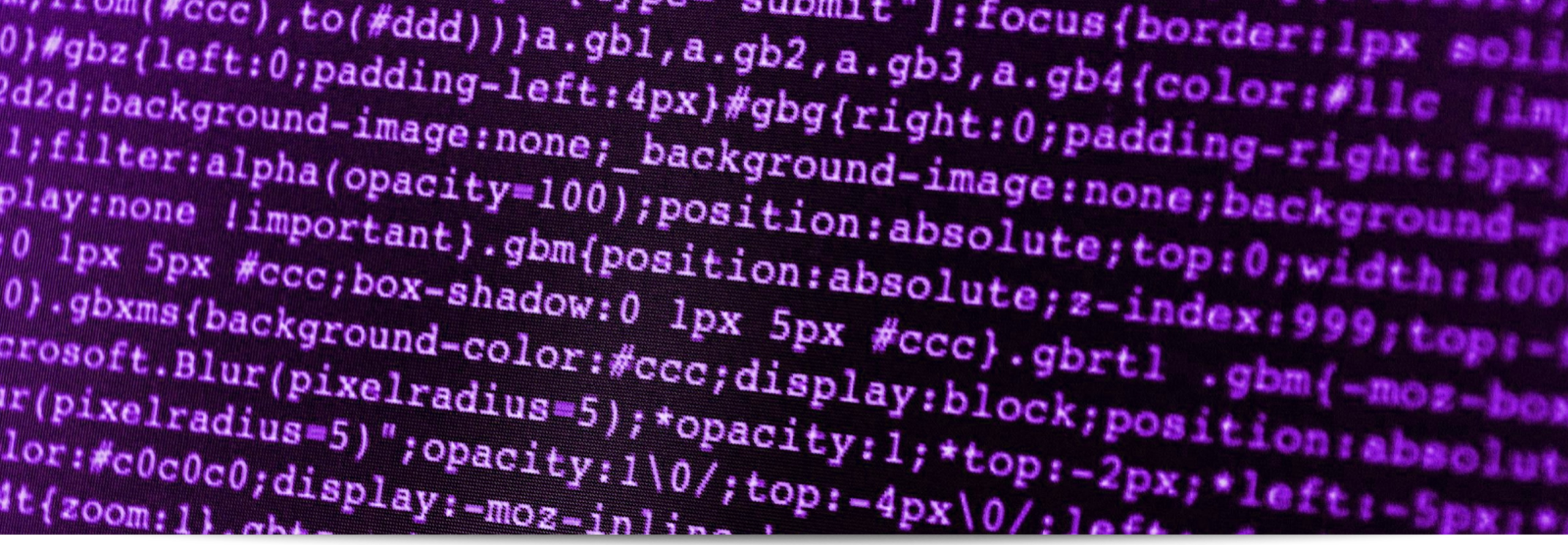


06

October 22-October 28, 2024

Unordered Maps and Sets, Enumerated Classes

Agenda

- Unordered Maps and Sets
- Enum classes
- Emplacement
- Using declarations
- Handwritten Problem

Announcements

- Lab 6 Handwritten is due IN LAB by **Monday, October 28**
- Lab 6 assignment: anonymous post-midterm feedback form! **due Monday, October 28**
  - Click here to access the feedback form.
  - **If more than 75% of the class submits, everyone will get credit.**
- Also, please provide IA/GSI feedback as well!
  - Helpful for all of us, so that we can address any concerns as we enter the second half of the semester.
  - Anonymous! Click here for the form.
- Lab 6 Quiz includes *optional, ungraded* questions. This is for your own practice! No need to submit anything besides surveys and handwritten.
- Project 3 releases on **Thursday, October 24,** and is due **Tuesday, November 12**
- Grade projections will be sent out soon.

Feedback Survey

(required)

https://forms.gle/djraTvV3AGdGxLh29

IA/GSI Survey

(Not required)

https://forms.gle/LkyvwfuaxWmCUu469

Lab 5 Handwritten Review

Handwritten Problem

- Given a vector with *n* elements with values of either 0, 1, or 2, devise an O(n) algorithm to sort this vector. You must do this in a **single pass** of the vector. You may **NOT** copy items, create arrays or strings, or do any other memory allocation. Make sure your algorithm works for all cases.
- You may use `std::swap` to swap two items in the vector.

