

# C++ Heap

## CSE 333 Spring 2023

**Instructor:** Chris Thachuk

**Teaching Assistants:**

Byron Jin

Deeksha Vawani

Humza Lala

Noa Ferman

Seulchan (Paul) Han

Tim Mandzyuk

CJ Reith

Edward Zhang

Lahari Nidadavolu

Saket Gollapudi

Timmy Yang

Wui Wu

# Relevant Course Information

- ❖ Exercise 7 out today, due next Monday (5/1)
  - Will build on Exercise 6 and use what a lot of is discussed today
- ❖ Homework 2 due Thursday (4/27)
  - 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!
- ❖ Midterm: May 3 - 5
  - Take home (Gradescope) and open notes
  - Will involve reflecting on previous assignments
  - Individual, but high-level discussion allowed ("Gilligan's Island Rule")

# 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.
  - 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 `deleted` 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

	<code>malloc()</code>	<code>new</code>
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 <code>void*</code> <i>(should be cast)</i>	appropriate pointer type <i>(doesn't need a cast)</i>
When out of memory	returns <code>NULL</code>	throws an exception
Deallocating	<code>free()</code>	<code>delete</code> or <code>delete []</code>



# Poll Everywhere

[pollev.com/cse333sp](http://pollev.com/cse333sp)

## 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;
}
```

# 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!)