COMP6771 Week 2.1

STL Containers & Iterators

I/O (Input / Output) in C++

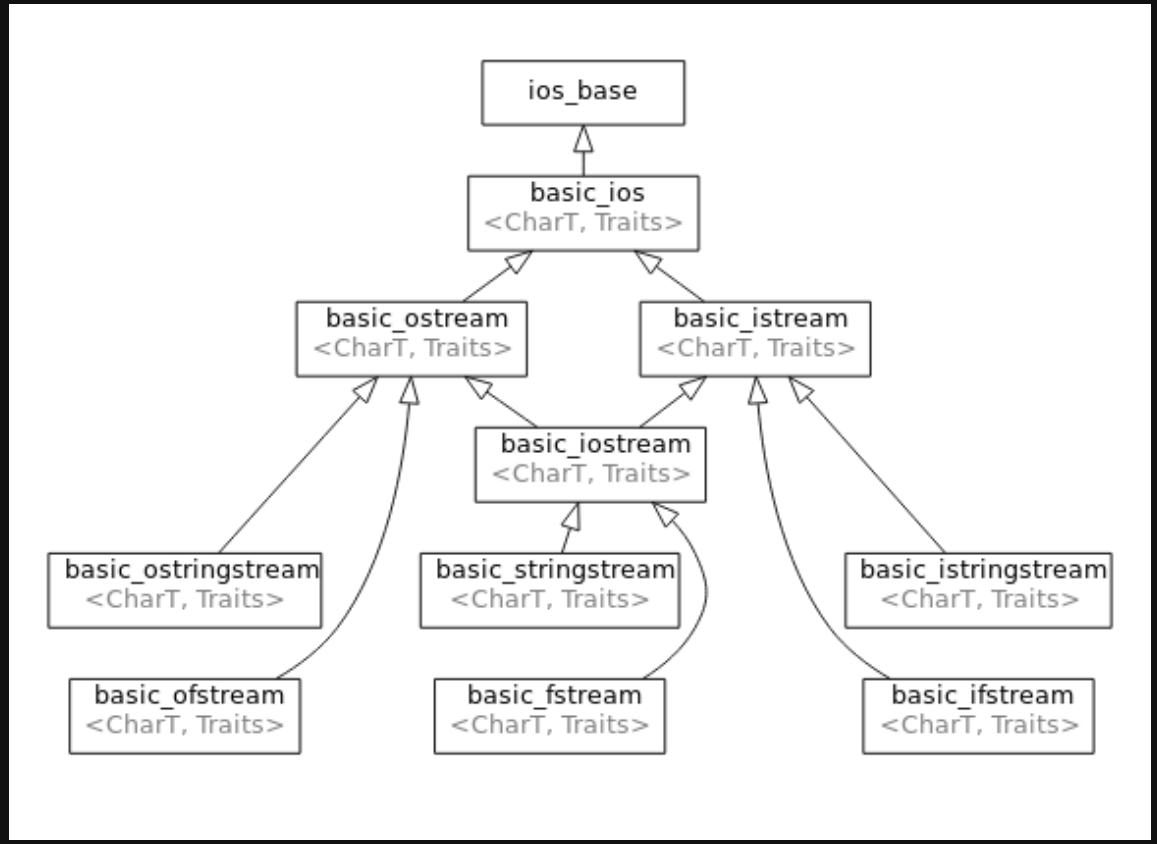
```cpp
#include <iostream>
#include <fstream>

int i;
std::ifstream fin{"data.in"};
while (fin >> i) {
  std::cout << i << "\n";
}
fin.close();
```

```cpp
std::ofstream fout{"data.out"};

fout << i;

fout.close();
```

Better input stream example

```cpp
#include <iostream>
#include <fstream>

int main () {
  // Below line only works C++17
  if (auto in = std::ifstream{"data.in"}; in) { // attempts to open file, checks it was opened
    for (auto i = 0; in >> i;) { // reads in
      std::cout << i << '\n';
    }
    if (in.bad()) {
      std::cerr << "unrecoverable error (e.g. disk disconnected?)\n";
    } else if (not in.eof()) {
      std::cerr << "bad input: didn't read an int\n";
    }
  } // closes file automatically <-- no need to close manually!
  else {
    std::cerr << "unable to read data.in\n";
  }
}
```

