C++ Heap, Deep Copies

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Exercise 9 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I prefer not to say

Administrivia

- Exercise 11 released today, due Wednesday
  - Implement `Vector`: dynamically allocated memory, practice with `friend` functions
  - Refer to `Str.h`/`Str.cc`
- Homework 2 due this Thursday (10/24)
  - 😱😱😱

Lecture Outline

- **Destructors!**
- Using the Heap in C++
  - `new` / `delete` / `delete[]`
- Deep Copies: Why Defaults Matter
- Operators and Friends

Destructors

- C++ has the notion of a destructor (dtor)
  - Invoked *automatically* when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)
  - Place to put your cleanup code
  - **A standard C++ idiom** for managing dynamic resources!
    - Slogan: "*Resource Acquisition Is Initialization*" (RAII)

```
Point::~Point() {   // destructor
  // do any cleanup needed when a Point object goes away
  // (nothing to do here since we have no dynamic resources)
}
```

```
FileCloser fc(fd);

Mutex m(&atomic_variable);

Deleter d(&mybuffer);
```

```
Rollbacker rb;
if (sqlQuery.success()) {
  rb.setCanCommitTxn(true);
}
```

Lecture Outline

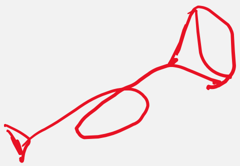
- Destructors!
- **Using the Heap in C++**
  - `new` / `delete` / `delete[]`
- Deep Copies: Why Defaults Matter
- Operators and Friends

An Aside: C++11 `nullptr`

- C/C++ have long used `NULL` as an invalid pointer value
- C++11 introduced a new literal for this: `nullptr`
  - New reserved word
  - Basically interchangeable with `NULL` … but typesafe!
    - It has type `T*` for any/every `T`, and is not an integer value
  - <u>Advice</u>: prefer `nullptr` in C++11 code

```
void foo(int i);        // #1
void foo(char *str);    // #2

foo(0);         // Calls #1
foo("bar");     // Calls #2
foo(NULL);      // Calls #1.  Why is there no sad trombone emoji?
```



Dynamically-allocated instances: `new`/`delete`

- To allocate on the heap, use the `new` keyword
  - Same for objects (*e.g.* `new Point`) and primitive types (*e.g.* `new int`)
  - Will call the appropriate constructor for class instances!
- To deallocate, use the `delete` keyword
- Built into the language; no need for `<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 Example

heappoint.cc

```
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;
}
```

```
#include "Point.h"
using std::cout;
using std::endl;

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

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

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

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

Dynamically-allocated arrays: `new/delete[]`

- To dynamically allocate an array:
  - Default initialize: `type *name = new type[size];`

new int[100];
new Point[100];

- To dynamically deallocate an array:
  - Use `delete[] name;`
  - It is *incorrect* to use "`delete name;`" on an array
    - The compiler probably won't catch this (!) -- it can't tell if `name*` was allocated with `new type[size]` or `new type;`
    - Result of wrong `delete` is undefined behavior

Arrays Example: (leaking some) primitives

arrays.cc

```
#include "Point.h"

int main(int argc, char **argv) {
  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;                // (1)
  delete heap_int_init;           // (2)
  delete[] heap_arr;              // (3) LEAK!
  delete[] heap_arr_init_val;     // (4)
  delete[] heap_arr_init_lst      // LEAK!
  return EXIT_SUCCESS;
}
```

Arrays Example: class objects

arrays.cc

```
#include "Point.h"

int main(int argc, char **argv) {
  ...

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

typically unhandled; just let program crash

Lecture Outline

- 💥 Destructors! 💥
- Using the Heap in C++
  - `new` / `delete` / `delete[]`
- **Deep Copies: Why Defaults Matter**
- Operators and Friends

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

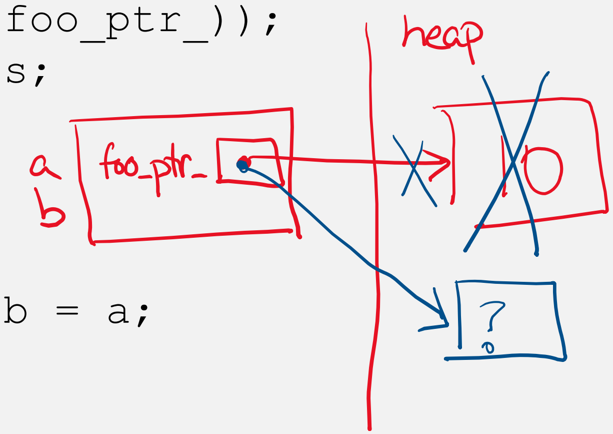
E. I'm not sure …

```
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);
  const Foo &b = a;
  a = b;
}
```



