Templates and Functions

CS 106L Spring 2020 – Avery Wang and Anna Zeng
Stanford University

27 April 2020

Game Plan

- template functions
- variadic template
- concept lifting
- lambda functions
- implicit interface
- SFINAE

template functions

Sidenote: ternary operator

```c
int my_min(int a, int b) {
  return a < b ? a : b;
}
```

if condition

return if true

return if false

```c
// equivalently
int my_min(int a, int b) {
  if (a < b) return a;
  else return b;
}
```

Can we handle different types?

```
int main() {
  auto min_int = my_min(1, 2);
  auto min_name = my_min(“Anna”, “Avery”);
}
```

One way: overloaded functions.

```
int my_min(int a, int b) {
  return a < b ? a : b;
}
```

```
string my_min(string a, string b) {
  return a < b ? a : b;
}
```

Bigger problem: how do you handle user defined types?

We now have a generic function!

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

Just in case we don't want to copy type T.

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

Example

Templates: syntax and initialization

We now have a generic function!

Declares the next declaration is a template.

Specifies T is some arbitrary type.

List of template arguments.

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

Scope of template argument T limited to function.

Explicit Instantiation of Templates

```
my_min<string>("Anna", "Avery");
```

Explicitly states T = string

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

Compiler replaces every T with string

Implicit Instantiation of Templates

```
my_min(3, 4);
```

Compiler deduces
T = int

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

Compiler replaces every T with int
(not perfect with our parameter
rules, but it's fine)

Be careful: type deduction can’t read your mind!

```
my_min(“Anna”, “Avery”);
```

Compiler deduces
T = char* (C-string)

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

Compiler replaces
every T with char*

Comparing pointers,
not what you want!

Our function isn't technically correct.

```
my_min(4, 3.2);
// this returns 3
```

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

Deduces T = int.

Be careful: type deduction can’t read your mind!

```
my_min(4, 3.2);
// this returns 3.2
```

```
template <typename T, typename U>
auto my_min(const T& a, const U& b) {
  return a < b ? a : b;
}
```

This is kinda complicated, so have the compiler figure it out.

Accounting for the fact that the types could be different.

You can overload non-template special cases.

```
char* my_min(const char*& a, const char*& b) {
  return string(a) < string(b) ? a : b;
}
```

If we get C-strings, run this special case.

```
template <typename T, typename U>
auto my_min(const T& a, const U& b) {
  return a < b ? a : b;
}
```

Otherwise, create a template.

Template Instantiation: creating an "instance" of your template.

When you call a template function, either:

- for explicit instantiation, compiler finds the relevant template and creates that function in the executable.
- for implicit instantiation, compiler looks at all possible overloads (template and non-template), picks the best one, deduces the template parameters, and creates that function in the executable.
- **After instantiation, the compiler looks as if you had written the instantiated version of the function yourself.**

Template functions are not functions

It's a recipe for generating functions via instantiation.

Distinction is important for separate compilation.

variadic templates

How can we make this function even more generic?

```
int main() {
  auto min1 = my_min(4.2, -7.9);



}
```

Say, an arbitrary number of parameters?

```
int main() {
  auto min1 = my_min(4.2, -7.9);
  auto min2 = my_min(4.2, -7.9, 8.223);
  auto min3 = my_min(4.2, -7.9, 8.223, 0.0);
  auto min4 = my_min(4.2, -7.9, 8.223, 0.0, 1.753);
}
```

Example

Variadic templates

Quick challenge: write a recursive version of the function which accepts a vector.

```
// assume nums is non-empty and T is comparable
template <typename T>
T my_min(vector<T>& nums) {





}
```

Quick challenge: write a recursive version of the function which accepts a vector.

```
// assume nums is non-empty and T is comparable
template <typename T>
T my_min(vector<T>& nums) {
  T elem = nums[0];
  if (nums.size() == 1) return elem;
  auto min = my_min(nums.subList(1));
  if (elem < min) min = elem;
  return min;
}
```

