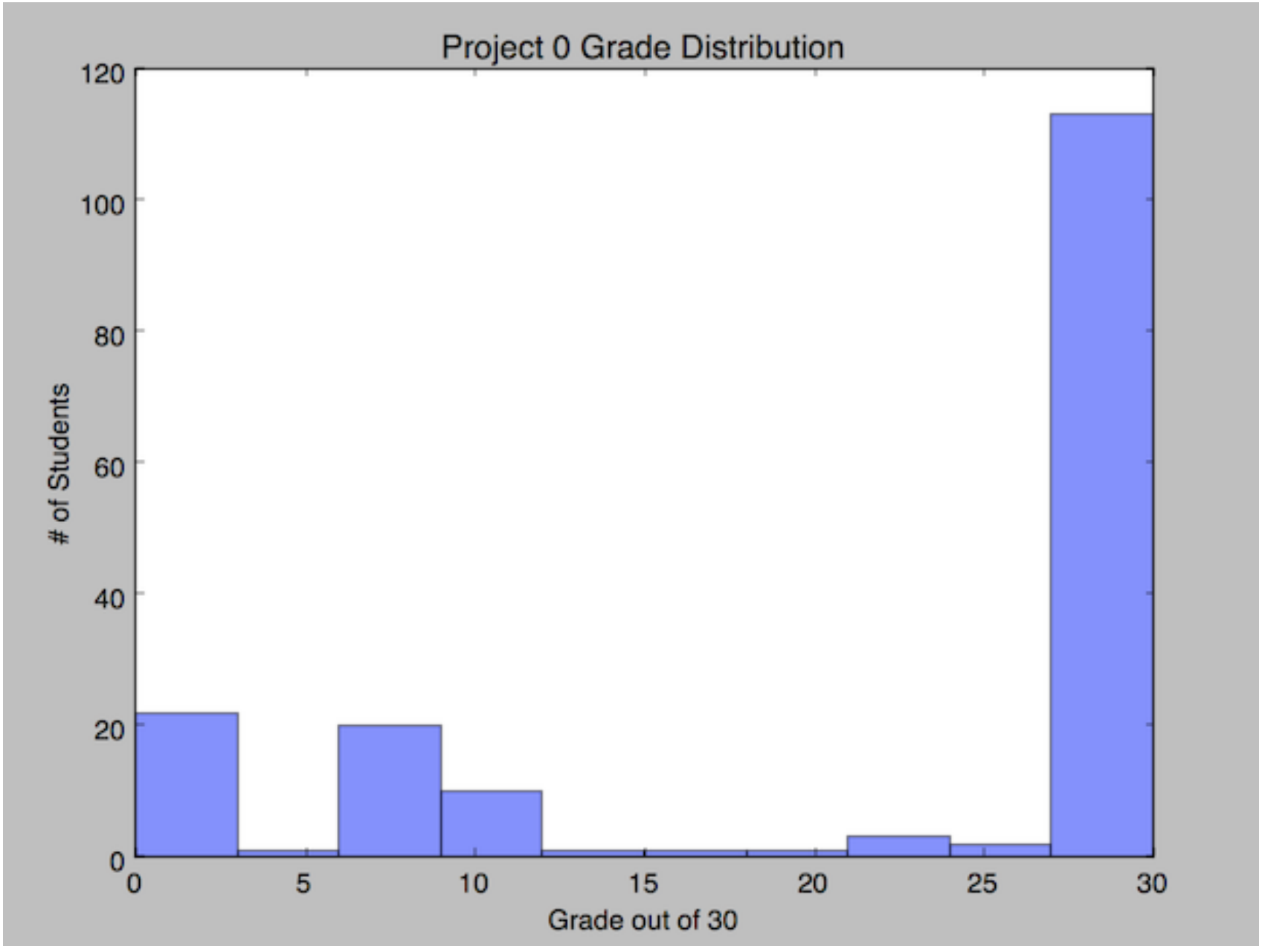
ECE 3574: Applied Software Design: C++ Standard Library

Chris Wyatt

Project 0


Project 0 Grade Distribution
# of Students
Grade out of 30
0
20
40
60
80
100
120
0
5
10
15
20
25
30

Project 0

- If you could not do this project ($<= 10$) you are in serious danger of failing.
- Please complete the Project 0 Survey on Canvas (2 quick questions, counts as an exercise)

The goal of today's meeting is to review the standard library

- Containers and Iterators
- Algorithms
- Exercise 5

"The best code is that already written and tested"

The C++ standard library is well-constructed and tested

- prefer to use containers and algorithms from the standard library rather than hand-coded data structures and algorithms.

In 2574 you saw how to implement data structures and common algorithms for sorting and searching. However, the C++ standard library provides implementations of these that are efficient and well tested, so you should prefer to use them over hand-coded approaches whenever feasible.

std::array is a wrapper around raw arrays

- supports standard access members (at, [], front, back)
- has a size() member
- supports fill and swap
- can be empty
- very low overhead

Example:

```
std::array<int,10> a;
a.fill(1);
assert(a[3] == 1);
assert(a.size() == 10);
```

std::vector is a dynamically sized array-based container

- the most useful linear data structure
- see members size, capacity, and reserve
- grows exponentially
- supports insert - much more efficient than you might think
- watch out for iterator invalidation

Example

```
std::vector<int> v;
std::cout << v.capacity() << std::endl;
for(int i = 0; i < 100; ++i){
    v.push_back(i);
    std::cout << v.capacity() << std::endl;
}
```

std::deque is a dynamically sized double ended queue

- not contiguous in memory
- access either end: push_front or push_back
- generally better performance than std::list

Example:

```
std::deque<int> d;
 for(int i = 0; i < 100; ++i){
   d.push_back(i);
   d.push_front(i);
 }

 return 0;
```

std::list and std::forward_list

- doubly and singly linked-lists respectively
- constant time insertion anywhere
- no random access
- std::list supports bidirectional iteration
- space efficient, no extra space as in std::vector
- can be less efficient than std::vector because of cache misses

adaptors provide wrappers around other containers

- stack (deque)
- queue (deque)
- priority_queue (a heap using vector for storage)

std::map and std::multimap are dictionaries (key,value)

- std::map requires unique keys and value
- implemented as red-black tree (balanced binary tree)
- index operator[] is very handy

Example:

```
std::map<std::string, int> occurances;
occurances["hello"] += 1;
occurances["hello"] += 1;
occurances["goodbye"] += 1;

for(auto it = occurances.begin();
    it != occurances.end();
    ++it)
  {
    std::cout << "You said " << it->first << " "
              << it->second << " times." << std::endl;
  }
```

See also std::set and std::multiset (no value, just a key)

Hash tables are in the C++ stdlib now!

- unordered_set / unordered_map
- unordered_multiset / unordered_multimap
- constant (amortized) time find, insert, remove

Same Example

```
std::unordered_map<std::string, int> occurances;
occurances["hello"] += 1;
occurances["hello"] += 1;
occurances["goodbye"] += 1;

for(auto it : occurances)
  {
    std::cout << "You said " << it->first << " "
              << it->second << " times." << std::endl;
  }
```

algorithms library

- Non-modifying sequence operations
- Modifying sequence operations
- Partitioning operations
- Binary search
- Set operations
- Heap operations
- min/max
- numeric (see random number generators too)

Exercise 5

See Website

Next Actions and Reminders

- Read The Pragmatic Programmer Sections 7, 8, 26
- Project 1 is released. Read through it ASAP.