <typename>
    typename Container
  >
  class stack {
    // other implementation...

    Container<T> cont_;
  };
}
```

Default Template Parameters

- We can provide default arguments to template parameters.
- The set of default template arguments accumulates over all declarations of a given template.
- Once a default is given, all subsequent parameters must also have default values.
  - Matches the same rules as default function arguments.

```
template <typename>
struct empty {};

template <
  typename T = int,
  auto I = 0,
  template <typename>
  typename Container = empty<T>
>
struct eclectic {};
```

Function Templates

- Function template: not actually a function.
  - Generalisation of algorithms.
- A blueprint for the compiler to synthesise particular instances of a function varying by type.
  - Single definition that can generate many definitions.
  - The compiler *instantiates* a function template only when a function by that name is needed and no proper function exists.

```cpp
#include <iostream>

template <typename T>
auto add_or_concat(const T &a, const T &b) -> T {
  return a + b;
}

int main() {
  // prints 3
  std::cout << add_or_concat(1, 2) << std::endl;

  // prints "hello world"
  std::string h = "hello", w = "world";
  std::cout << add_or_concat(h, w) << std::endl;

  // code is only generated for int and std::string
  // no other type was used, so no other version
  // was instantiated.

  return 0;
}
```

Function Template Argument Deduction

- To instantiate a function template, all the template parameters must be known.
  - But they don't need to be specified!
- The compiler will attempt to deduce any missing parameters from the function arguments.
- **Important**: implicit conversions:
  - For *type parameters*, implicit conversions do **not** occur.
  - For non-type template parameters, the compiler will perform implicit conversions.
- Some notable rules:
  - Most specific type is matched.
  - May match on type specifiers (const, volatile)
  - May match on modifiers (pointer *, reference &)
  - [Complete list of rules here]()

```cpp
template <typename T>
auto min(T &&x, T &&y) { return x < y ? x : y; }

template <typename T, std::size_t N>
T sum_array(const T(&arr)[N]) {
  return std::accumulate(arr, arr + N, T{});
}

int main() {
  // OK: T = int
  min(1, 2);
  // NOT OK: T = const char *; compare pointers
  min("hello", "world");

  // OK: T = std::string
  min(std::string{"hello"}, std::string{"world"});

  int nums[] = {3, 2, 1};
  // OK: T = int, N = 3
  std::cout << sum_array(nums) << std::endl;
}
```

auto Revisted

- Ever wondered what rules `auto` uses to deduce types?
  - `auto` uses the same rules as function template argument deduction!
- As of C++14, can use `auto` as lambda function parameters.
  - This creates a *generic* lambda.
  - A new lambda is created for each argument type combination.
- As of C++20, `auto` can also be used in function parameters.
  - This creates an *implicit* function template.
  - A new function is created for each argument type combination.

```
// generic lambda
auto min = [](const auto &x, const auto &y) {
  return x < y ? x : y;
}

// C++20-style function template
auto max(const auto &x, const auto &y) {
  return y < x ? y : x;
}

int main() {
  // prints 1, auto deduced to be int
  std::cout << min(1, 2) << std::endl;

  // prints 'a', auto deduced to be char
  std::cout << max('0', 'a') << std::endl;

  // won't compile: auto deduced const char *, int
  // pointers are not comparable to integers
  std::cout << max("hi", 6771) << std::endl;

  // prints 'a', auto deduced char, int
  // char implicitly convertible to int
  std::cout << min('a', 127) << std::endl;
} // altogether, 2 versions of min, 1 version of max
```

Explicit Template Argument Deduction

- If we need more control over the normal deduction process, we can explicitly specify the types being passed in.
- This will allow for implicit conversions of the passed arguments to the explicitly-stated types.
- It is possible to explicitly state a subset of the template parameters.
    - The remaining template parameters undergo normal template argument deduction.