Varadic Templates can use compile-time recursion.

```
template <typename T, typename ...Ts>
auto my_min(T num, Ts... args) {
  auto min = my_min(args...);
  if (num < min) min = num;
  return min;
}
```

Supplemental Material

Varadic Templates can use compile-time recursion.

```
template <typename T, typename ...Ts>
auto my_min(T num, Ts... args) {
  auto min = my_min(args...);
  if (num < min) min = num;
  return min;
}
```

Pack expansion:
comma-separated patterns

Parameter pack:
0 or more types.

Parameter Pack Expansion

```
// expands to f(&E1, &E2, &E3)
f(&args...);

// expands to f(n, ++E1, ++E2, ++E3);
f(n, ++args...);

// expands to f(++E1, ++E2, ++E3, n);
f(++args..., n);

// f(const_cast<const E1*>(&X1), const_cast<const E2*>(&X2), ...)
f(const_cast<const Args*>(&args)...);
```

What does this expand to?

```
f(h(args...) + args...);
```

```
// f(h(E1,E2,E3) + E1, h(E1,E2,E3) + E2, h(E1,E2,E3) + E3)
```

Don't forget a base case!

(well technically, it's not recursion)

```cpp
template <typename T, typename ...Ts>
auto my_min(const T& num, Ts... args) {
  auto min = my_min(args...);
  if (num < min) min = num;
  return min;
}

template <typename T>
auto my_min(const T& num) {
  return num;
}
```

Supplemental Material

Any call to the function is pattern matched to these templates, resolved at compile-time.

```
template <typename T, typename ...Ts>
pair<T, T> my_min(T num, Ts... args) {
    auto [min, max] = my_min(args...);
    if (num < min) min = num;
    if (num > max) max = num;
    return {min, max};
}
```

```
auto [min3, max3] = my_min(4.2, -7.9, 8.223, 0.0);

// T = int
// Ts = int, int, int

// num = 4.2
// args = -7.9, 8.223, 0.0
```

The parameter pack can be expanded into the comma-separated list.

```
template <typename T, typename ...Ts>
pair<T, T> my_min(T num, Ts... args) {
  auto [min, max] = my_min(args...);
  if (num < min) min = num;
  if (num > max) max = num;
  return {min, max};
}
```

```
Equivalent to:
auto [min3, max3] = my_min(-7.9, 8.223, 0.0);

Since:
// num = 4.2
// args = -7.9, 8.223, 0.0
```

The types do not have to be homogenous. More realistic example is printf.

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  int pos = format.find('%');
  if (pos == string::npos) return;
  cout << format.substr(0, pos) << value;
  printf(format.substr(pos+1), args...);
}
```

Note: this is a simple version without format flags.

The types do not have to be homogenous. More realistic example is printf.

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  int pos = format.find('%');
  if (pos == string::npos) return;
  cout << format.substr(0, pos) << value;
  printf(format.substr(pos+1), args...);
}

void printf(string format) {
  cout << format;
}
```

Note: this is a simple version without format flags.

At compile-time, the compiler instantiates the templates.

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  …
  printf(format.substr(pos+1), args...);
}
```

```
printf(“Lecture %: % (Week %)”, 7, “Templates”s, 4);
```

First using template deduction we deduce T and Ts.

At compile-time, the compiler instantiates the templates.

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  …
  printf(format.substr(pos+1), args...);
}
```

```
printf<int, string, int>
```

This function is generated.

At compile-time, the compiler instantiates the templates.

```
template <typename T, typename ...Ts>
void printf(string format, int value, string arg1, int arg2) {
  …
  printf(format.substr(pos+1), arg1, arg2);
}
```

```
printf<int, string, int>
```

And everything is replaced with the instantiated type.

At compile-time, the compiler instantiates the templates.

