Derived Classes and Resources

Inheritance hierarchies that access resources at multiple levels require intervention.  Managing relationships between the special member functions in a hierarchy with multiple resources involves ensuring that the appropriate calls between these functions are made.  The definitions of some copy constructors and copy assignment operators in the hierarchy may require explicit coding of the connections to their base class counterparts.

This chapter describes how to define the constructors and the copy assignment operators in a hierarchy that access multiple resources and how to call their base class counterparts.

Constructors and Destructor

Each constructor of a derived class calls a constructor of its base class.  By default, that constructor is the no-argument constructor.  To override this default, we insert an explicit call to the base class constructor.

Destructors in an inheritance hierarchy do not require any intervention, since each class in the hierarchy has but one destructor and each destructor calls its sole base class counterpart automatically.

Example

Let us upgrade the definition of our Student class to accommodate a client-defined number of grades.  We store the grades in dynamic memory and store the address of that memory in a resource instance pointer.

The upgraded definition of our Student class contains a resource instance pointer:

|  |
| --- |
| // Student.h  #include <iostream>  const int NC = 30;  class Person {  char name[NC+1];  public:  Person();  Person(const char\*);  void display(std::ostream&) const;  };  class Student : public Person {  int no;  float\* grade;  int ng;  public:  Student();  Student(int);  Student(const char\*, int, const float\*, int);  ~Student();  void display(std::ostream&) const;  }; |

Our four-argument constructor forwards the student's name to the single-argument constructor of the base class and then allocates memory for the grades.  Our destructor deallocates that memory.

|  |
| --- |
| // Student.cpp  #include <cstring>  #include "Student.h"  using namespace std;  Person::Person() {  name[0] = '\0';  }  Person::Person(const char\* nm) {  strncpy(name, nm, NC);  name[NC] = '\0';  }  void Person::display(ostream& os) const {  os << name << ' ';  }  Student::Student() {  no = 0;  ng = 0;  grade = nullptr;  }  Student::Student(int n) {  float g[] = {0.0f};  \*this = Student("", n, g, 0);  }  Student::Student(const char\* nm, int sn, const float\* g, int ng\_) : Person(nm) {  bool valid = sn > 0 && g != nullptr && ng\_ >= 0;  if (valid)  for (int i = 0; i < ng\_ && valid; i++)  valid = g[i] >= 0.0f && g[i] <= 100.0f;  if (valid) {  // accept the client's data  no = sn;  ng = ng\_;  if (ng > 0) {  grade = new float[ng\_];  for (int i = 0; i < ng; i++)  grade[i] = g[i];  } else {  grade = nullptr;  }  } else {  grade = nullptr;  \*this = Student();  }  }  Student::~Student() {  delete [] grade;  }  void Student::display(ostream& os) const {  if (no > 0) {  Person::display(os);  os << no << ":\n";  os.setf(ios::fixed);  os.precision(2);  for (int i = 0; i < ng; i++) {  os.width(6);  os << grade[i] << endl;  }  os.unsetf(ios::fixed);  os.precision(6);  } else {  os << "no data available" << endl;  }  } |

The following client uses this implementation to produce the output shown on the right:

|  |  |
| --- | --- |
| // Derived Class with a Resource Constructors  // dclassResourceCtor.cpp  #include <iostream>  #include "Student.h"  int main() {  Person jane("Jane");  float gh[] = {89.4f, 67.8f, 45.5f};  Student harry("Harry", 1234, gh, 3);  harry.display(std::cout);  jane.display(std::cout);  } | Harry 1234:  89.40  67.80  45.50  Jane |

Copy Constructor

The copy constructor of a derived class calls a constructor of the base class.  By default, that constructor is the no-argument constructor.  To override this default, we explicitly call the base class constructor of our choice.

The header in the definition of the copy constructor for a derived class takes the form

*Derived*(const *Derived*& *identifier*) : *Base*(*identifier*) {

// ...

}

The parameter receives an unmodifiable reference to an object of the derived class.  The argument in the call to the base class' constructor is the parameter's identifier.

