

CS 32 Intro to CS II

Week 2, Discussion 3A

Xiao Zeng



General Information

- Course info:

<http://web.cs.ucla.edu/classes/winter20/cs32/>

Midterms: 1/30 (Thu.) & 2/25 (Tue.); Final: 3/14 (Sat.)

- TA - Xiao (Steven) Zeng: stevennz@g.ucla.edu
- Discussion 3A:

Fridays 12:00 pm - 1:50 pm, Boelter 5422

- Office hours:

Mondays 11:30 am - 12:30 pm, 3:30 pm - 4:30 pm, Boelter 3256S

Fridays 3:30 pm - 4:30 pm, Boelter 3256S

LA - Sidharth Ramanan

- Email: sidharthramanan@gmail.com
- Office hours: Mondays 4:00 pm to 4:30 pm and Thursdays 8:30 am to 10:00 am @ Boelter 3256S

Today's Topics:

- Constructor & destructor
- Member initialization list
- Order of construction & destruction
- Copy constructor
- Assignment operator
- Worksheet
- Homework questions (tentative)

Constructor

A **constructor** is a special member function that automatically initializes **every new variable** we create of a class. It's called any time we **create a new variable** of a class.

A constructor is called **N times** when we create **an array of size N** (called for each array element). If a class variable is declared in a loop, it's newly constructed **in every iteration**.

```
// Circle.h

class Circle
{
public:
    Circle(double x, double y, double r);
    void draw() const;
    bool scale(double factor);
    double radius() const;
private:
    double m_x;
    double m_y;
    double m_r;
    // Class invariant: m_r > 0
};

double area(const Circle& circ);
```

```
Circle::Circle(double x, double y, double r)
: m_x(x), m_y(y), m_r(r)
{
    if (r <= 0)
    {
        cerr << "Cannot create a circle of radius " << r << endl;
        exit(1);
    }
}
```

```
int main()
{
    Circle c(-2, 5, 10);
    c.scale(2);
    c.draw();
    cout << area(c);
}
```

Member Initialization List

An **initialization/initializer list** can initialize an embedded class object and/or primitive member variables with a constructor.

It's required for any class member variable that does not have default constructor!

Advanced Class Composition

You **must** add an **initializer list** to **all** of your outer class's constructor(s).

```
class Stomach
{
public:
    Stomach()
    {
        myGas = 10;
    }
    ~Stomach() { cout << "Fart!\n"; }
    void eat() { myGas++; }
private:
    int myGas;
};
```

The **initializer list** sits between the constructor's prototype and its body.

It starts with a **colon**, followed by one or more member variables and their **parameters** in parentheses.

So here's what your revised C++ code looks like (without the C++ magic).

```
class HungryNerd
{
public:
    HungryNerd()
    : myBelly(10)
    {
        myBelly.eat();
        myBrain.think();
    }
private:
    Stomach myBelly;
    Brain myBrain;
};
```

Any time you have a member variable (e.g., **myBelly**) that **requires a parameter** for construction...

Advanced Class Composition

This line calls **myBelly's** constructor with a parameter value of **10**!

```
class Stomach
{
public:
    Stomach()
    {
        myGas = 10;
    }
    ~Stomach() { cout << "Fart!\n"; }
    void eat() { myGas++; }
private:
    int myGas;
};

class Brain
{
public:
    Brain() { myIQ = 100; }
    void think() { myIQ += 10; }
private:
    int myIQ;
};
```

Finally, once all of our embedded objects have been constructed, C++ will run the outer constructor's body!

```
class HungryNerd
{
public:
    HungryNerd()
    : myBelly(10)
    {
        myBrain.think();
        myBelly.eat();
        myBrain.think();
    }
private:
    Stomach myBelly;
    Brain myBrain;
};
```

Of course, C++ is still happy to implicitly construct any embedded objects with default constructors for us!

OK - let's see our new **HungryNerd** and **Stomach** classes in action!

carey

myBelly	myGas
	10
myBrain	myIQ
	100

```
int main(void)
{
    HungryNerd carey;
    ...
}
```

Destructor

A **destructor** de-initialize or destroy a class variable when it is goes away.

If we define an array of N items, a destructor is called **N times** when it goes away.

If we don't define a destructor for a class, compiler will create an implicit one for us. It works fine **unless we use dynamically allocated memory in our class** (will be discussed later).