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
   …
   printf(format.substr(pos+1), args...);
}
```

```
printf<int, string, int>
printf<string, int>
```

The recursive call tells us the next instantiation we need.

At compile-time, the compiler instantiates the templates.

```
template <typename T, typename ...Ts>
void printf(string format, string value, int arg1) {
  …
  printf(format.substr(pos+1), arg1);
}
```

```
printf<int, string, int>
printf<string, int>
```

Instantiation replaces the types.

At compile-time, the compiler instantiates the templates.

```
template <typename T, typename ...Ts>
void printf(string format, T value, Ts... args) {
  …
  printf(format.substr(pos+1), args...);
}
```

```
printf<int, string, int>
printf<string, int>
printf<int>
```

The recursive call tells us the next instantiation we need.

At compile-time, the compiler instantiates the templates.

```
template <typename T, typename ...Ts>
void printf(string format, string value) {
  …
  printf(format.substr(pos+1));
}
```

```
printf<int, string, int>
printf<string, int>
printf<int>
```

Here the parameter pack is empty.

At compile-time, the compiler instantiates the templates.

```
void printf(string format) {
  cout << format;
}
```

```
printf<int, string, int>
printf<string, int>
printf<int>
printf
```

One more function created – that is the non-template base function.

At compile-time, these functions are created.

```
printf<int, string, int>
printf<string, int>
printf<int>
printf
```

concept lifting

Concept Lifting

Looking at the assumptions you place on the parameters and questioning if they are really necessary.

Can you solve a more general problem by relaxing the constraints?

Why write generic functions?

Count how many times 4.7 appears in a list<double>.
Count how many times ‘X’ appears in a string.
Count how many times 5 appears in the second half of a vector<int>.
Count how many elements in the second half of a vector<int> are at most 5.

How many times does the integer [val] appear in an entire vector of integers?

```
template <>
int count_occurences(const vector<int>& vec, int val) {
  int count = 0;
  for (size_t i = 0; i < vec.size(); ++i) {
    if (vec[i] == val) ++count;
  }
  return count;
}
```

Example

Lifting count_occurences

How many times does the element satisfy [predicate] in [a range of elements]?

```
template <typename InputIt, typename DataType,
     typename UniPred>
int count_occurences(InputIt begin, InputIt end,
                                 UniPred predicate) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (predicate(*iter)) ++count;
  }
  return count;
}
```

A predicate is a function which takes in some number of arguments and returns a boolean.

```
// Unary Predicate (one argument)
bool is_equal_to_3(int val) {
    return val == 3;
}

// Binary Predicate (two arguments)
bool is_divisible_by(int dividend, int divisor) {
    return dividend % divisor == 0;
}
```

We can then call this function with a predicate.

```
bool is_less_than_5(int val) {
    return val < 5;
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};

  count_occurences(vec.begin(), vec.end(), is_less_than_5);
  // prints 2
}
```

Example

Experiments with improving function pointers.

Problems we've run into with function pointers.

From our experiments, we see that function pointers

- generalize poorly: we have to write a separate function if you wanted to change the limit from 5 to 6.
- cannot add more parameters: we call the predicate with one parameter.

The fundamental issue we've run into is scope. We need to pass a variable "limit" into the function without adding another parameter.

lambda functions

Old approach: function pointers

```
bool is_less_than_limit(int val) {
   return val < limit; // compiler error!
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 5;



  count_occurences(vec.begin(), vec.end(), is_less_than_limit);
  return 0;
}
```

New approach: lambda functions

```
bool is_less_than_limit(int val) {
   return val < limit;
}

int main() {
  vector<int> vec{1, 3, 5, 7, 9};
  int limit = 5;
  auto is_less_than_limit = [limit](auto val) {
    return val < limit;
  }
  count_occurences(vec.begin(), vec.end(), is_less_than_limit);
  return 0;
}
```

New approach: lambda functions

We don't know the type, ask compiler.

capture clause, gives access to outside variables

parameter list, can use auto!

return type, optional

```
auto is_less_than_limit = [limit](auto val) -> bool {
    return val < limit;
}
```

Accessible variables inside lambda limited to capture clause and parameter list.

You can also capture by reference.