```cpp
template<typename T, typename U>
auto min(T a, U b) {
  return a < b ? a : b;
}

auto main() -> int {
  auto i = 0;
  auto d = 3.0;
  // int min(int, double)
  min<int>(i, d);
  // int min(int, int)
  min<int, int>(i, static_cast<int>(d));

  // double min(double, double);
  min<double>(static_cast<double>(i), d);

  // double min(double, double)
  min<double, double>(i, d);
}
```

Overload Resolution Revisited

- The compiler changes how it performs overload resolution when function templates are involved:
  1. The compiler constructs the overload candidate set first *from real functions*.
     - This includes previously instantiated function templates.
  2. If there is no best match, then it will instantiate a new function with the appropriate types from the template.
     - It is important to remember function templates *are not* part of the overload set. Only functions synthesised from the template are.

```cpp
template <typename T>
void printer(const T *ptr) {
  std::cout << ptr << std::endl;
}

void printer(const int *ptr) {
  std::cout << ptr << std::endl;
}

int main() {
  int i = 0;
  double d = 0.0;

  // no function void printer(const double *)
  // synthesise one
  printer(&d);

  // found void printer(const int *)
  // don't even consider the template
  printer(&i);

  // found previous instantiation
  // void printer(const double *). Use that
  printer(&d);
}
```

Class Templates

- Similar to function template, a class template is a blueprint for synthesising a class-type.
- Is not actually a class.
- All of the members inside of the class template are parameterised based on the template parameters.

```cpp
// Before templates...
struct vec3i {
  int(&)[3] get_elems() const { /* ... */ }
  int elems[3];
};

struct vec4d {
  double(&)[4] get_elems() const { /* ... */ }
  double elems[4];
};


// ...after templates!
template <typename T, std::size_t N = 3>
struct vec {
  T(&)[N] get_elems() const { /* ... */ }
  T elems[N];
};
```

Member Function Templates

- Usually the member functions of a class template are parameterised on the class template's template arguments.
- It is possible to make the member function itself a template.
  - In this case, it would have two sets of parameters: the class template's and its own.
- This is useful for creating many overloads of a member function.
  - E.g., converting constructors.

```
// std::vector's iterator constructor probably
// looks something like this

namespace std {
  template <typename T, /* others... */>
  class vector {
    template <typename InputIt>
    vector(InputIt first, InputIt last) {
    // allocate memory for an array
    // copy elements between first and last
    // into the allocated array
    }
  };

  // another great example is std::set's
  // transparent comparator feature for .find().
  // read more about it here
  // (notably overloads 3 & 4)
}
```

Static Members of a Class Template

- Possible to create static data members for a class template.
  - Every instantiation has its own version of the static member.
  - Initialisation also looks like a template.
  - Can be defined inline if constant.
- Also possible to create static member functions.

```cpp
template <typename T>
struct t_is_small {
  // inline constant static data member
  static constexpr bool value = sizeof(T) <= 4;
};

template <typename T>
class rational {
public:
  // static data member
  static std::optional<rational> null;

  // static member function
  static auto make_rational(T n, T d) {
    /* implementation... */
  }

  // other implementation details...
};

// out-of-line defintion of static data member.
template <typename T>
std::optional<rational> rational<T>::null = {};
```

Friends of a Class Template (error)

- Class templates can also have `friends`.
- **Caution:** the friend declaration declares a non-template function (see example on the right).
  - To make a friend based on the class template's parameters, the friend itself also needs to be a template.
- Best to make all friends hidden friends to avoid confusion.

```cpp
// This looks like it would work, and it will even compile.
// However, the linker will fail to find operator<<.
template <typename T, int N>
class vec {
public:
  friend std::ostream &operator<<(
    std::ostream &os, const vec &v
  );

private:
  T elems_[N];
};

template<typename T, int N>
std::ostream &operator<<(
  std::ostream &os, const vec<T, N> &v
) {
  return os;
}

int main() {
  vec<int, 3> v;
  std::cout << v<< std::endl;
}
```

Friends of a Class Template (fixed)

- Class templates can also have `friends`.
- **Caution:** the friend declaration declares a non-template function (see example on the right).
  - To make a friend based on the class template's parameters, the friend itself also needs to be a template.
- Best to make all friends hidden friends to avoid confusion.

