More C++

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

pollev.com/tqm

- How are you?

Administrivia

- Third Assignment (HW02)
  - Released! Should have everything you need after this lecture
  - Due This Friday
  - Porting simple_string into C++ and adding some more complex functionality

- Simplekv (HW03)
  - To be released soon
  - Due next Friday (2/14)
  - Recommend taking a look sooner rather than later
    - Once you figure out what data members you need, consider talking to a TA or I about it
  - Is more work than previous assignments, not a lot though.

Aside: Const objects

- Just like with primitive types and structs, we can have a const object
- A const object cannot change its values

```
int main() {
  const Point p(3, 2);
}
```

- A const object cannot call a non-const member functions
  - Consider how we previously declared the point object:
  - If we left it as this, then this code would not compile

```
int main() {
  const Point p(3, 2);
  cout << p.get_x() << endl;
}
```

```
class Point {
 public:
  Point(int x, int y);
  int get_x() { return x_; }
  int get_y() { return y_; }
 …
```

Aside: Const objects

- We need to mark member functions that do not modify any data members as const.

```
class Point {
 public:
  Point(int x, int y);
  int get_x() const { return x_; }
  int get_y() const { return y_; }
 …
```

- This tells the compiler that it is ok for const objects to call these member functions.

- This code becomes OK now:

```
int main() {
  const Point p(3, 2);
  cout << p.get_x() << endl;
}
```

pollev.com/tqm

- What else could be marked const in Point?

```
class Point {
 public:
  Point(int x, int y);     // constructor
  int get_x() const { return x_; }
  int get_y() const { return y_; }
  double dot_prod(Point p);
  void set_location(int x, int y);

 private:
  int x_;  // data member
  int y_;  // data member
};  // class Point
```

```
Point::Point(int x, int y) {
  x_ = x;
  y_ = y;
}

double Point::dot_prod(Point p) {
  double prod = x_ * p.x_;
  prod += (y_ * p.y_);
  return prod;
}

void Point::set_location(int x, int y) {
  x_ = x;
  y_ = y;
}
```

Const for non-inline functions

- For functions that aren't only in the header (like the get_x and get_y functions). If we wanted to make them const, we need to do it inn the cpp and hpp files:

```
class Point {
 public:
  Point(int x, int y);     // constructor
  int get_x() const { return x_; }
  int get_y() const { return y_; }
  double dot_prod(Point p) const;
  void set_location(int x, int y);

 private:
  int x_;  // data member
  int y_;  // data member
};  // class Point
```

```
Point::Point(int x, int y) {
  x_ = x;
  y_ = y;
}

double Point::dot_prod(Point p) const {
  double prod = x_ * p.x_;
  prod += (y_ * p.y_);
  return prod;
}

void Point::set_location(int x, int y) {
  x_ = x;
  y_ = y;
}
```

Lecture Outline

- **Copying**
  - **Overhead**
- Destructors & Lifetimes
- More STL
  - std::variant
  - std::unordered_map
- Clang-tidy

Poll Everywhere

pollev.com/tqm

- How many vectors are made?
- If you have time:
  - How many times is a vector destructor run?
  - What does this print?

```
vector<int> prefix_sum(vector<int> input) {
  vector<int> res{};
  int sum = 0;
  for (int i : input) {
    sum += i;
    res.push_back(i);
  }
  return res;
}

int main() {
  vector<int> nums {3, 1, 2};
  vector<int> pre_sum = prefix_sum(nums);

  for (int i : pre_sum) {
    cout << i << endl;
  }
}
```

When Do Copies Happen?

- The copy constructor is invoked if:
  - You *initialize* an object from another object of the same type:

```
Point x;       // default ctor
Point y(x);    // copy ctor
Point z = y;   // copy ctor
```

  - You pass a non-reference object as a value parameter to a function:

```
void foo(Point x) { ... }

Point y;       // default ctor
foo(y);        // copy ctor
```

  - You return a non-reference object value from a function:

```
Point foo() {
  Point y;     // default ctor
  return y;    // copy ctor
}
```

Why Copying is an issue

- Consider this code again.
  - What happens when we run the copy constructor of a vector?
  - What do we do with memory?
  - What is the time complexity of a copy?