```
set<string> teas{“black”, “green”, “oolong”};
string banned = “boba”; // pls … this is not a tea
auto liked_by_Avery = [&teas, banned](auto type) {
   return teas.count(type) && type != banned;
};
```

You can also capture everything by value or reference.

```
// capture all by value, except teas is by reference
auto func1 = [=, &teas](parameters) -> return_value {
    // body
};

// capture all by reference, except banned is by value
auto func2 = [&, banned](parameters) -> return_value {
    // body
};
```

Summary of Lambdas

Lambdas are function objects that can capture variables that are not parameter.

Lambdas can be passed into template functions as a predicate.

implicit interfaces

The compiler literally replaces each template parameter with whatever you instantiate it with.

```
vector<int> v1{1, 2, 3, 1, 2, 3};
vector<int> v2{1, 2, 3};
count_occurences(v1.begin(), v1.end(), v2);
```

The compiler literally replaces each template parameter with whatever you instantiate it with.

```
vector<int> v1{1, 2, 3, 1, 2, 3};
vector<int> v2{1, 2, 3};
count_occurences(v1.begin(), v1.end(), v2);
```

`vector<int>::iterator`

`vector<int>::iterator`

`vector<int>`

The compiler literally replaces each template parameter with whatever you instantiate it with.

```
template <typename InputIt, typename DataType>
int count_occurences(InputIt begin, InputIt end,
                     DataType val) {

  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (*iter == val) ++count;
  }
  return count;
}
```

The compiler literally replaces each template parameter with whatever you instantiate it with.

```
template <typename InputIt, typename DataType>
int count_occurences(vector<int>::iterator begin,
                     vector<int>::iterator end,
                     vector<int> val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
```

*iter is an int, can’t == to a vector

A template function defines an implicit interface that each template parameter must satisfy.

```
template <typename InputIt, typename DataType>
int count_occurences(InputIt begin, InputIt end,
                     DataType val) {

  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (*iter == val) ++count;
  }
  return count;
}
```

What must be true of InputIt and DataType?

Each template parameter must have the operations the function assumes it has.

InputIt must support	// types which do not satisfy interface
• copy assignment (iter = begin)	// bad: streams
• prefix operator (++iter)	// bad: collections
• comparable to end (begin != end)	// bad: anything not an iterator
• dereference operator (*iter)	// bad: numeric types
DataType must support	
• comparable to *iter	// bad: vector<vector<int>>::iterator

Template interfaces: explicit vs. implicit