I/O (Input / Output) in C++



I/O (Input / Output) in C++

Be careful using these, these are not the best style

```cpp
#include <iostream>
#include <iomanip> // to use the setprecision manipulator

int main() {
  std::cout << 1331 << std::endl; // 1331
  std::cout << "In hex " << std::hex << 1331 << std::endl; // In hex 533
  std::cout << 1331.123456 << std::endl; // 1331.12
  std::cout.setf(std::ios::scientific, std::ios::floatfield);
  std::cout << 1331.123456 << std::endl; // 1.331123e+03
  std::cout << std::setprecision(3) << 1331.123456 << std::endl; // 1.331e+03
  std::cout << std::dec << 1331 << std::endl; // 1331
  std::cout.fill('X');
  std::cout.width(8);
  std::cout << 1331 << std::endl; // XXXX1331
  std::cout.setf(std::ios::left, std::ios::adjustfield);
  std::cout.width(8);
  std::cout << 1331 << std::endl; // 1331XXXX
}
```

Type Casting

- Types must be known at compile-time
- Type-checking happens at compile-time

```
1 std::string name = "Obama";
2 int age = 17;
3 age = name; // error
```

Type Conversions

- Implicit: Compiler-directed conversions
- Explicit: Programmer-specified conversions

```
1 // Implicit type conversion
2 #include <iostream>
3
4 int main() {
5   int age = 17;
6   double agePrecise = age;
7   std::cout << age;
8 }
```

Type Conversions - Explicit

- This is also known as static casting

```
1 double pi = 3.14;
2 int piInteger1 = pi; // What happens here?
3 int piInteger2 = (int)pi; // C-style
4 int piInteger3 = static_cast<int>(pi); // C++ style
5
6 int x = 5, y = 2;
7 // Note how the order of operations is not immediately obvious with C-style casts
8 double slope = (double)x / y; // C-style
9 double slope = static_cast<double>(x) / y; // C++ style
```

Iterating through arrays in C++

```cpp
#include <array>
#include <iostream>

int main() {
  // C-style. Don't do this
  // int ages[3] = { 18, 19, 20 };
  // for (int i = 0; i < 3; ++i) {
  //   std::cout << ages[i] << "\n";
  // }

  // C++ style. This can be used like any other C++ container.
  // It has iterators, safe accesses, and it doesn't act like a pointer.
  std::array<int, 3> ages{ 18, 19, 20 };

  for (int i = 0; i < ages.size(); ++i) {
    std::cout << ages[i] << "\n";
  }
  for (auto it = ages.begin(); it != ages.end(); ++it) {
    std::cout << *it << "\n";
  }
  for (const auto& age : ages) {
    std::cout << age << "\n";
  }
}
```

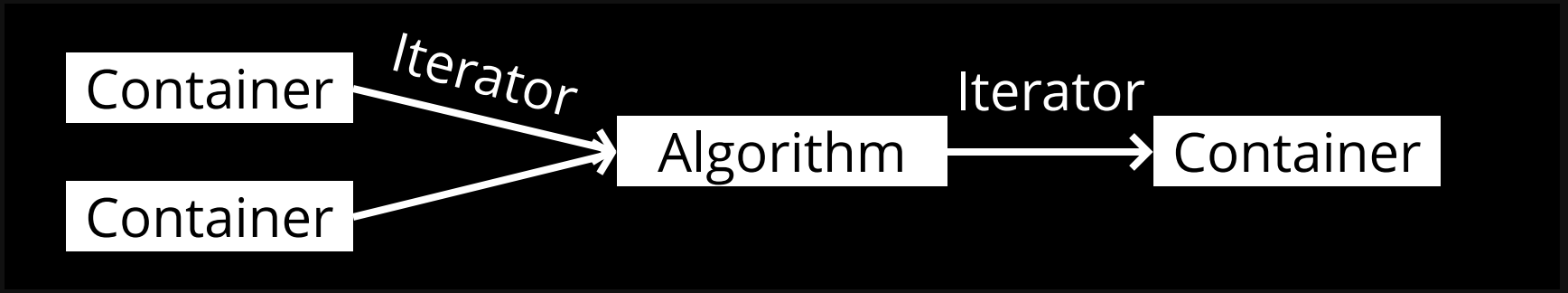
Function Templates

- A **function template** is a **prescription** for the **compiler** to generate particular instances of a function varying by type
  - A function template is **not a function**. It tells the compiler how to generate functions
- We will talk about this in a lot more detail in later weeks

```
1 template <typename T>
2 T Min(T a, T b) {
3   return a < b ? a : b;
4 }
5
6 int main() {
7   Min(1, 2); // uses int min(int, int);
8   Min(1.1, 2.2); // double min(double, double);
9 }
```

