C++ Heap

CSE 333 Fall 2023

Instructor: Chris Thachuk

Teaching Assistants:

Ann Baturytski
Yuquan Deng
Noa Ferman
James Froelich
Hannah Jiang
Yegor Kuznetsov
Humza Lala
Alan Li
Leanna Mi Nguyen
Chanh Truong
Jennifer Xu

Relevant Course Information

- Exercise 6 due tonight
- Exercise 7 due next Wednesday
  - Will build on Exercise 6

- Homework 2 due next Monday (10/30)
  - **Hw2 partner declaration due this Thursday (10/26)**

- Midterm this Friday in class (10/27)
  - A single 3”x5” index card with handwritten notes is allowed.

Lecture Outline

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

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.
  - <u>Advice</u>: 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];`

new still returns a pointer

- To dynamically deallocate an array:
  - Use `delete[] name;`
  - It is *incorrect* to use "`delete name;`" on an array

is this a pointer to a thing or an array of things?

    - 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;   // stack (uninitialized)
  int* heap_int = new int;   // heap (uninitialized)
  int* heap_int_init = new int(12);   // heap (value 12)

  int stack_arr[3];   // stack (uninitialized)
  int* heap_arr = new int[3];   // heap (uninitialized)

  int* heap_arr_init_val = new int[3]();   // heap (values 0)
  int* heap_arr_init_lst = new int[3]{4, 5};   // C++11
                                  // heap (initialized to {4,5,0})
  ...

  delete heap_int;              // correct!
  delete heap_int_init;         // correct!
  delete heap_arr;              // incorrect! should be delete[]
  delete[] heap_arr_init_val;   // correct!
  // memory leak of heap_arr_init_lst!
  return EXIT_SUCCESS;
}
```

Arrays Example (class objects)

arrays.cc

```
#include "Point.h"

int main() {
  ...

  Point stack_pt(1, 2);          //stack object
  Point* heap_pt = new Point(1, 2);    // heap object

X Point* heap_pt_arr_err = new Point[2];  // default constructed objects
                                          // error! no default constructor in Point

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

  ...

  delete heap_pt;                  // correct
  delete[] heap_pt_arr_init_lst;   // correct

  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 (always given a type)
Returns	a `void*` (*should be cast*)	(new T returns T*) appropriate pointer type (*doesn't need a cast*)
When out of memory	returns `NULL`	throws an exception (usually ignored)
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?

double delete error!

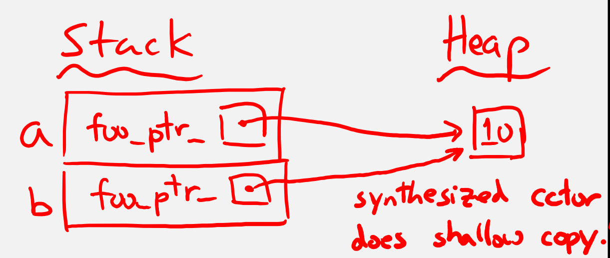
should define cctor to dynamically allocate space for copy of int

```
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;
}
```



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

}
```

Clone

- C++11 style guide tip:
  - If you disable them, then you instead may want an explicit "Clone" function that can be used when occasionally needed

Point_2011.h

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

sanepoint.cc

```
Point x(1, 2);  // OK
Point y(3, 4);  // OK
x.Clone(y);  // OK
```