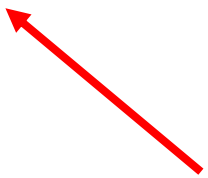
```
BEFORE: arr[] = {2, 1, 0, 0, 2, 1, 2, 2, 0, 1, 1, 1, 0}
AFTER: arr[] = {0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2}
```

```
// sort a vector of 0s, 1s, and 2s in linear time
void sort012(vector<int>& nums);
```

Handwritten Problem Solution

- The “Dutch National Flag Algorithm”

```
void sort012(vector<int>& nums) {
    int begin = 0, end = nums.size();
    for (int i = begin; i != end;){
        if (nums[i] == 0) {
            swap(nums[i++], nums[begin++]);
        } else if (nums[i] == 2) {
            swap(nums[i], nums[--end]);
        } else {
            i++;
         }
    }
}
```



Why not i++ here?
Consider the edge Case: [1, 2, 0]

Maps and Sets

A Blast From the Past

- Remember `isAnagram()`?
  - determine if two strings are anagrams (or if they have the same letter count)
- Required O(1) "lookup" for how many times a character occurs.
- Only 26 possible characters, so a vector of size 26 worked perfectly.
  - But what if we didn't want to index with numbers?

```
bool isAnagram(string s1, string s2) {
    vector<int> charsCount(26, 0);

    for (char letter: s1)
        if (letter != ' ')
            ++charsCount[letter - 'a'];

    for (char letter: s2) {
        if (letter != ' ') {
            if (--charsCount[letter - 'a'] < 0) {
                return false;
        }
    }

    for (int count: charsCount) {
        if (count > 0) {
            return false;
        }
    }

    return true;
}
```

Map of char -> int

"hash function"

Storing Student Data

- Motivating Problem:
  - you have to store the names of every student in EECS 281.
  - every student has a unique ID in the range [0, 750).
  - querying by student ID must be O(1).

Storing Student Data

- Motivating Problem:
  - you have to store the names of every student in EECS 281.
  - every student has a unique ID in the range [0, 750).
  - querying by student ID must be O(1).
- Solution:
  - store the names of each student in a vector of strings, and index by student ID to get the name of the student!

Storing Student Data

- Motivating Problem:
  - you have to store the names of every student in EECS 281.
  - every student has a **uniqname** that is a string of ASCII characters.
  - querying by **uniqname** must be O(1).
- Problem:
  - unlike the previous example, this isn't as easy... we could use a vector, but how would we know which index each uniqname would go to?
  - we would need to "map" each uniqname to some sort of index so that we know where to put each uniqname!

Storing Student Data

- Motivating Problem:
  - you have to store the names of every student in EECS 281.
  - every student has a **uniqname** that is a string of ASCII characters.
  - querying by **uniqname** must be O(1).
- Problem:
  - unlike the previous example, this isn't as easy... we could use a vector, but how would we know which index each uniqname would go to?
  - we would need to "map" each uniqname to some sort of index so that we know where to put each uniqname!
- Solution:
  - use an `unordered_map` and index by uniqname!

Storing Student Data

```
struct Student {
    string uniqname;
    string full_name;
    vector<double> grades;
};

std::unordered_map<string, Student> all_students;

string get_full_name(string uniqname) {
    return all_students[uniqname].full_name;
}
```

Storing Student Data

```
struct Student {
    string uniqname;
    string full_name;
    vector<double> grades;
};

std::unordered_map<string, Student> all_students;

void add_grade(string uniqname, double grade) {
      all_students[uniqname].grades.push_back(grade);
}
```

Advantages of an Unordered_Map

- Consider that a uniqname is between 3 and 8 characters - over 200 billion combinations!
- Using an `unordered_map` allows us to only consider uniqnames that currently exist!
  - Using a vector or array will make searching O(log n) with binary search if we sort the list beforehand.
  - Using an `unordered_map` allows for O(1) search by being directly indexable with a string which is a key value in our map.

