EE538: Computing and Software for Systems Engineers

Lecture 3: A Tour of the C++ Language Part 2

University of Southern California
Instructor: Arash Saifhashemi

Namespaces

```cpp
namespace ns1 {
int x = 1;
void Print() { std::cout << "Printing example 1." << std::endl; }
} // namespace ns1
namespace ns2 {
int x = 2;
void Print() { std::cout << "Printing example 2." << std::endl; }
} // namespace ns2

int main() {
 std::cout << "ns1::x: " << ns1::x << std::endl;
 std::cout << "ns2::x: " << ns2::x << std::endl;
 ns1::Print();
 ns2::Print();
}
```

Variable Scope

- Variable scope
    - Global
    - Local
        1. Inside functions
        2. Inside blocks (curly braces)

```
void MyFunc() {
 int a;
 {
   Person p;
   // Do something with p
  }
 // Compile Error:
 // std::cout << "p.name: " << p.name << std::endl;
}
```

```
// Global variable
int global_variable = 6;
int main() {
      {
         {
           Person p;
           int a = 1000;
         }
         int a;
         a = 109;
         {
           Person p;
           int a = 1000;
           std::cout << "a: " << a << std::endl;
         }

         std::cout << "a: " << a << std::endl;
       }
      }
}
```

Const

- Indicates no change
  - Variables
  - Pointers
  - Function arguments and return types
  - Class Data members
  - Class Member functions
  - Objects

```
int main() {
 const int i = 1;
 const int j = i + 1; // Initializing is ok
 i++;                 // Don't change the const!
}
```

Why Would We Use const?

- By making a variable const, we **prevent unintentional changes** that might cause bugs.
- It makes it easier to **reason about our code**.
- In C++ we use the compiler over and over to **catch our mistakes**:
  - Remember: Compile errors are your friends!

```
int main() {
 const int i = 1;
 const int j = i + 1; // Initializing is ok
 i++;                 // Don't change the const!
}
```

Const in Function Parameters

Use: const references for input parameters to:

1. Avoid copying
2. Avoid modification

You **CAN** pass a non-const to const

You **CANNOT** pass a const to non-const

```
int CalculateTax(int income) {
 income = income - 20; // It changes the copy
 return income * 0.3;
}
int CalculateTaxRef(int &income) {
 income = income - 20; // It changes original!
 return income * 0.3;
}
int CalculateTax(const int &income) {
 income = income - 20; // Don't touch my income!
 return income * 0.3;
}
```

Const in Function Parameters

Data type	Feature
Pass by value	• Copying, so original is protected • But, copying can have high cost
Pass by reference	• No copying ○ Original might be **changed** (can be good or bad) • No copy overhead
Pass by **const reference**	• No copying • Original cannot changed

Const in Classes

1. Member variables:
   a. They should **be initialized** by constructor
   b. Optional reading
2. Member Functions:
   a. They cannot change the member variables
   b. Optional reading
3. Const objects:
   a. Their member variables cannot change
   b. Should be initialized by constructor

```
// Const object
const Person q(/*_ssn=*/354545454);
// q._age = 21; // Error!


// Initializing ok
const Person r(354545454, 21);
```

const objects

Some Data Structures from STL

std::set

- Store a set of elements
- Important methods to know
  - size()
  - insert()
  - count()
  - find()
- Things to know about std::set:
  - Internally it is **sorted** based on keys
  - Access, Insert, and find complexity is **O(log(n))**
  - Reinserting the same key will just update the data, there is **no duplicate keys**

std::pair

- std::pair<T1, T2>
  - Couples together a pair of things (of type T1 to T2)
  - For a pair of items p:
    - Access the first item by p.first
    - Access the second item by p.second.

```
std::pair<std::string, int> p1("Ari", 3);
std::pair<std::string, int> p2("Ted", 4);


std::cout << "p1.first: " << p1.first << std::endl;
std::cout << "p1.second: " << p1.second << std::endl;
```

