CSE 333
Section 5

Dynamic Memory, Templates, and STL

Administrivia

HW2	Due Tonight 11:59pm
Exercise 7	Due Monday 11am

Dynamic Memory

New and Delete Operators

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

```
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

```
delete ptr;

delete[] heap_arr;
```

Exercise 1: Memory Leaks

Stack

Heap

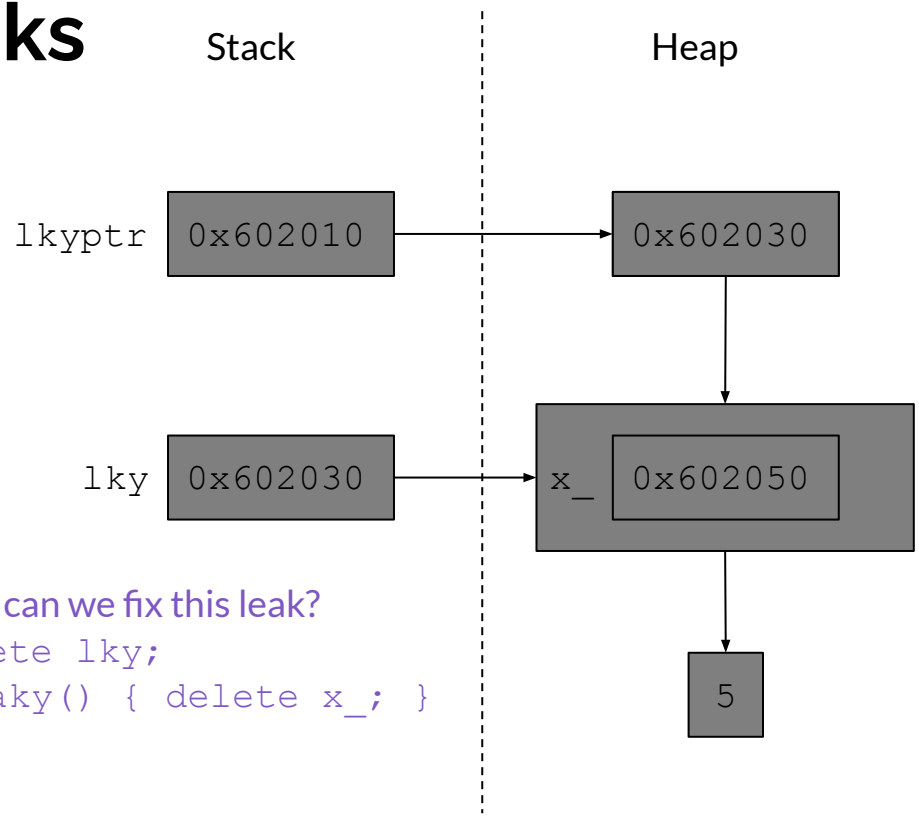
```
class Leaky {
 public:
  Leaky() { x_ = new int(5); }
 private:
  int *x_;
};

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



How can we fix this leak?

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

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;
}
```

Stack

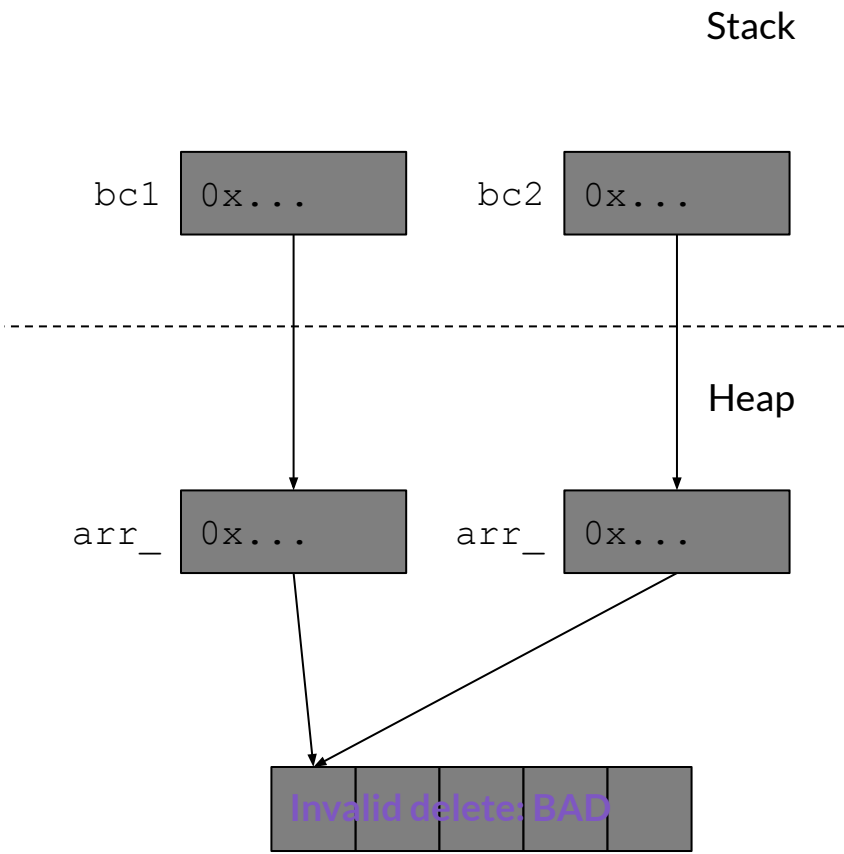
Heap

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!



C++ Templates

Templates

- C++ syntax to generate code that works with *generic types*
- Generates a new implementation in assembly for every type it is used with:
  - *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 num
```

Template Class

- A member variable of a template class can be declared using one of the class' template types
  - Very useful for implementing data structures that support *generic types*:

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

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

Exercise 3

Exercise 3 Solution

```
_______________                 // 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
};
```

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 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 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;
}
```