```cpp
// To fix, the class template and the friend template are
// declared first...
// Then a <> is added after the function name in the friend
// Finally, the friend is defined below.
// Avoid this - use hidden friends.

std::ostream &operator<<(std::ostream &os, const vec<T, N> &v);

template <typename T, int N>
class vec {
public: friend std::ostream &operator<< <>(
    std::ostream &os, const vec &v
);

private:
  T elems_[N];
};

template<typename T, int N>
std::ostream &operator<<(std::ostream &os, const vec<T, N> &v) {
  return os;
}
```

Class Template Argument Deduction

- NEW in C++17: Class Template Argument Deduction
  - Also called CTAD for short.
- Equivalent to function template argument deduction, but for class templates.
  - Same rules about not doing implicit conversions also apply.
- User must define the rules for deducing template arguments from a constructor call.
  - Then, the user doesn't need to specify the class template's parameters when declaring variables.
  - Syntax looks like the constructor signature.

```cpp
template <typename T>
class rational {
public:
  rational(T num, T denom);

private:
  T num_;
  T denom_;
};

// CTAD definition.
// When the rational(T, T) constructor would have been
// used, then deduce the template parameter to be T.
template <typename T>
rational(T, T) -> rational<T>;

int main() {
  // from initialiser, rat inferred to be rational<int>
  rational rat = {1, 2};

  // error: deduced two conflicting types!
  rational err = {1, 2.3};
}
```

Out-of-line Definitions

- A class template's methods can be defined out-of-line.
- For member templates, there are two sets of template parameters.
- Unless you have good reason to, prefer defining methods inline in the template definition.

```
template <typename T>
struct bar { /* definition... */ };

template <typename T>
struct foo {
  template <typename U>
  foo(const bar<U> &b);

  void baz();
};

template <typename T> // top-level template first
template <typename U> // bottom-level template 2nd
foo<T>::foo(const bar<U> &b) { /* ... */ }

template <typename T>
void foo<T>::baz() { /* ... */}
```

Variadic Templates

- NEW in C++11: variadic templates.
  - Also called "parameter packs"
- Allows templates to accept an arbitrary numbers of parameters and deal with them as a pack of types.
  - Expansion parameter pack (and variables of that type) done with ellipsis (...).
- Compiler performs pattern-matching when selecting which function or class template to use.
  - See example on right.

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

// base case for template instantiation
template <typename T>
void print_list(const T &e) {
  std::cout << e << std::endl;
}

// recursive case for instantiation
template <typename T, typename ...Rest>
void print_list(const T &e, const Rest& ...rest) {
  std::cout << e << ' ';
  print_list(rest...);
}

int main() {
  // signature of the initial call:
  // print_list<int, char, double, std::string>
  // compiler will recursively instantiate
  // print_list until the base is reached.

  // instantiations:
  // print_list<int, [char, double, std::string]>
  // print_list<char, [double, std::string]>
  // print_list<double, [std::string]>
  // print_list<std::string> (no leftover type params)
  print_list(1, 'a', 3.14, std::string{"hi"});
} // output: 1 a 3.14 hi
```

`sizeof...`

- It is possible to get the number of elements in a variadic template's parameter pack.
- This is done with the `sizeof...` operator.
  - This returns a `constexpr` `std::size_t`.
- No static `typeof...` operator, however.
  - Only way to get the types of a parameter pack is to expand it in a template.

```
template <typename T, int N>
struct array { T elems[N]; }

// CTAD guide for array using variadic templates.
// Note the "+ 1" for the N parameter.
// This is because the initial T should also be
// counted as part of the array size.
template <typename T, typename ...Ts>
array(T, Ts...) -> array<T, sizeof...(Ts) + 1>;

int main() {
  // If only there was a standard library type
  // that achieved the same goal as this...
  // (std::array, perhaps?)
  array arr = {1, 2, 3};
}

// It is as-if sizeof... is implemented like below
//
// std::size_t sizeof...() { return 0; }
//
// template <typename T, typename ...Ts>
// std::size_t sizeof...(const T&, const T&& ...ts) {
//   return 1 + sizeof...(ts...);
// }
```


