CSE 374 Programming Concepts and Tools

Lecture 19 - C++ Class Details

This Week

C++ Classes

Today: C++ Class Details

C++ Templates, STL

Today

More advice on writing classes in C++

- “Rule of Three”
- Non-member functions
- Access and Visibility
- Overloading
- `friend`

Using the Heap

- `new` and `delete`
- Dynamically allocating arrays

Review: Classes

Class definition syntax (in a `.h` file):

```
class ClassName {
 public:
  // public member definitions & declarations go here

 private:
  // private member definitions & declarations go here
};  // class Name
```

- Members can be functions (methods) or data (variables)

Class member function definition syntax (in a `.cc` file):

```
retType ClassName::MethodName(type1 param1, …, typeN
paramN) {
  // body statements
}
```

Default Class Members

In C++, we can define constructors (ctors), copy constructors (cctors), assignment operators (overloading =), and destructors (dtors) for a class

If we don't define one of the above for our class, C++ will create one for us

- In the case of cctors and assignment, it does a *shallow copy*

Destructors automatically tear down the class upon function return/program end

- Closes open files, deallocates memory, etc.

usepoint.cc

```
int main(int argc, char** argv) {
  Point p1(1, 2);
  Point p2(4, 6);

  cout << "p1 is: (" << p1.get_x() << ", ";
  cout << p1.get_y() << ")" << endl;

  cout << "p2 is: (" << p2.get_x() << ", ";
  cout << p2.get_y() << ")" << endl;

  cout << "dist : " << p1.Distance(p2) << endl;
  return EXIT_SUCCESS;
}
```

Rule of Three

If you define any of:

- Destructor
- Copy Constructor
- 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 "="
  // more stuff...
```

Non-member Functions

"Non-member functions" are just normal functions that happen to take a class as a parameter

- Called like a regular function instead of as a member of a class object instance (latter uses the dot operator)

These do *not* have access to the class' private members

- Can access fields via getters (if they are there)

Useful **non-member functions** often included as part of **interface** of a class

- Declaration goes in header file, but **outside** of class definition
- Operators that are commutative should typically be non-members (non-commutative things can be non members too)

Example

Member function

```
double Point::distance(Point&)
pt1.distance(pt2);

float Vector::operator*(Vector&)
vec1 * vec2;
```

Non-member function

```
double distance(Point&, Point&)
distance(pt1, pt2);

float operator*(Vector&, Vector&)
vec1 * vec2;
```

Access Control

Access modifiers for members:

- `public`: accessible to *all* parts of the program
- `private`: accessible to the member functions of the class
- `protected`: accessible to member functions of the class and any *derived* classes (subclasses – we'll talk about this next week)

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`

Operator Overloading

Can overload operators using **member functions**

- Restriction: left-hand side argument **must be a class you've made**

```
Complex& operator+=(const Complex& a) { ... }
```

Can overload operators using **non-member functions**

- No restriction on arguments (can specify any two)
  - **Our only option** when the left-hand side is a class you did not implement, like `ostream` or `istream`.
- But no access to private data members 😔

```
Complex operator+(const Complex& a, const Complex& b) { ... }
```

`friend` non-member Functions

A class can give a **non-member** 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`s are usually unnecessary if the class has the right "getter" functions

Complex.h

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

Complex.cc

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

Note: no Complex::

Demo: `Complex` Walkthrough

When to use non-member 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, and if you are not modifying

Polling Time!

Please answer in the LP19 assignment on Gradescope!

If we wanted to overload operator== to compare two points (based on their `x` and `y` coordinates), what type of function should it be?

Hints: Reminder that `Point` has getter functions for `x` and `y`. Does this operator need to modify either of the points?

- non-friend + member
- friend + member
- non-friend + non-member
- friend + non-member

Point.h

```
class Point {
 public:
  Point(const int x, const int y);
  int get_x() const { return x_; }
  int get_y() const { return y_; }
  double Distance(const Point& p) const;
  void SetLocation(const int x, const int y);

 private:
  int x_;
  int y_;
};  // class Point
```

Poll Question Explained

If we wanted to overload operator== to compare two points (based on their x and y coordinates), what type of function should it be?

A. non-friend + member
B. friend + member
C. non-friend + non-member
D. friend + non-member

We have getters to access the values of both points, and we aren't modifying either point.