Using an Unordered_Map

- Use a key value for identification and a mapped value to be stored
- Unique keys
- Not sorted
- Complexities for finding a value given a key and inserting a key into the hash table are average O(1) and worst case O(n)
- Keys must be hashable!
  - Hashing - Separate arbitrary data into smaller groups by exploiting the uniqueness of its value (similar to Rabin-Karp fingerprinting for strings)
  - hashing can be treated as an operator just like (), +, or *
  - Most of the primitive-types and `std::string` are hashable by default
- Elements within the `unordered_map` can be retrieved as a `std::pair`, where `first` represents the key and `second` represents the value.

Review: Pairs

- A **pair** is a data type that wraps two values (which can be of different types) and treats them as a single unit.
- The two values of a pair can be accessed using its members, `first` and `second`. This is similar to how a `struct` with two members would behave!
- Possible implementation:

```
template<typename T1, typename T2>
struct pair{
      T1 first;
      T2 second;
      pair(const T1 &f, const T2 &s) :first(f), second(s) {}
};
```

Constructing a Pair

```cpp
#include <utility>
#include <string>

// Method #1
auto pair1 = std::make_pair("eecs", 281);

// Method #2
std::pair<std::string, int> pair2{"eecs", 281};

// Method #3
std::pair<std::string, int> pair3 = {"eecs", 281};
```

Accessing Elements in a Pair

- To access elements of a pair, use `.first` for the first value and `.second` for the second value.

```
#include <utility>
#include <string>

// Make pair
std::pair<std::string, int> myPair = std::make_pair("pilots", 21);

// Modify the elements of the pair
myPair.first = "math";
myPair.second = 217;
```

Comparing Pairs

- Equality: Two pairs are equal if and only if they have identical `first` and `second` values.
- Less than and greater than comparisons: first compare the `first` elements and, in the case of a tie, compare the `second` elements.
  - If `pair1 = {"parrot", 2}` and `pair2 = {"carrot", 3}`

    `pair1 < pair2` would return **`false`** (since 'p' > 'c')
  - If `pair1 = {"parrot", 2}` and `pair2 = {"parrot", 3}`

    `pair1 < pair2` would return **`true`** (since "parrot" == "parrot" and 2 < 3)

Unordered_Map Operations

Function	Effect
`operator[]`	Gives a reference to the value-object with the corresponding key. Will create a **default** value-object if the key is not found. **Computes hash-function every time.**
`.find()`	Returns an iterator to the key-value pair matching a certain key, end() if it doesn't exist.
`.insert()`	Takes in a key-value pair, tries to insert the pair, and returns a pair containing an iterator to the key-value pair with a bool for whether the insertion actually took place (this function cannot change the existing value).
`.insert_or_assign()`	Same as insert, but will change an existing value.
`.erase()`	Removes a key-value pair from the hash table.
`.begin() and .end()`	Returns a **`ForwardIterator`** that will traverse key-value elements in *some* order.

Adding and Accessing Elements

```
using Name = string; // “Name” means “string” now
using FavColor = string;

int main() {
   unordered_map<Name, FavColor> favorite_colors;

      favorite_colors[“mrkevin”] = “orange”;
      favorite_colors[“paoletti”] = “grey”;

      cout << favorite_colors[“paoletti”] << endl; // prints “grey”
      cout << favorite_colors.size() << endl;      // prints 2
      cout << favorite_colors[“nobody”] << endl;   // ???
}
```

Adding and Accessing Elements

```
using Name = string; // “Name” means “string” now
using FavColor = string;

int main() {
   unordered_map<Name, FavColor> favorite_colors;

      favorite_colors[“mrkevin”] = “orange”;
      favorite_colors[“paoletti”] = “grey”;

      cout << favorite_colors[“paoletti”] << endl; // prints “grey”
      cout << favorite_colors.size() << endl;      // prints 2
      cout << favorite_colors[“nobody”] << endl;   // inserts “” into
table!
}
```