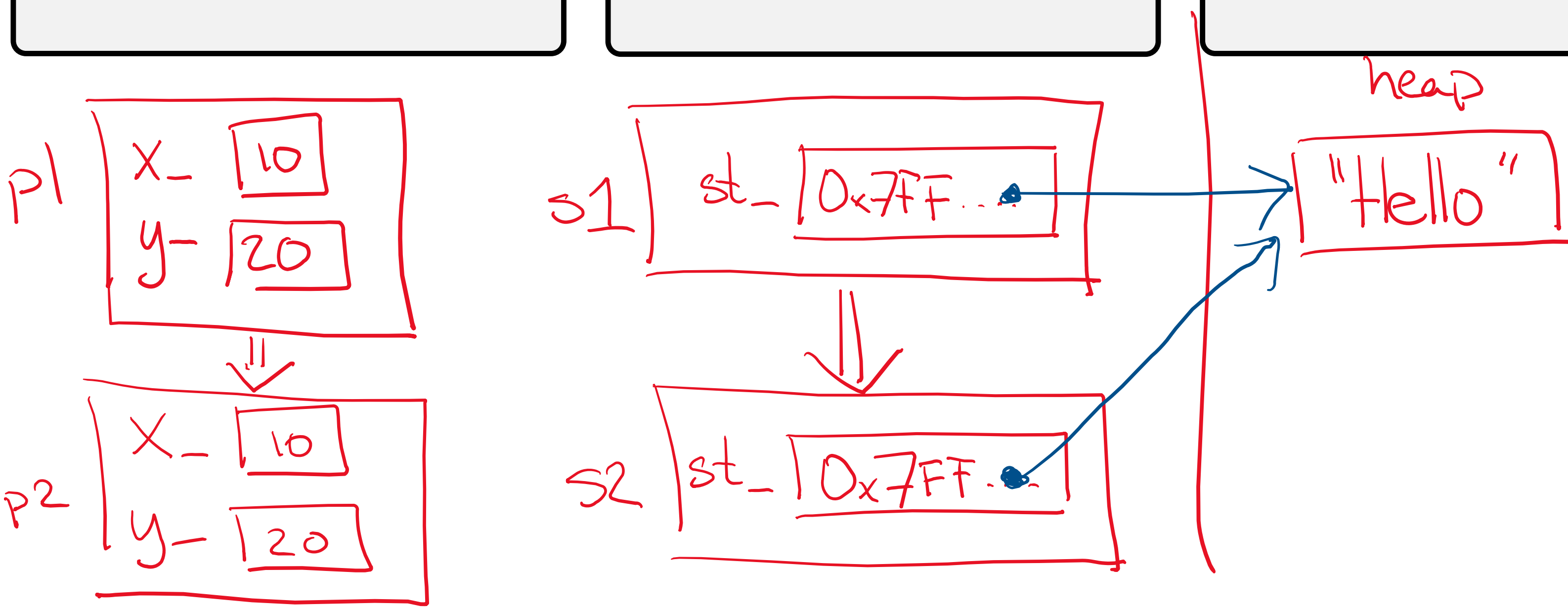
What's In a Default, Anyway?

- Compiler-provided cctor and **`operator=`** are basically **`memcpy`** when copied members are primitive types

```
class Point {
  // ...
 private:
  int x_, y_;
};
Point p2 = p1;
```

```
class Str {
  // ...
 private:
  char *st_;
};
Str s2 = s1;
```

```
class Node {
  // ...
 private:
  LLPayload_t payload_;
};
Node n2 = n1;
```



Shallow vs Deep Copies

- The byte-by-byte `memcpy`-style copy is a *shallow copy*
- Copying pointed-to fields is known as a *deep copy*
  - Necessary for more complex class definitions that must "release" internally-held resources (eg, file handles, dynamic memory)
  - If deep copies are necessary, must implement both the copy constructor and assignment operator

Rule of Three

"More what you would call a guideline than an actual rule"

- If you define any of:
  1) Destructor ← if the destructor is necessary
  2) Copy Constructor
  3) Assignment (`operator=`) } then overriding copies and assignment is probably also necessary