```
count_occurences(v1.begin(), v1.end(), v2);
```

- Semantic error: *iter == val compares int with vector<int>.
- Conceptual error: you can't find the min or max of two streams.

- The compiler deduces the types and literally replaces the types. Compiler will produce semantic errors, not conceptual error.
- Really not fun to debug.

Associative containers have an implicit interface: the type must support the < operator.

By default, the type (for sets) or key-type (for maps) must have a comparison (<) operator defined for them.

(There is an alternative that we will cover in next lecture)

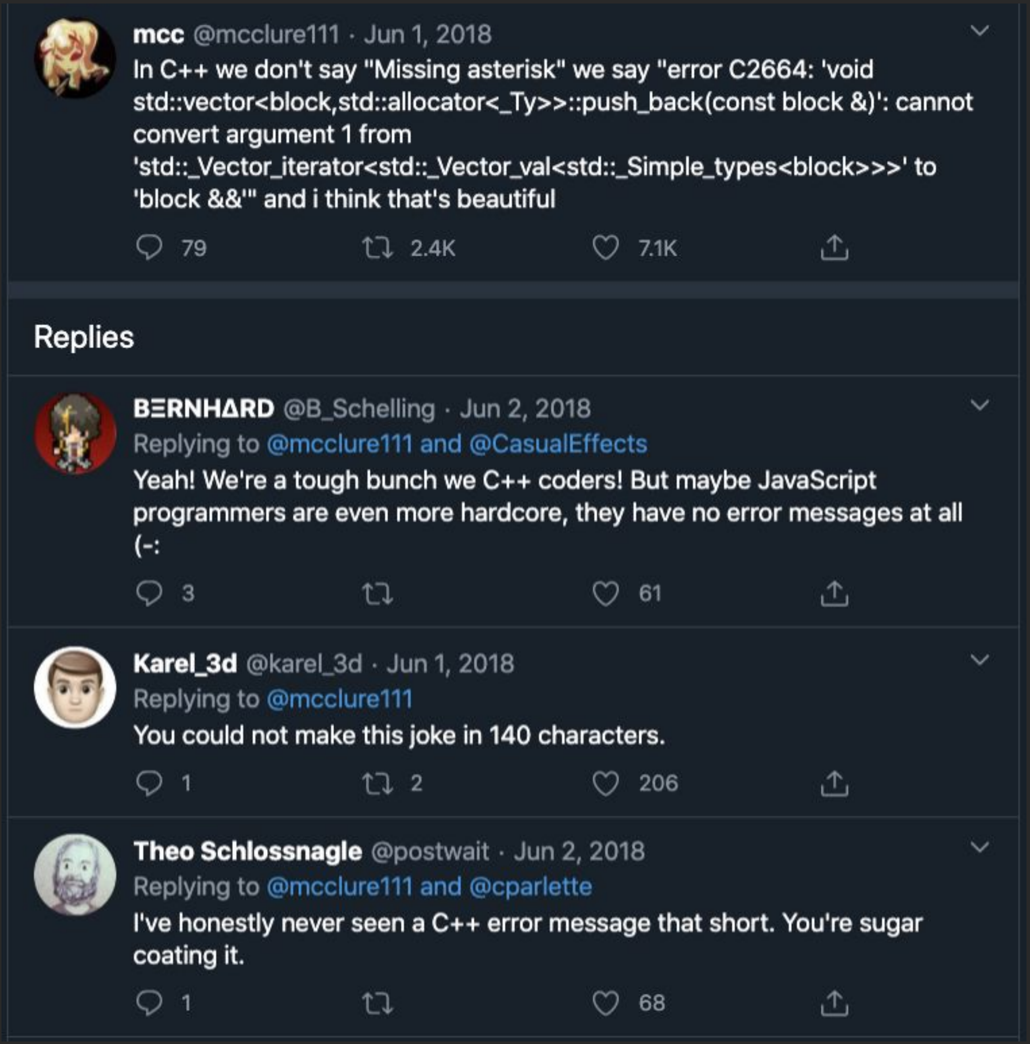
```
std::set<std::pair<int, int>> set1;  // OKAY - comparable
std::set<std::ifstream> set2;         // ERROR - not comparable

