CSC 211: Computer Programming

Copy Constructors, Operator Overloading, Dynamic Memory

Michael Conti

Department of Computer Science and Statistics
University of Rhode Island

Summer 2024

Administrative notes

More on constructors …

- So far …
  - default constructors, overloaded constructors
- C++ also defines **copy constructors**
  - used to create an object as a copy of an existing object
  - if you don't define your own, C++ will synthesize one copy constructor for you

```
Point2D obj1;               // default constructor
Point2D obj2(4.5, 3.2);     // overloaded constructor
Point2D obj3(obj2);         // copy constructor
Point2D obj4 = obj3;        // copy constructor
```


When are copy constructors invoked?

```
Point2D myfunc(Point2D obj) {
    Point2D newobj;
    // ...
    return newobj;
}

int main () {
    // copy constructor is invoked when an object is initialized from
    // another object of the same type
    Point2D obj2(4.5, 3.2);     // overloaded constructor
    Point2D obj3(obj2);         // copy constructor
    Point2D obj4 = obj3;        // copy constructor

    // copy constructor is invoked when a non-reference object is
    // passed to a function (to initialize parameter)
    myfunc(obj4);               // copy constructor

    // copy constructor is invoked when a non-reference object is
    // returned from a function
    Point2D obj5 = myfunc(obj2);
}
```

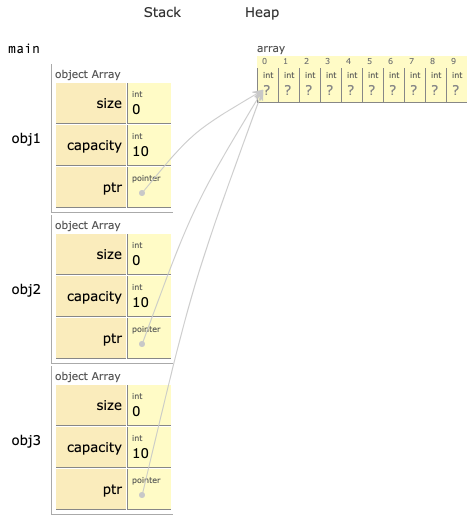
Shallow vs deep copies

- Synthesized copy constructors perform **shallow copies**
  - a shallow copy is a byte-to-byte copy of all data members (works fine most of the cases, except when pointers are used)

```
Point2D::Point2D(const Point2D& obj) {
    x = obj.x;
    y = obj.y;
    // ...
}
```

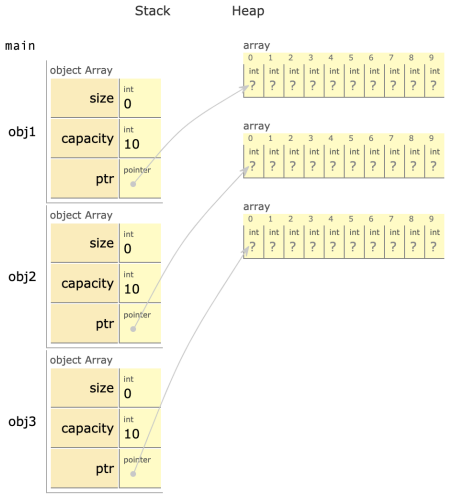
- Sometimes a **deep copy** is necessary (can handle more complex objects)
  - must define your own copy constructor


```
class Array {
    public:
        Array(int cap);
        ~Array();
    private:
        int size;
        int capacity;
        int *ptr;
};

Array::Array(int cap) {
    size = 0;
    capacity = cap;
    ptr = new int[cap];
}

Array::~Array() {
    delete [] ptr;
}

int main () {
    Array obj1(10);
    Array obj2(obj1);
    Array obj3 = obj2;
}
```



shallow copies


```
Array::Array(int cap) {
    size = 0;
    capacity = cap;
    ptr = new int[cap];
}

Array::Array(Array& obj) {
    size = obj.size;
    capacity = obj.capacity;
    ptr = new int[capacity];
    for (int i = 0 ; i < size ; i++) {
        ptr[i] = obj.ptr[i];
    }
}

Array::~Array() {
    delete [] ptr;
}

int main () {
    Array obj1(10);
    Array obj2(obj1);
    Array obj3 = obj2;
}
```



deep copies

The **assignment** operator =

- Assignment is not construction
- The assignment operator '=' assigns an object to an existing object (already constructed)

```
Point2D obj1;               // default constructor
Point2D obj2(4.5, 3.2);     // overloaded constructor
Point2D obj3(obj2);         // copy constructor
Point2D obj4 = obj3;        // copy constructor
obj1 = obj4;                // assignment operator
```