```
bool operator==(const Point& a, const Point& b) {
    return a.get_x() == b.get_x() && a.get_y() == b.get_y();
}
```

Namespaces

Each namespace is a separate scope

- Useful for avoiding symbol collisions!

```
LL::Iterator
HT::Iterator
```

Same name, but different namespace

Namespace definition:

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

using namespace LL;
using namespace HT;

- Doesn't end with a semicolon 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.* `namespace_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

Using the Heap

Null in C++?

C and C++ have long used `NULL` as a pointer value that references **nothing**

- What is `NULL`, really? (reference page)

C++11 introduced a new literal for this: **`nullptr`** (reference page)

- New reserved keyword
- Interchangeable with `NULL` for all practical purposes, but it has type `T*` for any/every type `T`, and is not an integer value
  - Still can convert to/from integer `0` for tests, assignment, etc.
- Advice: prefer **`nullptr`** in C++11 or later code
  - Though `NULL` will also be around for a long, long time 😓

new/delete

To allocate on the heap using C++, instead of **malloc()**, you use the **new keyword** (`#include <new>`)

- 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, instead of **free()**, use the **delete keyword** (included in `<new>`)

- **Don't mix and match!**
  - *Never* **free()** something allocated with `new`
  - *Never* `delete` something allocated with **malloc()**
  - ⚠️ **Be careful** if you're using **legacy C code** with 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(374)`)
- If no initialization specified, it will use default constructor for objects, garbage for primitives (integer, float, character, boolean, double)
  - You don't need to check whether `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"
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 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.cc

```
#include "Point.h"

int main() {
  int stack_int;                        // stack (garbage)
  int* heap_int = new int;              // heap (garbage)
  int* heap_int_init = new int(12);     // heap (12)

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

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

  // more stuff...

  delete heap_int;                  // ok
  delete heap_int_init;             // ok
  delete heap_arr;                  // BAD
  delete[] heap_arr_init_val;       // ok

  return EXIT_SUCCESS;
}
```

Arrays Example (primitives: int)

arrays.cc

```
#include "Point.h"

int main() {
  // primitive stuff...

  Point stack_pt(1, 2);                // stack 2-arg constructor
  Point* heap_pt = new Point(1, 2);    // heap 2-arg constructor

  Point* heap_pt_arr_err = new Point[2]; // heap default ctor?
                                         // fails cause no default ctor

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

  delete heap_pt;
  delete[] heap_pt_arr_init_lst;

  return EXIT_SUCCESS;
}
```

Arrays Example (class objects)

`malloc` vs. `new`

	`malloc()`	`new`
What is it?	a function	an operator / 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 `void*` (*should be cast*)	appropriate pointer type (*doesn't need a cast*)
When out of memory	returns `NULL`	throws an exception
Deallocating	**`free`**`()`	`delete` or `delete[]`

Polling Time!

Please answer in the LP19 assignment on Gradescope!

This code has a memory error!

How should we fix it?

- Add "`delete ptr2`"
- Add "`delete ref3`"
- Remove "`delete ptr1`"
- Move "`delete ptr1`" to right before `main` returns

```
void print_int_ptr(int* ptr2) {
  cout << *ptr2 << endl;
  // delete ptr2;
}

void print_int_ref(int& ref3) {
  cout << ref3 << endl;
  // delete ref3;
}

int main() {
  int* ptr1 = new int;
  *ptr1 = 42;
  print_int_ptr(ptr1);
  delete ptr1;
  print_int_ref(*ptr1);
  return EXIT_SUCCESS;
}
```

Polling Question Explained

This code has a memory error!

How should we fix it?

- Add "`delete ptr2`"
- Add "`delete ref3`"
- Remove "`delete ptr1`"
- Move "`delete ptr1`" to right before `main` returns

```
void print_int_ptr(int* ptr2) {
  cout << *ptr2 << endl;
  // delete ptr2;
}

void print_int_ref(int& ref3) {
  cout << ref3 << endl;
  // delete ref3;
}

int main() {
  int* ptr1 = new int;
  *ptr1 = 42;
  print_int_ptr(ptr1);
  print_int_ref(*ptr1);
  delete ptr1; // moved here
  return EXIT_SUCCESS;
}
```

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?

Str Class Walkthrough

Str.h

