C++ Class Details, Heap
CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:
Justin Tysdal
Sayuj Shahi
Nicholas Batchelder
Leanna Mi Nguyen
Administrivia

- Homework 2 due Thursday night
  - **Check your work!!** Allocate time to clone the repo when you’re done, do `git checkout hw2-final; cd hw1` and `copy/build libhw1.a; cd hw2; make; then test everything looks good`
  - Reminder: **do not modify header files**
  - Reminder: commit/push your work regularly, not all at once at the
Administrivia

- Exercise 9 due this morning, Exercise 10 due on Wednesday

- Unfortunately, Exercise 11 needs to be due before the midterm...
Administrivia

- Midterm exam in a week:
  Monday 7/22, 1:10 - 2:10 in SMI 211
  - Topic list and old exams on website now (see exams link on resources page)
  - Closed book, slides, etc., but you may have one 5x8 notecard with whatever handwritten notes you want on both sides
Administrivia

- Extra midterm points for coming to office hours this week
  - +5 points on the midterm (out of 100), but can’t go above 100 total
  - Must go to an existing, in-person office hours and bring a problem set to work on; either from the extra-problems in the slides, or an old midterm question
  - Make sure the TA writes down your name
Administrivia

- Midterm review in section this week
Lecture Outline

❖ Class Details
  ▪ Rule of Three / Making Copies
  ▪ Access Controls and Friends
  ▪ Namespaces

❖ Using the Heap
  ▪ \texttt{new/\texttt{delete}/\texttt{delete}[]}
  ▪ String Class walkthrough
Rule of Three

❖ If you define any of:

1) Destructor
2) Copy Constructor
3) Assignment (operator=)

❖ Then you should normally define all three
   ▪ Can explicitly ask for default synthesized versions (C++11 & later):

```cpp
class Point {
public:
    Point() = default; // the default ctor
    ~Point() = default; // the default dtor
    Point(const Point& copyme) = default; // the defaultcctor
    Point& operator=(const Point& rhs) = default; // the default "="
    ...
```
Dealing with the insanity

❖ C++ style guide tip:

- If possible, disable the copy constructor and assignment operator if not needed — avoids implicit invocation and excessive copying. C++11 and later have direct syntax to indicate this:

```cpp
class Point {
public:
    Point(const int x, const int y) : x_(x), y_(y) { } // ctor
    ... Point(const Point& copyme) = delete; // declare cctor and "=" to
    Point& operator=(const Point& rhs) = delete; // be deleted (C++11)
private:
    ...
}; // class Point
```

```cpp
Point w; // compiler error (no default constructor)
Point x(1, 2); // OK!
Point y = w; // compiler error (no copy constructor)
y = x; // compiler error (no assignment operator)
```
If you’re dealing with old code …

- In pre-C++11 code the copy constructor and assignment were often disabled by making them private and not implementing them (you may see this)…

```cpp
class Point {
    public:
    Point(const int x, const int y) : x_(x), y_(y) { } // ctor
...
    private:
    Point(const Point& copyme); // disable cctor (no def.)
    Point& operator=(const Point& rhs); // disable "=" (no def.)
...
}; // class Point

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

❖ Class Details
  ▪ Rule of Three / Making Copies
  ▪ Access Controls and Friends
  ▪ Namespaces

❖ Using the Heap
  ▪ new / delete / delete[]
  ▪ String Class walkthrough
Access Control

❖ **Access modifiers** for members:

▪ **public**: accessible to *all* parts of the program

▪ **private**: accessible to the member functions of the class
  • Private to *class*, not object instances

▪ **protected**: accessible to member functions of the class and any *derived* classes (subclasses – more to come, later)

❖ **Reminders**:

▪ Access modifiers apply to *all* members that follow until another access modifier is reached
struct vs. class

❖ In C, a `struct` can only contain data fields
  ▪ Has no methods and all fields are always accessible
  ▪ In `struct foo`, the `foo` is a “struct tag”, not an ordinary data type

❖ In C++, `struct` and `class` are (nearly) the same!
  ▪ Both define a new type (the `struct` or `class` name)
  ▪ Both can have methods and member visibility (public/private/protected)
  ▪ Only real (minor) difference: members are default `public` in a `struct` and default `private` in a `class`
struct vs. class

❖ Common style/usage convention:
  ▪ Use struct for simple bundles of data
    • Convenience constructors can make sense though
  ▪ Use class for abstractions with data + functions
Nonmember Functions

- "Nonmember functions" are just normal functions that happen to use some class
  - Called like a regular function instead of as a member of a class object instance
  - These do not have access to the class’ private members

- Useful nonmember functions often included as part of the interface to a class
  - Declaration goes in header file, but outside of class definition
    - But inside the same namespace as the class, if it has one
Nonmember Functions

❖ “Nonmember functions” are just normal functions that happen to use some class
  ▪ Called like a regular function instead of as a member of a class object instance
  ▪ These do not have access to the class’ private members
  ▪ Often included as part of the interface to a class

```cpp
class Complex { ... }

void ReadFromStream(std::istream& in, Complex& a);
```

```cpp
void ReadFromStream(std::istream& in, Complex& a) {
    double r;
    in >> r
    a.set_real(r);
    // ... etc ...
}
```
Nonmember Operators

- Operators can be member methods or non-member functions
  - Eg, overloaded operators (operator+, etc.), stream I/O (operator<<), etc. …
Review: Operator Overloading

- Can overload operators using **member functions**
  - Restriction: left-hand side argument must be a class you are implementing
    ```cpp
    Complex& Complex::operator+=(const Complex &a) { ... }
    ```

- 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
    ```cpp
    Complex operator+(const Complex &a, const Complex &b) { ... }
    ```
friend Nonmember Functions

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

