COP3503

maps

std::map<k,v> class

```
#include <map>
using std::map;
```

- A map is a **non-contiguous**, template data structure that stores data in **pairs**.
- A pair is made of a two different parts:
  - A **key**: An identifier, some label for the other part of the pair.
  - A **value**: the information being labeled, the "real" data in the map.

Both keys and values can be any data type.

Keys are commonly strings or an **integral** (whole number) value.

We tend to label things with strings and numbers, and floating-point numbers may have accuracy problems.

- Internally, a map is implemented as a type of **binary tree** (you'll cover trees in later courses).
- They can very quickly store and retrieve elements.
- They are automatically sorted (we'll see an example later)

Creating map Variables

This is how you access it.

This is what we are really interested in.

```
map<key, value> someVariable;
```

```
map<int, string> myMap;
```

(storing strings)

```
map<int, int> myMap;
```

(also storing ints)

```
map<string, int> myMap;
```

(storing ints)

```
map<short, double> myMap;
```

(storing doubles)

```
map<int, Hero> myMap;
```

(storing Heroes, a user-defined class)

```
map<char, float> myMap;
```

(storing floats)

```
map<string, vector<int>> data;
```

(storing vector<int>)

Containers holding containers… sure, why not?

Think of a box containing smaller boxes, a backpack containing a pencil case, etc.

Adding Data to a map With `emplace()`

```
map<int, string> myMap;              // Integer keys, string values
myMap.emplace(12, "Bob");            // Add a pair<int, string> to the map
myMap.emplace(0, "Superman");        // Add a second pair
```

12 and 0 are arbitrary keys for this example.

Keys are needed to later access that data.

```
map<string, int> characters;          // String keys, integer values
characters.emplace("Spider-Man", 17);   // Key: Spider-Man  Value: 17
characters.emplace("Yoda", 900);
```

```
map<string, string> courses;     // String keys, string values
courses.emplace("COP3503", "Programming Fundamentals 2");
courses.emplace("COT3100", "Applications of Discrete Structures");
```

Adding and Modifying Data with operator[]

- The subscript operator can be used to add key/value pairs, **or** modify values

```
map<int, string> myMap;
```

```
// If the specified key doesn't already exist, create a new
// pair<int, string> with that key, then assign it a value
myMap[41] = "Hello";          // Create a pair<41, "Hello">
myMap[25] = "Thor Odinson";   // Create a pair<25, "Thor Odinson">
myMap[616] = "Peter Parker"; // Create a pair<616, "Peter Parker">
```

You don't have to pre-allocate space for a map—it can do so dynamically.

Maps are **node-based** data structures, like a linked list—each pair is a node.

```
// If a key already, overwrite the VALUE of the existing pair
myMap[41] = "A new value for the key of 41";
myMap[25] = "The Mighty Thor";
myMap[616] = "Miles Morales";
```

The subscript operator **returns a reference** to the value.

This allows for fast access **and** modification of existing values.

Maps Retrieve Values by Key, Not Index

- Like vectors, the map has `operator[]` and `at()`.
- They don't look up elements in array but return a **value** from a **pair** that has the **provided key** (possibly creating a new pair if one doesn't exist!).

```
vector<int> numbers;
/* Assume some initialization */
```

```
map<int, int> aMap;
/* Assume some initialization */
```

```
cout << numbers[0];  // Print the first element
cout << aMap[0];     // Print the value associated with KEY 0
```

The key of 0 might not exist! Even if it does, it's not necessarily the first key

```
cout << numbers[numbers.size()-1]; // Print the last element
cout << aMap[aMap.size()-1];   // Key might not exist
```

```
cout << numbers[-20]; // A negative INDEX is always bad!
cout << aMap[-20];    // -20 is a valid KEY (but it might not exist in THIS map)
```

```
// .at() works similarly for both classes
cout << numbers.at(-5) // Throws an exception (out of range)
cout << aMap.at(-5); // MIGHT throw an exception, IF the key doesn't exist
```

Using Strings as Keys

- Computers can understand index values without issue—they (only!) understand numbers.
- Humans tend to use words (strings) as labels instead of numbers.
- Consider writing a dictionary program:

```
struct Word
{
    string word;
    string definition;
};

vector<Word> words;

// Read a list of words from a file

// Where is the word "program"?
// What INDEX is that word stored at?
cout << words[???].definition;
```

In a massive collection, finding a specific value by **index** isn't easy.

```
// Key: a word
// Value: its definition
map<string, string> dictionary;

// Read a list of words from a file
// Add using emplace(word, definition)
// or [word] = definition

// Where is the word "program"?
cout << dictionary["program"];
```

With a map, finding a specific value by **key** is simple! (That's the whole point!)

Reminder: This is not an index!

