Lecture 13 part 1

Fall 2020

CME 211: Lecture 13(a)

Topics:

- C++ containers
- `vector`
- `tuple`
- `map`
- `set`
- and more

C++ containers

- Static arrays are created on stack and can hold limited amount of data; their size must be known at compile time.
- You could use dynamic arrays to build your own data structures like lists, dictionaries, etc. from scratch!
- But, the C++ standard library includes many containers that are similar to what you have already seen in Python.
- Some of these include: `vector`, `map`, `set`, `tuple`, etc.

Vector

- A vector in C++ standard librarry is analogous to a list in Python.
- Vectors are objects, so they have methods associated with them.
- Just like the Python list, a vector can change in size to accommodate the addition or removal of items. They support constant-time append and extraction methods.
- Unlike Python lists, the vector is restricted to containing homogeneous data, i.e. all vector elements must be of the same type.

Our first vector

`src/vector1.cpp`:

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

int main()
{
  std::vector<int> v;
```

```
  // The call to .size() is a call to a library method.
  std::cout << "v.size() = " << v.size() << std::endl;
  // The same is true for .empty()
  if (v.empty())
    std::cout << "v is empty" << std::endl;
  else
    std::cout << "v is not empty" << std::endl;

  // We can append an element in O(1) time.
  v.push_back(42);

  std::cout << "v.size() = " << v.size() << std::endl;

  if (v.empty())
    std::cout << "v is empty" << std::endl;
  else
    std::cout << "v is not empty" << std::endl;

  return 0;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion vector1.cpp -o vector1
$ ./vector1
v.size() = 0
v is empty
v.size() = 1
v is not empty
$
```

Printing a vector

C++ does not have a built-in facility to print out a `vector`; attempting to naively print out a vector results in an *error*:

src/vector2.cpp:

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

int main()
{
  std::vector<int> v;
  v.push_back(42);

  std::cout << "v = " << v << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion    vector2.cpp   -o vector2
vector2.cpp: In function ‘int main()’:
vector2.cpp:9:13: error: cannot bind ‘std::basic_ostream<char>’ lvalue to ‘std::basic_ostream<char>&&’
   std::cout << "v = " << v << std::endl;
             ^
In file included from /usr/include/c++/4.9.2/iostream:39:0,
```

```
                 from vector2.cpp:1:
/usr/include/c++/4.9.2/ostream:602:5: note: initializing argument 1 of ‘std::basic_ostream<_CharT, _Traits
     operator<<(basic_ostream<_CharT, _Traits>&& __os, const _Tp& __x)
     ^
<builtin>: recipe for target 'vector2' failed
```

Printing a vector

We must write our own loop to print a vector. We use square brackets `[]` to access an item of a `vector`.

src/vector3.cpp:

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

int main()
{
  std::vector<int> v; // default constructor, creates an empty vector
  v.push_back(42);
  v.push_back(-7);
  v.push_back(19);

  for(unsigned int n = 0; n < v.size(); n++)
    std::cout << "v[" << n << "] = " << v[n] << std::endl;

  return 0;
}
```

When we access an element using the extraction operator `[]`, we can retrieve the element in constant time.

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion    vector3.cpp   -o vector3
$ ./vector3
v[0] = 42
v[1] = -7
v[2] = 19
```

Subscript `operator[]`

On C++ containers, like `vector`, the square brakets `[]` are called `operator[]`. This is a special method for C++ objects and may be overloaded. For now, we just need to use them for `vectors`.