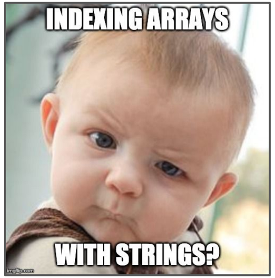
First	Second
Ari	3

First	Second
Ted	4

std::map

- Store elements in a mapped fashion
  - AKA **Associative Array** or **Dictionary**
- A Collection of **pairs** (key, value)
  - Essentially implements a table of items each having a key and value.
  - Keys are unique
  - Values can be duplicated
- Important methods to know
  - size()
  - insert()
  - count()
- Things to know about std::map:
  - Internally it is **sorted** based on keys
  - **Access**, **Insert**, and **find** complexity is **O(log(n))**
  - Map is really a collection of pairs
  - Accessing a non-existent key using [ ], creates that key
  - No duplicate keys

Key (Name)	Value (Grade)
Ari	3
Ted	3
Jessica	3

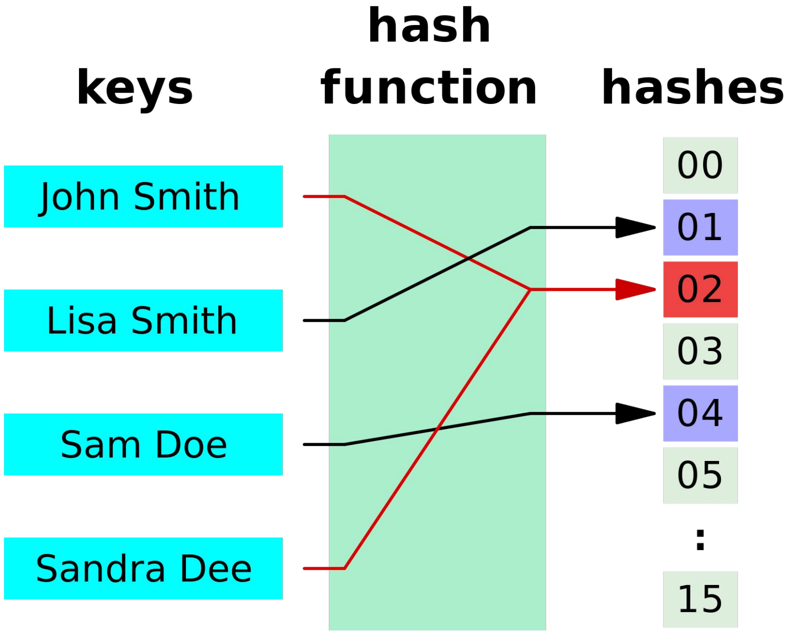



```
std::map<std::string, int> persons;

persons["Ari"] = 3;
persons["Ted"] = 4;
persons["Jessica"] = 3;
```

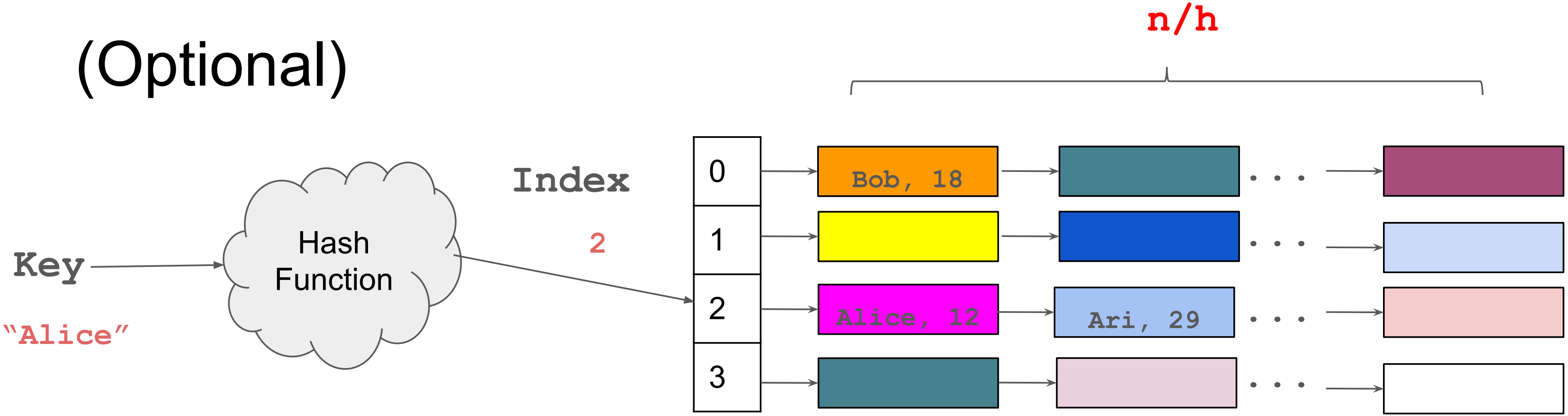
Hash Function (Optional)