```
#include <iostream>
using namespace std;

class Str {
 public:
  Str();                // default ctor: create empty string
  Str(const char* s);   // c-string ctor: create Str from c-string s
  Str(const Str& s);    // copy ctor: initialize this to be a copy of s
  ~Str();               // dtor

  int length() const;  // return length of string
  char* c_str() const; // return a copy of st_ on heap
  void append(const Str& s); // append contents of s to the end of this
string

  Str& operator=(const Str& s);  // string assignment

  friend std::ostream& operator<<(std::ostream& out, const Str& s); // output

 private:
  char* st_;  // c-string on heap (terminated by '\0')
};  // class Str
```

Demo: `Str` Example Walkthrough

Assignment Reminders

HW5 due this Friday

- Implement the library you tested in HW4

HW6 (C++) will be released this week and due before Exam 3

- The only C++ programming homework
- Should be much shorter than HW5

Exam 3 will be held on the last day of class from 10:50-12:50 in PCAR 291

- Paper exam, 1 page of handwritten notes allowed
- C/C++ coding, covers all content in the course except the final lecture
- Past exams available on the course website

Extra Exercises 

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!)

Extra Exercise #2: Desugaring

Create a new file `testcomplexexplicit.cc` that has the same behavior as `testcomplex.cc`, but explicitly calls the appropriate overloaded operators, e.g.,

- for addition (+), use `operator+(<arg1>, <arg2>)`
- for equals (=), use `<object1>.operator=(<arg>)`
- etc.

You should also do this with C++ standard library operators (e.g., from `<iostream>`)

Extra Exercise #3: More Complex-ity

Alter the `Complex.h` and `Complex.cc` files to add additional typical complex number operations, such as taking the conjugate or the magnitude, or perhaps taking the square of a complex number. Test them out in `testcomplex.cc`

Extra Exercise #4

In the Complex example, was it actually necessary to make `operator>>` a friend non-member function?

If it wasn't necessary, implement it as a plain non-member function *without* friend access.

If it is necessary, explain why.

Extra Exercise #5

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

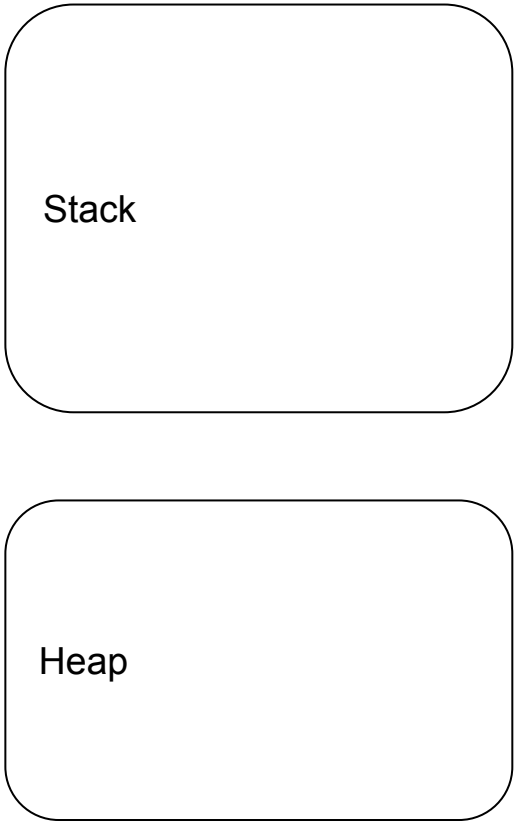
```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