Valid `vector` indices for a vector named `v` are in the range `[0,v.size())`. Attempting to access element outside of those bounds leads to *undefined* behavior.

src/vector4a.cpp:

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

int main()
{
  std::vector<int> v(3); // Constructor creating vector with 3 elements
  v[0] = 42;
  v[1] = -7;
  v[2] = 19;

  // -1 is clearly OOB, and so is 3 if we recall that we are 0-indexed.
```

```
  std::cout << "v[-1] = " << v[-1] << std::endl;
  std::cout << "v[3] = " << v[3] << std::endl;

  return 0;
}
```

Output (non-deterministic, unreliable, and in general relies on undefined behavior):

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion    vector4a.cpp   -o vector4a
$ ./vector4a
v[-1] = 0
v[3] = 0
```

Hmm, nothing bad happened yet! It is hard to track down these bugs.

Address Sanitizer

Let's explore this a little bit further. In the file `src/vector4b.cpp` we are only going to attempt accessing `v[-1]` and use the `-fsanitize=address` compiler flag.

Part of `src/vector4b.cpp`

```
  std::cout << "v[-1] = " << v[-1] << std::endl;
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    vector4b.cpp   -o vector4b
$ ./vector4b
=================================================================
==7470==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000efac at pc 0x40131c bp 0x7ffc87
READ of size 4 at 0x60200000efac thread T0       <-- First key line!
    #0 0x40131b in main /home/nwh/Dropbox/courses/2015-Q4-cme211/lecture-prep/lecture-19-work/src/vector4b
    #1 0x7f77d9383fdf in __libc_start_main (/lib64/libc.so.6+0x1ffdf)
    #2 0x401118 (/home/nwh/Dropbox/courses/2015-Q4-cme211/lecture-prep/lecture-19-work/src/vector4b+0x4011
...
```

Notice that address sanitizer is saying we are making an invalid read of size 4 bytes (the size of an integer, what we are storing in the vector) starting on line 11 of `vector4b.cpp`. Now, in the file `src/vector4c.cpp` we are going to attempt accessing `v[3]` with `-fsanitize=address` and see what happens.

Part of `src/vector4c.cpp`

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

int main()
{
  std::vector<int> v(3);
  v[0] = 42;
  v[1] = -7;
  v[2] = 19;

  std::cout << "v[3] = " << v[3] << std::endl;

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    vector4c.cpp   -o vector4c
$ ./vector4c
v[3] = -1094795586
```

The program compiled and ran with no problem. Of course we got junk output for `v[3]` because that part of memory had not been initialized.

What happened here:

- When a `vector` is declared in C++, some amount of memory is allocated on the heap for the storage of the element. Often, more storage is allocated than initially needed by the vector to allow for efficient addition of new items at the end of the vector.
- Thus, trying to access `v[3]` in this case does not access memory out of bounds from the context of the lower level memory allocation, but is still undefined behavior. There is not guarantee that there will be extra space.
- Subscript `operator[]` for `vector` takes in an unsigned integer as its argument. There for in `v[-1]` the `-1` is converted to a very large positive integer, which turns out to be out of range of the allocated memory for the vector. This leads to the address sanitizer churning out error messages.

`at()`

The `at()` method for a vector performs bounds checking. As a result `at()` is slower than `operator[]`, but also safer to use.

`src/vector5.cpp`:

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

int main()
{
  std::vector<int> v(3);
  v[0] = 42;
  v[1] = -7;
  v[2] = 19;

  std::cout << "v.at(1) = " << v.at(1) << std::endl;
  std::cout << "v.at(3) = " << v.at(3) << std::endl;

  return 0;
}
```

Output:

```
$ clang++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    vector5.cpp   -o vector5
$ ./vector5
v.at(1) = -7
libc++abi.dylib: terminating with uncaught exception of type std::out_of_range: vector
```

Note that in some places you will see `clang++` as the compiler. For the context of this class consider this to be equivalent to `g++`.

Modifying an element

Basically: this works as you would expect.