- Suppose we want to:
  - hold information of **Persons** in a vector
  - index it with their **names** rather than numbers



Hash function: f(keys) -> indices

(Optional)



n: Number of items

h: Table size

With a good Hash Function, push time is: O (1 + n/h)

If n/h is small, then search time is O(1)

💡 **Idea:** As n grows, we **increase** h

std::**map** vs std::**unordered_map**

- Similar functionality to std::map
- Used for mapping **unique** **Keys** to **Values.**
  - Example: Mapping **SSN** to Person
  - Example: Count the number of each word in a book: Map of words to numbers.
- Both provide similar APIs

std::map	**std::unordered_map**
Internally sorted	Not sorted
Implemented using balanced trees (red-black trees)	Implemented using a hash table
Search, **removal**, and **insertion** operations have logarithmic complexity: **O(log n)**	**Search**, **insertion**, and **removal** of elements have **average** time of **O(1)**, but worst case can be **O(n)**

std::**set** vs std::**unordered_set**

- Used for keeping a list of **unique** items
  - A set is really the list of keys in a map
- Both provide similar APIs

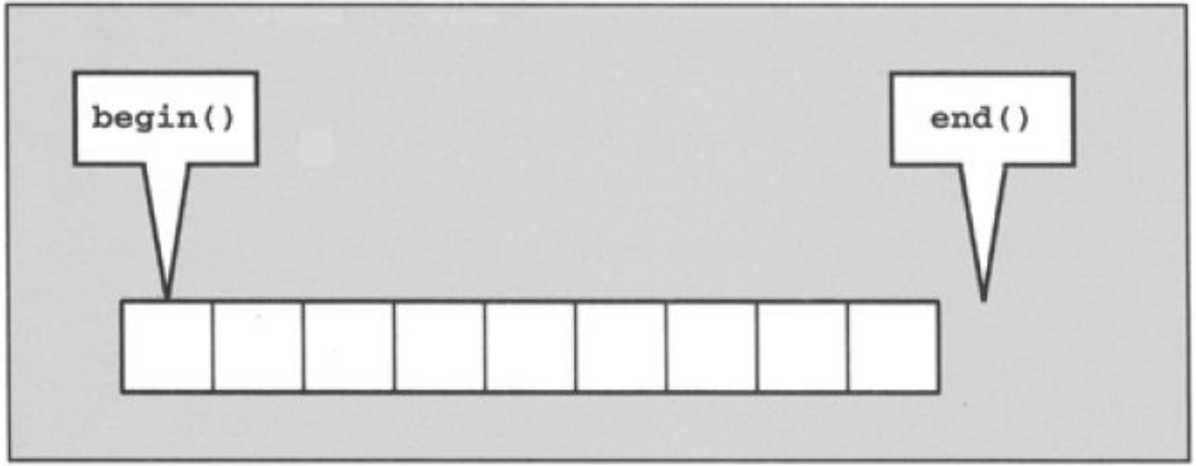
std::set	**std::unordered_set**
Internally sorted	Not sorted
Implemented using balanced trees (red-black trees)	Implemented using a hash table
Search, **removal**, and **insertion** operations have logarithmic complexity: **O(log n)**	**Search**, **insertion**, and **removal** of elements have average time of **O(1)**, but worst case can be **O(n)**

Iterators

STL

- A set of C++ template classes and functions
- Four components
  - Algorithms
  - Containers: vector, set, map
  - Functions
  - **Iterators**
- So far we have seen:
  - Vectors, sets, strings, and maps

Iterators



Iterators

- Used for iteration of STL objects
  - An internal variable (similar to a pointer) of the class that moves one step in the collection as we iterate

```
std::vector<int>::iterator it;
```

Definition

```
int n = *it;
```

Dereferencing

```
std::vector<int> v = {1, 2, 3, 4, 5};
 // An easy way of iteration
 for (int n : v) {
   std::cout << "n: " << n << std::endl;
 }
 // General way of iteration
 std::vector<int>::iterator it;
 for (it = v.begin(); it != v.end(); ++it) {
   int n = *it;
   std::cout << "n: " << n << std::endl;
 }
```

What do we need to know about iterators?