STL: Standard Template Library

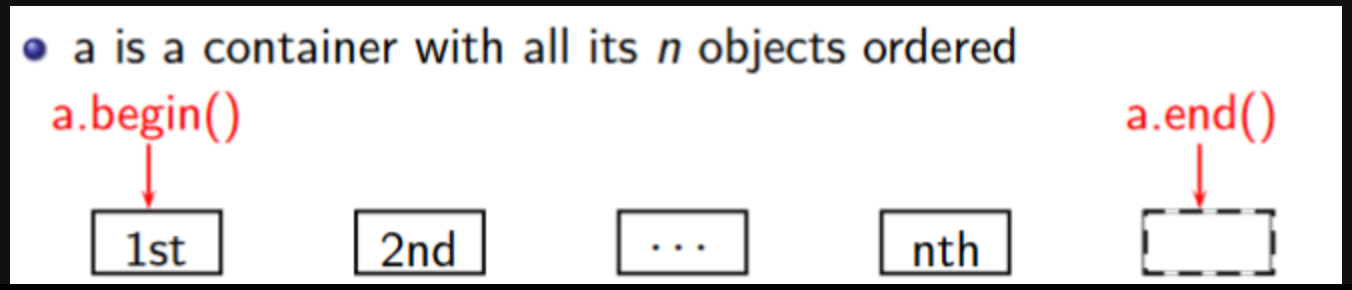
- STL is an architecture and design philosophy for managing generic and abstract collections of data with algorithms
- All components of the STL are templates
- Containers store data, but don't know about algorithms
- Iterators are an API to access items within a container in a particular order, agnostic of the container used
  - Each container has its own iterator types
- Algorithms manipulate values referenced by iterators, but don't know about containers



STL: Iterators

- Iterator is an abstract notion of a **pointer**
- Iterators are types that abstract container data as a sequence of objects
  - The glue between containers and algorithms

STL: Iterators




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

int main() {
  std::vector<std::string> names;
  for (auto iter = names.begin(); iter != names.end(); ++iter) {
    std::cout << *iter << "\n";
  }
  for (std::vector<std::string>::iterator iter = names.begin(); iter != names.end(); ++iter) {
    std::cout << *iter << "\n";
  }
}
```

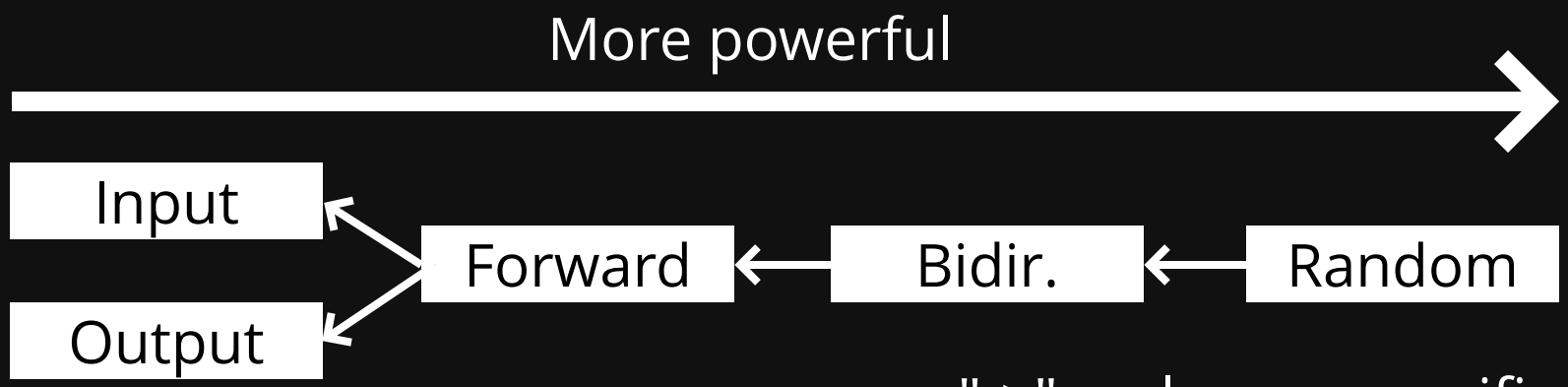
- **a.begin()**: abstractly "points" to the first element
- **a.end()**: abstractly "points" to one past the last element
  - a.end() is not an invalid iterator value
- If iter abstractly points to the **k-th** element, then:
  - ***p** is the object it abstractly points to
  - **++p** abstractly points to the (k + 1)-st element

Iterators, Constness, Reverse

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> ages;
  ages.push_back(18);
  ages.push_back(19);
  ages.push_back(20);

  // type of iter would be std::vector<int>::iterator
  for (auto iter = ages.begin(); iter != ages.end(); ++iter) {
    (*iter)++; // OK
  }

  // type of iter would be std::vector<int>::const_iterator
  for (auto iter = ages.cbegin(); iter != ages.cend(); ++iter) {
    //(*iter)++; // NOT OK
  }

  // type of iter would be std::vector<int>::const_iterator
  for (auto iter = ages.rbegin(); iter != ages.rend(); ++iter) {
    std::cout << *iter << "\n"; // prints 20, 19, 18
  }

  // Can also use crbegin and crend
}
```