The diagram illustrates the syntax for a C++ destructor within a class definition. It features a code block for `class SomeClass` with a public destructor `~SomeClass()` and a private section. Four callout boxes provide key rules: 1) Destructors must NOT have any parameters. 2) It looks just like a constructor function except for the tilde ~ which identifies it as a destructor. 3) To define a destructor function, place a tilde ~ character in front of the name of the class. 4) Destructors must NOT return a value either.

```
class SomeClass
{
public:
    ~SomeClass()
    {
        // your destructor
        // code goes here
    }
private:
    ...
};
```

Destructors must **NOT** have any parameters.

It looks just like a **constructor** function except for the **tilde ~** which identifies it as a destructor.

To define a destructor function, place a **tilde ~** character in front of the **name** of the class.

Destructors must **NOT** return a value either.

Order of Construction & Destruction

When a class contains a *member variable of a class type*, it **construct** the member variable **first**, and **destruct** the member variable **last**.

Q: What's the output?

```
using namespace std;

class Tire
{
public:
    Tire() { cout << "T "; }
    ~Tire() { cout << "~T "; }
};

class Wheel
{
public:
    Wheel() { cout << "W "; }
    ~Wheel() { cout << "~W "; }
private:
    Tire m_tire;
};
```

```
class Motorcycle
{
public:
    Motorcycle() { cout << "M "; }
    ~Motorcycle() { cout << "~M "; }
private:
    Wheel m_wheels[2];
};

int main()
{
    Motorcycle m;
    cout << endl;
    cout << "====" << endl;
}
```

Try it out: <https://repl.it/@BruinUCLA/Order-of-construction>

Manage Resource

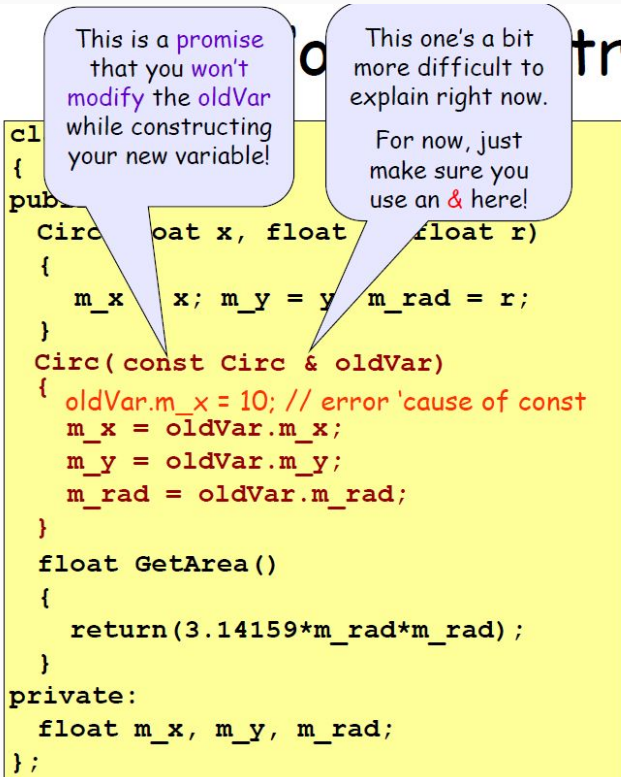
Determine the memory we need for a program during runtime is necessary sometimes (e.g., user inputs). In these cases, we use **new** and **delete** to dynamically allocate memory.

- Allocate memory to a single element: *Ptr = new sometype*
- Allocate memory to a block (array) of elements: *Ptr = new sometype [# of elements]*
- Free memory containing a single element: *delete Ptr*
- Free memory containing a block (array) of elements: *delete [] Ptr*

We should write our own destructor if our class use dynamically allocated memory to **prevent memory leak** before the class instance is destroyed.

Copy Constructor

A **copy constructor** initializes a new object using an existing object of the **same class**.



```
class Circle {
public:
    Circle(float x, float y, float r)
    {
        m_x = x; m_y = y; m_rad = r;
    }
    Circle(const Circle &oldVar)
    {
        oldVar.m_x = 10; // error 'cause of const
        m_x = oldVar.m_x;
        m_y = oldVar.m_y;
        m_rad = oldVar.m_rad;
    }
    float GetArea()
    {
        return(3.14159*m_rad*m_rad);
    }
private:
    float m_x, m_y, m_rad;
};
```

Construction

The parameter to your copy constructor *should* be *const*!

The parameter to your copy constructor *must* be a *reference*!

The *type* of your parameter must be the *same type* as the *class* itself!

```
int main()
{
    Circ a(1,1,5);

    Circ b(a);

    cout << b.GetArea();
}
```

Note: C++ also allows you to write *Circ b = a*, instead of *Circ b(a)*.