Adding and Accessing Elements

```
using Name = string; // “Name” means “string” now
using FavColor = string;

int main() {
   unordered_map<Name, FavColor> favorite_colors;

      favorite_colors[“mrkevin”] = “orange”;
      favorite_colors[“paoletti”] = “grey”;

      cout << favorite_colors[“paoletti”] << endl; // prints “grey”
      cout << favorite_colors.size() << endl;      // prints 2
      cout << favorite_colors[“nobody”] << endl;   // inserts “” into
table!

      cout << favorite_colors.size() << endl; // prints ???

}
```

Adding and Accessing Elements

```
using Name = string; // “Name” means “string” now
using FavColor = string;

int main() {
   unordered_map<Name, FavColor> favorite_colors;

      favorite_colors[“mrkevin”] = “orange”;
      favorite_colors[“paoletti”] = “grey”;

      cout << favorite_colors[“paoletti”] << endl; // prints “grey”
      cout << favorite_colors.size() << endl;      // prints 2
      cout << favorite_colors[“nobody”] << endl;   // inserts “” into
table!

      cout << favorite_colors.size() << endl; // prints 3

}
```

Using find

```
unordered_map<Name, FavColor> favorite_colors;

favorite_colors[“mrkevin”] = “orange”;
favorite_colors[“paoletti”] = “grey”;

Name input_name = argv[1];

auto found_it = favorite_colors.find(input_name);
if (found_it == favorite_colors.end()) {
   cout << “Name not found: ” << input_name << endl;
} else {
   cout << found_it->first << “‘s favorite color is: ” << found_it->second;
}
```

prevents operator[] from adding in keys that do not exist!

Common performance pitfall

```
// Two lookups

if (map.find(x) != map.end()) {
      std::cout << “Found: ” << map[x] << std::endl;
}


// One lookup

if (auto iter = map.find(x); iter != map.end()) {
      std::cout << “Found: ” << iter->second << std::endl;
}
```

Common performance pitfall

```
// Multiple lookups
employees[x].salary -= 1000;
employees[x].title = “manager”;
employees[x].years_working = 3;

// One lookup
auto iter = employees.find(x);
iter->second.salary -= 1000;
iter->second.title = “manager”;
iter->second.years_working = 3;

// One lookup, better style
auto& employee = employees[x];
employee.salary -= 1000;
employee.title = “manager”;
employee.years_working = 3;
```

watchout for multiple instances of operator[] use with the same key

Another pitfall: forgetting reference specifier on auto

Multiple Choice Practice

Hash Applications

For which of the following applications would a hash table **NOT** be appropriate? Select all that apply.

1. printing out all of the elements in sorted order
2. storing passwords that can be looked up by uniqname
3. returning a person's name given their phone number
4. finding the kth largest element in an array
5. creating an index for an online book

Hash Applications

For which of the following applications would a hash table **NOT** be appropriate? Select all that apply.

1. **printing out all of the elements in sorted order**
2. storing passwords that can be looked up by uniqname
3. returning a person's name given their phone number
4. **finding the kth largest element in an array**
5. creating an index for an online book

Symmetric Pairs

Two pairs (a, b) and (c, d) are symmetric if b = c and a = d. Suppose you want to find all symmetric pairs in the array. The first element of all pairs is distinct.

For example, given {(14, 23), (11, 2), (52, 83), (49, 38), (38, 49), (2, 11)},

the symmetric pairs are {(11, 2), (2, 11)} and {(49, 38), (38, 49)}.

What is the average-case time complexity of accomplishing this task if you use the most efficient algorithm? If hashing is involved, assume that both search and insert methods work in Θ(1) time.

1. Θ(1)
2. Θ(log n)
3. Θ(n)
4. Θ(n log n)
5. Θ(n^2)

Symmetric Pairs

Two pairs (a, b) and (c, d) are symmetric if b = c and a = d. Suppose you want to find all symmetric pairs in the array. The first element of all pairs is distinct.

For example, given {(14, 23), (11, 2), (52, 83), (49, 38), (38, 49), (2, 11)},

the symmetric pairs are {(11, 2), (2, 11)} and {(49, 38), (38, 49)}.

What is the average-case time complexity of accomplishing this task if you use the most efficient algorithm? If hashing is involved, assume that both search and insert methods work in Θ(1) time.