Using Strings as Keys

```
map<string, Level> gameLevels; // Level == user-created class
gameLevels.emplace("tutorial", Level(/*some constructor data*/));
gameLevels.emplace("level1", Level());
gameLevels.emplace("secret_level", Level());
gameLevels["level2"] = Level(); // [] or emplace can both work
gameLevels["level3"] = Level(); // [] or emplace can both work

vector<Level> vecLevels;
/* Add the same data to the vector */
vecLevels.push_back(Level(/*tutorial data*/);
vecLevels.push_back(Level(/*level1 data*/);
// etc...

// Accessing level 2 in a map:
gameLevels["level2"].DoSomething(); // [] returns a reference to a Level object

// Accessing level 2 in a vector:
// vecLevels[1]? Maybe? May not be guaranteed
for (unsigned int i = 0; i < vecLevels.size(); i++)
    if (vecLevels[i].GetName() == "level2")
        vecLevels[i].DoSomething();
```

Maps make searching much easier (the map has this implemented already).

Keys in a map Must Be Unique

- A map cannot store duplicate keys.

```
map<string, string> myMap;
myMap.emplace("Batman", "The Caped Crusader");
myMap.emplace("Batman", "The Dark Knight"); // emplace() won't add duplicate keys
myMap.emplace("Robin", "The Boy Wonder");
myMap.emplace("Superman", "The Man of Steel");

cout << myMap["Batman"] << endl; // "The Caped Crusader" (sorry, Dark Knight!)
```

```
// emplace() won't add a new pair or overwrite if a key already exists
// But [] will!
myMap["Batman"] = "The Dark Knight"; // Set the value for the key "Batman"
cout << myMap["Batman"]; // Prints "The Dark Knight"
```

- What if we wanted “multiple values” for a single key? (i.e., more than one nickname?)

```
map<string, vector<string>> nicknames;
nicknames["Batman"].push_back("The Caped Crusader");
nicknames["Batman"].push_back("The Dark Knight");
```

Keys in a map Must Be Unique

- You might not want to “blindly” use a given key—what if it doesn’t exist?

```
map<string, string> userData; // Stores usernames (key) and passwords (value)
/* assume some initialization, maybe from a file*/

string username, password;
cout << "Username: ";
getline(cin, username);
cout << "Password: ";
getline(cin, password);

// Verify data
if (userData[username] == password)
    // Do some login stuff...
/*=== OR ===*/
if (userData.at(username) == password)
    // Do some login stuff...
```

If the key doesn’t exist… let’s say the user made a typo for their name (“Bbo123” instead of “Bob123”).

This will create a new key/value pair: **Key**: Bbo123, **Value**: “”

This will throw an out_of_range exception, key does not exist.

Keys in a map Must Be Unique

`someMap.find(key)`

`find()` returns an **iterator** to the pair with a matching key.
If no match was found, returns an iterator to `map::end()`.

Iterators
A way of accessing elements in complex containers. We'll cover this more as a standalone topic.

map::end()
Returns a result you should never use directly, similar to nullptr for pointers.

```
// If the result of find() is equal to end(), don't use it!
if (userData.find(username) == userData.end())
{
    cout << "User " << username << " not found!";
    // Throw exception, ask for additional input, etc
}

// If the results are NOT equal to the end()... a match was found!
if (userData.find(username) != userData.end())
{
    if (userData[username] == password)
        // Username found, NOW check the password...
}
```

A little bit of error checking can go a long way!

Iteration Through maps

- A typical for loop most likely won't work when it comes to maps.
- The **keys** of most map objects are will be incompatible, either by data type or values.

```
map<int, string> heroes;
heroes.emplace(22, "Spider-Man");
heroes.emplace(19, "Captain America");
heroes.emplace(30, "Thor");
heroes.emplace(17, "Iron Man");

for (int i = 0; i < heroes.size(); i++)
{
    cout << heroes[i] << endl;
}
```

Who's at heroes[0]? heroes[1]? [2]?

You might get **SOME** results…
but don't count on it.

```
map<string, string> dictionary;
/* assume initialization */
for (int i = 0; i < dictionary.size(); i++)
{
    cout << dictionary[i] << endl;
}
```

An `int` loop counter won't even compile with this map!

For this type of container, we need an **iterator**.

Iterators

A standard way of iterating through STL containers

```
map<string, int> example;

// Create an iterator, using the same template types as the map
map<string, int>::iterator iter;

// Set the iterator to th
iter = example.begin();

// As long as the iterator isn't at the end...
// Run the body of the loop, and move to the next element (++iter)
for (; iter != example.end(); ++iter)
{
    cout << "This element's key:   " << iter->first;
    cout << "This element's value: " << iter->second;
}
```

Having an abstraction like `begin()` makes it easier to use the class—you don't need to know all the internal details.

