Wrapping Up The STL

Bad Dad Joke of the Day:

- How did the hamburger introduce his wife?
- Meat patty.

Creds: Julie

Abstraction in the STL

Abstractions allow us to express the general structure of a problem instead of the particulars of its implementation.

Rather than solve specific instances, solve the problem in a general setting!

do is express general problems rather than specific ones

Abstraction in the STL

We started off with basic types:

- char
- int
- double
- ...

Each type was conceptually a "single value".

sort of taste of this we start off in computer

Abstraction in the STL

Can we keep track of a collection of basic types, regardless of what the type is?

of a collection of basic types no matter what the type is

Basic Types

Abstraction in the STL

Many programs require a collection of basic types:

A `vector<int>` representing student ages

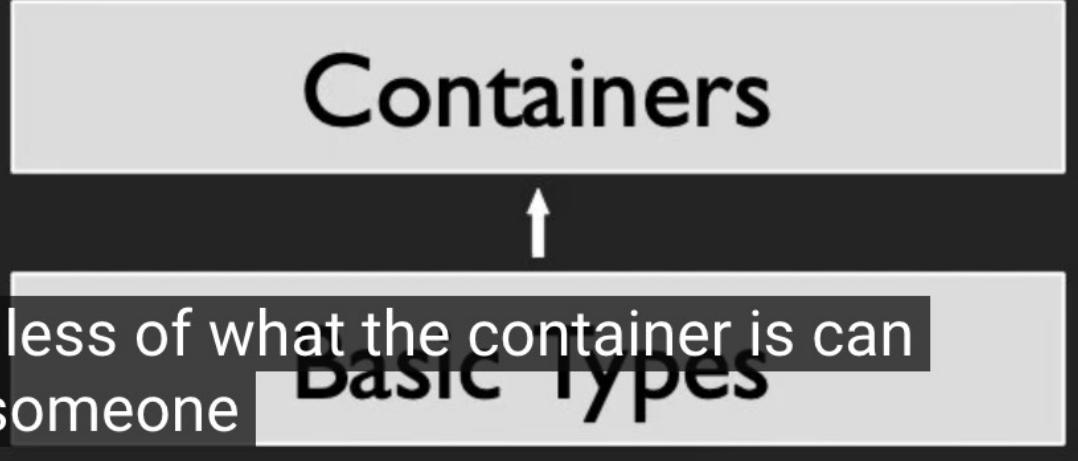
A `map<string, int>` of names to phone numbers

Containers let us perform operations on basic types, regardless of what the basic type is.

what the type is so abstracting away
from the type absolutely and you guys

Abstraction in the STL

Can we perform operations on containers regardless of what the container is?



regardless of what the container is can
does someone

Abstraction in the STL

Iterators allow us to abstract away from the container being used.

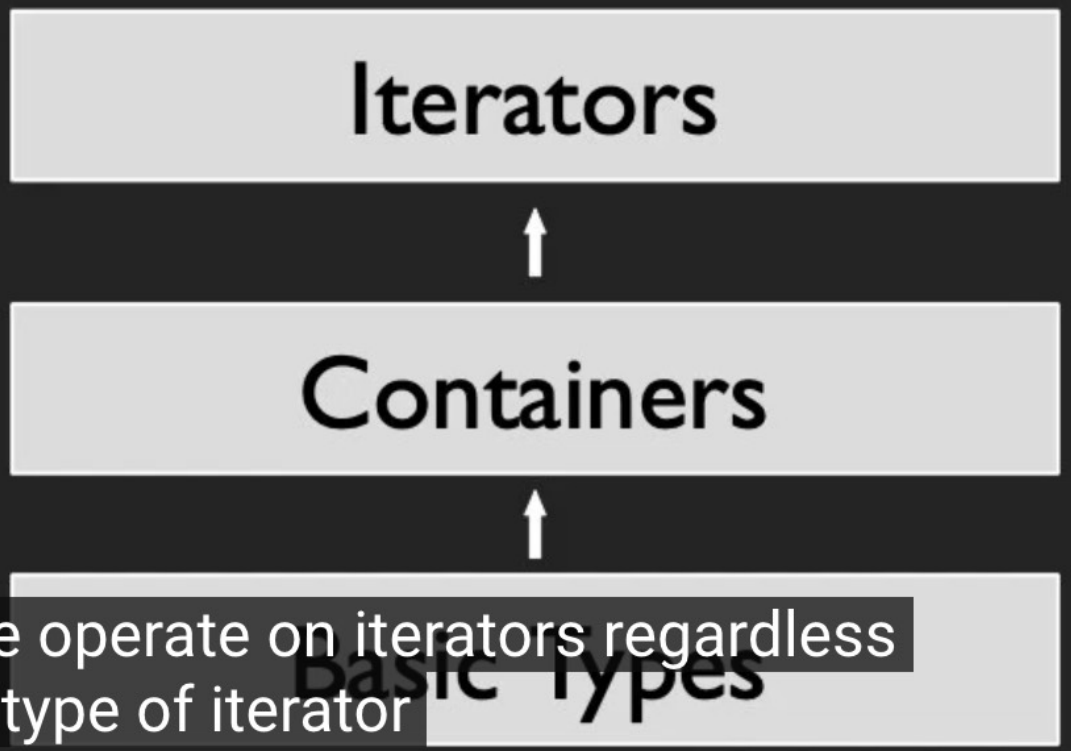
Similar to how containers allow us to abstract away from the basic type being used.

