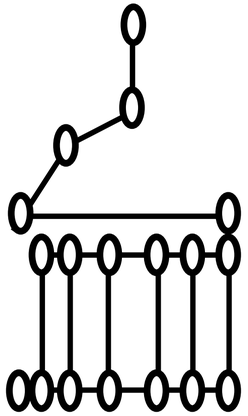
Lecture: More on Classes

ENGR 2730: Computers in Engineering



Constructors & the Member Initializer List

Constructors

Software Engineering Observation 3.1

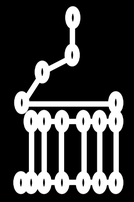
Unless default initialization of your class's data members is acceptable, you should generally provide a custom constructor to ensure that your data members are properly initialized with meaningful values when each new object of your class is created.

- Recall: C++ does not initialize primitive types to zero! Default values for primitive types are undefined!

Assignment inside constructor is sub-optimal

```
class MyClass {
public:
    MyClass()
    {
        m_x = 0;
        m_y = 0;
    }
    void setX(int x) { m_x = x; }
    void setY(int y) { m_y = y; }
    int getX() const { return m_x; }
    int getY() const { return m_y; }
private:
    int m_x;
    int m_y;
};
```

member assignment in body of constructor



Constructor: Member list initialization

member initialization

```
class MyClass {
public:
    MyClass(): m_x(0), m_y(0) { }

    void setX(int x) { m_x = x; }
    void setY(int y) { m_y = y; }
    int getX() const { return m_x; }
    int getY() const { return m_y; }
private:
    int m_x;
    int m_y;

};
```

Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body.

If an object appears in the member initializer list, then its constructor is called before executing code in the constructor's body.

Initialization vs. Assignment

```
class MyClass {
public:
    MyClass(int x=0, int y=0): m_x(x), m_y(y)
    {

    }
…
```

Great!

```
class MyClass {
public:
    MyClass(int x=0, int y=0)
    {
        m_x=x;
        m_y=y;
    }
…
```

Sub-optimal.

Performance Tip 9.4

Initialize member objects explicitly through member initializers. This eliminates the overhead of "doubly initializing" member objects—once when the member object's default constructor is called and again when set *functions are called in the constructor body (or later) to initialize the member object.*

Constructors: Prefer initialization to Assignment

- C++ calls the constructor when each object is created—ideal point to initialize an object's data members.
- Constructor is a special member function that **must** have the same name as the class.
- A constructor can have parameters—the corresponding argument values help initialize the object's data members.
- A constructor has NO RETURN TYPE, not even void

```
// Using data member initialization
MyClass::MyClass(int x=0, int y=0):
        m_x(x), m_y(y)
{

}

-- OR --

// Using data member assignment
MyClass::MyClass(int x=0, int y=0)
{
    m_x=x;
    m_y=y;
}
```

Construction Ordering

Software Engineering Observation 9.8

Data members are constructed in the order in which they're declared in the class definition (not in the order they're listed in the constructor's member-initializer list) and before their enclosing class objects are constructed.

Good Programming Practice 9.2

For clarity, list the member initializers in the order that the class's data members are declared.

Constructors with Parameters

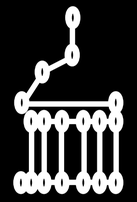
Instantiating Objects: Prefer Initialization to Assignment

- When creating objects, we often know the desired initial state (i.e. values of the member data). We should construct the object with that initial state
- Sometimes you want an object to have the same constant state so that it can never be changed!

```
int main()
{
  // create Time object
  Time t1;

  t1.set(3,30,45);
}
```

Prefer Initialization

```
int main()
{
  // create Time object
  const Time t1(3,30,45);
}
```



Defining a constructor with parameters

```
class MyClass {
public:
    MyClass(int x, int y): m_x(x), m_y(y) {}

    void setX(int x) { m_x = x; }
    void setY(int y) { m_y = y; }

    int getX() const { return m_x; }
    int getY() const { return m_y; }
private:
    int m_x;
    int m_y;
};
```

← a constructor with parameters

*The compiler does **NOT** automatically provide a default constructor if you define a constructor with parameters.*

Output: 1 1

```
int main()
{
    //MyClass myClassObject; // not allowed: default constructor was not automatically provided since
                             // user-defined constructor was defined
    MyClass myClassObject(1,1);

    cout << myClassObject.getX() << " " << myClassObject.getY() << endl;
}
```

Defining many constructors

```
class MyClass {
public:
    MyClass(): m_x(0), m_y(0)  {}

    MyClass(int x): m_x(x), m_y(0) {}

    MyClass(int x, int y): m_x(x), m_y(y) {}

    void setX(int x) { m_x = x; }
    void setY(int y) { m_y = y; }
    int getX() const { return m_x; }
    int getY() const { return m_y; }
private:
    int m_x;
    int m_y;
};
```

You can define as many overloaded constructors as you want (as long as they have different parameter sets)

```
Output: 0 0
        2 0
        3 3
```

```
int main()
{
    MyClass myClassObject1;
    MyClass myClassObject2(2);
    MyClass myClassObject3(3,3);

    cout << myClassObject1.getX() << " " << myClassObject1.getY() << endl;
    cout << myClassObject2.getX() << " " << myClassObject2.getY() << endl;
    cout << myClassObject3.getX() << " " << myClassObject3.getY() << endl;
}
```

