ADVANCED PROGRAMMING IN C++

Dynamic Memory 2

Patrick Bader · SS 2023

Dynamic Memory Management in Value Types

```
class Widget {
private:
  char* m_buffer;
  int m_size;
public:
  Widget(int bufferSize)
    : m_buffer(new char[bufferSize]),
      m_size(bufferSize) {

      for(int i = 0; i < bufferSize; ++i) {
        m_buffer[i] = 0;
      }
  }

  //...
  ~Widget() {
    delete[] m_buffer;
  }
};
```

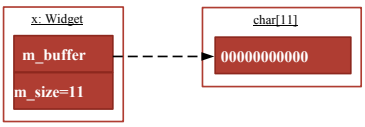
- Dynamic Memory is allocated in constructor
- Memory is freed in destructor
- Destructor is guaranteed to run before lifetime of Widget ends
- **Lifetime of dynamic memory is bound to lifetime of enclosing object**

Copying Value Types with Pointer Members

```
int main() {
// -->
  Widget x(11);

  {
    Widget y(x);
  }

  return 0;
}
```

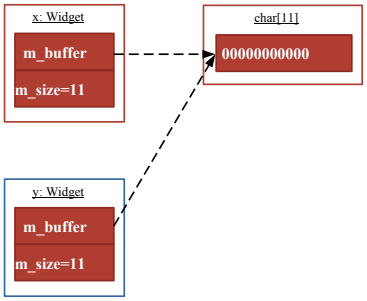
Copying Value Types with Pointer Members

```
int main() {
  Widget x(11);
// -->

  {
    Widget y(x);
  }

  return 0;
}
```



Copying Value Types with Pointer Members

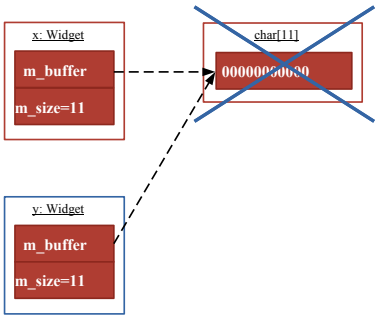
```
int main() {
  Widget x(11);

  {
    Widget y(x);
// -->
  }

  return 0;
}
```



Shallow copy: Memory of buffer is shared by `x` and `y`.

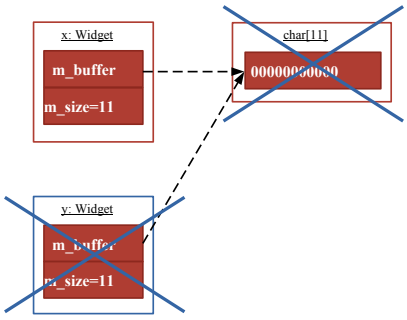
Copying Value Types with Pointer Members

```
int main() {
  Widget x(11);

  {
    Widget y(x);
  }
// -->
  return 0;
}
```



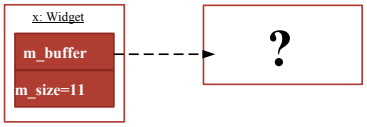
Copying Value Types with Pointer Members

```
int main() {
  Widget x(11);

  {
    Widget y(x);
  }
// -->
  return 0;
}
```



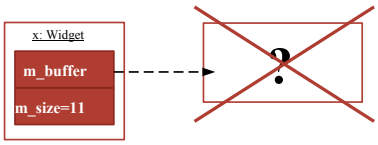
Copying Value Types with Pointer Members

```
int main() {
  Widget x(11);

  {
    Widget y(x);
  }

// -->
  return 0;
}
```



Copying Value Types with Pointer Members

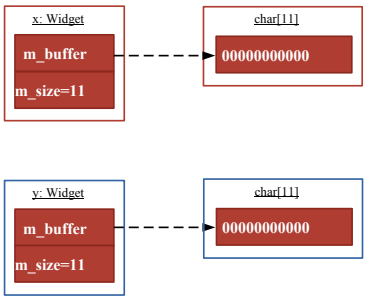
```
int main() {
  Widget x(11);

  {
    Widget y(x);
  }

  return 0;
// -->
}
```



Error: `delete` called twice on same memory location

How to implement correct behavior?

