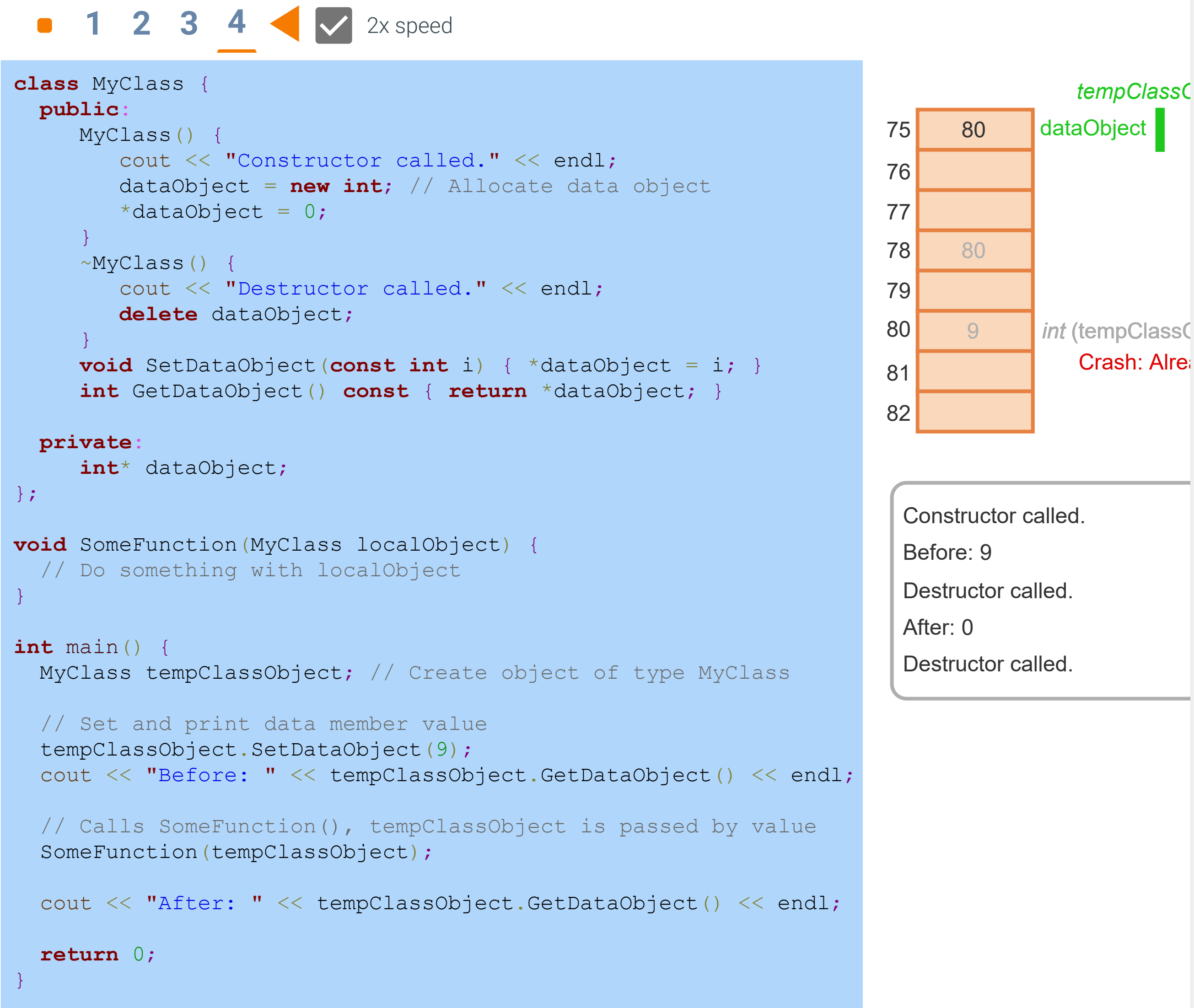
8.9 Copy constructors

Copying an object without a copy constructor

The animation below shows a typical problem that arises when an object is passed by value to a function and no copy constructor exists for the object.