std::map<std::set<int>, int> map1;   // OKAY - comparable
std::map<std::function, int> map2;   // ERROR - not comparable
```

sidenote: why are we calling this an l-value reference?
We'll see this in a few slides...

Example

When templates go wrong.

Template error messages are not fun to debug.



mcc @mcclure111 · Jun 1, 2018

In C++ we don't say "Missing asterisk" we say "error C2664: 'void std::vector<block,std::allocator<_Ty>>::push_back(const block &)': cannot convert argument 1 from 'std::_Vector_iterator<std::_Vector_val<std::_Simple_types<block>>>' to 'block &&'" and i think that's beautiful

79 2.4K 7.1K

Replies

BΞRNHΔRD @B_Schelling · Jun 2, 2018

Replying to @mcclure111 and @CasualEffects

Yeah! We're a tough bunch we C++ coders! But maybe JavaScript programmers are even more hardcore, they have no error messages at all (-:

3 61

Karel_3d @karel_3d · Jun 1, 2018

Replying to @mcclure111

You could not make this joke in 140 characters.

1 2 206

Theo Schlossnagle @postwait · Jun 2, 2018

Replying to @mcclure111 and @cparlette

I've honestly never seen a C++ error message that short. You're sugar coating it.

1 68

Summary of Implicit Interface

A template is unconstrained: the compiler will let you instantiate the types with whatever you ask it to.

The actual error comes from the exact line there is a type error.

This is why template errors are extra scary.

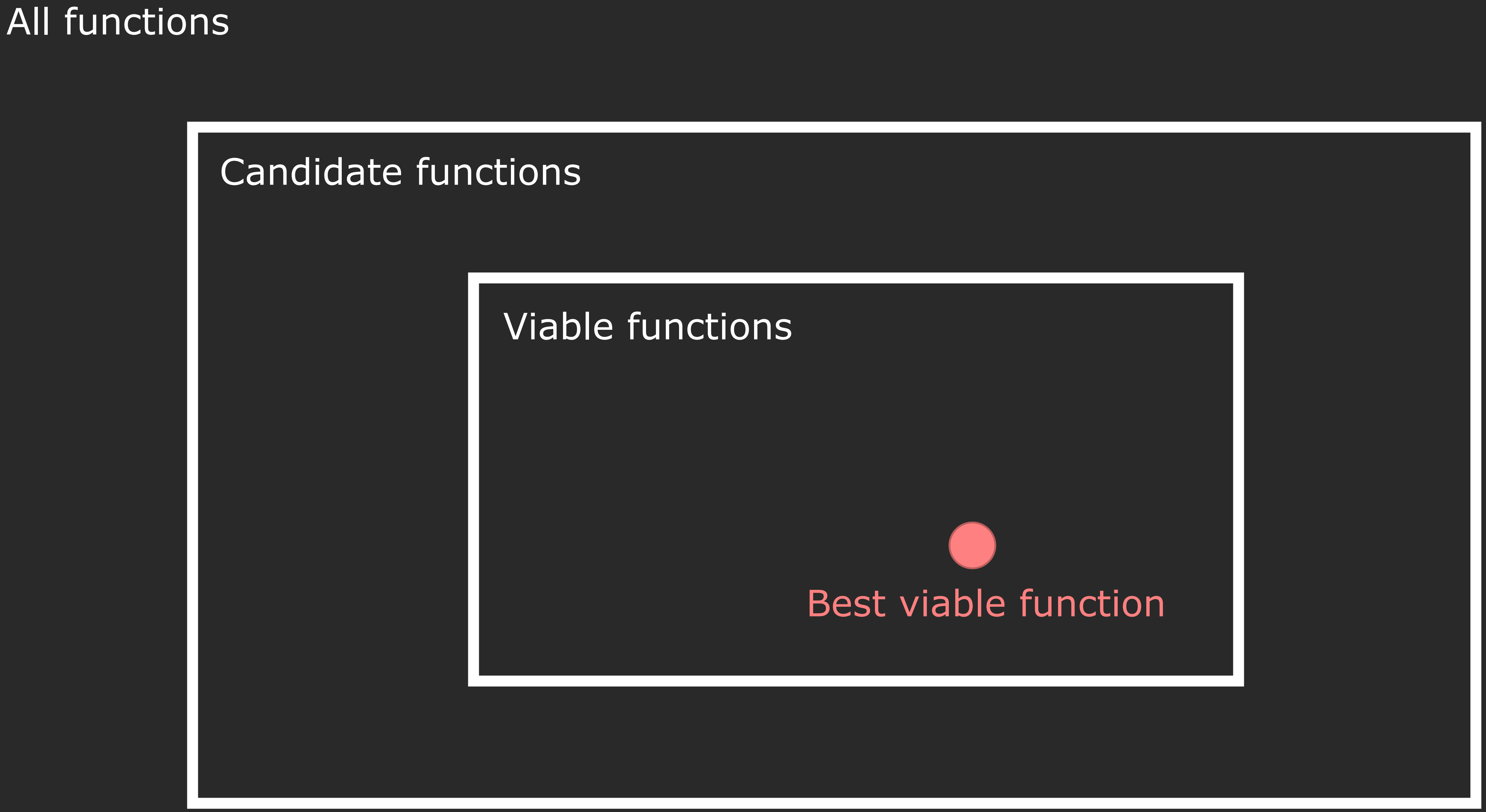
overload resolution

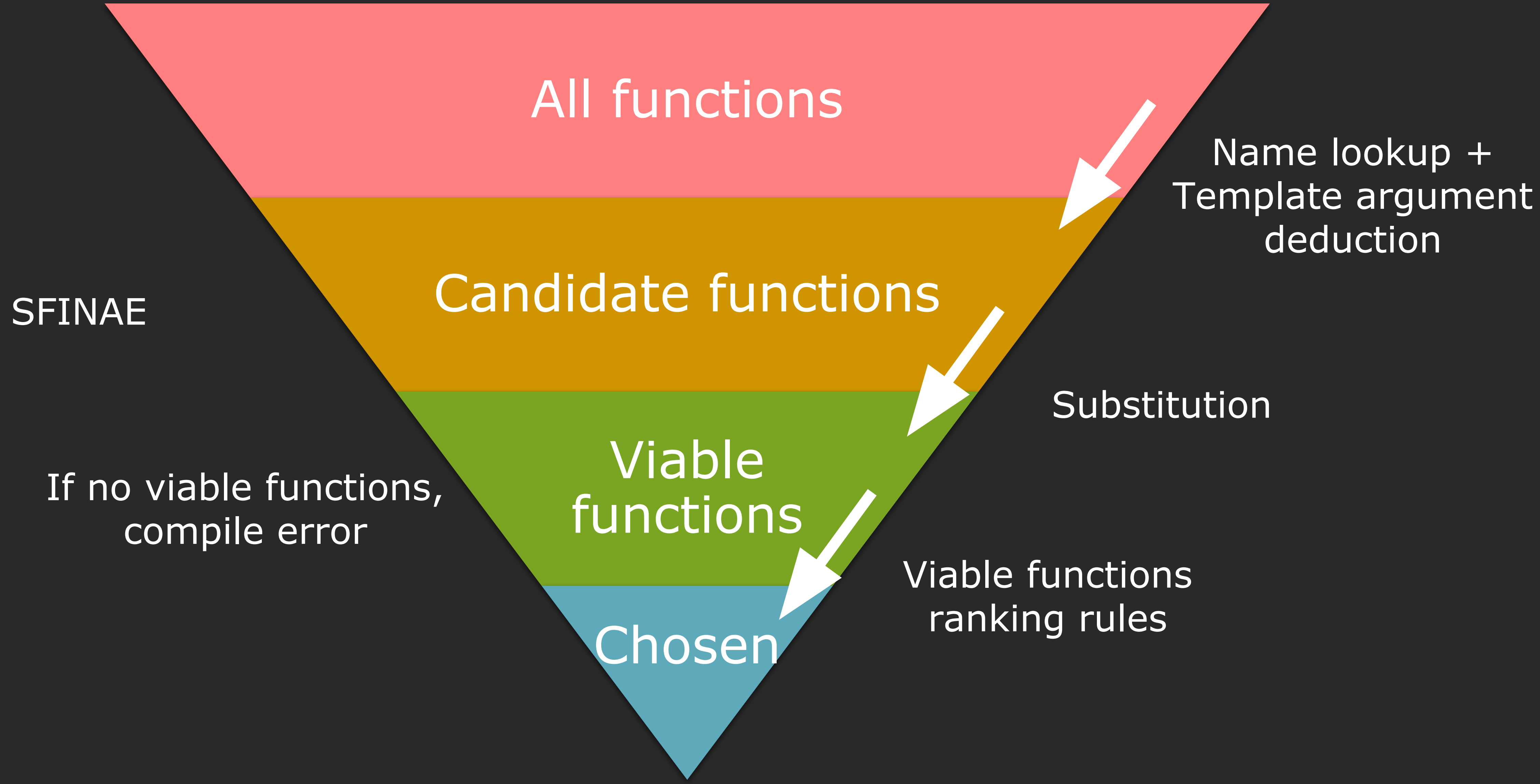
Warning: the topics here will seem "hacky".

Indeed, most of template metaprogramming was discovered by accident. It was not "designed" this way, but it turned out to be useful technique used to implement a lot of the STL.

“what if there are multiple potential templates functions?”

This turns out to be extremely, extremely useful.


All functions
Candidate functions
Viable functions
Best viable function


All functions
Candidate functions
Viable functions
Chosen
Name lookup + Template argument deduction
Substitution
Viable functions ranking rules
SFINAE
If no viable functions, compile error

Overload resolution steps

- From all functions within scope, look up all functions that match the name of function call. If template is found, deduce the type.
- From all candidate functions, check the number and types of the parameters. For template instantiations, try substituting and see if implicit interface satisfied. If fails, remove these instantiations.
- From all viable functions, rank the viable functions based on the type conversions necessary and the priority of various template types. Choose the best viable function.

SFINAE

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror
- When substituting the deduced types fails (in the immediate context) because the type doesn’t satisfy implicit interfaces, this does not result in a compile error.
- Instead, this candidate function is not part of the viable set. The other candidates will still be processed.

Power of SFINAE

- We can implement this logic:

If substituted type does not satisfy some condition,
remove this overload from the candidate function set.

For Java users, this should remind you of Reflection.

For Python users, this might remind you of hasattr().

Very common SFINAE template: use this overload if [expression] compiles