- Think of an iterator as a pointer
  - Initially it points to nothing
  - **begin() :** address of the **first** item
  - **end():** address **AFTER the last item** or **NULL**
    - As long as the iterator is less than or not equal end(), you are safe
  - You can perform ++ and -- on them
    - Each time check against **begin**() and **end**()

```
// General way of iteration
std::vector<int>::iterator it;
for (it = v.begin(); it != v.end(); ++it) {
  int n = *it;
  std::cout << "n: " << n << std::endl;
}
```

Definition, initialization, check for end(), increment, and dereferencing

Why Iterators?

- Iterating the container
  - Duh!
- Reuse code
- Container manipulation

```
std::vector<int> v = {1, 2, 3, 4, 5};
std::vector<int>::iterator v_it;
for (it = v.begin(); it != v.end(); ++it) {
  int n = *it;
  std::cout << "n: " << n << std::endl;
}


std::set<int> s = {1, 2, 3, 4, 5};
std::set<int>::iterator s_it;
for (s_it = s.begin(); s_it != s.end(); ++s_it) {
  int n = *s_it;
  std::cout << "n: " << n << std::endl;
}
```

A More Modern Way of Using Iterators

- Using auto
  - The compiler deduces the type for us
  - Use a const reference if you don’t want to modify the items and prevent copy

```
// using auto
 std::set<int> s = {1, 2, 3, 4, 5};
 for (auto it = s.begin(); it != s.end(); ++it) {
   const int &n = *it;
   std::cout << "n: " << n << std::endl;
 }
```

Insert and Delete

- Iterators can tell us where to insert or delete
  - Irrespective of the type of the container


```cpp
// Insert a number before 3
std::vector<int> v = {1, 2, 3, 4, 5};
for (auto it = v.begin(); it != v.end(); ++it) {
  const int &n = *it;
  if (n == 3) {
    it = v.insert(it, 12);
    // break;
  }
}
```

- Side note:
  - When using iterators, if you **modify the collection size**, your iterator **may become invalid** afterwards and you should not use it anymore. That's why we **break** right after inserting an item in the vector. After this point, the iterator may not be valid anymore.
  - You can get a new valid one using **v.begin()**.

STL

- A set of C++ template classes and functions
- Four components
  - **Algorithms**
  - Containers: vector, set, map
  - Functions
  - Iterators
- So far we have seen:
  - Vectors, sets, strings, and maps

STL Algorithms

- Iterators are used to generalize them
  - Sort
  - Find
  - Reverse
  - Max_element
  - Min_element
  - Accumulate
  - Count
  - Count_if
  - Transform

```
std::vector<int> v = {12, -2, 0, 13, 3, 5};
   auto it = std::find(v.begin(), v.end(), 4);
   if (it != v.end()) {
     const auto &n = *it;
     std::cout << "Found n: " << n << std::endl;
   } else {
     std::cout << "Didn't find." << std::endl;
   }
```

See src/stl_algorithm/main.cc in cpp_tour repo for examples

Passing Functions As Parameters

Passing Function as A Parameter

- Old C way: function pointers
  - `void func ( void (*f)(int) );`
- Using std::function

```
int BinaryOperation(int a, int b, std::function<int(int, int)> func) {
 return func(a, b);
}
int Add(int a, int b) { return a + b; }
int Mult(int a, int b) { return a * b; }

int main() {
  int result1 = BinaryOperation(10, 20, Add);
  int result2 = BinaryOperation(10, 20, Mult);
}
```

STL Algorithms

- We can pass functions to them

```
bool IsOdd(int i) { return ((i % 2) == 1); }
bool IsEven(int i) { return ((i % 2) == 0); }

int main() {
   std::vector<int> v = {12, -2, 0, 0, 1, 12, 5, 3, 13, 3, 5};
   auto count_odd = std::count_if(v.begin(), v.end(), IsOdd);
   auto count_even = std::count_if(v.begin(), v.end(), IsEven);
   return 0;
}
```

Transform (AKA Map)