- Copying means we have to make an independent copy of the vector.
  - **This requires iterating over the elements**
  - **This requires dynamic memory allocation.**

```
vector<int> prefix_sum(vector<int> input) {
  vector<int> res{};
  int sum = 0;
  for (int i : input) {
    sum += i;
    res.push_back(i);
  }
  return res;
}

int main() {
  vector<int> nums {3, 1, 2};
  vector<int> pre_sum = prefix_sum(nums);

  for (int i : pre_sum) {
    cout << i << endl;
  }
}
```

Why Copying is an issue: Depending on the type

- Copying means we have to make an independent copy of the vector.
  - **This requires iterating over the elements**
  - **This requires dynamic memory allocation.**

- The cost to make a copy varies on what type we are copying.
  - Do you think this "point" class takes a lot to copy?
    - No: low time complexity and no memory allocation
  - What about a Hash Map?
    - Yes: need to reallocate all key/value pairs and iterate over the original hash map.

```
class Point {
  ...
 private:
  int x_;
  int y_;
};
```

Not all operations are equal time

- Previously: We focus on time complexity for how expensive an algorithm is
  - Not all operations are equal!
  - Adding two numbers is really quick
  - Using the heap can take a lot longer.
    - Allocation involves searching the heap for a space big enough to handle our request. Searching the heap can take some time
    - Deallocating memory is usually quicker than allocating, but some work may be done still to make future allocations take less time

- Dynamic memory is still needed, we should just not over use it
  - This also applies to other languages as well, you just don't have to worry about when to deallocate things.

Const reference most parameters

- To avoid creating copies, we pass most things in as a const references to the function.
  - References make it so we do not copy the object
  - Const makes it so that we do not accidentally modify the value if we do not need it to be modified

- We do similar things in range for loops:

```
vector<int> prefix_sum(const vector<int>& input) {
  vector<int> res{};
  int sum = 0;
  for (const int& i : input) {
    sum += i;
    res.push_back(i);
  }
  return res;
}
```

The Heap

KEY TAKEAWAY: allocating on the heap is not free, it has overhead

- The Heap is a large pool of available memory to use for Dynamic allocation
- This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- **`new`**:
  - searches for a large enough unused block of memory
  - marks the memory as allocated.
  - Returns a pointer to the beginning of that memory
- **`delete`**:
  - Takes in a pointer to a previously allocated address
  - Marks the memory as free to use.

Free Lists

- One way that allocation can be implemented is by maintaining an implicit list of the space available and space allocated.
- Before each chunk of allocated/free memory, we'll also have this metadata:

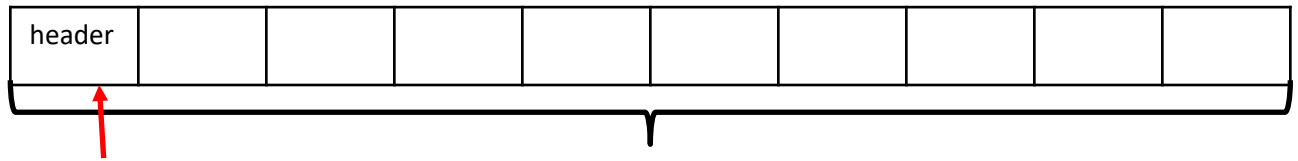
```
// this is simplified
// not what malloc/new really does
struct alloc_info {
  alloc_info* prev;
  alloc_info* next;
  bool allocated;
  size_t size;
};
```

Dynamic Memory Example

```
int main() {
  short* ptr = new short(16);
  double* ptr2 = new double(3.14);
  ...              // do stuff with ptr
  delete ptr;
  delete ptr2;
}
```

This diagram is not to scale

- free_list ->




```
{
 NULL,
 NULL,
 false,
 1024
}
```

The metadata is at the beginning of the chunk of memory

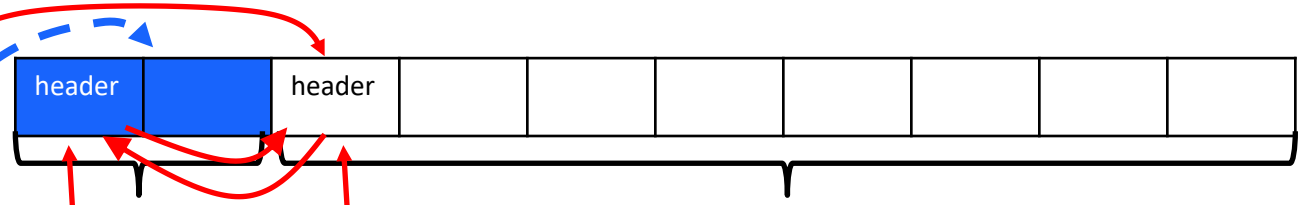
Dynamic Memory Example

```
int main() {
  short* ptr = new short(16);
  double* ptr2 = new double(3.14);
  ...              // do stuff with ptr
  delete ptr;
  delete ptr2;
}
```