This process is the same for any kind of map—just change the data type of the iterator!

Much of this process is identical for any container that uses iterators.

A map Sorts Based on Keys

```
map<string, int> data;
data.emplace("Thanos", 50);
data.emplace("Star Lord", 83);
data.emplace("Nova", 50);
data.emplace("Nightcrawler", 72);
data.emplace("Colossus", 110);

map<string, int>::iterator iter = data.begin();
for (; iter != data.end(); iter++)
{
    cout << "Key: " << iter->first << endl;
    cout << "Value: " << iter->second << endl;
}
```

```
Key: Colossus
Value: 110
Key: Nightcrawler
Value: 72
Key: Nova
Value: 50
Key: Star Lord
Value: 83
Key: Thanos
Value: 50
Press any key to continue . . .
```

- The data is sorted by the **key**, in ascending order.
- Want to use map, but don't need sorted data? (Sorting takes processing power!)

```
// Use the unordered_map instead!
// Same functionality, faster performance!
#include <unordered_map>
using std::unordered_map;
unordered_map<string, int> data;
```

When Shouldn't We Use maps?

- You don't have any specific key-value association.
- If your "key" is really just an index…

```
map<int, string> someData;
someData[0] = "Bob";
someData[1] = "Iron Man";
someData[2] = "Super Smash Bros.";
someData[3] = "Everything is in order, why use a map?";

vector<string> strings;
strings.push_back("Bob");
strings.push_back("Iron Man");
strings.push_back("Super Smash Bros.");
strings.push_back("Everything is in order, why use a map?");

// Print out the first entry of each container
cout << someData[0] << endl;
cout << strings[0] << endl;
```

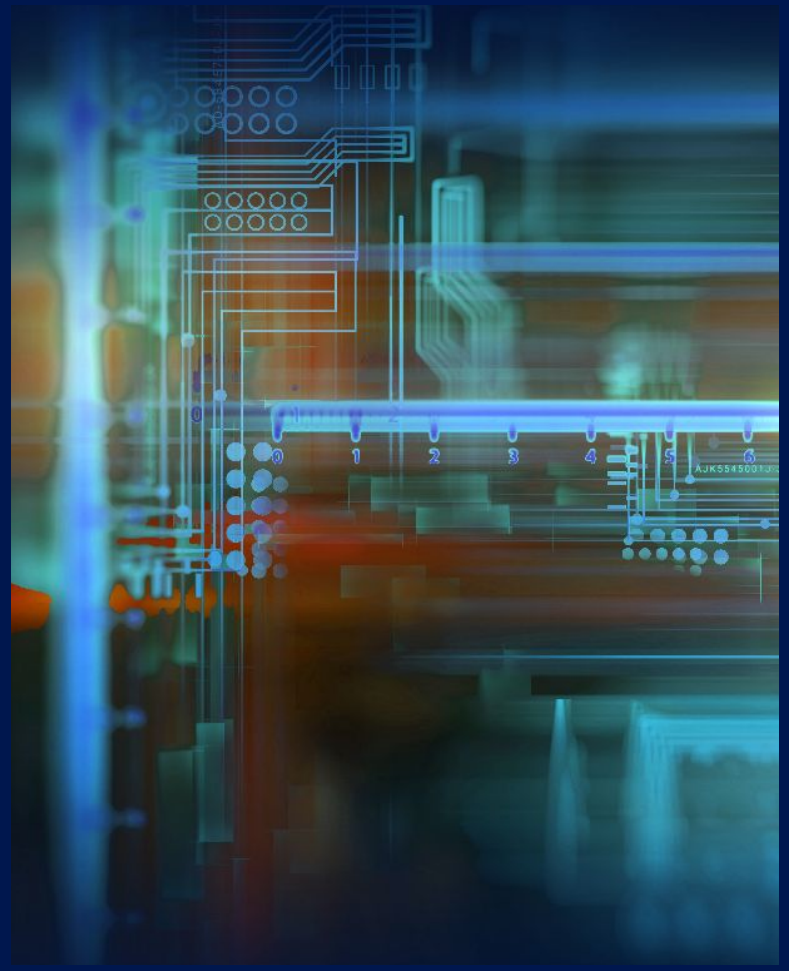
You use a map because keys may have no specific order or pattern (or they aren't numbers!).

If you want data in the same order you added it, use a vector or an array.

Recap

- maps (ordered or unordered) are containers which store **pairs**:
  - A **key**, a label for a value
  - A **value**, the data being labeled by the key
- Storage and retrieval is based on **keys, not index values** like arrays or vectors.
- **Keys are unique**—maps won't store duplicate keys.
- maps are useful when you know **what** you're looking for (a key) but don't know **where** the data is.
- Like many containers, maps **use iterators** to access all of the elements in a loop.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.