# Classes And Resources

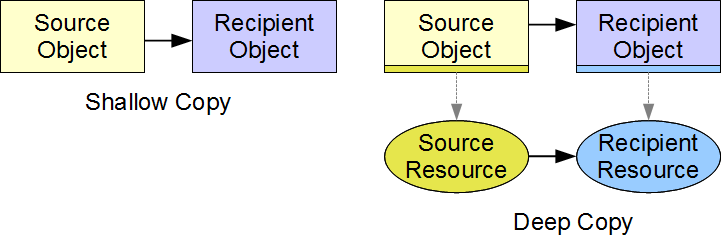
Memory that an object allocated at run-time represents a resource to its class. Management of this resource requires additional logic that was unnecessary in simpler designs. The additional logic ensures proper handling of the resource and is often called deep copying and deep assignment.

The member functions that manage resources are constructors, the assignment operator and the destructor.

**Deep Copies and Assignments**

* In designing a class with a resource, we expect the resource associated with one object to be independent of the resource associated with another object.
* That is, if we change the resource data in one object, we expect the resource data in the other object to remain unchanged.
* In copying and assigning objects we ensure **resource independence through deep copying and deep assigning** involves copying the resource data.
* Shallow copying and assigning involve copying the instance variable only and are only appropriate for non-resource instance variables

.



* In each deep copy, we allocate memory for the underlying resource and copy the content of the source resource into the destination memory.
* We shallow copy the instance variables that are NOT resource instance variables

Two Special member functions manage allocations and deallocations associated with deep copying and deep copy assigning:

* The copy constructor
* The copy assignment operator

If we do not declare a copy constructor, the compiler inserts code that implements a shallow copy.

If we do not declare a copy assignment operator, the compiler inserts code that implements a shallow assignment.

**Copy Constructor**

The copy constructor **contains the logic for copying** **for a source object to a newly created object** of the same type. The compiler calls this constructor when the client code.

* **Creates an object** by initializing it to an existing object
* **Copies an object** by value in a function call
* **Returns an objec**t by value from a function

Declaration

| **Type(const Type&);** |
| --- |

* Where **Type** is the name of the class/
* To define a copy constructor, we insert its declartion into the class

| **Student(const Student&);** |
| --- |

Definition

The definition of a copy constructor contains logic to:

* **Perform a shallow copy on all non-resource** instance variables
* **Allocate memory** for each new resource.
* **Copy data from the source resource** to the newly created resource

| **Student::Student(const Student& src) {  // shallow copy  no = src.no;  ng = src.ng;   // allocate dynamic memory for grades  if (src.grade != nullptr) {  grade = new float[ng];  // copy data from the source resource  // to the newly allocated resource  for (int i = 0; i < ng; i++)  grade[i] = src.grade[i];  }  else {  grade = nullptr;  } }** |
| --- |

**Copy Assignment Operator**

The copy assignment operator **contains the logic for copying data from an existing object to an existing object.**

| **identifier = identifier** |
| --- |

Declaration

* The **left type** is the return type and the **right type** is the type fo the source operand.

| **Type& operator=(const Type&);** |
| --- |

| **Student& operator=(const Student&);** |
| --- |

Definition

The definition of the copy assignment operator contains logic to:

* **Check** for self-assignment
* **Deallocate** any previously allocated memory for the resource associated with the current object.
* **Shallow copy** the non-resouce instance variables to destination variables.
* **Allocate** a new memory for the resource associated with the current object.
* **Copy** resource data from the source object to the newly allocate memory of the current object

| **Student& Student::operator=(const Student& source) {  // 1. check for self-assignment (and NOTHING else)  if (this != &source)  {  // 2. clean up (deallocate previously allocated dynamic memory)  delete[] grade;   // 3. shallow copy (copy non-resource variables)  no = source.no;  ng = source.ng;   // 4. deep copy (copy the resource)  if (source.grade != nullptr) {  // 4.1 allocate new dynamic memory, if needed  grade = new float[ng];  // 4.2 copy the resource data  for (int i = 0; i < ng; i++)  grade[i] = source.grade[i];  }  else {  grade = nullptr;  }  }  return \*this; }** |
| --- |

**Localization**

The code in our definition of the copy constructor is identical to most of the code in our definition of the assignment operator. To avoid such duplication and thereby improve maintainability we can localize the logic in a:

* **Private member function:** **localize the common code** in a private member function and **call** that member function **from both** the constructor and the copy assignment operator.
* **Direct Call:** Call the assignment operator directly from the copy constructor.

**Private Member Function**

| **void Student::init(const Student& source) {  no = source.no;  ng = source.ng;  if (source.grade != nullptr) {  grade = new float[ng];  for (int i = 0; i < ng; i++)  grade[i] = source.grade[i];  }  else {  grade = nullptr;  } }  Student::Student(const Student& source) {  init(source); }  Student& Student::operator=(const Student& source) {  if (this != &source) { // check for self-assignment  // deallocate previously allocated dynamic memory  delete[] grade;  init(source);  }  return \*this; }** |
| --- |

**Direct Call**

The following solution initialises the resource instance variable in the copy constructor to nullptr and call tsh the copy assignment operator directly:

| **Student::Student(const Student& source) {  // copy-assignment operator will deallocate `grade`  // We must ensure that the `grade` doesn't contain some random value.  grade = nullptr;  \*this = source; // calls copy-assignment operator }  Student& Student::operator=(const Student& source) {  // 1. check for self-assignment (and NOTHING else)  if (this != &source)  {  // 2. clean up (deallocate previously allocated dynamic memory)  delete[] grade;   // 3. shallow copy (copy non-resource variables)  no = source.no;  ng = source.ng;   // 4. deep copy (copy the resource)  if (source.grade != nullptr) {  // 4.1 allocate new dynamic memory, if needed  grade = new float[ng];  // 4.2 copy the resource data  for (int i = 0; i < ng; i++)  grade[i] = source.grade[i];  }  else {  grade = nullptr;  }  }  return \*this; }** |
| --- |

Assigning grade to nullptr in the copy constructor ensures that the assignment operator does not deallocate any memory if called by the copy constructor.

**Copies Prohibited**

* Certain class designs require prohibiting client code from copying or copy assigning any instance of a class.
* To prohibit copying and/or copy assigning, we declare the copy constructor and/or the copy assignment operator as deleted members of our class.

| **Student(const Student& source) = delete; Student& operator=(const Student& source) = delete;** |
| --- |