Operations like sorting, searching, filtering, partitioning, and more can be written to work with almost any container.

really yeah just like well this is an
imperfect

Abstraction in the STL

Can we operate on iterators regardless of what type of container the iterator is for?



can we operate on iterators regardless of the type of iterator

Abstraction in the STL

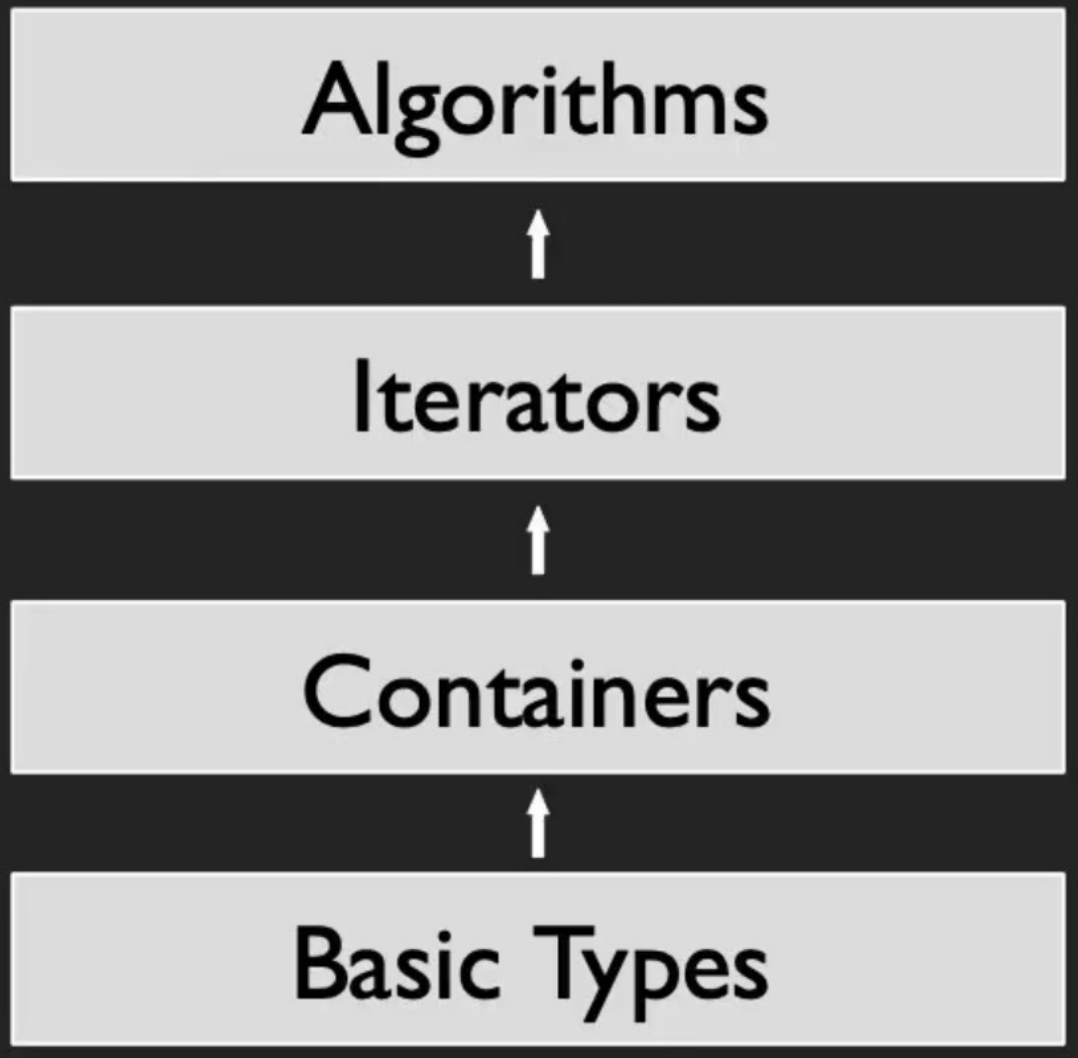
The STL contains pre-written algorithms that:

