## C++ Class Details, Heap CSE 333 Winter 2021

**Instructor:** John Zahorjan

#### **Teaching Assistants:**

Matthew Arnold	Nonthakit Chaiwong	Jacob Cohen
Elizabeth Haker	Henry Hung	Chase Lee
Leo Liao	Tim Mandzyuk	Benjamin Shmidt
Guramrit Singh		

# **Lecture Outline**

- **\*** Class Details
  - Filling in some gaps from last time
- Using the Heap
  - new/delete/delete[]

#### **Synthesized Constructors / Destructor / Assignment**

 You can explicitly indicate you want the compiler to synthesize them

- Why?
  - Communicate to another programmer that this really is your intention
  - Cause constructor synthesis even when you have defined another constructor

### **Synthesized Constructors / Assignment**

When you intend that they shouldn't be used, make sure they're not!

```
class Farm {
  public:
    Farm() = delete;
    Point(const Farm& copyme) = delete;
    Farm& operator=(const Point& rhs) = delete;
    private:
    Address *p_address; // some new'ed memory
}; // class Point
```

✤ Ridiculous side note:

Yes, you can delete the destructor, ~Point(), but then your code compiles only if you create a Point only using new and create a memory leak by never deleting a Point

### 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
- Common style/usage convention:
  - Use struct for simple bundles of data
    - Convenience constructors can make sense though
  - 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
- Rules:
  - Access modifiers apply to *all* members that follow until another access modifier is reached
  - If no access modifier is specified, <then there's some rule>
    - Never don't specify access modifiers

## **Operator Overloading**

C++ identifies operators syntactically

```
6 + x
--my_obj
my_obj * your_obj
this_obj = that_obj + the_other_obj
```

- Okay, you've found the operators. Now what?
  - The type(s) of the operand(s) determine what "method" the operator is

# Why Would You Customize Operators?

- Assignment is special in that the compiler has a default meaning for =
  - Customize when that meanings is wrong for your application
- What about other operators
  - +, -, \*, /, &, (), <<, >>, ..., ,, etc.
- Compiler has default meanings for those as well
  - at least for some types of operands
- In Java, string\_1 + string\_2 is built into the language, because class string is part of the language
- In C++, string\_1 + string\_2 is created by library programmers who implemented the String class using a generally available feature of the language

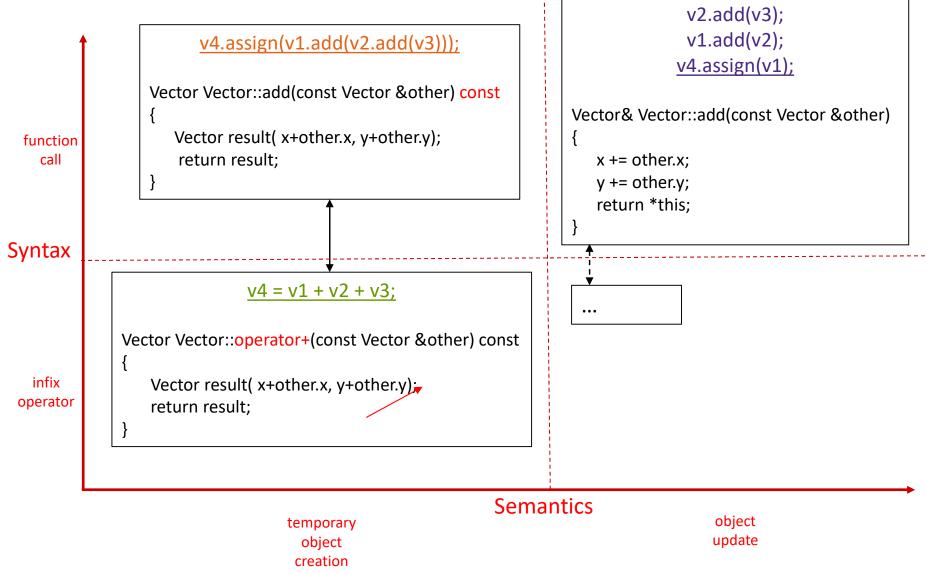
# Why Customize (Overload) Operators?

- There's nothing you can compute with overloaded operators you can't compute without them
- But sometimes you prefer the syntax of operators to function call syntax
  - What syntax do you want (your and your clients) to use?
  - What syntax is most likely to be used correctly / not to be mis-used?

```
Vector v1, v2, v3, v4;
```

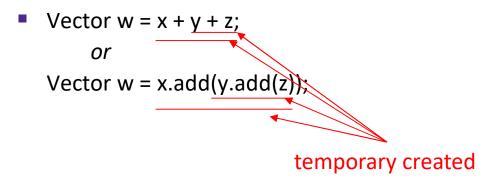
*There are <u>syntax distinctions</u>. There are a <u>side-effects distinctions</u>. There are a performance distinctions*.

# What Are the Distinctions?



# **Semantic Choice / Temporaries**

 Being able to create <u>compiler managed</u> temporaries often leads to simpler, cleaner, less drive-you-crazy code



- For the compiler to deal with destruction, the temporaries cannot be pointers or references, they must be objects
- Object creation/destruction can be expensive

# **Implementing Operator Overloading**

- Can overload operators using member functions
  - For binary operators, look at the class of the argument on the left my\_obj + 6 my\_obj \* your\_obj
- Can overload operators using nonmember functions

```
MyClass& operator+(MyClass& o, int x)
{
    o.set(o.get()+x);
    return o;
}
```

## friend Functions

- A class can give a nonmember function (or class) access to its nonpublic members by declaring it as a friend
  - 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

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

- It is common to overload ostream insertion (<<) and istream extraction (>>)
  - std::cout << "Point A: " << pointA << " Point B: " << pointB << std::endl</p>
- This isn't the only way to get this effect in C++ though...

# **Lecture Outline**

- Class Details
  - Filling in some gaps from last time
- **Solution** Using the Heap
  - new/delete/delete[]

### nullptr (as of C++11)

- In C we used NULL to be a special pointer value
  - Used to indicate errors
  - Dereferencing NULL is a run-time error
  - NULL is 0 as an int, false as a Boolean
  - NULL is typically a void\*
- In C++, we have nullptr
  - It's a pointer type
  - It will implicitly convert to every other pointer type
  - It will resist becoming an integer

# new/delete

- To allocate on the heap in 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!
    - <u>Never</u> free () something allocated with new
    - <u>Never</u> delete something allocated with malloc()

# new/delete Example

```
#include "Point.h"
int main() {
    Point* x = new Point(1, 2);
    int* y = new int(3);
    std::cout << "Point: " << *x << std:: endl;
    std:: cout << "int: " << *y << std:: endl;
    delete x;
    delete y;
    return 0;
}</pre>
```

# **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 <u>undefined</u> behavior

# **Arrays Example (primitive)**

```
#include "Point.h"
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; // ok
  delete heap init int; // ok
 delete heap_arr; // error - must be delete[]
delete[] heap_init_arr; // ok
 return 0;
```

## **Arrays Example (class objects)**

```
#include "Point.h"
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
  . . .
  delete heap point;
  . . .
  return 0;
```

### malloc vs. new

	malloc()	new
What is it?	a function	an operator or keyword
How often used (in C)?	often	never
How often used (in C++)?	rarely	often
Allocates	Memory bytes	arrays, structs, objects, primitives
Calls Constructor	No	Yes
Returns	a void* (should be cast)	appropriate pointer type (doesn't need a cast)
When out of memory	returns NULL	throws an exception
Deallocation	free()	delete <b>or</b> delete[]