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;
}
```

Dynamically Allocated Class Members



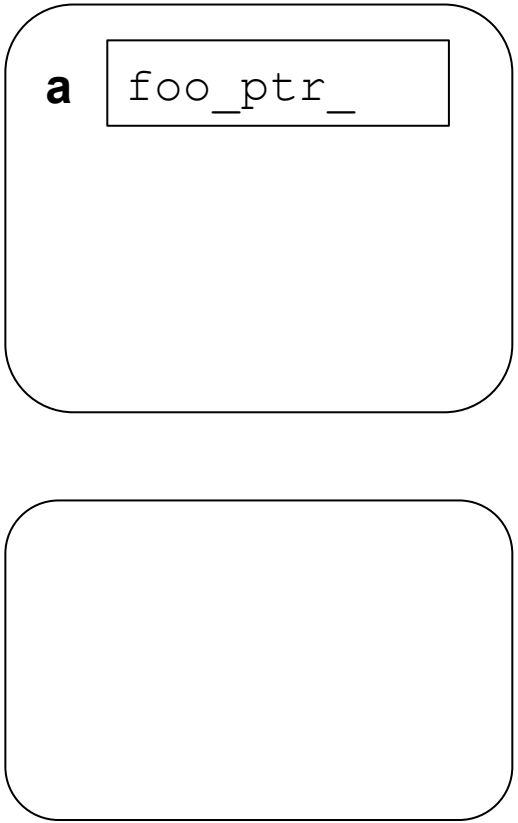

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

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;
}
```

Dynamically Allocated Class Members



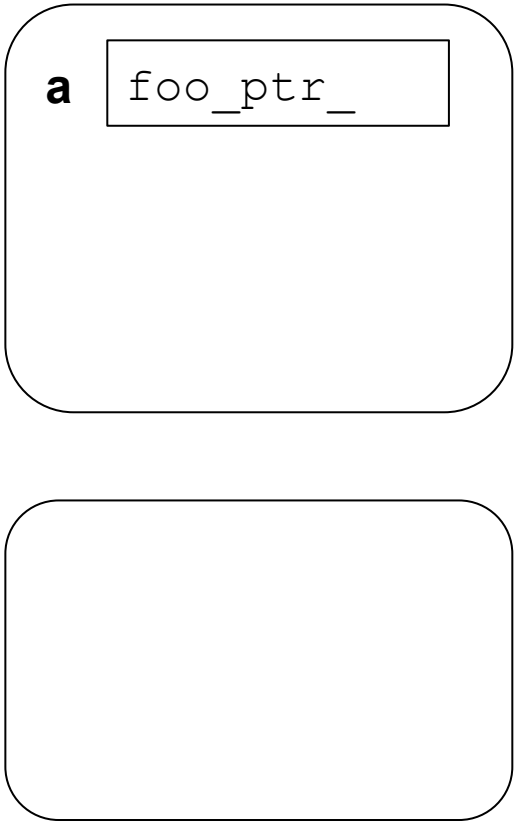

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

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;
}
```

Dynamically Allocated Class Members



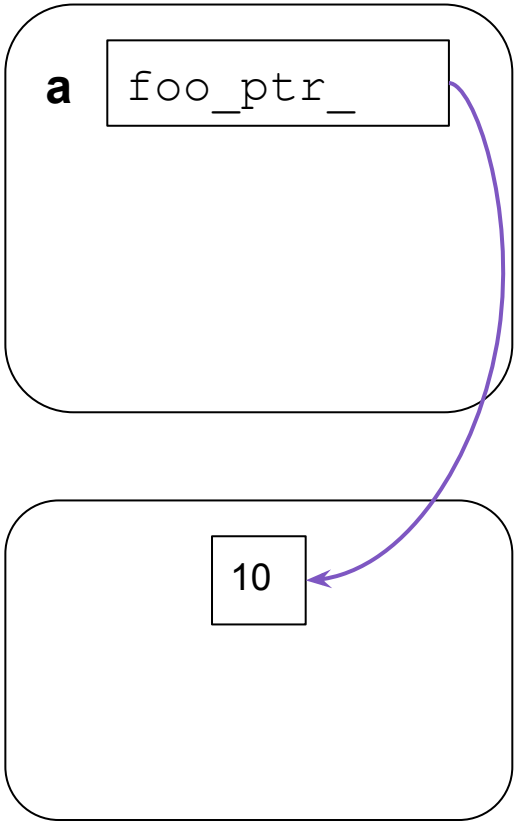

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

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;
}
```