- Compiler generated **copy constructor** copies all members
- **Copies** just the **address** stored in pointer members
- Implement a user defined copy constructor
- Perform a **deep copy** of the object.

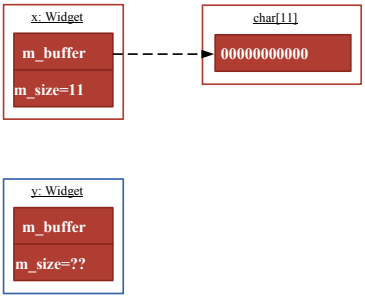


User Defined Copy Constructor

```
class Widget {
private:
  char* m_buffer;
  int m_size;
public:
  Widget(int bufferSize);

  // copy constructor:
  Widget(const Widget& rhs);

  //...
  ~Widget();
};
```

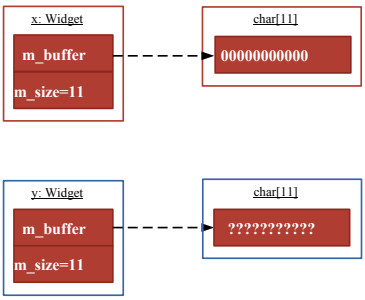
User Defined Copy Constructor

```
  Widget::Widget(const Widget& rhs) {

  }
};
```



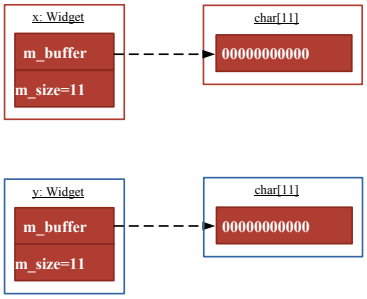
User Defined Copy Constructor

```
  Widget::Widget(const Widget& rhs)
    : m_buffer(new char[rhs.m_size]),
      m_size(rhs.m_size) {

  }
};
```



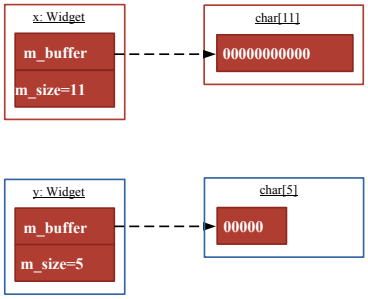
User Defined Copy Constructor

```
  Widget::Widget(const Widget& rhs)
    : m_buffer(new char[rhs.m_size]),
      m_size(rhs.m_size) {

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }
  }
};
```



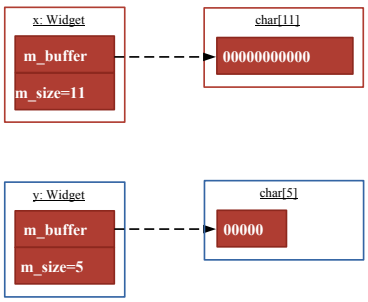
What about assignment?

```
int main() {
  Widget x(11);

  {
    Widget y(5);
//-->
    y = x;
  }

  return 0;
}
```



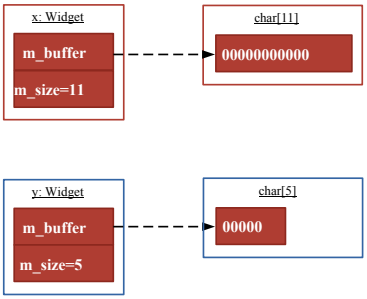
User Defined Copy Assignment Operator

```
class Widget {
private:
  char* m_buffer;
  int m_size;
public:
  Widget(int bufferSize);

  // copy constructor:
  Widget(const Widget& rhs);
  // copy assignment operator
  Widget& operator=(const Widget& rhs);

  //...
  ~Widget();
};
```



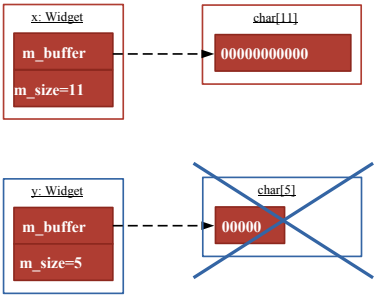
User Defined Copy Assignment Operator

```
  Widget& Widget::operator=(const Widget& rhs) {
    return *this;
  }
};
```