Iterator Categories

Operation	Output	Input	Forward	Bidirectional	Random Access
Read		=*p	=*p	=*p	=*p
Access		->	->	->	-> []
Write	*p=		*p=	*p=	*p=
Iteration	++	++	++	++ --	++ -- + - += -=
Compare		== !=	== !=	== !=	== != < > <= >=



"->" no longer specified as of C++20

Iterator Categories

An **algorithm** requires certain kinds of iterators for their operations

- **input**: find(), equal()
- **output:** copy()
- **forward:** replace(), binary_search()
- **bi-directional**: reverse()
- **random:** sort()

A **container's** iterator falls into a certain category

- **forward:** forward_list
- **bi-directional**: map, list
- **random:** vector, deque

stack, queue are container adapters, and do not have iterators

Other Iterators: Streams

```
1  #include <fstream>
2  #include <iostream>
3  #include <iterator>
4
5  int main() {
6    std::ifstream in("data.in");
7
8    std::istream_iterator<int>begin(in);
9    std::istream_iterator<int> end;
10   std::cout << *begin++ << "\n"; // read the first int
11
12   ++begin; // skip the 2nd int
13   std::cout << *begin++ << "\n"; // read the third int
14   while (begin != end) {
15     std::cout << *begin++ << "\n"; // read and print the rest
16   }
17 }
```

STL: Containers

- STL containers are abstractions of common data structures
- cppreference has a summary of them here.
- Different containers have different time complexity of the same operation (see right)

Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

O(1)+ means amortised constant time

Sequential Containers

- Elements have specific order controlled by programmer
- These are all templates (but class templates, not function templates)
- Note that vector is still your go-to type for this sort of data. It is typically faster, so don't use another type without a good reason.

Sequential container	**Description**
std::vector	Dynamically sized array
std::list	Doubly linked list
std::forward_list	Singly linked list
std::deque	<vector> with fast operations for element at beginning
std::array	C-style array wrapper

Vector (Container)

- Array-like container most used is <vector>
  - Abstract, dynamically resizable array
  - In later weeks we will learn about various ways to construct a vector

```
#include <iostream>
#include <vector>

// Begin with numbers 1, 2, 3 in the list already
int main() {
  // In C++17 we can omit the int if the compiler can determine the type.
  std::vector<int> numbers {1, 2, 3};
  int input;
  while (std::cin >> input) {
    numbers.push_back(input);
  }
  std::cout << "1st element: " << numbers.at(0) << "\n"; // slower, safer
  std::cout << "2nd element: " << numbers[1] << "\n"; // faster, less safe
  std::cout << "Max size before realloc: " << numbers.capacity() << "\n";
  for (int n : numbers) {
    std::cout << n << "n"
  }
}
```

Associative Containers

- A **value type** is accessed through a second data type, the **key**.
- Associative containers include:
  - map<T>
    - log(n) for most operations, probably stored as a red-black tree
    - Ordered by key value (requires key to be comparable with <)
    - Iterators will iterate through in order of key, not by insertion time
  - unordered_map<T>
    - O(1) for most operations
    - Stored as a hash table (requires keys to be hashable)
    - Iterators will iterate through in an arbitrary, undefined order
  - set<T>
    - Search, removal, insertion have log(n) complexity
    - Contains sorted set of unique objects of type Key

std::map example

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

int main() {
  std::map<std::string, double> m;
  // The insert function takes in a key-value pair.
  std::pair<std::string, double> p1{"bat", 14.75};
  m.insert(p1);
  // The compiler will automatically construct values as
  // required when it knows the required type.
  m.insert({"cat", 10.157});
  // This is the preferred way of using a map
  m.emplace("cat", 10.157);

  // This is very dangerous, and one of the most common causes of mistakes in C++.
  std::cout << m["bat"] << '\n';

  auto it = m.find("bat"); // Iterator to bat if present, otherwise m.end()

  // This is a great example of when to use auto, but we want to show you what type it is.
  for (const std::pair<const std::string, double>& kv : m) {
    std::cout << kv.first << ' ' << kv.second << '\n';
  }
}
```