Copy Constructor

A default copy constructor provided by C++ just copies all member variables to the new object (**shallow copy**) and may cause problems. Thus in those cases we must define our own copy constructor.

Q: What would a failure case be?

Try it out:

<https://repl.it/@BruinUCLA/Copy-Constructor-Example?language=cpp11&folderId=>

Assignment Operator (Operator=)

An assignment operator copies the value of an **existing** object to the value of another **existing** object with the **same class**. It does the copy job too without construction.

The Assignment Operator

The **const** keyword guarantees that the source object (src) is not modified during the copy.

Now let's see what a real assignment operator

You MUST pass a **reference** to the source object. This means you have to have the **&** here!!!

```
Circ(float x, float y, float r)
{
    m_x = x; m_y = y; m_rad = r;
}

Circ &operator= (const Circ &src)
{
    m_x = src.m_x;
    m_y = src.m_y;
    m_rad = src.m_rad;
    return *this;
}

float GetArea()
{
    return (3.14159*m_rad*m_rad);
}

private:
    float m_x, m_y, m_rad;
};
```

1. The function name is **operator=**
2. The function return type is a **reference to the class**.
3. The function returns ***this** when it's done.

I'll explain this more in a bit...

```
int main()
{
    Circ foo(1,2,3);

    Circ bar(4,5,6);

    bar = foo;
}
```

Existing variable

Existing variable

Assignment Operator (Operator=)

A default assignment operator provided by C++ does **shallow copy** too and may cause problems. Thus in those cases we must define our own assignment operator.

Another example:

```
class PiNerd
{
public:
    PiNerd(int n) { ... }
    ~PiNerd(){ delete[]m_pi; }

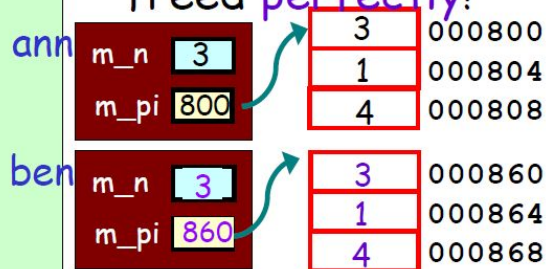
    // assignment operator:
    PiNerd &operator=(const PiNerd &src)
    {
        delete [] m_pi;
        m_n = src.m_n;
        m_pi = new int[m_n];
        for (int j=0;j<m_n;j++)
            m_pi[j] = src.m_pi[j];
        return *this;
    }
    void showOff() { ... }
private:
    int *m_pi, m_n;
};
```

```
int main()
{
    PiNerd ann(3);
    PiNerd ben(4);

    ben = ann;

    }// ann's d'tor called, then ben's
```

... and everything is
freed **perfectly!**



Assignment Operator (Operator=)

We must also check to see if a variable is being assigned to itself, and if so do nothing:

If the **right-hand**
variable's address...

Is the same as the **left-hand**
variable's address...

```
...  
PiNerd & operator=(const PiNerd &src)  
{  
    if (&src == this)  
        return *this; // do nothing  
    delete [] m_pi;  
    m_n = src.m_n;  
    m_pi = new int[m_n];  
    for (int j=0;j<m_n;j++)  
        m_pi[j] = src.m_pi[j];  
    return *this;  
}  
...  
};
```

Then they're the **same variable**!
We simply **return a reference** to
the variable and do nothing else!

And we're done!

A More Elegant Way of Assignment Operator

In real world, issues like exception safety (won't cover in CS32) make the classic way of implementing an assignment operator problematic. A more elegant modern approach is using **copy-and-swap idiom** to implement `Operator=`.

We give our class a swap function that swaps the values of two Strings. Then implement assignment operator using a copy constructor and swap function:

```
void String::swap(String& other)
{
    ... // exchange the m_len and other.m_len ints
    ... // exchange the m_text and other.m_text pointers
}
```

```
String& String::operator=(const String& rhs)
{
    if (this != &rhs)
    {
        String temp(rhs);
        swap(temp);
    }
    return *this;
}
```

Questions?

- You can find code examples used by Prof. Smallberg on course website “Lecs. 2&3 (Smallberg)” section.
- My slides take the following materials as references:
 - Prof. Smallberg’s code examples
 - Prof. Nachenberg’s lecture slides
 - Jack Gong’s discussion slides
 - Mark Edmonds’ website about CS32 from previous years - <https://mjedmonds.com/CS32/CS32.html>

Worksheet!