- operate on iterators, which lets them work on many types of containers, and
- often apply functors, which allows generalization of the algorithm's applications

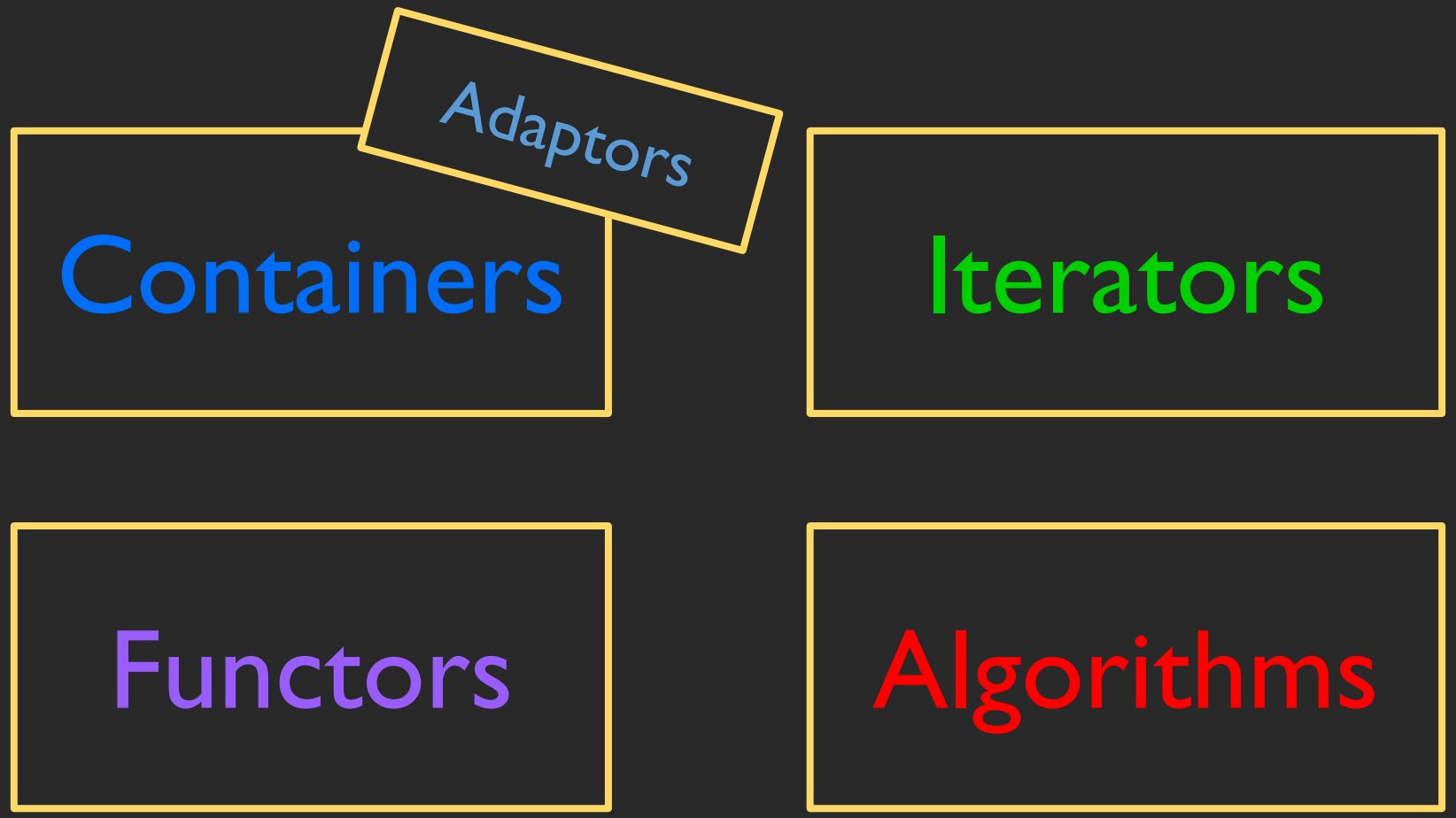
of the type of iterator that it is and
it's exactly as we already said