Free chunks can be split to allocate blocks of specific size

new gets a pointer to just after the metadata

- free_list



"new" return value

```
{
 NULL,
 0x…,
 true,
 4
}
```

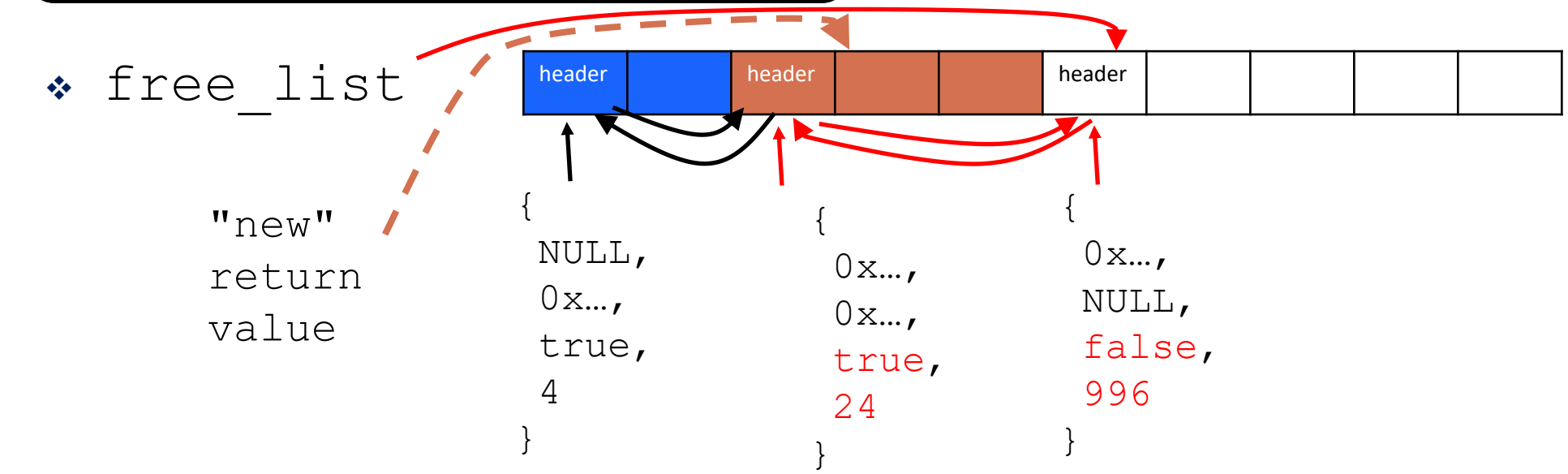
```
{
 0x…,
 NULL,
 false,
 1020
}
```

free_list points to first free chunk

Dynamic Memory Example

```
int main() {
  short* ptr = new short(16);
  double* ptr2 = new double(3.14);
  ...              // do stuff with ptr
  delete ptr;
  delete ptr2;
}
```