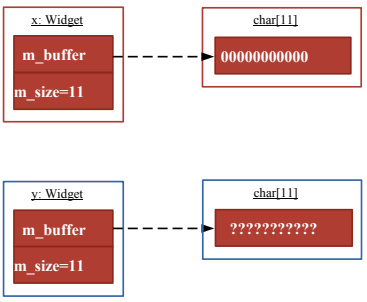
User Defined Copy Assignment Operator

```
  Widget& Widget::operator=(const Widget& rhs) {
    delete[] m_buffer;

    return *this;
  }
};
```



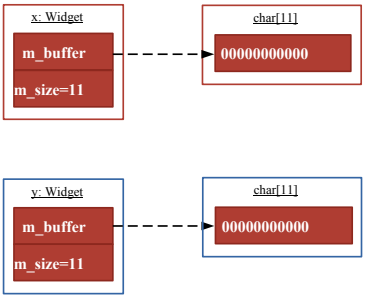
User Defined Copy Assignment Operator

```
  Widget& Widget::operator=(const Widget& rhs) {
    delete[] m_buffer;

    m_size = rhs.m_size;
    m_buffer = new char[rhs.m_size];

    return *this;
  }
};
```



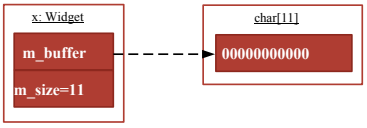
User Defined Copy Assignment Operator

```
  Widget& Widget::operator=(const Widget& rhs) {
    delete[] m_buffer;

    m_size = rhs.m_size;
    m_buffer = new char[rhs.m_size];

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }

    return *this;
  }
};
```



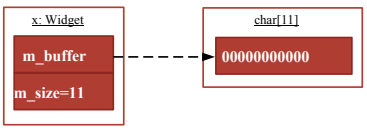
What about self-assignment?

```
int main() {
  Widget x(11);

  x = x;

  return 0;
}
```



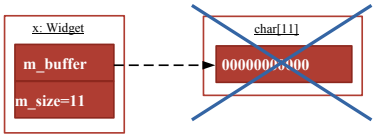
User Defined Copy Assignment Operator - Self Assignment

```
  Widget& Widget::operator=(const Widget& rhs) {
//-->
    delete[] m_buffer;

    m_size = rhs.m_size;
    m_buffer = new char[rhs.m_size];

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }

    return *this;
  }
};
```



User Defined Copy Assignment Operator - Self Assignment

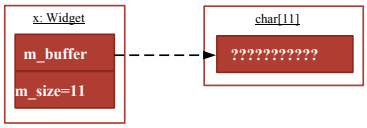
```
  Widget& Widget::operator=(const Widget& rhs) {
    delete[] m_buffer;
//-->

    m_size = rhs.m_size;
    m_buffer = new char[rhs.m_size];

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }

    return *this;
  }
};
```



Beware: Original state of object is destroyed

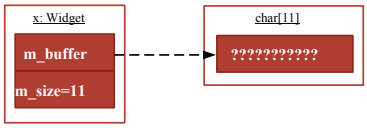
User Defined Copy Assignment Operator - Self Assignment

```
  Widget& Widget::operator=(const Widget& rhs) {
    delete[] m_buffer;

    m_size = rhs.m_size;
    m_buffer = new char[rhs.m_size];
//-->

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }

    return *this;
  }
};
```



User Defined Copy Assignment Operator - Self Assignment

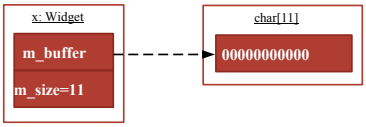
```
  Widget& Widget::operator=(const Widget& rhs) {
    delete[] m_buffer;

    m_size = rhs.m_size;
    m_buffer = new char[rhs.m_size];
//-->

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }

    return *this;
  }
};
```



Error: Copying uninitialized data is undefined behavior!