`src/vector6.cpp`:

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

int main()
{
  std::vector<int> v(3);
```

```
  v[0] = 42;
  v[1] = -7;
  v[2] = 19;

  v[1] = 73;

  for(unsigned int n = 0; n < v.size(); n++)
    std::cout << "v[" << n << "] = " << v[n] << std::endl;

  return 0;
}
```

Output:

```
$ clang++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    vector6.cpp   -o vector6
$ ./vector6
v[0] = 42
v[1] = 73
v[2] = 19
```

Insert

Costs linear time, and requires use of *iterators*!

`src/vector7.cpp`:

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

int main()
{
  std::vector<int> v(3);
  v[0] = 42;
  v[1] = -7;
  v[2] = 19;

  v.insert(1, 73);

  for(unsigned int n = 0; n < v.size(); n++)
    std::cout << "v[" << n << "] = " << v[n] << std::endl;

  return 0;
}
```

Output:

```
clang++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    vector7.cpp   -o vector7
vector7.cpp:11:5: error: no matching member function for call to 'insert'
  v.insert(1, 73);
  ~~^~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/v
      candidate function not viable: no known conversion from 'int' to
      'const_iterator' (aka '__wrap_iter<const_pointer>') for 1st argument
    iterator insert(const_iterator __position, value_type&& __x);
             ^
```

C++ `vector` does *not allow* insertion at an integer index!

Iterators

We have to us an **iterator** for this.

src/vector8.cpp:

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

int main()
{
  std::vector<int> v(3);
  v[0] = 42;
  v[1] = -7;
  v[2] = 19;

  //  Declare an iterator
  std::vector<int>::iterator iter;

  // Set iterator to start of vector

  iter = v.begin();

  // Advance iterator by two positions
  iter += 2;

  // Use iterator to insert a new value into the vector
  v.insert(iter, 73);

  for(unsigned int n = 0; n < v.size(); n++)
    std::cout << "v[" << n << "] = " << v[n] << std::endl;

  return 0;
}
```

Output:

```
$ clang++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    vector8.cpp   -o vector8
$ ./vector8
v[0] = 42
v[1] = -7
v[2] = 73
v[3] = 19
```

Erase

The `erase()` method also uses an iterator.

src/vector9.cpp:

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

int main()
{
  std::vector<int> v(3);
  v[0] = 42;
  v[1] = -7;
  v[2] = 19;
```

```
  v[3] = 73;
  v[4] =  0;

  // remove fourth element
  v.erase(v.begin()+3);

  for(unsigned int n = 0; n < v.size(); n++)
    std::cout << "v[" << n << "] = " << v[n] << std::endl;

  return 0;
}
```

Output:

```
$ clang++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    vector9.cpp   -o vector9
$ ./vector9
v[0] = 42
v[1] = -7
v[2] = 19
v[3] = 0
```

Sort

src/sort.cpp:

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

int main()
{
  // Using initializer list to initialize vector
  std::vector<int> v {42, -7, 19, 73, 0};

  std::sort(v.begin(), v.end());

  for(unsigned int n = 0; n < v.size(); n++)
    std::cout << "v[" << n << "] = " << v[n] << std::endl;

  return 0;
}
```

Output:

```
$ clang++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    sort.cpp   -o sort
$ ./sort
v[0] = -7
v[1] = 0
v[2] = 19
v[3] = 42
v[4] = 73
```

Accumulate

src/accumulate.cpp:

```
#include <iostream>
#include <numeric>
```

```
#include <vector>

int main()
{
  std::vector<int> v {42, -7, 19, 73, 0};

  int sum = std::accumulate(v.begin(), v.end(), 0);
  std::cout << "sum = " << sum << std::endl;

  return 0;
}
```

Output:

```
$ ./accumulate
sum = 127
```

Copy or reference?

What happens when we use the assignment operator on a vector? We get a *copy*!

src/vector10.cpp:

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

int main()
{
  std::vector<int> v {42, -7, 19};

  std::vector<int> v2 = v1;
  v2[1] = 73;

  for (unsigned int n = 0; n < v1.size(); n++) {
    std::cout << "v1[" << n << "] = " << v1[n] << std::endl;
  }
  for (unsigned int n = 0; n < v2.size(); n++) {
    std::cout << "v2[" << n << "] = " << v2[n] << std::endl;
  }
  return 0;
}
```

Output:

```
$ clang++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    vector10.cpp   -o vector10
$ ./vector10
v1[0] = 42
v1[1] = -7
v1[2] = 19
v2[0] = 42
v2[1] = 73
v2[2] = 19
```

Assignment operator `=` creates a **deep** copy of the vector.