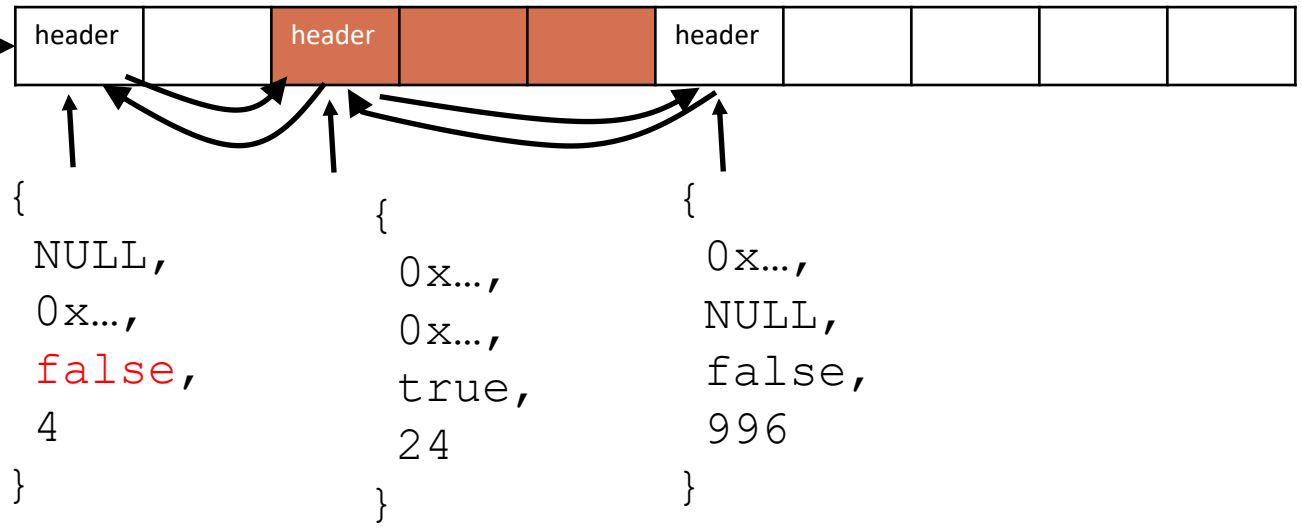
- free_list



Dynamic Memory Example

```
int main() {
  short* ptr = new short(16);
  double* ptr2 = new double(3.14);
  ...              // do stuff with ptr
  delete ptr;
  delete ptr2;
}
```

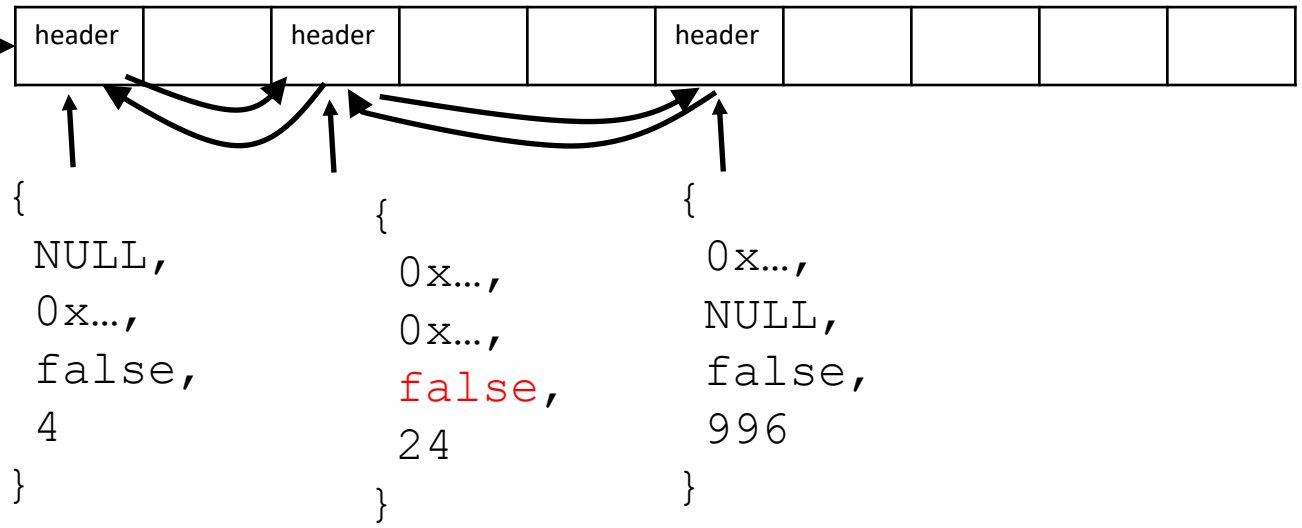
- free_list



Dynamic Memory Example

```
int main() {
  short* ptr = new short(16);
  double* ptr2 = new double(3.14);
  ...              // do stuff with ptr
  delete ptr;
  delete ptr2;
}
```

- free_list



Dynamic Memory Example

```
int main() {
  short* ptr = new short(16);
  double* ptr2 = new double(3.14);
  ...              // do stuff with ptr
  delete ptr;
  delete ptr2;
}
```

Once a block has been freed, we can try to "coalesce" it with their neighbors

The first delete couldn't be coalesced, only neighbor was allocated

- free_list


```
{
 NULL,
 0x…,
 false,
 1024
}
```

Key Takeaway

- **Dynamic memory allocation is not free and can have considerable overhead**

- Performant C++ code minimizes the number of dynamic allocations and/or custom allocators

Lecture Outline

- Copying
  - Overhead
- **Destructors & Lifetimes**
- More STL
  - std::variant
  - std::unordered_map
- Clang-tidy

Temporal Safety

- A concern in systems programming is that we can sometimes still try to access/use some data after it no longer exists
  - After the data is deallocated from the heap
  - After the data is popped off of the stack
  - The object is destructed
  - Etc.