```
template <typename T>
auto function(const T& a)
        -> decltype((void) [expression], [return type]()) {


  // function implementation


}
```

Example

SFNIAE example and enable_if

This template is valid if a.size() compiles.

```
template <typename T>
auto print_size(const T& a)
        -> decltype((void) a.size(), size_t()) {

  cout << “printing with size member function: ”;
  cout << a.size() << endl;

  return a.size();
}

// T = int (fail)
// T = vector<int> (success)
// T = vector<int>* (fail)
```

This template is valid if -a compiles.

```
template <typename T>
auto print_size(const T& a)
        -> decltype((void) -a, size_t()) {

  cout << "printing with negative numeric function: ";
  cout << -a << endl;

  return -a;
}

// T = int (success)
// T = vector<int> (fail)
// T = vector<int>* (fail)
```

This template is valid if a->size() compiles.

```
template <typename T>
auto print_size(const T& a)
        -> decltype((void) a->size(), size_t()) {

  cout << “printing with pointer function: ”;
  cout << a->size() << endl;

  return a->size();
}

// T = int (fail)
// T = vector<int> (fail)
// T = vector<int>* (success)
```

SFNIAE removes the overloads which do not compile, allowing you to call printSize on different types!

```
int main() {
  vector<int> vec{1, 2, 3};
  print_size(vec);       // calls first overload
  print_size(vec[1]);    // calls second overload
  print_size(&vec);      // calls third overload
  print_size(nullptr);  // compiler error
}
```

Power of SFINAE

```
std::enable_if<Predicate>
```

If Predicate is satisfied, proceed as normal.

If Predicate is not satisfied, <u>purposely</u> create a compiler error!

template metaprogramming

Welcome to the deep dark corners of C++.

Whoops - no time left :/

Guess it'll have to be an optional topic!

Next time

Algorithms

Homework

CS 106B section problems…
but you'll have to use STL containers + iterators.