- If you don't define your own, C++ will synthesize one assignment operator for you (performs **shallow copy**)

The `this` pointer

- Pointer accessible only within member functions of a class
  - it points to the object for which the member function is called
  - **static member functions** do not have this pointer

```
void Date::set_year(int y) {
    // statements below are equivalent
    year = y;
    this->year = y;
    (*this).year = y;
}
```


Overloading Operators

How to overload the '=' operator?

```
Point2D& Point2D::operator=(const Point2D &obj) {
    // always check against self-assignment
    // especially when performing deep copies
    if (this != &obj) {
        this->x = obj.x;
        this->y = obj.y;
    }
    // always return *this, necessary for
    // cascade assignments (a = b = c)
    return *this;
}
```

Modify the self object reference and return it

can perform either shallow or deep copies

How many copy constructor calls?

```
Point2D myfunc(const Point2D& obj) {
    Point2D newobj;
    newobj = obj;
    // ...
    return newobj;
}

int main () {
    Point2D obj2(4.3, 1.1);
    Point2D obj3(obj2);
    Point2D obj4 = myfunc(obj3);
    Point2D obj5;
    obj5 = obj4 = obj2;
}
```

Dynamic Memory Allocation

The `new` and `delete` operators

- Used to create and destroy variables, objects, or arrays while the program is running
- Memory allocated with the `new` operator does **NOT** use the **call stack**
  - new allocations go into the **heap** (area of memory reserved for dynamic memory allocation)
- Programmer **must** destroy all variables, objects, and arrays created dynamically
  - using the `delete` operator

Heap vs Stack

- Dynamic (heap) memory
  - allocated during run time
  - exact sizes or amounts don't need to be known
  - must use pointers
  - alternative to local stack memory
- Static (stack) memory
  - exact size and type of memory must be known at compile time.
  - local variables are allocated automatically when a function is called and they are deallocated automatically when the function exits.

When do we need dynamic memory?

- **When you need a lot of memory.**
  - Typical stack size is 1 MB, so anything bigger than 50-100KB should better be dynamically allocated, or you're risking crash.
- **When the memory must live after the function returns.**
  - Stack memory gets destroyed when function ends, dynamic memory is freed when you want.
- **Size that is unknown at runtime**
  - When you're building a structure (like array, or graph) that dynamically changes or is too hard to precalculate.
- **Allocate storage space while the program is running**
  - We cannot create new variable names "on the fly"

Then why does this work?

- There is a GCC extension to the standard that makes this work
- Not part of the standard C++ specification, but it is supported by some compilers as an extension from the C99 standard of the C language.

```
int n = 0;
int i = 0;

std::cout << "Enter size: ";
std::cin >> n;
int myarray[n];

for (i=0; i<n; i++)
{
  myarray[i] = i;
}
```

Source: https://stackoverflow.com/questions/53760170/why-do-i-need-dynamic-memory-allocation-if-i-can-just-create-an-array


```
#include <iostream>

int main( ) {
    int *p1, *p2;

    p1 = new int;
    *p1 = 10;
    p2 = p1;
    *p2 = 20;
    p1 = new int;
    *p1 = 30;

    std::cout << *p1 << ' ' << *p2 << '\n';

    delete p1;
    delete p2;

    return 0;
}
```

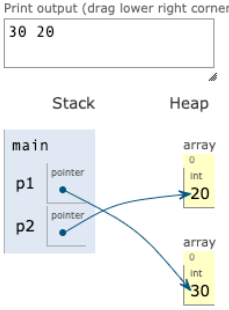

Tracing the code

C++ (gcc 4.8, C++11)
EXPERIMENTAL! known limitations

```
1  #include <iostream>
2
3  int main( ) {
4      int *p1, *p2;
5
6      p1 = new int;
7      *p1 = 10;
8      p2 = p1;
9      *p2 = 20;
10     p1 = new int;
11     *p1 = 30;
12
13     std::cout << *p1 << ' ' << *p2 << '\n';
14
15     delete p1;
16     delete p2;
17
18     return 0;
19 }
```