- An important part of understanding how our basic data structures work, is so that we know how these issues can come up.

Temporal Safety

- What is the issue in this code?

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

using namespace std;


int main(int argc, char** argv) {
  vector<int> v {3, 4, 5};
  int& first = v.front();

  cout << first << endl;

  v.push_back(6);

  cout << v.size() << endl;
  cout << first << endl;
}
```

Lifetimes & Reference Invalidation

- If you read the documentation for many C++ classes, there will be sections on iterator / reference invalidation.

- An example from push_back:
  - If after the operation the new size() is greater than old capacity() a reallocation takes place, in which case all iterators (including the end() iterator) and all references to the elements are invalidated.

- Even if we don't have to allocate and deallocate things ourselves much in C++, we must still be aware of it.

Poll Everywhere

pollev.com/tqm

- What is the issue in this code?

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

using namespace std;


void func(vector<int>& v1, vector<int>& v2) {
  v1.push_back(v2.front());
}

int main() {
  vector<int> x{3, 4, 5};
  func(x, x);
}
```

Poll Everywhere

pollev.com/tqm

- What is the issue in this code?

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

using namespace std;


void func(vector<int>& v1, vector<int>& v2) {
  v1.push_back(v2.front());
}

int main() {
  vector<int> x{3, 4, 5};
  func(x, x);
}
```

push_back takes in an int&

push_back may need to resize, if it does, the reference to its front becomes invalid

Lecture Outline

- Copying
  - Overhead
- Destructors & Lifetimes
- **More STL**
  - **std::variant**
  - **std::unordered_map**
- Clang-tidy

std::variant

- Similar to how std::optional can store 1 type or nothing, std::variant can store **one of** two or more different values

```
int main() {
  variant<int, string> var {3};

  cout << holds_alternative<int>(var) << endl;
  cout << get<int>(var) << endl

  cout << holds_alternative<string>(var) << endl;
  cout << get<string>(var) << endl;
}
```

Can hold an int or string currently holding a string

Prints “true” and “3”

Prints “false”

Throws an exception

STL `unordered_map`

- One of C++'s *associative* containers: a key/value table, implemented as a Chaining Hash Map
  - http://www.cplusplus.com/reference/unordered_map/
  - General form: `unordered_map<key_type, value_type> name;`
  - Keys must be *unique*
    - **`multimap`** allows duplicate keys
  - Efficient lookup (O(1)) and insertion (O(1))
    - Access `value` via **`operator[]`** (example: **`map_name[key]`**)
      - if key doesn't exist in map, it is added to the map with a "default" value
  - Elements are type `pair<key_type, value_type>` (key is field **`first`**, value is field **`second`**)
    - Key type must be hashable

Independent types

unordered_map Example

#include <unordered_map>

```
int main(int argc, char** argv) {
  unordered_map<int,string> table{};
  unordered_map<int,string>::iterator it{};

  table.insert(pair<int,string>(2, "hello"));
  table[4] = "NGNM";
  table[6] = "mutual aid"; // inserts a value
  table[6] = "sleep";  // updates a value
  cout << "table[6]:" << table[6] << endl;



  if (table.contains(4)) {
    cout << "4 exists as a key in the map" << endl;
  }

  cout << "iterating:" << endl;
  for (auto& p : table) {
    cout << "[" << p.first << "," << p.second << "]" << endl;
  }
  return 0;
}
```

Map elements

Equivalent behavior

use .contains() to see if a key exists

Access the key and value stored in the pair

STL `unordered_set`

- One of C++'s *associative* containers: a container of unique values, implemented as a hash set
  - http://www.cplusplus.com/reference/unordered_set/
  - General form: `unordered_set<element_type> name;`
  - elements must be *unique*
    - **`multiset`** allows duplicate elements
  - Efficient lookup (O(1)) and insertion (O(1))
  - Inserting an element that already exists does nothing
  - Can use `contains(element)` to see if the element exists
  - Elements are stored in *unsorted* order
    - element type must be hashable

unordered_set Example