UNSW SYDNEY

Fold Expressions

- NEW in C++17: fold expressions.
- Use a parameter pack in expressions containing unary or binary operators to perform a left or right fold.
  - Parentheses around the fold expression are mandatory.
- For the binary operator case:
  - The operators must be the same.
  - Optionally can take an initial value.
- Can be used to replace some recursive function templates.

```cpp
#include <iostream>

// Pre-C++17
template <typename T>
auto sum(T t1) { return t1 + 0; }

template <typename T, typename ...Ts>
auto sum(T t, Ts ...ts) { return t + sum(ts...); };

// Post-C++17

template <typename ...Ts>
auto sum_binary(Ts ...ts) {
  return (ts + ... + 6765); // 6765 is the initial value
}

template <typename ...Ts>
auto sum_unary(Ts ...ts) {
   return (... + ts); // no initial value
}

int main() {
  std::cout << sum_binary(3, 1, 2) << std::endl;
  std::cout << sum_unary(89, 96, 02) << std::endl;
}
// Output:
// 6771
// 187
```

Partial & Explicit Specialisation

The templates we've defined so far are completely generic.

- There are two ways we can refine our generic, primary templates for something more specific:
- Partial specialisation:
  - Refining ("specialising") the primary template to work with a subset of types.
  - T*
  - std::vector

Explicit specialisation:

Refining the template for a specific, non-generic type.

  - std::string
  - int

Note: not all the template varieties can be partially specialised.

- Notably, function templates cannot be partially specialised.

When to Specialise

- You need to preserve existing semantics for something that would not otherwise work.
  - `std::is_pointer` is partially specialised over pointers.
- You want to write a type trait (coming next week).
  - `std::is_integral` is fully specialised for `int`, `long`, etc.
- There is an optimisation you can make for a specific type or family of types.
  - `std::vector<bool>` is fully specialised to reduce memory footprint.

When NOT to Specialise

- Don't specialise function templates
  - A function template cannot be partially specialised.
  - Fully specialised function templates are better done with overloads.
  - Herb Sutter has an article on this
    - http://www.gotw.ca/publications/mill17.htm
- You think it would be cool if you changed some feature of the class for a specific type.
  - People assume a class works the same for all types.
  - Don't violate assumptions!

(Not) Specialising Function Templates

- Though function templates cannot be partially specialised, they can be explicitly specialised.
- This creates incredibly confusing bugs.
- Do **NOT** specialise function templates.
  - Use overloads instead.

```
#include <iostream>

template <typename T>
/* A */ void foo(T) { std::cout << "A\n"; }

template <>
/* B */ void foo(int *) { std::cout << "B\n"; }

template <typename T>
/* C */ void foo(T *) { std::cout << "C\n"; }

/* D */ void foo(int *) { std::cout << "D\n"; }

int main() {
  int p = 0;
  foo(&p); // which of A, B, C, D is used?

  // Answer: D
}
```

(Not) Specialising Function Templates

- Though function templates cannot be partially specialised, they can be explicitly specialised.
- This creates incredibly confusing bugs.
- Do **NOT** specialised function templates.
    - Use overloads instead.

```cpp
#include <iostream>

template <typename T>
/* A */ void foo(T) { std::cout << "A\n"; }

template <>
/* B */ void foo(int *) { std::cout << "B\n"; }

template <typename T>
/* C */ void foo(T *) { std::cout << "C\n"; }

int main() {
  int p = 0;
  foo(&p); // which of A, B, C is used?

  // Answer: C!!!
}

// The compiler considers only primary templates
// when deciding when if it should instantiate
// a function template. Once it has selected the
// primary template, only then will it look for any
// specialisations.
// Here, (C) is a better match than (A), so (C) is used
```