PARTICIPATION ACTIVITY | 8.9.1: Copying an object without a copy constructor.



```
class MyClass {
  public:
     MyClass() {
        cout << "Constructor called." << endl;
        dataObject = new int; // Allocate data object
        *dataObject = 0;
     }
     ~MyClass() {
        cout << "Destructor called." << endl;
        delete dataObject;
     }
     void SetDataObject(const int i) { *dataObject = i; }
     int GetDataObject() const { return *dataObject; }

  private:
     int* dataObject;
};

void SomeFunction(MyClass localObject) {
  // Do something with localObject
}

int main() {
  MyClass tempClassObject; // Create object of type MyClass

  // Set and print data member value
  tempClassObject.SetDataObject(9);
  cout << "Before: " << tempClassObject.GetDataObject() << endl;

  // Calls SomeFunction(), tempClassObject is passed by value
  SomeFunction(tempClassObject);

  cout << "After: " << tempClassObject.GetDataObject() << endl;

  return 0;
}
```

```
Constructor called.
Before: 9
Destructor called.
After: 0
Destructor called.
```

When main() returns, the MyClass destructor is called again, attempting to free the dataObject's m again, causing the program to crash.

Feedback?

PARTICIPATION ACTIVITY | 8.9.2: Copying an object without a copy constructor. ✓

1) If an object with an int sub-object is passed by value to a function, the program will complete execution with no errors.

● True

○ False

Correct

When passed by value to a function, a local copy of the object is made, which also makes a local copy of the int sub-object. When the function terminates, both local objects are destroyed, causing no problems for the rest of the program. Common data types like int and double can be locally copied without a copy constructor.

✓

2) If an object with an int* sub-object is passed by value to a function, the program will complete execution with no errors.

○ True

● False

Correct

When passed by value to a function, a local copy of the object is made, which directs a copy of the int* sub-object to point at the same data as the original object. When the function terminates, the destructor destroys the local object, freeing the sub-object's memory. When main() terminates, the destructor is called again to free the already freed memory, causing a crash.

✓

3) If an object with an int* sub-object is passed by value to a function, the program will call the class constructor to create a local copy of the object.

○ True

● False

Correct

Passing an object by value creates a local copy of the object, but not by calling the constructor, which would make a new object. The local copy's sub-object points to the same memory location as the original object's sub-object, causing problems when the function terminates.

✓

Feedback?

Copy constructor

The solution is to create a ***copy constructor***, a constructor that is automatically called when an object of the class type is passed by value to a function and when an object is initialized by copying another object during declaration. Ex: `MyClass classObj2 = classObj1;` or `obj2Ptr = new MyClass(classObj1);`. The copy constructor makes a new copy of all data members (including pointers), known as a ***deep copy***.

If the programmer doesn't define a copy constructor, then the compiler implicitly defines a constructor with statements that perform a memberwise copy, which simply copies each member using assignment: `newObj.member1 = origObj.member1`, `newObj.member2 = origObj.member2`, etc. Creating a copy of an object by copying only the data members' values creates a ***shallow copy*** of the object. A shallow copy is fine for many classes, but typically a deep copy is desired for objects that have data members pointing to dynamically allocated memory.

The copy constructor can be called with a single pass-by-reference argument of the class type, representing an original object to be copied to the newly-created object. A programmer may define a copy constructor, typically having the form: `MyClass(const MyClass& origClass);`

Construct 8.9.1: Copy constructor.

```
class MyClass {
   public:
      ...
      MyClass(const MyClass& origClass);
      ...
};
```

Feedback?

The program below adds a copy constructor to the earlier example, which makes a deep copy of the data member dataObject within the MyClass object. The copy constructor is automatically called during the call to SomeFunction(). Destruction of the local object upon return from SomeFunction() frees the newly created dataObject for the local object, leaving the original tempClassObject's dataObject untouched. Printing after the function call correctly prints 9, and destruction of tempClassObject during the return from main() produces no error.

Figure 8.9.1: Problem solved by creating a copy constructor that does a deep copy.

```
#include <iostream>
using namespace std;

class MyClass {
public:
   MyClass();
   MyClass(const MyClass& origClass); // Copy constructor
   ~MyClass();

   // Set member value dataObject
   void SetDataObject(const int setVal) {
      *dataObject = setVal;
   }

   // Return member value dataObject
   int GetDataObject() const {
      return *dataObject;
   }
private:
   int* dataObject;// Data member
};

// Default constructor
MyClass::MyClass() {
   cout << "Constructor called." << endl;
   dataObject = new int; // Allocate mem for data
   *dataObject = 0;
}

// Copy constructor
MyClass::MyClass(const MyClass& origClass) {
   cout << "Copy constructor called." << endl;
   dataObject = new int; // Allocate sub-object
   *dataObject = *(origClass.dataObject);
}

// Destructor
MyClass::~MyClass() {
   cout << "Destructor called." << endl;
   delete dataObject;
}

void SomeFunction(MyClass localObj) {
   // Do something with localObj
}

int main() {
   MyClass tempClassObject; // Create object of type MyClass

   // Set and print data member value
   tempClassObject.SetDataObject(9);
   cout << "Before: " << tempClassObject.GetDataObject() << endl;

   // Calls SomeFunction(), tempClassObject is passed by value
   SomeFunction(tempClassObject);

   // Print data member value
   cout << "After: " << tempClassObject.GetDataObject() << endl;

   return 0;
}
```

```
Constructor called.
Before: 9
Copy constructor called.
Destructor called.
After: 9
Destructor called.
```

Feedback?