Copying occurs in two distinct stages and four steps altogether:

1. copy the base class part of the existing object
   1. allocate memory for the instance variables of the base class in the order of their declaration
   2. execute the base class' copy constructor
2. copy the derived class part of the existing object
   1. allocate memory for the instance variables of the derived class in the order of their declaration
   2. execute the derived class' copy constructor

Example

Let us declare our own definition of the copy constructor for our Student class, but use the default definition for the Person class:

|  |
| --- |
| // Student.h  #include <iostream>  const int NC = 30;  class Person {  char name[NC+1];  public:  Person();  Person(const char\*);  void display(std::ostream&) const;  };  class Student : public Person {  int no;  float\* grade;  int ng;  public:  Student();  Student(int);  Student(const char\*, int, const float\*, int);  Student(const Student&);  ~Student();  void display(std::ostream&) const;  }; |

We implement the copying steps as follows:

1. shallow copy the Person part of the source object
   * allocate static memory for name in the base class part of the newly created object
   * copy into name the string at address src.name
2. copy the Student part of the source object
   * allocate static memory for no, \*grade and ng in the derived part of the newly created object
   * shallow copy src.no into no
   * shallow copy src.ng into ng
   * allocate dynamic memory for a copy of src.grade
   * deep copy the elements at src.grade into grade

The default copy constructor for the base class performs a shallow copy.  The copy constructor for the derived class calls the base class copy constructor and performs the deep copy itself:

|  |
| --- |
| // Student.cpp  #include <cstring>  #include "Student.h"  using namespace std;  Person::Person() {  name[0] = '\0';  }  Person::Person(const char\* nm) {  strncpy(name, nm, NC);  name[NC] = '\0';  }  void Person::display(ostream& os) const {  os << name << ' ';  }  Student::Student() {  no = 0;  ng = 0;  grade = nullptr;  }  Student::Student(int n) {  float g[] = {0.0f};  \*this = Student("", n, g, 0);  }  Student::Student(const char\* nm, int sn, const float\* g, int ng\_) : Person(nm) {  bool valid = sn > 0 && g != nullptr && ng\_ >= 0;  if (valid)  for (int i = 0; i < ng\_ && valid; i++)  valid = g[i] >= 0.0f && g[i] <= 100.0f;  if (valid) {  // accept the client's data  no = sn;  ng = ng\_;  if (ng > 0) {  grade = new float[ng\_];  for (int i = 0; i < ng; i++)  grade[i] = g[i];  } else {  grade = nullptr;  }  } else {  grade = nullptr;  \*this = Student();  }  }  Student::Student(const Student& src) : Person(src) {  no = src.no;  ng = src.ng;  if (src.grade != nullptr && ng > 0) {  grade = new float[ng];  for (int i = 0; i < ng; i++)  grade[i] = src.grade[i];  }  else  grade = nullptr;  }  Student::~Student() {  delete [] grade;  }  void Student::display(ostream& os) const {  if (no > 0) {  Person::display(os);  os << no << ":\n";  os.setf(ios::fixed);  os.precision(2);  for (int i = 0; i < ng; i++) {  os.width(6);  os << grade[i] << endl;  }  os.unsetf(ios::fixed);  os.precision(6);  } else {  os << "no data available" << endl;  }  } |

The Student copy constructor executes its logic after the Person copy constructor has executed its logic.

The following client uses this implementation to produce the output shown on the right:

|  |  |
| --- | --- |
| // Derived Class with a Resource Copy Constructor  // dclassResourceCopyCtor.cpp  #include <iostream>  #include "Student.h"  int main() {  float gh[] = {89.4f, 67.8f, 45.5f};  Student harry("Harry", 1234, gh, 3);  Student harrz = harry; // calls copy constructor  harry.display(std::cout);  harrz.display(std::cout);  } | Harry 1234:  89.40  67.80  45.50  Harry 1234:  89.40  67.80  45.50 |

Copy Assignment Operator

The default copy assignment operator of a derived class calls the copy assignment operator of its base class.  However, any custom copy assignment operator of a derived class DOES NOT by default call the copy assignment operator of its base class.  Accordingly, a custom copy assignment operator of a derived class with a resource requires an explicit call to the base class copy assignment operator.

We call the base class copy assignment operator from within the body of the definition of the derived class assignment operator.  The call takes one of the following forms:

* the functional form
* the cast assignment form

The functional expression takes the form

*Base*::operator=(*identifier*);

The assignment expression takes the form

(*Base*&)\*this = *identifier*;

*Base* is the name of the base class and *identifier* is the name of the right operand, which is the source object for the assignment.  Note that the address of the derived object is the same as the address of the base class part of that object.  The compiler distinguishes the call to the base class operator from a call to the derived class operator by the type of the left operand.