1. Θ(1)
2. Θ(log n)
3. **Θ(n)**
4. Θ(n log n)
5. Θ(n^2)

Use a hashtable! The first element of each pair is the key, and the second element is the value.

Go through the array once. For each current pair, check if the current second element is in the hashtable, and if it is, check that its value is the same as the current first element. If it matches, it's a symmetric pair.

Interview Problems

Interview Problem

- Given a vector of integers, find the first non-repeated integer.
  - Example:
    - the first non-repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is 10.

Interview Problem

- Given a vector of integers, find the first non-repeated integer.
  - Example:
    - the first non-repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is 10.

- Use an `unordered_map`:
  - scan the vector from left to right and construct an `unordered_map<int,int>` to count the number of appearances for each number
  - after the `unordered_map` is completed, scan the vector from left to right and check the count for each character. If an element has a count of 1, return it.
  - can't scan the `unordered_map`, because it's unordered, need the FIRST one
  - time complexity O(n) *on average*, space complexity O(n)

Modifying the Problem

- Given a vector of integers, find the first non-repeated integer.
  - Example:
    - the first non-repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is 10.

- Modification: each element can only appear at most twice
  - can we improve the efficiency of our previous solution?

Modifying the Problem

- Given a vector of integers, find the first non-repeated integer.
  - Example:
    - the first non-repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is 10.

- Modification: each element can only appear at most twice
  - can we improve the efficiency of our previous solution?
  - still need two passes, since we want the *first* non-repeated element
  - do we have the potential to save some space?

Modifying the Problem

- Given a vector of integers, find the first non-repeated integer.
  - Example:
    - the first non-repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is 10.

- Modification: each element can only appear at most twice
  - can we improve the efficiency of our previous solution?
  - still need two passes, since we want the *first* non-repeated element
  - do we have the potential to save some space?

- If we see an element twice, can we erase it from the hash table?
  - Yep! At this point we know we'll never have to consider this element again.

Modifying the Problem

- Given a vector of integers, find the first **repeated** integer.
  - Example:
    - the first repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is -234.

- How does finding the first repeated (rather than non-repeated) integer change the problem?

Modifying the Problem

- Given a vector of integers, find the first **repeated** integer.
  - Example:
    - the first repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is -234.

- How does finding the first repeated (rather than non-repeated) integer change the problem? **We no longer care about the count! As long as the integer is in our hash table, we must have seen it before.**

- As a result, the "value" associated with each key doesn't really matter to us. Is there an alternative container we could use?

Unordered_Sets

- An unordered_set stores unique elements in no particular order (i.e. keys are also values).
- Complexities for finding/inserting are average O(1) and worst-case O(n).
- The solution to the previous problem:

```
#include <unordered_set>
void find_first_duplicate(const vector<int>& input) {
    unordered_set<int> s;
    for (auto number : input) {
            if (s.find(number) == s.end()) {
                    s.insert(number);
        } else {
                    cout << “First repeated number is: ” << number << endl;
                    break;
        }
    }
}
```

once we find an element in the set, we must have seen it before, so we return it!

Unordered vs. Ordered

- As the name implies, the ordered versions of this container maintain sorted order.
  - Useful when we need to do an operation on a range - like printing elements between X and Y
  - Also means **complexities differ** - O(log n) insertion and find on ordered map/set.
- (ordered) set: unique elements, **following a specific order (sorted using comparator)**. Declare using

```
#include <set>
set<int> s;
```

- (ordered) map: elements formed by a combination of a key value and a mapped value, **following a specific order (sorted using comparator)**. Declare using

```
#include <map>
map<int, char> m;
```

Enum classes

Enumerated Type (enum class)

- A type whose values are restricted to a set of values
  - more efficient than using strings (e.g. “stack_mode” and “queue_mode” in P1)
  - more readable than using ints or characters (‘s’ or ‘q’ or 120 or 121)
- Intuition: implies a set of related constants, more readable

Using an Enumerated Class

```
enum class Color { red, green, blue };
Color favorite_color = Color::red;
switch (favorite_color) {
    case Color::red:
        std::cout << "red" << std::endl;
        break;
    case Color::green:
        std::cout << "green" << std::endl;
        break;
    case Color::blue:
        std::cout << "blue" << std::endl;
        break;
}
```

