Poll Everywhere

pollev.com/cse333

Where are you so far on Homework 2?

A. Haven't started yet
B. Working on Part A (File Parser)
C. Working on Part B (File Crawler and Indexer)
D. Working on Part C (Query Processor)
E. Done!
F. Prefer not to say

C++ Heap, Templates (start)

CSE 333 Summer 2023

Instructor: Timmy Yang

Teaching Assistants:

Jennifer Xu Leanna Nguyen Pedro Amarante

Sara Deutscher Tanmay Shah

Relevant Course Information

- Exercise 6 due Monday (7/17)
- Exercise 7 out Today (7/14)
  - Due next Wednesday (7/19)
  - Will build on Exercise 6 and use what a lot of is discussed today
- Homework 2 due Thursday (7/20)
  - File system crawler, indexer, and search engine
  - Don't forget to clone your repo to double-/triple-/quadruple-check compilation!
  - Don't modify the header files!

Poll Everywhere

pollev.com/cse333

- How many times does the ***destructor*** get invoked?
  - Assume `Point` with everything defined (ctor, cctor, =, dtor)
  - Assume no compiler optimizations

count.cc

```
void PrintPoint(Point pt);

int main(int argc, char** argv) {
  Point origin(0, 0);
  Point zero = origin;
  PrintPoint(zero);
  return EXIT_SUCCESS;
}

void PrintPoint(Point pt) {
  cout << "(" << pt.get_x() << ", ";
  cout << pt.get_y() << ")" << endl;
}
```

A. 1
B. 2
C. 3
D. 4
E. We're lost…

Class Definition (from last lecture)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(int x, int y);                    // constructor
  int get_x() const { return x_; }        // inline member function
  int get_y() const { return y_; }        // inline member function
  double Distance(const Point& p) const;      // member function
  void SetLocation(int x, int y);             // member function

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

#endif  // POINT_H_
```

Poll Everywhere

pollev.com/cse333

- How many times does the ***destructor*** get invoked?

ctor	cctor	op=	dtor

count.cc

```
void PrintPoint(Point pt);

int main(int argc, char** argv) {
  Point origin(0, 0);
  Point zero = origin;
  PrintPoint(zero);
  return EXIT_SUCCESS;
}

void PrintPoint(Point pt) {
  cout << "(" << pt.get_x() << ", ";
  cout << pt.get_y() << ")" << endl;
}
```

Lecture Outline

- **Using the Heap**
  - **`new` / `delete` / `delete[]`**
- Templates

C++11 `nullptr`

- C and C++ have long used `NULL` as a pointer value that references nothing

- C++11 introduced a new literal for this: `nullptr`
  - New reserved word
  - Interchangeable with `NULL` for all practical purposes, but it has type `T*` for any/every `T`, and is not an integer value
    - Avoids funny edge cases (see C++ references for details)
    - Still can convert to/from integer `0` for tests, assignment, etc.
  - Advice: prefer `nullptr` in C++11 code
    - Though `NULL` will also be around for a long, long time

new/delete

- To allocate on the heap using C++, you use the `new` keyword instead of **`malloc`**`()` from `stdlib.h`
  - You can use new to allocate an object (*e.g.*, `new Point`)
  - You can use new to allocate a primitive type (*e.g.*, `new int`)

- To deallocate a heap-allocated object or primitive, use the `delete` keyword instead of **`free`**`()` from `stdlib.h`
  - Don't mix and match!
    - *Never* **`free`**`()` something allocated with `new`
    - *Never* `delete` something allocated with **`malloc`**`()`
    - Careful if you're using a legacy C code library or module in C++

`new`/`delete` Behavior

- `new` behavior:
  - When allocating you can specify a constructor or initial value
    - *e.g.*, `new Point(1, 2)`, `new int(333)`
  - If no initialization specified, it will use default constructor for objects and uninitialized ("mystery") data for primitives
  - You don't need to check that `new` returns `nullptr`
    - When an error is encountered, an exception is thrown (that we won't worry about)

- `delete` behavior:
  - If you `delete` already `delete`d memory, then you will get undefined behavior (same as when you double **`free`** in C)

new/delete Example

```
int* AllocateInt(int x) {
  int* heapy_int = new int;
  *heapy_int = x;
  return heapy_int;
}
```

```
Point* AllocatePoint(int x, int y) {
  Point* heapy_pt = new Point(x,y);
  return heapy_pt;
}
```

heappoint.cc

```
#include "Point.h"

...  // definitions of AllocateInt() and AllocatePoint()

int main() {
  Point* x = AllocatePoint(1, 2);
  int* y = AllocateInt(3);

  cout << "x's x_ coord: " << x->get_x() << endl;
  cout << "y: " << y << ", *y: " << *y << endl;

  delete x;
  delete y;
  return EXIT_SUCCESS;
}
```

Dynamically Allocated Arrays

- To dynamically allocate an array:
  - Default initialize: `type* name = new type[size];`
- To dynamically deallocate an array:
  - Use `delete[] name;`
  - It is an *incorrect* to use "`delete name;`" on an array
    - The compiler probably won't catch this, though (**!**) because it can't always tell if `name*` was allocated with `new type[size];` or `new type;`
      - Especially inside a function where a pointer parameter could point to a single item or an array and there's no way to tell which!
    - Result of wrong `delete` is undefined behavior

Arrays Example (primitive)

arrays.cc

```
#include "Point.h"

int main() {
  int stack_int;
  int* heap_int = new int;
  int* heap_int_init = new int(12);

  int stack_arr[3];
  int* heap_arr = new int[3];

  int* heap_arr_init_val = new int[3]();
  int* heap_arr_init_lst = new int[3]{4, 5};  // C++11

  ...

  delete heap_int;              //
  delete heap_int_init;         //
  delete heap_arr;              //
  delete[] heap_arr_init_val;   //

  return EXIT_SUCCESS;
}
```

Arrays Example (class objects)

arrays.cc

```
#include "Point.h"

int main() {
  ...