Dynamically Allocated Class Members



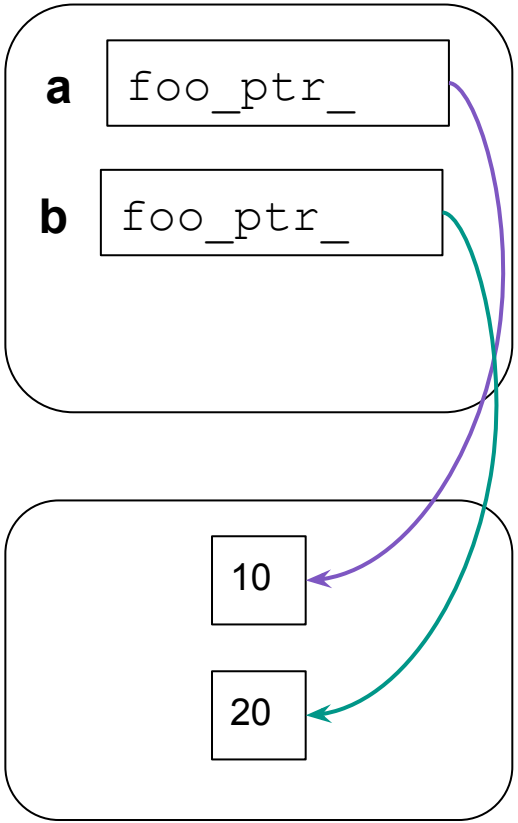

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

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;
}
```

Dynamically Allocated Class Members



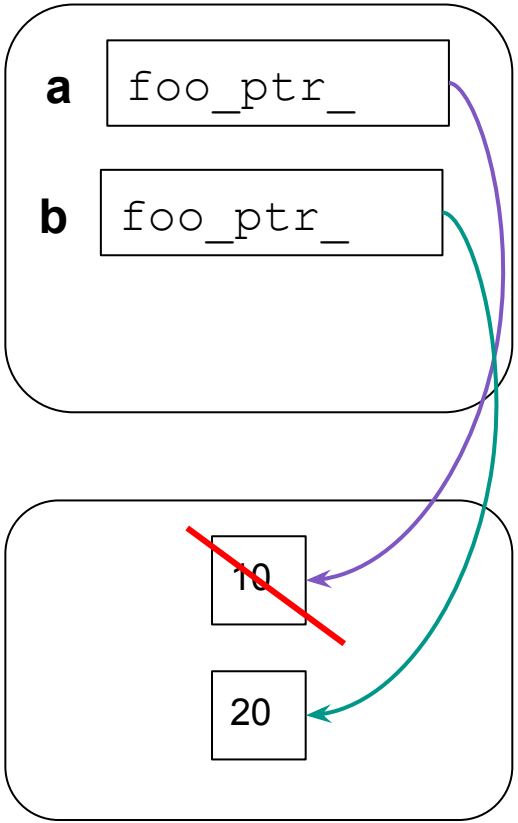

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

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;
}
```

Dynamically Allocated Class Members



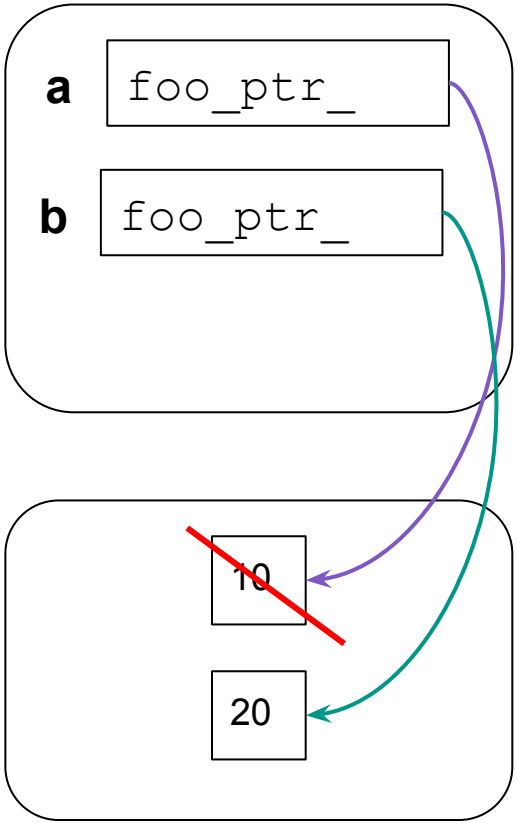

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

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;
}
```

Dynamically Allocated Class Members




```
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;
}
```

```
if(&rhs!=this){

}
```