Function that returns a vector

src/vector11.cpp:

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

std::vector<int> ReadNumbers(std::string filename) {
  std::vector<int> v;
  std::ifstream f(filename.c_str());
  if (f.is_open()) {
    int val;
    while (f >> val) v.push_back(val);
    f.close();
  }
  return v;
}

int main() {
  std::vector<int> v = ReadNumbers("numbers.txt");

  for(unsigned int n = 0; n < v.size(); n++)
    std::cout << "v[" << n << "] = " << v[n] << std::endl;

  return 0;
}
```

Output:

```
$ cat numbers.txt
42
17
-5
73
$ ./vector11
v[0] = 42
v[1] = 17
v[2] = -5
v[3] = 73
```

Notice that here we are creating the vector inside the function stack-frame and returning it; this wouldn't be safe to do with a static array!

Copy or reference?

When we pass vectors to functions, they get copied.

src/vector12.cpp:

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

void increment(std::vector<int> v) {
  for (unsigned int n = 0; n < v.size(); n++) {
    v[n]++;
    std::cout << "v[" << n << "] = " << v[n] << std::endl;
  }
}

int main() {
  std::vector<int> v;
```

```
  v.push_back(42);
  v.push_back(-7);
  v.push_back(19);

  increment(v);

  for (unsigned int n = 0; n < v.size(); n++) {
    std::cout << "v[" << n << "] = " << v[n] << std::endl;
  }
  return 0;
}
```

Passing vector by value creates a deep copy inside the function scope. Once the function returns, the copy of the vector is destroyed.

Output:

```
$ ./vector12
v[0] = 43
v[1] = -6
v[2] = 20
v[0] = 42
v[1] = -7
v[2] = 19
```

Pass by reference

If we want to pass by reference, we can do so using by declaring the input argument to be a reference; this is done using the `&` character, but should not be confused with obtaining the memory address of an object!

`src/passing.cpp`:

```
#include <iostream>

void increment(int& a)
{
  a++;
  std::cout << "a = " << a << std::endl;
}

int main()
{
  int a = 2;

  increment(a);
  std::cout << "a = " << a << std::endl;

  return 0;
}
```

Notice that we declared `increment(int& a)`, i.e. to accept an integer as reference; when we call the function, we simply pass an integer as argument! We'll learn more about nuanced differences between references and pointers in 212.

Output:

```
$ ./passing
a = 3
a = 3
$
```

Pass by reference and const

For functions that don't intend to modify the data, it's more efficient to pass by reference but we can also be clear to the compiler (and reader of the program) that we aren't mutating the underlying data.

src/vector13.cpp:

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

void increment(std::vector<int>& v)
{
  for (unsigned int n = 0; n < v.size(); n++) {
    v[n]++;
    std::cout << "v[" << n << "] = " << v[n] << std::endl;
  }
}

void print(const std::vector<int>& v)
{
  // Trying to mutate 'v' inside this function would result in a compiler error.
  for (unsigned int n = 0; n < v.size(); n++) {
    std::cout << "v[" << n << "] = " << v[n] << std::endl;
  }
}

int main()
{
  std::vector<int> v;
  v.push_back(42);
  v.push_back(-7);
  v.push_back(19);

  increment(v);
  print(v);

  return 0;
}
```

Output:

