CSE 333 Section 5

Heap, Templates, STL

UNIVERSITY of WASHINGTON

Relevant Course Information

- Mid-quarter survey
  - Link: https://canvas.uw.edu/courses/1556298/quizzes/1703097
- HW2
  - Due **Thursday** (TONIGHT!) (7/21) @ 11:59pm
- Exercise 7
  - Due **Monday** (7/25) @ 11am

Dynamic Memory

New and Delete Operators

New: Allocates the type on the heap, calling its constructor if it is a class type

Syntax:

```
type* ptr = new type;

type* heap_arr = new type[num];
```

Delete: Deallocates the type from the heap, calling the destructor if it is a class type. For anything you called new on, you should at some point call `delete` to clean it up

Syntax:

```
delete ptr;

delete[] heap_arr;
```

Rule of 3

If you define any of:

1. Destructor
2. Copy Constructor
3. Assignment (`operator=`)

Then you should normally define all three. Otherwise you may run into strange behavior.

This is especially true if your class needs to allocate something on the heap. (Resources may not get allocated/deallocated correctly)

Design Considerations

- What happens if you don't define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad?
  - In C++, if you don't define any of these, one will be synthesized for you
  - The synthesized copy constructor does a shallow copy of all fields
  - The synthesized assignment operator does a shallow copy of all fields
  - The synthesized destructor calls the default destructors of any fields that have them

- How can you get the default the copy constructor/assignment operator/destructor?

Set their prototypes equal to the keyword "default": `~SomeClass() = default;`

Exercise 1

Exercise 1: Memory Leaks

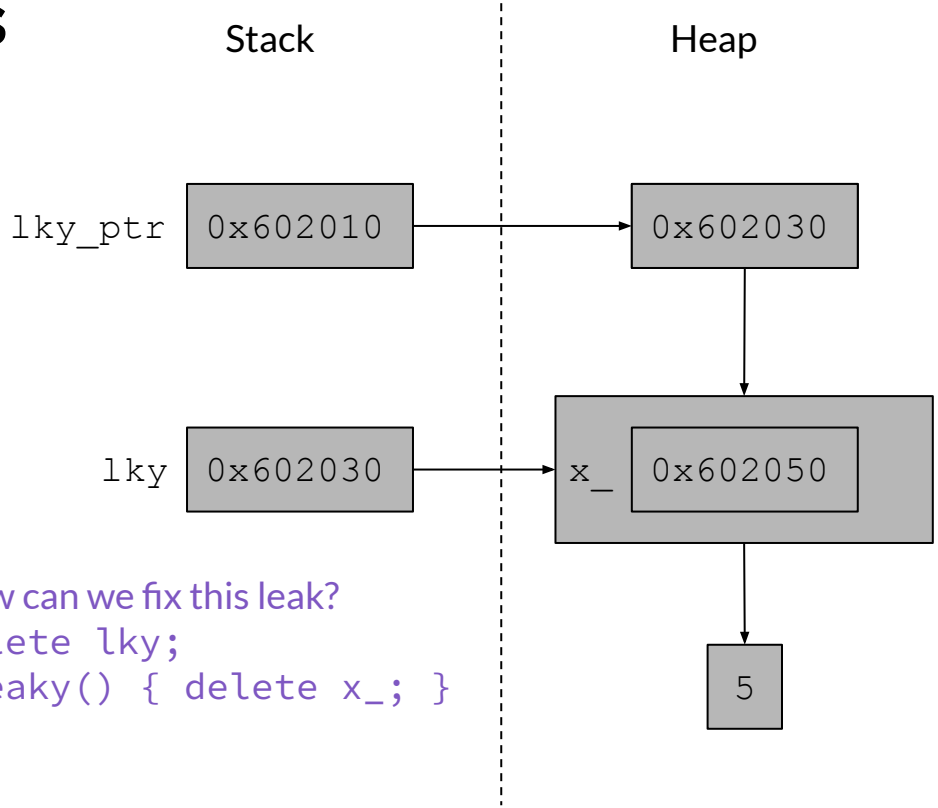
Stack

Heap

```
class Leaky {
 public:
  Leaky() { x_ = new int(5); }
 private:
  int* x_;
};

int main(int argc, char** argv) {
  Leaky** lky_ptr = new Leaky*;
  Leaky* lky = new Leaky();
  *lky_ptr = lky;
  delete lky_ptr;
  return EXIT_SUCCESS;
}
```