- Map each **x** in the container to **f(x)**

```
int IncrementByTen(int a) { return a + 10; }

int main() {
 std::vector<int> inputs = {0, 1, 2, 3, 4, 5, 6, 7, 8};
 std::vector<int> outputs(inputs.size());
 // Increment all of them
 std::transform(inputs.begin(), inputs.end(),
                outputs.begin(), IncrementByTen);
}
```

Copy_if (AKA Filter)

- Keep item **x** of the container if **f(x)==true**

```
bool IsOdd(int i) { return ((i % 2) == 1); }

int main() {
 std::vector<int> inputs = {0, 1, 2, 3, 4, 5, 6, 7, 8};
 std::vector<int> outputs(inputs.size());
 // Increment all of them
 std::copy_if(inputs.begin(), inputs.end(),
                   outputs.begin(), IsOdd);
}
```

Accumulate (AKA Reduce)

- Homework: implement using std::accumulate

Struct and Class

C++ Struct

- A collection of different data types.

```
struct Point {
 int x;
 int y;
 std::string name;
};

void PrintPoint(Point &p) {
 std::cout << "p.x: " << p.x <<
                   ", p.y: " << p.y <<
                   ", p.name: " << p.name
              << std::endl;
}
```

```
int main() {
 Point p1;
 Point p2;
 p1.x = 20;
 p1.y = 30;
 p1.name = "My Point 1";
 PrintPoint(p1);
 p2 = p1;
 p2.x++;
 p2.name = "My Point 2";
 PrintPoint(p2);
}
```

Main Parts of a Class

- **Member** variables
  - What data must be stored?
- Constructor(s)
  - How do you build an instance?
- **Member functions** AKA **Methods**
  - How does the user need to interact with the stored data?
- Destructor
  - How do you clean up an after an instance?

```cpp
class Person {
public:
  Person() { name_ = "UNKNOWN"; }
  ~Person() { std::cout << "Destructor!" << std::endl; }

  std::string GetSSN() {
    return social_security_number_.empty()
        ? "NONE"
        : "***-**-****";
  }

  void SetSSN(const std::string& ssn) {
    social_security_number_ = encrypt(ssn);
  }

  std::string name_;

private:
  int age_;
  std::string social_security_number_;
};
```

Main Parts of a Class

- Public or private
    - Defaults is private (only class methods can access)
    - Must explicitly declare something public
- Most common C++ operators will not work by default
    - (e.g. ==, +, <<, >>, etc.)
- May be used just like other data types
    - Get pointers/references to them
    - Pass them to functions (by copy, reference or pointer)
    - Dynamically allocate them
    - Return them from functions

```cpp
class Person {
public:
 Person() { name_ = "UNKNOWN"; }
 ~Person() { std::cout << "Destructor!" << std::endl; }

 std::string GetSSN() {
   return social_security_number_.empty()
      ? "NONE"
      : "***-**-****";
 }

 void SetSSN(const std::string& ssn) {
   social_security_number_ = encrypt(ssn);
 }

 std::string name_;

private:
 int age_;
 std::string social_security_number_;
};
```

C++ Class vs Struct

- For the most part they are the same.
  - Members of struct are by default public (Unlike class)
  - By **convention**, we use a struct when there is no method.

```
struct Point {
 int x;
 int y;
 std::string name;
};

void PrintPoint(Point &p) {
 std::cout << "p.x: " << p.x <<
              ", p.y: " << p.y <<
              ", p.name: " << p.name
           << std::endl;
}
```

```
class Point {
 int x;
 int y;
 std::string name;
};

void PrintPoint(Point &p) {
 std::cout << "p.x: " << p.x <<
              ", p.y: " << p.y <<
              ", p.name: " << p.name
```

Class, Instance, Object

- A **Class** is a type. It specifies a group of similar objects.
- An **Object** is an **Instance** of that class, i.e it's a variable of that type.

```
class Point {
public:

 Point(int i, int j) {
   i_ = i;
   j_ = j;
 }

 int GetI() const { return i_; }
 int GetJ() const { return j_; }
 void SetI(int i) { i_ = i; }
 void SetJ(int j) { j_ = j; }

private:
 int i_;
 int j_;
};
```

```
int main() {
 Point p1(1, 2);
 Point p2(1, 2);
 Point p3 = p1;
}
```