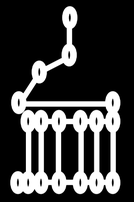
Defining many constructors by taking advantage of default parameters

```
class MyClass {
public:
    MyClass(int x=0, int y=0): m_x(x), m_y(y) {}

    void setX(int x) { m_x = x; }
    void setY(int y) { m_y = y; }
    int getX() const { return m_x; }
    int getY() const { return m_y; }
private:
    int m_x;
    int m_y;
};
```

← Three constructors in one

```
int main()
{
    MyClass myClassObject1;
    MyClass myClassObject2(2);
    MyClass myClassObject3(3,3);

    cout << myClassObject1.getX() << " " << myClassObject1.getY() << endl;
    cout << myClassObject2.getX() << " " << myClassObject2.getY() << endl;
    cout << myClassObject3.getX() << " " << myClassObject3.getY() << endl;
}
```

```
Output: 0 0
        2 0
        3 3
```



CQ: Suppose you wish to add an additional constructor to the class defined below. Which of the following prototypes would be allowed?

```
class MyClass {
public:
    MyClass(int x=0, int y=0): m_x(x), m_y(y) {}

    void setX(int x) { m_x = x; }
    void setY(int y) { m_y = y; }
    int getX() const { return m_x; }
    int getY() const { return m_y; }
private:
    int m_x;
    int m_y;
};
```

A. MyClass(int x, int y, int z);

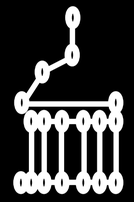
B. MyClass();

C. void MyClass();

D. B and C only

E. A, B and C are all valid

Problem with B : "no-parameter" constructor already defined by default parameters
Problem with C : return types (even void) not allowed for constructors

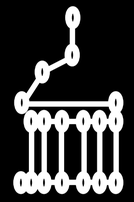


Const Member functions

```
// Time class definition
class Time {


public:
   Time(int h=0, int m=0 , int s=0) {
       setTime(h, m, s);
   }

   void setTime(int hours, int min, int sec);
   string toUniversalString() const;
   string toStandardString()  const;


private:
   int hour;   // 0 - 23
   int minute; // 0 - 59
   int second; // 0 - 59
};
```

Error-Prevention Tip 3.1

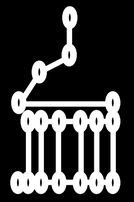
Declaring a member function with `const` *to the right of the parameter list tells the compiler, "this function should* not *modify the object on which it's called—if it does, please issue a compilation error." This can help you locate errors if you accidentally insert in the member function code that would modify the object.*



Only const member functions can be accessed from const objects

```
// Time class definition
class Time {


public:
   Time(int h=0, int m=0 , int s=0) {
       setTime(h, m, s);
   }

   void setTime(int hours, int min, int sec);
   string toUniversalString() const;
   string toStandardString()  const;


private:
   int hour;   // 0 - 23
   int minute; // 0 - 59
   int second; // 0 - 59
};
```

```
// const objects
void main{

  const Time myTime;

  // COMPILER ERROR CATCHES CODING ERROR!
  // invalid because myTime can not change state
  myTime.setTime(1,2,3);

  // OK because only reading values
  myTime.toUniversalString();
  myTime.toStandardString();
};
```



The 'this' implicit parameter

- Object members can be accessed as this->member
- Example:

```
class point2D {
  public:
    void setX (float x) {this->m_x = x;}
    float getX () const {return this->m_x;}

    void setY (float y) {this->m_y = y;}
    float getY () const {return this->m_y;}

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

Equivalent Code

```
class point2D {
  public:
    void setX (float x) {m_x = x;}
    float getX () const {return m_x;}

    void setY (float y) {m_y = y;}
    float getY () const {return m_y;}

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

Overloading Operators: Complex Number Example

C++ allows us to overload the +, -, *, and / operators so that we can add, subtract, multiply and divide complex number objects.

Assume: x = 2 + j2 and y = 1 + j5.

Overloading the binary operations above, we can do things like
z = x + y
z = x * y
z = 3 * x + 4 * y
Etc.

Overloading operators is nice feature of the C++ language.

It is useful to overload the stream insertion operator << to print objects to the screen.

Complex Number Example (Overloaded + and << operators)

```
int main() {

    Complex x(2, 2);
    Complex y(1, 5);

    Complex z = x + y;

    cout << "z = " << z << endl;

    return 0;
}
```

Output:

z = 3 + j7

```
class Complex{
public:
    double real;
    double imag;

    Complex operator+(const Complex & rightNum){
        Complex result;
        result.real = this->real + rightNum.real;
        result.imag = this->imag + rightNum.imag;
        return result;
    }

    friend ostream& operator<<(ostream& output, const Complex& num){
        output << num.real << " + j" << num.imag << endl;
    }
};
```

How does this work?

Answer: x + y is treated as x.operator+(y)