Abstraction in the STL

STL核心概念：
– Iterator:
iterator 链接container和algorithms，所有对container操作的算法都离不开iterator



Overview of STL



THE BIG STL RECAP,
aka the BSTLR*

*not an official C++ acronym

The BSTLR

1. Streams
2. Sequence containers + container adaptors
3. Associative containers
4. Iterators
5. Templates
6. Lambdas
7. Algorithms

Streams Recap

when to use stream

console & keyboard

a string

files

stringstream:
- stringstream ss("Hello", stringstream::ate);
- ss << 106;
- string myString; ss >> myString;

filestream:
- fstream fs(filename);
- string line; getline(fs, line);

State bits:
- good, fail, eof, bad
- fail fails silently!

cout/cin:
- cout << "Hello" << endl;
- cin >> myInt >> myString;

Def of streams: buffer + pos ptr(controlled by diff operators)

>> operator: how does it work? (nightmare: cin >> ....)

<< operator

getline()

stringstream vs. string

When should I use a stringstream?

1. Processing strings
   - Simplify “/./a/b/..” to “/a”
2. Formatting input/output
   - uppercase, hex, and other stream manipulators
3. Parsing different types
   - stringToInteger() from previous lectures

If you’re just concatenating strings, str.append() is faster than using a stringstream!

Sequence Containers + Adaptors Recap

Sequence Containers:
- vector
  (fast access of middle)
- deque
  (fast insert begin/end)
- list, forward_list

Container Adaptors:
- stack
  ➜ secretly a vector or deque
- queue
  ➜ secretly a deque

vector:
- vector<int> v{1, 0, 6};
- v.push_back(-100);
- v.pop_back();
- v[3] fails silently! v.at(3) throws error

deque:
- deque<int> d{1, 0, 6};
- same as a vector
- d.push_front(42);
- d.pop_front();

stack: stack<int> s{1, 0, 6}; s.push(5); s.pop();
queue: queue<int> q{1, 0, 6}; q.push(5); q.pop();

Associative Containers Recap

Associative Containers:

(sorted, fast for range:)
- map
- set

set is a map only with keys

- multimap
- multiset

(allows repeated keys)

(hashed, fast for single elems:)
- unordered_map
- unordered_set
- unordered_multimap
- unordered_multiset

map:
- map<int, string> m{{5, “Hi”}, {80, “Bye”}};
- m[106] = “C++”;
- m.count(106);
- m.at(99) = “Hey”; // throws error
- m[99] = “Hey”;   // creates new entry

.count()/std::find: whether an key/element exist or no

set:
- set<int> s{1, 0, 6};
- really a map to 0/1, without .at() and []