User Defined Copy Assignment Operator - Self Assignment

```
  Widget& Widget::operator=(const Widget& rhs) {
    if(this == &rhs) {
      return *this;
    }

    delete[] m_buffer;

    m_size = rhs.m_size;
    m_buffer = new char[rhs.m_size];

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }

    return *this;
  }
};
```



Important: Check for self-assignment first!

User Defined Copy Constructor and Copy Assignment Operator - Summary

Needed if you manage resources yourself i.e. the class has a **user defined destructor**

Single-responsibility principle: A module/class/function should do exactly one thing.

Rule of three: If you implement a destructor, also implement a copy constructor and copy assignment operator

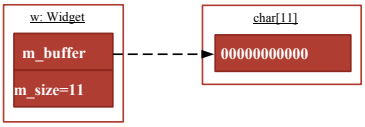
Rule of zero: Implement neither destructor nor copy operations.

Copying can be expensive...

```
Widget createWidget(int size) {
  Widget w(size + 1);

  w[0] = 1;

  return w;
}

int main() {
//-->
  Widget x = createWidget(10);

  return 0;
}
```

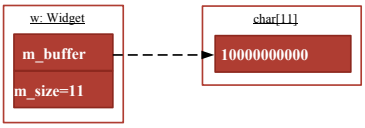
Copying can be expensive...

```
Widget createWidget(int size) {
  Widget w(size + 1);
//-->

  w[0] = 1;

  return w;
}

int main() {
  Widget x = createWidget(10);

  return 0;
}
```



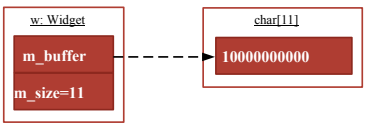
Copying can be expensive...

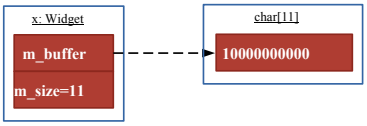
```
Widget createWidget(int size) {
  Widget w(size + 1);

  w[0] = 1;
//-->

  return w;
}

int main() {
  Widget x = createWidget(10);

  return 0;
}
```



Copying can be expensive...

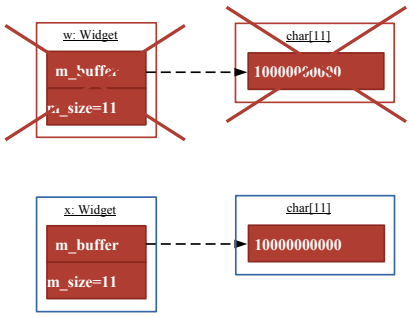
```
Widget createWidget(int size) {
  Widget w(size + 1);

  w[0] = 1;

  return w;
//-->
}

int main() {
  Widget x = createWidget(10);

  return 0;
}
```





Copying can be expensive...

```
Widget createWidget(int size) {
  Widget w(size + 1);

  w[0] = 1;

  return w;
}

int main() {
  Widget x = createWidget(10);
//-->

  return 0;
}
```

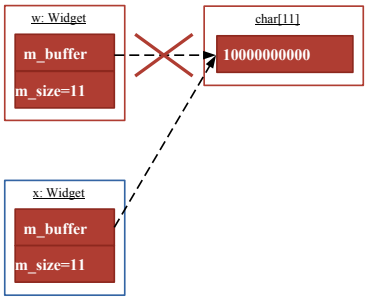


Idea: What if we could steal the buffer from `w`?

```
Widget createWidget(int size) {
  Widget w(size + 1);

  w[0] = 1;

  return w;
//-->
}

int main() {
  Widget x = createWidget(10);

  return 0;
}
```

Idea: What if we could steal the buffer from `w`?

```
Widget createWidget(int size) {
  Widget w(size + 1);

  w[0] = 1;

  return w;
//-->
}

int main() {
  Widget x = createWidget(10);

  return 0;
}
```