```
#include <unordered_set>

int main(int argc, char** argv) {
  unordered_set<string> names {};

  names.insert("bjarne");
  names.insert("ken");
  names.insert("dennis");
  names.insert("travis");
  names.insert("bjarne");

  bool exists = names.contains("bjarne");
  cout << "Is bjarne in the set?: " << exists << endl;

  numbers.erase("travis");

  for (string& name : names) {
    cout << name << endl;
  }
  return EXIT_SUCCESS;
}
```

Doesn't insert duplicate elements

prints "true"

Removes the element "travis"

Prints every name in the set

Ordered Containers (C++11)

- `map`, `set`
  - Average case for key access is O(log(n)), so generally not preferred
    - But range iterators can be more efficient than unordered `map`/`set`
  - See *C++ Primer*, online references for details

Unordered vs Ordered Containers

- The comparison between `unordered_map` vs `map` is similar to how HashMap vs TreeMap are related in java.
  - Both use the same interface
  - Have different implementations
  - If you want things to be in sorted order, use `map` (TreeMap)
  - In almost all other cases, use `unordered_map` (HashMap)

map Example

```
#include <map>
int main(int argc, char** argv) {
  map<int,string> table{};
  map<int,string>::iterator it{};

  table.insert(pair<int,string>(2, "hello"));
  table[4] = "NGNM";
  table[6] = "mutual aid"; // inserts a value
  table[6] = "sleep";  // updates a value
  cout << "table[6]:" << table[6] << endl;

  it = table.find(4);

  if (it != table.end()) {
    cout << "4 exists as a key in the map" << endl;
  }

  cout << "iterating:" << endl;
  for (pair<int, string>& p : table) {
    cout << "[" << p.first << "," << p.second << "]" << endl;
  }
  return 0;
}
```

Poll Everywhere

pollev.com/tqm

- Does this code work as expected?

```
// given a sequence of integers, return a map that maps
// each value to how many times it shows up.
// e.g.
// count_nums([5, 9 , 5, 0, 32]) should result in:
// {
//    (5, 2),
//    (9, 1),
//    (0, 1),
//    (32, 1),
// }
unordered_map<int, int> count_nums(const vector<int>& nums) {
  unordered_map<int, int> res;

  for (const int i : nums) {
    res[i] += 1;
  }

  return res;
}
```

Lecture Outline

- Copying
  - Overhead
- Destructors & Lifetimes
- More STL
  - std::variant
  - std::unordered_map
- **Clang-tidy**

Clang Tidy

- Starting in HW03, we will be using something that will automatically check the style of your code
- You can run it inside your docker container to fix things locally
  - **`make tidy-check`**

- If it runs and you have no style errors, it will look something like this:

```
Error while trying to load a compilation database:
Could not auto-detect compilation database for file "SimpleKV.cpp"
No compilation database found in /workspace/simplekv or any parent directory
fixed-compilation-database: Error while opening fixed database: No such file or directory
json-compilation-database: Error while opening JSON database: No such file or directory
Running without flags.
4 warnings generated.
Suppressed 4 warnings (4 in non-user code).
Use -header-filter=.* to display errors from all non-system headers. Use -system-headers to di
errors from system headers as well.
```

Cognitive Complexity

- Most errors are straight forward enough just from reading what the error says.
  - e.g.

```
error: parameter name 'i' is too short, expected at least 3 characters
[readability-identifier-length,-warnings-as-errors]
                    size_t i,
```

- There is one that is not clear and shows up enough to be worth going over now: “Cognitive complexity”
  - The tool calculates “cognitive complexity” of your code and will complain about anything that is too complex. This means you should think about how to break your code into helpers, because if you don’t, clang-tidy will complain and you will face a deduction.

Cognitive Complecity

- This function has Cognitive Complexity of *3*.

```
int function3(bool var1, bool var2) {
  if(var1) { // +1, nesting level +1
    if(var2)  // +2 (1 + current nesting level of 1), nesting level +1
      return 42;
  }

  return 0;
}
```

- Each if statement, loop, etc adds a +1. How “nested” it is can make it worth more

Cognitive Complecity

- Consider the code on the left
- It has a much higher complexity than the one on the right

```
bool foo(string param) {
  if ( !error1 ) {
    if ( !error2 ) {
      if ( !error3 ){
        // do some computation
        return true;
      }
    }
  }
  return false;
}
```

```
bool foo(string param) {
  if ( error1 ) {
    return false;
  }
  if ( error2 ) {
    return false;
  }
  if ( error3 ) {
    return false;
  }
  // do some computation
  return true;
}
```

That's it for now!

- More next lecture ☺