Example

The derived class definition declares the assignment operator and a private member function for the copying operations:

|  |
| --- |
| // Student.h  #include <iostream>  const int NC = 30;  class Person {  char name[NC+1];  public:  Person();  Person(const char\*);  void display(std::ostream&) const;  };  class Student : public Person {  int no;  float\* grade;  int ng;  void init(int, int, const float\*);  public:  Student();  Student(int);  Student(const char\*, int, const float\*, int);  Student(const Student&);  Student& operator=(const Student& src);  ~Student();  void display(std::ostream&) const;  }; |

The private init() contains the copying logic shared by the constructors and the assignment operator:

|  |
| --- |
| // Student.cpp  #include <cstring>  #include "Student.h"  using namespace std;  Person::Person() {  name[0] = '\0';  }  Person::Person(const char\* nm) {  strncpy(name, nm, NC);  name[NC] = '\0';  }  void Person::display(std::ostream& os) const {  os << name << ' ';  }  void Student::init(int no\_, int ng\_, const float\* g) {  no = no\_;  ng = ng\_;  if (g != nullptr && ng > 0) {  grade = new float[ng\_];  for (int i = 0; i < ng; i++)  grade[i] = g[i];  } else {  grade = nullptr;  }  }  Student::Student() {  no = 0;  ng = 0;  grade = nullptr;  }  Student::Student(int n) {  float g[] = {0.0f};  \*this = Student("", n, g, 0);  }  Student::Student(const char\* nm, int sn, const float\* g, int ng\_) : Person(nm) {  bool valid = sn > 0 && g != nullptr && ng\_ >= 0;  if (valid)  for (int i = 0; i < ng\_ && valid; i++)  valid = g[i] >= 0.0f && g[i] <= 100.0f;  if (valid) {  init(sn, ng\_, g);  } else {  grade = nullptr;  \*this = Student();  }  }  Student::Student(const Student& src) : Person(src) {  init(src.no, src.ng, src.grade);  }  Student& Student::operator=(const Student& src) {  if (this != &src) {  // Base class assignment  // 1 - functional expression  // Person::operator=(src);  // 2 - assignment expression  (Person&)\*this = src; // call base class assignment operator  delete [] grade;  init(src.no, src.ng, src.grade);  }  return \*this;  }  Student::~Student() {  delete [] grade;  }  void Student::display(ostream& os) const {  if (no > 0) {  Person::display(os);  os << no << ":\n";  os.setf(ios::fixed);  os.precision(2);  for (int i = 0; i < ng; i++) {  os.width(6);  os << grade[i] << endl;  }  os.unsetf(ios::fixed);  os.precision(6);  } else {  os << "no data available" << endl;  }  } |

Sharing a private member function is one way of coding the copy constructor and assignment operator for the derived class.

The following client uses this implementation to produce the output shown on the right:

|  |  |
| --- | --- |
| // Derived Class with a Resource Copy Assignment  // dclassResourceCopyAssmnt.cpp  #include <iostream>  #include "Student.h"  int main() {  float gh[] = {89.4f, 67.8f, 45.5f};  Student harry("Harry", 1234, gh, 3), harrz;  harrz = harry; // calls copy assignment  harry.display(std::cout);  harrz.display(std::cout);  } | Harry 1234:  89.40  67.80  45.50  Harry 1234:  89.40  67.80  45.50 |

Direct Call Copy Constructor

The alternative to sharing a private member function is a direct call from the copy constructor to the copy assignment operator (as in the chapter entitled [Classes and Resources](https://mwatler.github.io/sep200/Week3/ClassesResources.docx)).  In a direct call, the assignment operator copies the base class part of the object and any call to the base class copy constructor is redundant.

|  |
| --- |
| Student::Student(const Student& src) { // calls no-argument base constructor  grade = nullptr;  \*this = src;  }  Student& Student::operator=(const Student& src) {  if (this != &src) {  // Base class assignment  // 1 - functional expression  // Person::operator=(src);  // 2 - assignment expression  Person& person = \*this; // only copies address  person = src; // call base class operator  delete [] grade;  no = src.no;  ng = src.ng;  if (src.ng > 0) {  grade = new float[ng];  for (int i = 0; i < ng; i++)  grade[i] = src.grade[i];  }  else  grade = nullptr;  }  return \*this;  } |

Summary

* a derived class with a resource requires explicit definitions of its special member functions - constructors, copy assignment operator and destructor
* an explicitly defined derived class copy constructor without a call to the base class' copy constructor calls the base class' no-argument constructor
* the derived class' copy constructor executes the logic in the base class' copy constructor first
* an explicitly defined derived class copy assignment operator does NOT automatically call the base class assignment operator.
* the derived class assignment operator executes the base class assignment operator entirely within the scope of the copy derived class assignment operator
* the destructor of a derived class automatically calls the destructor of the base class