Object-Oriented Programming

# Virtual Destructors

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

# virtual

```cpp
class Parser {
public:
    // constructor
    Parser(std::string_view encoding);

    // parse the buffer using a calculated encoding
    void parse();

private:
    // creates a buffer of the proper encoding
    virtual std::string createBuffer(std::string_view encoding) = 0;

    // hook during parsing for progress updates
    virtual void updateProgress(int bytes);

public:
    // destructor
    ~Parser();
};
```

- Constructors **cannot** be virtual
- Methods can be *virtual* or *non-virtual*
- What about *destructors*?

# Example Classes

```
class BaseData final {
public:
    ~BaseData();
};

class Base {
public:
    ~Base();
private:
    BaseData data;
};
```

```
class DerivedData final {
public:
    ~DerivedData();
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

# Usage

```
class Base {
public:
    ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

```
{
    Base b;
}

{
    Derived d;
}

{
    Base* pb = new Base;

    delete pb;
}

{
    Base* pb = new Derived;

    delete pb;
}
```

# Run

```cpp
class Base {
public:
    ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

```cpp
{
    Base b;
}

{
    Derived d;
}

{
    Base* pb = new Base;

    delete pb;
}

{
    Base* pb = new Derived;

    delete pb;
}
```

```
~Base()
~BaseData()

~Derived()
~DerivedData()
~Base()
~BaseData()

~Base()
~BaseData()

~Base()
~BaseData()
```

# Problem

```
~Base()
~BaseData()

~Derived()
~DerivedData()
~Base()
~BaseData()


~Base()
~BaseData()


~Base()
~BaseData()
```

- When using a Base pointer with a Derived object, the Derived destructor is not called
- Not calling the destructor means that any destructor cleanup is not performed, e.g., RAII
- Not only is the Derived destructor not called, but destructors for all fields/data members are also not called
- Why? The compiler uses static dispatch to set up a call to the destructor
- Solution: *virtual destructor*

# Virtual Destructor Classes

```
class Base {
public:
    virtual ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

# Virtual Destructor Usage

```
class Base {
public:
    virtual ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

```
{
    Base b;
}

{
    Derived d;
}

{
    Base* pb = new Base;

    delete pb;
}

{
    Base* pb = new Derived;

    delete pb;
}
```

# Virtual Destructor Run

```cpp
class Base {
public:
    virtual ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

```cpp
{
    Base b;
}

{
    Derived d;
}

{
    Base* pb = new Base;

    delete pb;
}

{
    Base* pb = new Derived;

    delete pb;
}
```

```
~Base()
~BaseData()

~Derived()
~DerivedData()
~Base()
~BaseData()

~Base()
~BaseData()

~Derived()
~DerivedData()
~Base()
~BaseData()
```

# Class Comparison

```
class Base {
public:
    ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

```
class Base {
public:
    virtual ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

# Run Comparison

```
class Base {
public:
    ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

```
{
    Base b;
}

{
    Derived d;
}

{
    Base* pb = new Base;

    delete pb;
}

{
    Base* pb = new Derived;

    delete pb;
}
```

```
~Base()
~BaseData()

~Derived()
~DerivedData()
~Base()
~BaseData()

~Base()
~BaseData()

~Base()
~BaseData()
```

```
class Base {
public:
    virtual ~Base();
private:
    BaseData data;
};

class Derived : public Base {
public:
    ~Derived();
private:
    DerivedData data;
};
```

```
{
    Base b;
}

{
    Derived d;
}

{
    Base* pb = new Base;

    delete pb;
}

{
    Base* pb = new Derived;

    delete pb;
}
```

```
~Base()
~BaseData()

~Derived()
~DerivedData()
~Base()
~BaseData()

~Base()
~BaseData()

~Derived()
~DerivedData()
~Base()
~BaseData()
```

```
class Base {
public:
    virtual ~Base() = default;
private:
    BaseData data;
};
```

# Conclusion

- Any base class with a virtual method should have a virtual destructor
- If there is nothing to do in the destructor, use the `default` function specifier
- Default destructors are compiler generated and can be more efficient than user-created destructors
- Prefer to declare a default destructor in the include file, but if there is an issue, you can declare it in the implementation file, e.g.,

```
Base::~Base() = default;
```