Exercise 1: Memory Leaks

```
class Leaky {
 public:
  Leaky() { x_ = new int(5); }
 private:
  int* x_;
};

int main(int argc, char** argv) {
  Leaky** lky_ptr = new Leaky*;
  Leaky* lky = new Leaky();
  *lky_ptr = lky;
  delete lky_ptr;
  return EXIT_SUCCESS;
}
```



How can we fix this leak?

```
delete lky;
~Leaky() { delete x_; }
```

Exercise 2

Exercise 2: Bad Copy

Stack

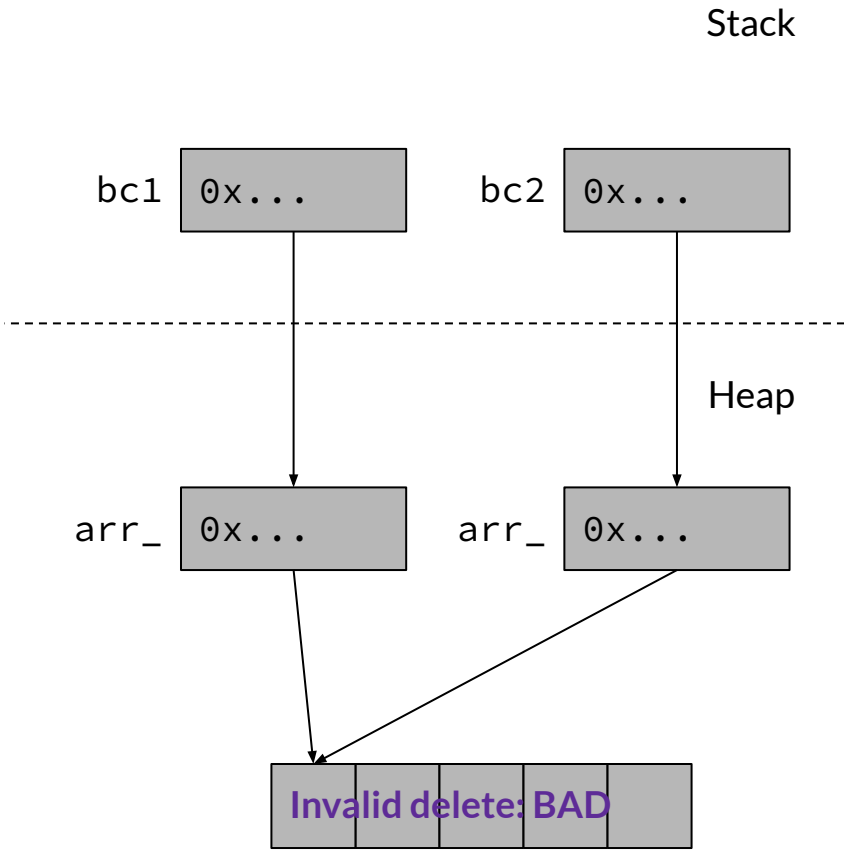
Heap

```
class BadCopy {
 public:
  BadCopy()  { arr_ = new int[5]; }
  ~BadCopy() { delete [] arr_; }
 private:
  int* arr_;
};

int main(int argc, char** argv) {
  BadCopy* bc1 = new BadCopy;
  BadCopy* bc2 = new BadCopy(*bc1); // cctor
  delete bc1;
  delete bc2;
  return EXIT_SUCCESS;
}
```

Exercise 2: Bad Copy

```
class BadCopy {
 public:
  BadCopy()  { arr_ = new int[5]; }
  ~BadCopy() { delete [] arr_; }
 private:
  int* arr_;
};

int main(int argc, char** argv) {
  BadCopy* bc1 = new BadCopy;
  BadCopy* bc2 = new BadCopy(*bc1);
  delete bc1;
  delete bc2;
  return EXIT_SUCCESS;
}
```

as if!