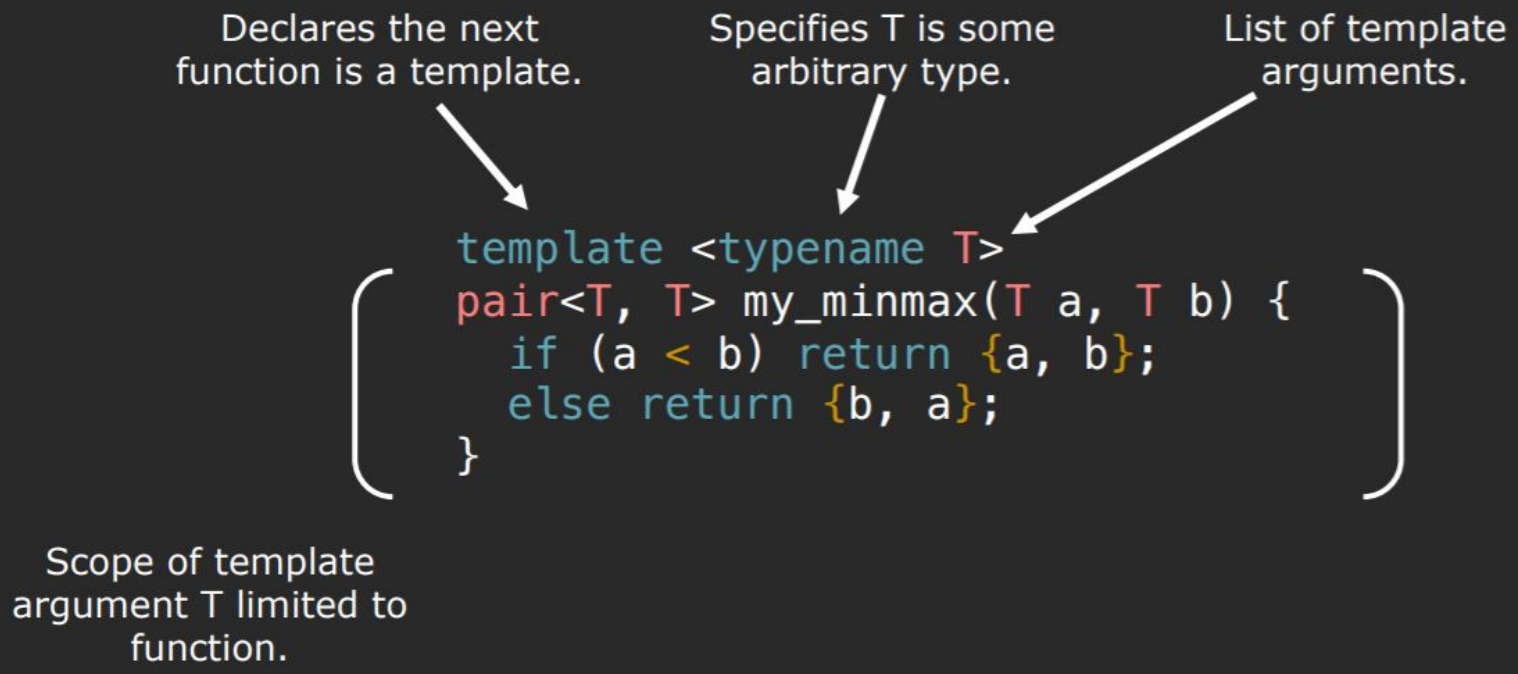
Iterators Recap

Types:
- input
  (one-pass, read-only)
- output
  (one-pass, write-only)
- forward
  - (multi-pass, read and write)
- bidirectional
  (multi-pass, read and write, can decrement)
- random access
  (multi-pass, read and write, can incr/decr by arbitrary amounts)

Basic syntax:
- set<int> s{1, 0, 6};
- set<int>::iterator it = s.begin();
- auto it2 = s.end();
- ++it;
- *it = 3;
- if (it != it2) ...
- map<int, string> m{{1, “Hi”}, {6, “Bye”}};
- auto [key, val] = m.begin();
- (m.begin())->first = 3

for-each loop:
- for (int i : s) ...
  is implemented as
- for (auto it = s.begin(); it != s.end(); ++it) ...