  Point stack_pt(1, 2);
  Point* heap_pt = new Point(1, 2);

  Point* heap_pt_arr_err = new Point[2];

  Point* heap_pt_arr_init_lst = new Point[2]{{1, 2}, {3, 4}};
                                                    // C++11

  ...

  delete heap_pt;
  delete[] heap_pt_arr_init_lst;

  return EXIT_SUCCESS;
}
```

`malloc` vs. `new`

	`malloc()`	`new`
What is it?	a function	an operator or keyword
How often used (in C)?	often	never
How often used (in C++)?	rarely	often
Allocated memory for	anything	arrays, structs, objects, primitives
Returns	a `void*` (*should be cast*)	appropriate pointer type (*doesn't need a cast*)
When out of memory	returns `NULL`	throws an exception
Deallocating	`free()`	`delete` or `delete[]`

Poll Everywhere

pollev.com/cse333

What will happen when we invoke `Bar()`?

- If there is an error, how would you fix it?

A. Bad dereference
B. Bad delete
C. Memory leak
D. "Works" fine
E. We're lost…

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

void Foo::Init(int val) {
   foo_ptr_ = new int;
  *foo_ptr_ = val;
}

Foo& Foo::operator=(const Foo& rhs) {
  delete foo_ptr_;
  Init(*(rhs.foo_ptr_));
  return *this;
}

void Bar() {
  Foo a(10);
  Foo b(20);
  a = a;
}
```

Rule of Three, Revisited

- Now what will happen when we invoke **Bar**()?
  - If there is an error, how would you fix it?

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

void Foo::Init(int val) {
   foo_ptr_ = new int;
  *foo_ptr_ = val;
}

Foo& Foo::operator=(const Foo& rhs) {
  if (&rhs != this) {
    delete foo_ptr_;
    Init(*(rhs.foo_ptr_));
  }
  return *this;
}

void Bar() {
  Foo a(10);
  Foo b = a;
}
```

Lecture Outline

- Using the Heap
  - `new` / `delete` / `delete[]`
- **Templates**

Suppose that…

- You want to write a function to compare two `int`s
- You want to write a function to compare two `string`s
  - Function overloading!

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int& value1, const int& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const string& value1, const string& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}
```

Hm…

- The two implementations of `compare` are nearly identical!
  - What if we wanted a version of `compare` for *every* comparable type?
  - We could write (many) more functions, but that's obviously wasteful and redundant

- What we'd prefer to do is write "*generic code*"
  - Code that is type-independent
  - Code that is compile-type polymorphic across types

C++ Parametric Polymorphism

- C++ has the notion of templates
  - A function or class that accepts a ***type*** as a parameter
    - You define the function or class once in a type-agnostic way
    - When you invoke the function or instantiate the class, you specify (one or more) types or values as arguments to it
  - At ***compile-time***, the compiler will generate the "specialized" code from your template using the types you provided
    - Your template definition is NOT runnable code
    - Code is *only* generated if you use your template

Function Templates

- Template to **compare** two "things":

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

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>   // <...> can also be written <class T>
int compare(const T &value1, const T &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare<int>(10, 20) << std::endl;
  std::cout << compare<std::string>(h, w) << std::endl;
  std::cout << compare<double>(50.5, 50.6) << std::endl;
  return EXIT_SUCCESS;
}
```

functiontemplate.cc

Compiler Inference

- Same thing, but letting the compiler infer the types:

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

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>
int compare(const T &value1, const T &value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare(10, 20) << std::endl; // ok
  std::cout << compare(h, w) << std::endl;   // ok
  std::cout << compare("Hello", "World") << std::endl;  // hm…
  return EXIT_SUCCESS;
}
```

functiontemplate_infer.cc

Extra Exercise #1

- Write a C++ function that:
  - Uses `new` to dynamically allocate an array of strings and uses `delete[]` to free it
  - Uses `new` to dynamically allocate an array of pointers to strings
    - Assign each entry of the array to a string allocated using `new`
  - Cleans up before exiting
    - Use `delete` to delete each allocated string
    - Uses `delete[]` to delete the string pointer array
    - (whew!)

BONUS SLIDES

An extra example for practice with class design and heap-allocated data: a C-string wrapper class classed `Str`.

Heap Member (extra example)

- Let's build a class to simulate some of the functionality of the C++ string
  - Internal representation: c-string to hold characters

null-terminated char *

- What might we want to implement in the class?

default constructor → "" string is '\0'

constructor from char*

print to ostream

length → reminder: this doesn't count the null terminator

concatenation → we'll do append instead, which is similar

copy constructor

destructor → clean up internal mem!

Str Class

Str.h

```
#include <iostream>
using namespace std;    // should replace this

class Str {
 public:
  Str();                // default ctor
  Str(const char* s);   // c-string ctor
  Str(const Str& s);    // copy ctor
  ~Str();               // dtor

  int length() const;   // return length of string
  char* c_str() const; // return a copy of st_
  void append(const Str& s);

  Str& operator=(const Str& s);  // string assignment

  friend std::ostream& operator<<(std::ostream& out, const Str& s);

 private:
  char* st_;  // c-string on heap (terminated by '\0')
};  // class Str
```

Str::append (extra example)

- Complete the **append()** member function:
  - `char* strncpy(char* dst, char* src, size_t num);`
  - `char* strncat(char* dst, char* src, size_t num);`

```
#include <cstring>
#include "Str.h"
// append contents of s to the end of this string
void Str::append(const Str& s) {



}
```

see Str.cc