C++ Class Details, Heap
CSE 333 Winter 2019

Instructor: Hal Perkins

Teaching Assistants:
Alexey Beall
David Porter
Yifan Xu
Renshu Gu
Forrest Timour
Sujie Zhou
Harshita Neti
Soumya Vasisht
Yifan Xu
Administrivia

- Yet another exercise released today, due Wed. Mon.
  - Rework exercise 10 but with dynamic memory this time
    - Fine to use ex10 solution as a starting point for ex11

- Should be due before Fri. lecture, but can’t because...

- ...Homework 2 due Thursday (tomorrow!) night
  - File system crawler, indexer, and search engine

- Not clear what to do about exercise based on Fri. lecture
  - 2nd exercise due Monday? (🤔 but maybe need to catch up?)
Lecture Outline

- **Class Details**
  - Filling in some gaps from last time

- **Using the Heap**
  - `new/delete/delete[]`
Rule of Three

- If you define any of:
  1) Destructor
  2) Copy Constructor
  3) Assignment (\texttt{operator=})

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

```cpp
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 "="
    ...
```
Dealing with the Insanity

- C++ style guide tip:
  - If possible, disable the copy constructor and assignment operator by declaring as private and not defining them (pre-C++11)

```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)
```
Disabling in C++11

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

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

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

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

```cpp
Point x(1, 2); // OK
Point y(3, 4); // OK
x.CopyFrom(y); // OK
```
**struct vs. class**

- In C, a `struct` can only contain data fields
  - Has no methods and all fields are always accessible

- In C++, `struct` and `class` are (nearly) the same!
  - 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`

- Common style/usage convention:
  - Use `struct` for simple bundles of data
  - Use `class` for abstractions with data + functions
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
  - If no access modifier is specified, *struct* members default to `public` and *class* members default to `private`
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
    - This gets a little weird when we talk about operators...
  - 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
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

std::istream& operator>>(std::istream& in, Complex& a) {
    ...
}
```
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 \textit{adds to the existing namespace} (!)
    - This means that components (classes, functions, etc.) of a namespace can be defined in multiple 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
    - 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
  - Filling in some gaps from last time

- Using the Heap
  - `new / delete / delete[]`
C++11 `nullptr`

- C and C++ have long used `nullptr` 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 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;
}
```

```cpp
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
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
#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
    ...

    delete heap_int; //
    delete heap_init_int; //
    delete heap_arr; //
    delete[] heap_init_arr; //

    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];  // no Point() ctor
    Point* err2_pt_arr = new Point[10](1,2);  // bad syntax
    ...
    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>Allocated memory for</strong></td>
<td>anything</td>
<td>arrays, structs, objects, primitives</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td><code>void*</code> (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 <code>NULL</code></td>
<td>throws an exception</td>
</tr>
<tr>
<td><strong>Deallocating</strong></td>
<td><code>free()</code></td>
<td><code>delete</code> or <code>delete[]</code></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?
#include <iostream>
using namespace std;

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 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!)