Using an Enumerated Class

- Enum values are actually represented as an integer type (`int` by default).
- If the first enumerator does not have an initializer, the associated value is zero.
- For an enumerator without an initializer, its associated value is the previous plus one:

```
enum class Nums { zero /*0*/, one /*1*/, two /*2*/, three /*3*/ };
```

- You can also choose constant values for each enum:

```
enum class Foo { a /*0*/, b /*1*/, c = 10, d /*11*/, e = 1, f /*2*/, g = f + c /*12*/ };
```

- You can get these values by `static_cast<int>(enum_val)`; enum values also have <, >, ==, >=, <=, !=.

- If `int` is too big (or too small), you can also use a different underlying integer type:

```
enum class RoutingMode : char { queue, stack };
```

Emplacement

Emplace

Problem:

```
std::vector<Foo> v;
v.push_back(Foo(70, ‘g’));
```

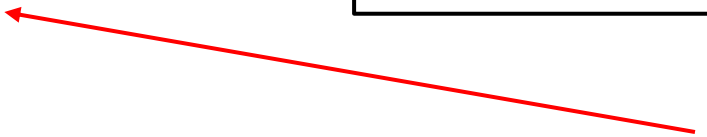
```
struct Foo{
        int a;
        char b;
        Foo(int a_, char b_) : a(a_), b(b_) {}
};
```

Emplace

Problem:

```
std::vector<Foo> v;
v.push_back(Foo(70, ‘g’));
```

```
struct Foo{
        int a;
        char b;
        Foo(int a_, char b_) : a(a_), b(b_) {}
};
```



This calls default Foo constructor, and then Foo’s move or copy constructor. Unnecessary work!

Emplace

Problem:

```
std::vector<Foo> v;
v.push_back(Foo(70, ‘g’));
```

```
struct Foo{
        int a;
        char b;
        Foo(int a_, char b_) : a(a_), b(b_) {}
};
```

This calls default Foo constructor, and then Foo’s move or copy constructor. Unnecessary work!

Solution:

```
v.emplace_back(70, ‘g’);
```

emplace_back forwards constructor arguments and creates Foo in-place. The extra move/copy is avoided.

Emplace

emplace is supported by every STL container with push or insert support

Forwards constructor arguments to have the container construct in-place and avoid big copies

- **Parameters must match a constructor for the container's data type**

Example:

```
struct Foo{
        int a;
        char b;
        Foo(int a_, char b_) : a(a_), b(b_) {}
};
```

```
std::vector<Foo> fighters;
//void push_back(Foo);
fighters.push_back(Foo(70, 'g'));
// void emplace_back(int, char);
fighters.emplace_back(70, 'g'); // <- matches constructor
```

Using declarations

Using declarations

Problem: Long type names but we can't or don't want to use templates...

```
std::unordered_map<KeyType, std::vector<OtherType>> var;
var[x_type] = std::vector<OtherType>(N, dflt);
```

Is there a way to declare an alias so we don't have to type everything out every time?

Using declarations

```
using Row = std::vector<OtherType>;
```

Now we can make things shorter

```
std::unordered_map<KeyType, std::vector<OtherType>> var;
std::unordered_map<KeyType, Row> var;

var[x_type] = std::vector<OtherType>(N, dflt);
var[x_type] = Row(N, dflt);
```

Handwritten Problem

Handwritten Problem

Given an array of **distinct** integers, find if there are two pairs (a, b) and (c, d) such that a+b=c+d, and a, b, c, and d are distinct elements. If there are multiple elements, the function should print **all** pairs that work. You may assume that for any pair (a, b), there is at most one other pair (c, d) that sums to a+b.

Examples:

Input: `[3, 4, 7, 1, 2, 8]`

Output:

`(3, 2) and (4, 1)`

`(3, 7) and (2, 8)`

`(3, 8) and (4, 7)`

`(7, 2) and (1, 8)`

Input: `[65, 30, 7, 90, 1, 9, 8]`

Output (nothing):

Expected Runtime: $O(n^2)$

```
// Prints out all different pairs in input vec that have same sum.
```