(Not) Specialising Function Templates

- Though function templates cannot be partially specialised, they can be explicitly specialised.
- This creates incredibly confusing bugs.
- Do **NOT** specialised function templates.
    - Use overloads instead.

```cpp
#include <iostream>

template <typename T>
/* A */ void foo(T) { std::cout << "A\n"; }

template <typename T>
/* C */ void foo(T *) { std::cout << "C\n"; }

template <>
/* B */ void foo(int *) { std::cout << "B\n"; }

int main() {
  int p = 0;
  foo(&p); // which of A, B, C is used?

  // Answer: B!!!
}

// The compiler finds (C) as being the better primary
// template for the call to foo.
// Once it finds (C), it checks to see if there are
// any specialisations. Here, since (B) is declared after
// C, the compiler thinks it is a specialisation of (C),
// and so it will select (B) to call.
```

Partial Specialisation of Class Template

- You can partially specialise class types.
  - You cannot partially specialise a particular method of a class in isolation, however.
- Partial specialisation of classes is particularly useful when writing type traits.
- Compiler performs pattern matching on the given template arguments and the expected parameter.
  - If the primary template expects a `T` and the partial specialisation expects a `T*` and an `int*` is given, the compiler will select the partial specialisation since `T*` "matches" `int*` better than `T`.

```
template <typename T>
struct is_a_pointer {
  static constexpr auto value = false;
};

template <typename T>
struct is_a_pointer<T*> {
  static constexpr auto value = true;
};

// An example of a simple type trait.
// Starting generically, we assume any and every
// generic type is not a pointer.
// So, the answer to the question "is T a pointer?"
// is no (false).
// Through partial specialisation over pointer types,
// we can refine our answer to yes! (true)

// answer: false
constexpr auto int_is_pointer = is_a_pointer<int>::value;

// answer: true
constexpr auto ptr_is_pointer = is_a_pointer<void*>::value;
```

Explicit Specialisation of Class Template

- Explicit specialisation should only be done on class and variable templates.
- std::vector<bool> is an interesting example and here too.
  - Surprisingly, `std::vector<bool>::reference` is not a `bool&`.
- In addition to the primary template, create a fully specialised version for a specific parameter (or set of parameters) for a template.

```cpp
#include <iostream>

template <typename T>
struct is_void {
  static constexpr auto value = false;
};

template<>
struct is_void<void> {
static constexpr auto value = true;
};

// The answer to the question:
// "Is this type void?"
// in general is "no".
// However, for void (and only void!),
// the answer to: "is this type void?"
// is unambiguously yes.
// So, encode that information using
// explicit specialisation.
int main() {
  std::cout << std::boolalpha <<
               is_void<int>::value << ' ' <<
               is_void<void>::value << std::endl;
}
// output: false true
```

(Not) Specialising Alias Templates

- Alias template cannot be partially specialised.
- Alias templates cannot be explicitly specialised.
  - That would just be a regular alias!

Specialising Variable Templates

- Variable templates can be partially or fully specialised.
- Not many use cases of specialising variable templates.

```cpp
#include <iostream>

// Actually, the mathematical constants
// are specified in the standard like this:
namespace std::numbers {
  template <typename T>
  constexpr T pi = /* unspecified */;

  template <>
  constexpr auto pi<double> = double(3.141592653589793238 5L);

  template <>
  constexpr auto pi<float> = float(3.141592653589793238 5L);
}

// The reason for this is to disallow instantiating pi
// with an arbitrary type and losing precision.
// As above, only the provided explicit specialisations of
// pi are allowed to be used and trying to specialise or
// instantiate pi outside of this set is unspecified behaviour.
```

Implicit Instantiation

- We know the compiler instantiates templates.
  - But when exactly does it do it?
- The compiler **implicitly instantiates** a template only at its first point of *use*.
  - Thus, if you never use a template, it is never instantiated.
  - Further uses of the template used the cached instantiation.

```
template <typename T>
bool is_less_than(T t1, T t2) {
  return t1 < t2;
}

template <typename T>
  bool is_greater_than(T t1, T t2) {
return t1 > t2;
}

int main() {
  // first use of is_less_than<int, int>
  // this template is implicitly instantiated
  is_less_than(1, 4);

  // second use of is_less_than<int, int>
  // previous instantiation is used
  is_less_than(2, 5);

  // first use of is_less_than<char, char>
  // the template is implicitly instantiated
  is_less_than(2,2);
}
// no use of is_greater_than?
// no instantiations of it.
```