```cpp
class Complex {
  ...
  friend std::istream& operator>>(std::istream& in, Complex& a);
  ...
};  // class Complex
```
When to use Nonmember and friend

❖ Member functions:
  ▪ Operators that modify the object being called on
    • Assignment operator (operator=)
  ▪ “Core” non-operator functionality that is part of the class interface

❖ Nonmember functions:
  ▪ Used for commutative operators
    • *e.g.*, so v1 + v2 is invoked as operator+ (v1, v2) instead of v1.operator+ (v2)
  ▪ If operating on two types and the class is on the right-hand side
    • *e.g.*, cin >> complex;
  ▪ Returning a “new” object, not modifying an existing one
  ▪ Only grant friend permission if you NEED to
Lecture Outline

❖ Class Details
  ▪ Rule of Three / Making Copies
  ▪ Access Controls and Friends
  ▪ Namespaces

❖ Using the Heap
  ▪ `new` / `delete` / `delete[]`
  ▪ String Class walkthrough
Namespaces

- Each namespace is a separate scope
  - Useful for avoiding symbol collisions

Namespace definition:

- `namespace name {  
  // declarations go here  
}  

- Creates a new namespace name if it did not exist, otherwise *adds to the existing namespace* (!)
  - This means that components (classes, functions, etc.) of a namespace can be defined in multiple source files
    - All of the standard library is in namespace `std` but it has many source files
Classes vs. Namespaces

❖ They seem somewhat similar, but classes are not namespaces:

- There are no instances/objects of a namespace; a namespace is just a group of logically-related things (classes, functions, etc.)

- To access a member of a namespace, you must use the fully qualified name (*i.e.* `nsp_name::member`)
  - Unless you are using that namespace or individual member item
  - You only used the fully qualified name of a class member when you are defining it outside of the scope of the class definition
Lecture Outline

❖ Class Details
  ▪ Rule of Three / Making Copies
  ▪ Access Controls and Friends
  ▪ Namespaces

❖ Using the Heap
  ▪ `new`/`delete`/`delete[]`
  ▪ String Class walkthrough
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, especially with function overloading (`f(int) vs f(T*)`; 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`)
    - Will execute appropriate constructor as part of object allocate/create
  - 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 Example

```cpp
#include "Point.h"
using namespace std;

... // 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 0;
}
```
new/delete Example

g++ -Wall -g -std=c++17 -o heappoint \ 
    heappoint.cc Point.cc
valgrind ./heappoint

==3167334== Memcheck, a memory error detector
==3167334== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3167334== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==3167334== Command: ./heappoint
==3167334== Calling Point constructor
  x's x_ coordinate: 1
  distance between x and self: 0
  y: 0x4daa110, *y: 3
==3167334== HEAP SUMMARY:
==3167334== in use at exit: 0 bytes in 0 blocks
==3167334== total heap usage: 4 allocs, 4 frees, 73,740 bytes allocated
==3167334== All heap blocks were freed -- no leaks are possible
==3167334== For lists of detected and suppressed errors, rerun with: -s
==3167334== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
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)
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)

```cpp
#include "Point.h"
using namespace std;

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

    int stack_arr[10];
    int* heap_arr = new int[10];

    int* heap_init_arr = new int[10](); // uncommon usage
    int* heap_init_error = new int[10](12); // bad syntax
    int* heap_init_error = new int[10]{12}; // C++11 allows
                                           // (uncommon)

    delete heap_int;                  // ok
    delete heap_init_int;            // ok
    delete heap_arr;                 // error - must be delete[]
    delete[] heap_init_arr;          // ok

    return 0;
}
```
# Arrays Example (class objects)

```cpp
#include "Point.h"
using namespace std;

int main() {
    ...
    Point stack_point(1, 2);
    Point* heap_point = new Point(1, 2);

    Point* err_pt_arr = new Point[10]; // bug-no Point() ctr
    Point* err2_pt_arr = new Point[10](1, 2); // bad syntax
    Point* err2_pt_arr = new Point[10]{1, 2}; // C++11 allows
    ... // (uncommon)

    delete heap_point;
    ...
    return 0;
}
```
## malloc vs. new

<table>
<thead>
<tr>
<th></th>
<th>malloc()</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What is it?</strong></td>
<td>a function</td>
<td>an operator or keyword</td>
</tr>
<tr>
<td><strong>How often used (in C)?</strong></td>
<td>often</td>
<td>never</td>
</tr>
<tr>
<td><strong>How often used (in C++)?</strong></td>
<td>rarely</td>
<td>often</td>
</tr>
<tr>
<td><strong>Typed</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>a void* (should be cast)</td>
<td>appropriate pointer type (doesn’t need a cast)</td>
</tr>
<tr>
<td><strong>When out of memory</strong></td>
<td>returns NULL</td>
<td>throws an exception</td>
</tr>
<tr>
<td><strong>Deallocating</strong></td>
<td>free()</td>
<td>delete or delete[]</td>
</tr>
</tbody>
</table>
Heap Member Example

- Let’s build a class to simulate some of the functionality of the C++ string
  - Internal representation: c-string to hold characters
- What might we want to implement in the class?
Lecture Outline

❖ Class Details
  ▪ Rule of Three / Making Copies
  ▪ Access Controls and Friends
  ▪ Namespaces

❖ Using the Heap
  ▪ `new/delete/delete[]`
  ▪ `String` Class walkthrough
Str Example Walkthrough

See:
Str.h
Str.cc
strtest.cc

- Look carefully at assignment operator=
  - self-assignment test is especially important here
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!)