Print output (drag lower right corner to resize)

```
30 20
```

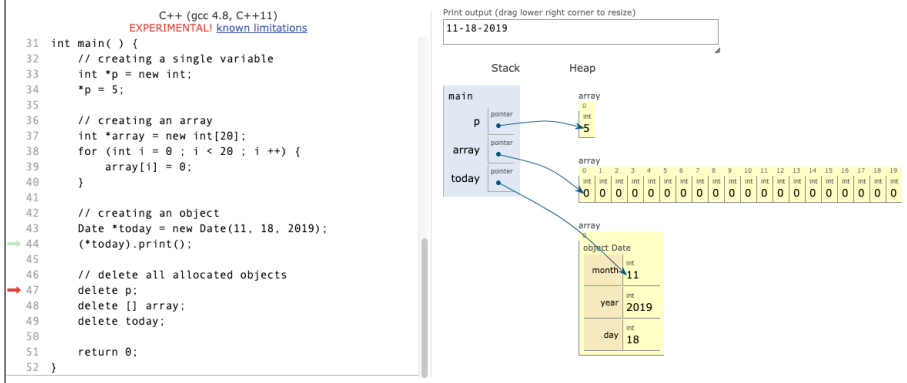


http://pythontutor.com/cpp.html#mode=edit

Syntax for `new` and `delete`

```
#include "date.h"
#include <iostream>

int main( ) {
    // creating a single variable
    int *p = new int;
    *p = 5;

    // creating an array
    int *array = new int[20];
    for (int i = 0 ; i < 20 ; i ++) {
        array[i] = 0;
    }

    // creating an object
    Date *today = new Date(11, 18, 2019);
    (*today).print();

    // delete all allocated objects
    delete p;
    delete [] array;
    delete today;

    return 0;
}
```

Tracing the code



http://pythontutor.com/cpp.html#mode=edit

Array Resizing

```
int size = 5;

int * list = new int[size];

for(int i =0; i < 5; i++){
    list[i] = i;
}

/// need to add more space later on

int * temp = new int[size + 5];

 for (int i = 0; i < size; i++){
    temp[i] = list[i];
 }

  delete [] list;  // this deletes the array pointed to by "list"

   list = temp;
```

https://pythontutor.com

Pointers and objects

- Data members and methods of an object can be accessed by dereferencing a pointer

```
Date *today = new Date(11, 18, 2019);
(*today).print();
```

- Or … can use the **–> operator**

```
Date *today = new Date(11, 18, 2019);
today->print();
```


Memory Leaks

Memory Leak

- A memory leak occurs when a piece of memory which was previously allocated by the programmer. Then it is not deallocated properly by programmer.
- That memory is no longer in use by the program. So that memory location is reserved for no reason.

Memory Leak

```
void my_func() {
   int *data = new int;
   *data = 50;
}
```

👎

```
void my_func() {
   int *data = new int;
   *data = 50;
   delete data;
}
```

👍

Destructors

Destructor

- Special `method` automatically called when objects are destroyed
  - it is used to delete any memory created **dynamically**
- Objects are destroyed when …
  - … they exist in the stack and go out of scope
  - … they exist in the heap and the delete operator is used
- A destructor …
  - … is a member function (usually `public`)
  - … must have the same name as its class preceded by a ~
  - … is automatically called when an object is destroyed
  - … does not have a return type (not even `void`)
  - … takes no arguments

Destructor Syntax

```
//Syntax for defining the destructor within the class
~ <classname>()
{
//body
}


//Syntax for defining the destructor outside the class
<classname>::~<classname>()
{
//body
}
```


Destructor Syntax

```
class Test
{
    public:
        Test()
        {
            std::cout<<"\n Constructor executed";
        }

        ~Test()
        {
            std::cout<<"\n Destructor executed";
        }
};

int main(){

    Test t,t1,t2,t3;
    return 0;
}
```


Destructor Syntax

```
class Test
{
    public:
        Test()
        {
            std::cout<<"\n Constructor executed";
        }

        ~Test()
        {
            std::cout<<"\n Destructor executed";
        }
};

int main(){

    Test t,t1,t2,t3;
    return 0;
}
```

```
 Constructor executed
 Constructor executed
 Constructor executed
 Constructor executed
 Destructor executed
 Destructor executed
 Destructor executed
 Destructor executed%
michaelconti@Michaels-MacBook-Pro-2 Desktop %
```