Templates!

Templates

- C++ syntax to generate code that works with *generic types*
- Generates **at compile time** instructions on each way a function is used:
  - *e.g.*, calls to `foo<int>()` and `foo<double>()` generate two implementations
  - *e.g.*, calls to `foo<int>()` and another `foo<int>()` require only one implementation
  - *e.g.*, if `foo` is never used, zero implementations are generated

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);
add3<char*>("a str");
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");   // uses add3<char*>, return ->"tr"
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");   // uses add3<char*>, return ->"tr"
add3<string>("a str"); // Compiler error! No `+` for string
                        //                 and int
```

Template Class

- Meant for supporting *generic types*:

```
typedef uint64_t HTKey_t;
typedef void*    HTValue_t;
typedef struct {
  HTKey_t   key;
  HTValue_t value;
} HTKeyValue_t;
```

```
template<typename K, typename V>
struct HTKeyValue {
  K HTKey;
  V* HTValue;
};
```

Exercise 3

Exercise 3

```
_______________                    // template type definition
struct Node {
  _______________                  // two-argument constructor

  ~Node() { delete value; }        // destructor cleans up the payload

  _______________                  // public field value
  _______________                  // public field next
};
```

Exercise 3 Solution

```
template <typename T>          // template type definition
struct Node {
  ______________              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  ______________              // public field value
  ______________              // public field next
};
```

Exercise 3 Solution

```
template <typename T>          // template type definition
struct Node {
  ______________              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  T* value                    // public field value
  Node<T>* next               // public field next
};
```

Exercise 3 Solution

```
template <typename T>          // template type definition
struct Node {
  Node(T* val, Node<T>* node): value(val), next(node) {}
                              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  T* value                    // public field value
  Node<T>* next               // public field next
};
```

Containers!

C++ standard lib is built around templates

- ***Containers*** store data using various underlying data structures
  - The specifics of the data structures define properties and operations for the container
- ***Iterators*** allow you to traverse container data
  - Iterators form the common interface to containers
  - Different flavors based on underlying data structure
- ***Algorithms*** perform common, useful operations on containers
  - Use the common interface of iterators, but different algorithms require different ‘complexities’ of iterators

Common C++ STL Containers (and Java equiv)

- *Sequence* containers can be accessed sequentially
  - `vector<Item>` uses a dynamically-sized contiguous array (like `ArrayList`)
  - `list<Item>` uses a doubly-linked list (like `LinkedList`)
- *Associative* containers use search trees and are sorted by keys
  - `set<Key>` only stores keys (like `TreeSet`)
  - `map<Key,Value>` stores key-value `pair<>`'s (like `TreeMap`)
- *Unordered associative* containers are hashed
  - `unordered_map<Key,Value>` (like `HashMap`)

Common C++ STL Methods

	vector	list	set	map	unordered_map
`.size()` // get number of elements	✔	✔	✔	✔	✔
`.push_back()` // add element to back `.pop_back()` // remove back element	✔	✔			
`.push_front()` // add element to front `.pop_front()` // remove front element		✔			
`.operator[]()` // random access element	✔			✔	✔
`.insert()` // insert key			✔	✔	✔
`.find()` // find key			✔	✔	✔

Common STL Containers

Many more containers and methods!

See full documentation here:
http://www.cplusplus.com/reference/stl

Exercise 4

Exercise 4

```
using namespace std;
vector<string> ChangeWords(const vector<string>& words,
                           map<string,string>& subs) {









}
```

Exercise 4 Solution

```
using namespace std;
vector<string> ChangeWords(const vector<string>& words,
                           map<string,string>& subs) {
  vector<string> result;
  for (auto& word : words) {
    if (subs.find(word) != subs.end()) {
      result.push_back(subs[word]);
    } else {
      result.push_back(word);
    }
  }
  return result;
}
```