Inclusion Compilation Model

- When it comes to templates, we implement them in header files.
  - This is because template definitions need to be known at **compile time**.
- Will expose implementation details in the header file.
- Can cause slowdown in compilation as every file using the header file will have to instantiate the template.
  - It's then up the linker to ensure there is only 1 instantiation.
  - Alot of generated code is simply thrown away.

```cpp
// in min.h
template <typename T>
T min(T t1, T t2) {
  return t1 < t2 ? t1 : t2;
}

// in me.cpp
#include "min.h"
void foo() {
  // complex calculation...
  auto m = min(var1, var2);
  // more complex calculations...
}

// in you.cpp
#include "min.h"
void bar() {
  // simple calculation
  auto mm = min(var0, var3);
  // more "simple" calculations...
}

// The function template min() was included in two
// different .cpp files. This means the compiler had
// to parse, instantiate, and store the same template
// twice, only for the linker to throw one version away...
```

Explicit Instantiation

- Sometimes, we want explicit control of when the compiler instantiates a template.
  - But should be avoided if it can be.
- This can alleviate some of the performance costs of using templates since the compiler only instantiates the template once.
  - It still has to parse the template definition, though.
- We can tell the compiler to only instantiate a template once and link the generated code after compilation.
  - This is especially useful for common instantiations of a template, such as `std::vector<int>`.

```cpp
// in vec.h
template <typename T, int N>
struct vec {
  // other, very mathematical, implementation
T elems[N];
};

// extern says that the template instantiation
// is defined in another .cpp file.
extern template struct vec<double, 3>;
extern template struct vec<double, 4>;

// in vec.cpp

// explicitly instantiate the extern templates.
// other .cpp files will use the generated code
// from this translation unit.
template struct vec<double, 3>;
template struct vec<double, 4>;
```

Two-Phase Translation

- Compiler processes each template in two phases:
  1. When compiler reaches the definition.
     - Happens once for each template for each translation unit.
  2. When compiler instantiates the template.
     - Happens once for each combination of template parameters.
- Error messages vary based on which phase the error was detected.
  - For **syntactical** issues, the error is reported when the compiler reaches the template definition.
  - For **semantic** (type-related) issues, the error is reported when the compiler instantiates the template.
- This has the benefit of reducing compiler work when a template is not used, but can lead to **dependent scope ambiguity**.

Dependent Scope Ambiguity

Consider `example()` on the right.

Question: what is this?

- If `bar` is a value, then this code is a multiplication.
- If `bar` is a type, this is a variable definition.
- If `bar` is a template, this code is ill-formed.

Since `bar` depends on the template parameter, which is not known until instantiation, we have to tell the compiler what we expect `bar` to be:

- By default, the compiler expects `bar` to be a value (nothing to do).
- If `bar` is a type, we need to prepend typename before using `bar`.
- If `bar` is a template, we need to prepend template before using `bar`.

```
int a;

template <typename T>
void example() {
  T::bar * a; // what is this?
}

template <typename T>
void example_value() {
  // this is a multiplication of
  // T::bar and the global variable a
  T::bar * a;
}

template <typename T>
void example_type() {
  // this is definition of a local variable a
  // (a pointer to the type T::bar)
  typename T::bar * a;
}

template <typename T>
void example_template() {
  T:: template bar * a; // this code is ill-formed.
}
```

Considerations for Templates

- C++ templates are an extremely powerful way to do generic programming.
  - Fast, efficient, and *automatic*.
- Tenet of C++: don't pay for what you don't use.
  - Use of templates has a cost.
  - Overuse of templates in large projects absolutely cripples compilation times.
- Some guidelines when using templates:
  - For general datastructures and generic, relatively small, and modular algorithms, template use is completely fine.
  - Always prefer functions, function overloads, and real class-types over templates unless you have a reason to generalise.
    - YAGNI still applies – **Y**a **A**in't **G**onna **N**eed **I**t!
- If compilation time is too long, consider using explicit instantiation.

Feedback (stop recording)