Copy constructors in more complicated situations

The above examples use a trivially-simple class having a dataObject whose type is a pointer to an integer, to focus attention on the key issue. Real situations typically involve classes with multiple data members and with data objects whose types are pointers to class-type objects.

PARTICIPATION ACTIVITY 8.9.3: Determining which constructor will be called. ✓

Given the following class declaration and variable declaration, determine which constructor will be called for each of the following statements.

```
class EncBlock {
public:
   EncBlock();                         // Default constructor
   EncBlock(const EncBlock& origObj); // Copy constructor
   EncBlock(int blockSize);            // Constructor with int parameter
   ~EncBlock();                        // Destructor
   ...
};

EncBlock myBlock;
```

1) `EncBlock* aBlock = new EncBlock(5);`

○ `EncBlock();`

○ `EncBlock(const EncBlock& origObj);`

● `EncBlock(int blockSize);`

Correct ✓

Constructor with int parameter is called because EncBlock(5) passes a single integer value to the constructor.

2) `EncBlock testBlock;`

● `EncBlock();`

○ `EncBlock(const EncBlock& origObj);`

○ `EncBlock(int blockSize);`

Correct ✓

The default constructor is called because no arguments are passed to the constructor.

3) `EncBlock* lastBlock = new`

Correct ✓

`EncBlock(myBlock);`

- ○ `EncBlock();`
- ● `EncBlock(const EncBlock& origObj);`
- ○ `EncBlock(int blockSize);`

The copy constructor is explicitly called because a variable of the same class type is passed to the constructor.

4) `EncBlock vidBlock = myBlock;`

- ○ `EncBlock();`
- ● `EncBlock(const EncBlock& origObj);`
- ○ `EncBlock(int blockSize);`

✓

Correct

The copy constructor is implicitly called with myBlock because a variable is initialized during the variable declaration to an existing variable of the same class type.

Feedback?

Exploring further:

- More on Copy Constructors from cplusplus.com

CHALLENGE ACTIVITY 8.9.1: Enter the output of the copy constructors. ✓

Jump to level 1

Type the program's output.

```
Copy:
5 3
```

```
#include <iostream>
using namespace std;

class IntNode {
   public:
      IntNode(int value) {
         numVal = new int;
         *numVal = value;
      }
      IntNode(const IntNode& origObject) {
         cout << "Copying " << *(origObject.numVal) << endl;
         numVal = new int;
         *numVal = *(origObject.numVal);
      }
      ~IntNode() {
         delete numVal;
      }
      void SetNumVal(int val) { *numVal = val; }
      int GetNumVal() { return *numVal; }
   private:
      int* numVal;
};

int main() {
   IntNode node1(3);
   IntNode node2 = node1;

   node2.SetNumVal(5);
   cout << node2.GetNumVal() << " " << node1.GetNumVal() << endl;

   return 0;
}
```

1

Check Next **Done**. Click any level to practice more. Completion is preserv

✓ `IntNode node2 = node1` calls the copy constructor, which assigns node2's numVal with a with the value stored by node1's numVal (so, a deep copy). `node2.SetNumVal(5)` changes the but not the value stored at node1's numVal because node2 was a deep copy. So, node1.GetNu than node2.GetNumVal().

Yours

```
Copying 3
5 3
```

Expected

```
Copying 3
5 3
```

Feedback?

CHALLENGE ACTIVITY 8.9.2: Write a copy constructor. ✓

Write a copy constructor for CarCounter that assigns origCarCounter.carCount to the constructed object's carCount. Sample output for the given program:

```
Cars counted: 5
```

```
   carCount = 0;
}

// FIXME add copy constructor

/* Your solution goes here  */
CarCounter::CarCounter(const CarCounter& origCarCounter) {
   //cout << "Copy constructor called." << endl;
   //carCount = new int; // Allocate sub-object
   carCount = origCarCounter.carCount;
}



void CountPrinter(CarCounter carCntr) {
   cout << "Cars counted: " << carCntr.GetCarCount();
}

int main() {
   CarCounter parkingLot;
   int count;

```

Run ✓ All tests passed

✓ Testing carCount assigned 5

Your output
```
Cars counted: 5
```

✓ Testing carCount assigned 9

Your output
```
Cars counted: 9
```

Feedback?