Templates Recap



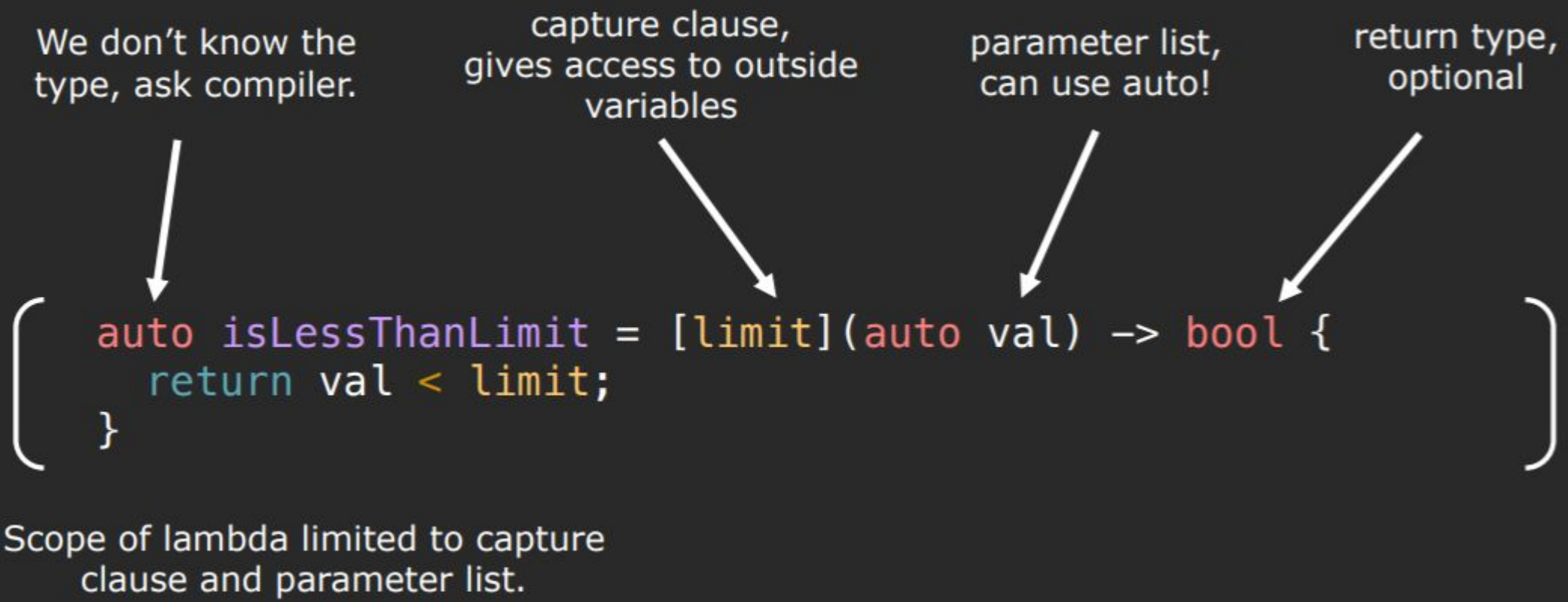
Explicit instantiation:
- my_minmax<string>("Avery", "Anna");

Implicit instantiation:
- my_minmax(3, 6);
- my_minmax("Avery", "Anna"); won't do as you expect! Will deduce C-strings

generic programming: concept lifting: relaxing the constraints/assumptions imposed on parameters

Concept (C++20) = turns the implicit assumptions that your code is making, into explicit requirements

Lambdas Recap



Capture clause:

- [=, &obj] ➔ captures everything by value, except obj by reference
- [&, limit] ➔ captures everything by reference, except limit by value

Algorithms Recap

Algorithms we've seen:
- std::sort
- std::find
- std::count
- std::nth_element
- std::stable_partition
- std::copy
- std::copy_if
- std::remove_if
- and more!

Special iterators:

iterator adaptor

- back_inserter
  - e.g., std::copy(vec.begin(), vec.end(), std::back_inserter(newVec));
- stream_iterator
  - e.g., std::copy(vec.begin(), vec.end(), std::ostream_iterator<int>(cout, ", "));

Erase-remove idiom using algorithms*:

```
std::erase(
    std::remove(v.begin(), v.end()), v.end()
);
```

*many containers will define their own erase function which does this for you - this only applies if you use the STL erase/remove algorithms