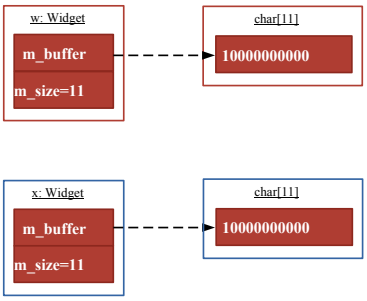


User Defined Move Constructor

```
class Widget {
private:
  char* m_buffer;
  int m_size;
public:
  Widget(int bufferSize);
  // copy constructor:
  Widget(const Widget& rhs);
  // copy assignment operator
  Widget& operator=(const Widget& rhs);
  // move constructor:
  Widget(Widget&& rhs);

  //...
  ~Widget();
};
```

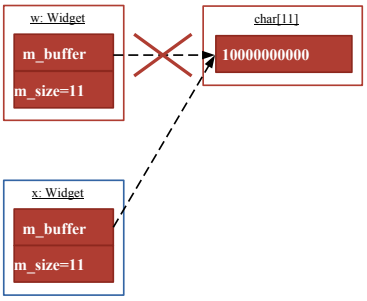
From Copy Constructor...

```
  Widget::Widget(const Widget& rhs)
    : m_buffer(new char[rhs.m_size]),
      m_size(rhs.m_size) {

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }
  }
};
```



... To Move Constructor

```
  Widget::Widget(Widget&& rhs)
    : m_buffer(rhs.m_buffer),
      m_size(rhs.m_size) {

    rhs.m_buffer = nullptr;
  }
};
```



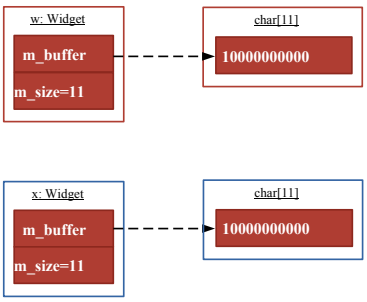
Note: Calling `delete`/`delete[]` on `nullptr` is allowed and does nothing.

User Defined Move Assignment Operator

```
class Widget {
private:
  char* m_buffer;
  int m_size;
public:
  Widget(int bufferSize);
  // copy constructor:
  Widget(const Widget& rhs);
  // copy assignment operator
  Widget& operator=(const Widget& rhs);
  // move constructor:
  Widget(Widget&& rhs);
  // move assignment operator
  Widget& operator=(Widget&& rhs);

  //...
  ~Widget();
};
```

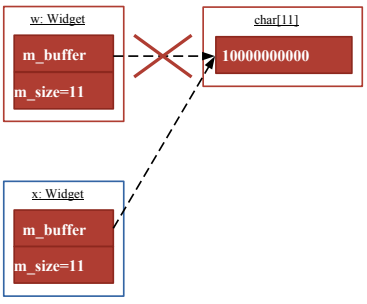
From Copy Assignment Operator...

```
  Widget& Widget::operator=(const Widget& rhs) {
    if(this == &rhs) {
      return *this;
    }

    delete[] m_buffer;

    m_size = rhs.m_size;
    m_buffer = new char[rhs.m_size];

    for(int i = 0; i < m_size; ++i) {
      m_buffer[i] = rhs.m_buffer[i];
    }

    return *this;
  }
};
```



... To Move Assignment Operator

```
  Widget& Widget::operator=(Widget&& rhs) {
    if(this == &rhs) {
      return *this;
    }

    delete[] m_buffer;

    m_size = rhs.m_size;
    m_buffer = rhs.m_buffer;

    rhs.m_buffer = nullptr;

    return *this;
  }
};
```



User Defined Move Constructor and Move Assignment Operator - Summary

Automatically called when right-hand side of operation is a **temporary** object, or goes **out of scope**.

When Move operations are not available, fall back to copying.

Suggested if you manage resources yourself *and* **copying is expensive**

Rule of five: If you implement a destructor, implement all .

Note: You may also implement only the destructor and move operations if copying does not make sense for the type, e.g. your resource is a file.

RAII: Resource Acquisition Is Initialization

Acquire resources during object creation, i.e. **in constructor**.

Release resources during object destruction, i.e. **in destructor**.

Variables are **guaranteed** to be **destructed** when their **scope ends**: `}`.

We tie the **lifetime of dynamic resources** to the **lifetime of** their **enclosing object**.

If the enclosing object is not leaked, the associated resource is also not leaked.