```
$ ./vector13
v[0] = 43
v[1] = -6
v[2] = 20
v[0] = 43
v[1] = -6
v[2] = 20
```

Only a reference to `std::vector` is passed to functions `increment()` and `print()`, so it's not as expensive as copying the entirety of the container. If the vector is passed to a function by constant reference the function cannot modify the vector, as in the case for `print()`.

Tuple

- A tuple is another sequence object available in C++.
- Tuples have fixed size established at the time of creation. The data-types are also fixed at the time of creation, since C++ is statically typed.

- Elements in the tuple can be modified.
- Elements need not be homogeneous, but the data types cannot be changed after you create the tuple.

Our first tuple

src/tuple1.cpp:

```
#include <iostream>
#include <string>
#include <tuple>

int main()
{
  std::string h = "Hello";
  int a = 42;

  std::tuple<std::string, int> t(h, a); // tuple constructor

  std::cout << "t[0] = " << std::get<0>(t) << std::endl;
  std::cout << "t[1] = " << std::get<1>(t) << std::endl;

  std::get<1>(t) = 19;

  std::cout << "t[1] = " << std::get<1>(t) << std::endl;

  return 0;
}
```

It may seem unfortunate that we can't use the subscript operator with a tuple, but this is because of advanced technical reasons that relate to determining the return type and how this must be done at compile time... `get<0>(t)` and `get<1>(t)` turn out to be different *functions* defined at compile time. This is in contrast to `operator[]` which can return a value of a fixed type at run-time.

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion tuple1.cpp -o tuple1
$ ./tuple1
t[0] = Hello
t[1] = 42
t[1] = 19
$
```

Vector of tuples

src/tuple2.cpp:

```
#include <iostream>
#include <fstream>
#include <tuple>
#include <vector>

int main() {
  std::ifstream f;
  std::vector<std::tuple<std::string,float,float,int>> names;

  f.open("dist.female.first");
  if (f.is_open()) {
```

```
    std::string name;
    double perc1, perc2;
    int rank;
    while (f >> name >> perc1 >> perc2 >> rank) {
      names.emplace_back(name, perc1, perc2, rank); // emplace method takes
    }                                               // constructor's arguments
    f.close();
  }
  else {
    std::cerr << "ERROR: Failed to open file" << std::endl;
  }

  for(unsigned int n = 0; n < names.size(); n++) {
    std::cout << std::get<0>(names[n]) << " " << std::get<1>(names[n]) << std::endl;
  }

  return 0;
}
```

We'll talk more about `emplace_back` vs. `push_back` in 212, but for now just understand that `emplace_back` constructs the object we're inserting into the container on the fly.

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    tuple2.cpp   -o tuple2
$ ./tuple2
MARY 2.629
PATRICIA 1.073
LINDA 1.035
BARBARA 0.98
ELIZABETH 0.937
JENNIFER 0.932
MARIA 0.828
SUSAN 0.794
MARGARET 0.768
DOROTHY 0.727
```

Newer style iteration

src/tuple3.cpp:

```
#include <iostream>
#include <fstream>
#include <tuple>
#include <vector>

int main() {
  std::ifstream f;
  std::vector<std::tuple<int,int,int,int>> data;

  f.open("u.data");
  if (f.is_open()) {
    int uid, mid, rating, time;
    while (f >> uid >> mid >> rating >> time) {
      data.emplace_back(uid, mid, rating, time);
    }
    f.close();
  }
```

```
  else {
    std::cerr << "ERROR: Failed to open file" << std::endl;
  }

  for (auto d : data) {
    std::cout << std::get<0>(d) << " " << std::get<1>(d);
    std::cout << " " << std::get<2>(d) << std::endl;
  }

  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion -g -fsanitize=address    tuple3.cpp   -o tuple3
$ ./tuple3
196 242 3
186 302 3
22 377 1
244 51 2
166 346 1
298 474 4
115 265 2
253 465 5
305 451 3
6 86 3
```

Reading

- **C++ Primer, Fifth Edition** by Lippman et al.
- Chapter 9: Sequential Containers: Sections 9.1 - 9.4