- Then you probably need to define all three
  - Can explicitly ask for default synthesized versions (C++11):

```
class Point {
 public:
  Point() = default;                                  // the default ctor
  ~Point() = default;                                 // the default dtor
  Point(const Point& copyme) = default;               // the default cctor
  Point& operator=(const Point& rhs) = default;       // the default "="
  ...
```

Avoiding the Insanity (of deep copies)

- Thanks to C++ destructors, we can do complicated (but cool) things with object lifetimes

- But now we have to be thoughtful about copy semantics
  - What does it mean to "copy" an object that manages a dynamically-allocated buffer?
  - What does it mean to "assign" a mutex?

- **Best practice**: Implement both xor disable both

Avoiding the Insanity (of deep copies)

- Pre-C++11:
  - Disable the copy constructor and assignment operator by *declaring* as private and *not defining* them

UncopyablePoint.h

```
class UncopyablePoint {
 public:
  UncopyablePoint(int x, int y) : x_(x), y_(y) { }  // ctor
  ...
 private:
  UncopyablePoint(const UncopyablePoint& copyme);
  UncopyablePoint& operator=(const UncopyablePoint& rhs);
  ...
};  // class Point

UncopyablePoint w;         // compiler error (no default constructor)
UncopyablePoint x(1, 2);   // OK!
UncopyablePoint y = w;     // compiler error (no copy constructor)
y = x;                     // compiler error (no assignment operator)
```

Avoiding the Insanity (of deep copies)

- C++11 added new syntax to do this directly
  - **This is the better choice** in C++11 code

UncopyablePoint.h

```
class UncopyablePoint {
 public:
  UncopyablePoint(int x, int y) : x_(x), y_(y) { }
  ...
  UncopyablePoint(const UncopyablePoint& copyme) = delete;
  UncopyablePoint& operator=(const UncopyablePoint& rhs) = delete;
 private:
  ...
};  // class UncopyablePoint

UncopyablePoint w;          // compiler error (no default constructor)
UncopyablePoint x(1, 2);    // OK!
UncopyablePoint y = w;      // compiler error (no copy constructor)
y = x;                      // compiler error (no assignment operator)
```

Avoiding the Insanity (of deep copies)

- A `CopyFrom` function can be used manually by the caller when occasionally needed
- Or you can use it to implement both cctor and assign op

UncopyablePoint2011.h

```
class UncopyablePoint {
 public:
  UncopyablePoint(int x, int y) : x_(x), y_(y) { }  // ctor
  void CopyFrom(const UncopyablePoint &copyme);
  ...
  UncopyablePoint(const Point &copyme) = delete;
  UncopyablePoint& operator=(const UncopyablePoint &rhs) = delete;
 private:
  ...
};  // class UncopyablePoint
```

sanepoint.cc

```
UncopyablePoint x(1, 2);  // OK
UncopyablePoint y(3, 4);  // OK
x.CopyFrom(y);  // OK
```

Lecture Outline

- 💥 Destructors! 💥
- Using the Heap in C++
  - `new` / `delete` / `delete[]`
- Deep Copies: Why Defaults Matter
- **Operators and Friends**

Review: Nonmember Functions

- "Nonmember functions" are just normal functions that happen to use some class
  - Called like a regular function, not as a member of a class instance
  - These do *not* have access to the class' private members
- Useful nonmember functions often included as part of interface to a class
  - Declaration goes in header file, but *outside* of class definition

Review: Operator Overloading

- Can overload operators using **member functions**
  - Restriction: left-hand side argument must be a class you are implementing

```
Str& operator+=(const Str &s) { ... }
```

- Can overload operators using **nonmember functions**
  - No restriction on arguments (can specify any two)
    - **Our only option** when the left-hand side is a class you do not have control over, like `ostream` or `istream`.
  - But no access to private data members

```
Str operator+(const Str &a, const Str &b) { ... }
```

friend Nonmember Functions

- A class can give a nonmember function (or class) access to its non-`public` members by declaring it as a `friend` within its definition
  - Not a class member, but has access privileges as if it were
  - `friend` functions are usually unnecessary if your class includes appropriate public "getter" functions

Str.h

```
class Str {
  ...
  friend std::ostream& operator<<(std::ostream &out, Str &s);
  ...
};  // class Point
```

```
std::ostream& operator<<(std::ostream &out, Str &s) {
  ...
}
```

Str.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!)