

C++ Class Details, Heap

CSE 333 Autumn 2021

Instructor: Chris Thachuk

Teaching Assistants:

Arpad (John) Depaszthory

Ian Hsiao

Logan Gnanapragasam

Mengqi (Frank) Chen

Angela Xu

Khang Vinh Phan

Maruchi Kim

Cosmo Wang

Administrivia

- ❖ Exercise 9 due Wednesday
- ❖ Homework 2 due Thursday
 - File system crawler, indexer, and search engine
 - Don't forget to clone your repo to double-/triple-/quadruple-check compilation on attu! (Checkout code with **hw2-final** tag.)
 - Don't modify the header files!

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 (`operator=`)
- ❖ Then you should normally 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 "="  
    ...  
};
```

Dealing with the Insanity (C++11)

❖ C++ style guide tip:

- **Disabling** the copy constructor and assignment operator can avoid confusion from implicit invocation and excessive copying

Point_2011.h

```
class Point {
public:
    Point(const int x, const int y) : x_(x), y_(y) { } // ctor
    ...
    Point(const Point& copyme) = delete; // declare cctor and "="
    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)
```

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 ctor
    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
```

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 direct 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 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 “getter” public functions

Complex.h

```
class Complex {  
    ...  
    friend std::istream& operator>>(std::istream& in, Complex& a);  
    ...  
}; // class Complex
```

```
std::istream& operator>>(std::istream& in, Complex& a) {  
    ...  
}
```

Complex.cc 10

When to use Nonmember and `friend`



❖ Member functions:

Good rules of thumb, be sure to think about these carefully

- Operators that modify the object being called on
 - Assignment operator (`operator=`)
- “Core” non-operator functionality that is part of the class interface

❖ Nonmember function

There is more to C++ object design that we don't have time to get to ☹

- Used for commutative operators
 - 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 it



pollev.com/cse333

If we wanted to overload operator== to compare two points, what type of function should it be?

- ❖ Reminder that Point has getters and a setter

- A. **non-friend + member**
- B. **friend + member**
- C. **non-friend + non-member**
- D. **friend + non-member**
- E. **We're lost...**

Namespaces

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

- ❖ Namespace definition:

- ```
namespace name {
 // declarations go here
} // namespace name
```

- Doesn't end with a semi-colon and doesn't add to the indentation of its contents
- Creates a new namespace name if it did not exist, otherwise *adds to the existing namespace (!)*
  - This means that components (*e.g.* classes, functions) of a namespace can be defined in multiple source files

# Classes vs. Namespaces

- ❖ They seems 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

# Complex Example Walkthrough

See:

`Complex.h`

`Complex.cc`

`testcomplex.cc`

# 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 `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)`) or (e.g. `new int(333)`)
- If no initialization specified, it will use default constructor for objects, garbage 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><br><i>(should be cast)</i> | appropriate pointer type<br><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/cse333](http://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...**

```
class Foo{
public:
 Foo(int val) { Init(val); }
 ~Foo() { delete foo_ptr_; }
 void Init(int val) {
 foo_ptr_ = new int(val);
 }
 Foo& operator=(const Foo& rhs) {
 delete foo_ptr_;
 Init(*(rhs.foo_ptr_));
 return *this;
 }
private:
 int* foo_ptr_;
};

void bar() {
 Foo a(10);
 Foo b(20);
 a = a;
}
```

# Dynamically Memory & Rule of three

- ❖ What will happen when we invoke `bar()` after modifying it?
  - If there is an error, how would you fix it?

```
class Foo{
public:
 Foo(int val) { Init(val); }
 ~Foo() { delete foo_ptr_; }
 void Foo::Init(int val) {
 foo_ptr_ = new int(val);
 }
 Foo& operator=(const Foo& rhs) {
 if (&rhs != this) {
 delete foo_ptr_;
 Init(*(rhs.foo_ptr_));
 }
 return *this;
 }
private:
 int* foo_ptr_;
};

void bar() {
 Foo a(10);
 Foo b = a;
}
```

# Heap Member (Extra Exercise)

- ❖ 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?

# Str Class Walkthrough

Str.h

```
#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::append

❖ 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) {

}
}
```

# 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!)