CS3520 Programming in C++
Copying, Assigning, Moving

Ken Baclawski
Fall 2016

Outline

- Assignment and Initialization
- Special Member Functions
- The enroll package
- Assignment #9

Assignment and Initialization

Assignment is not initialization

- Common misconception
  - Languages like Java do not distinguish them
  - Important feature of C++
- Initialization is used for a new object
  - Previous values are *never* replaced
- Assignment is used for changed an existing object
  - Previous values are *always* replaced

Constants

Closely related to the `const` keyword

- A `const` data member, parameter or variable can only be initialized
- Only `const` methods can be called on such an object
- By definition, such an object can never be assigned

Initialization

- Uses of initialization
  - Variable declarations are initializations
    - `int x = 5;` is an initialization, *not* an assignment
  - Parameters are initialized when a function is called
  - The return value of a function is initialized
  - Constructor initialization
    - `Molecule("benzene")` initializes an anonymous object
- Constructors are responsible for all initializations

Forms of Initialization

- Direct initialization
  - MyClass mc(5);
- Copy initialization
  - MyClass mc = MyClass(5);
- List initialization
  - MyClass mc{5};
- Different contexts require different forms
- Little difference in practice
- If one does not compile, try another one

Assignment

- Requires an lvalue on the left side and any expression on the right
- Defined by the `operator=` method of the class
- For example,

```
double dist2 = x * x + y * y;

...

double d2 = 0.0;

...

d2 = x * x + y * y;
```

Special Member Functions

Automatically Generated Functions

- C++ automatically generates up to 6 member functions if they are used
  - Called *special member functions*
- Many of the sample programs and assignments have been using special member functions already
- Prior to C++11 there was a “rule of three” because of the special member functions at the time
  - The name comes from the fact that if you need any one of them, then you probably need three more
- The rule should now be called “rule of five”

Shallow versus Deep

- A *shallow* operation is one that only operates at the highest level
- A *deep* operation is one that operates at levels below the highest level
- The next slides illustrate shallow versus deep for a list of strings

Shallow Operation

```
{
  vector<Action*> labels;
  labels.push_back(new Action("produce"));
  labels.push_back(new Action("consume"));
}
```

When `labels` goes out of scope, it is destructed.
However, the destructor for `vector` is shallow.
The vector and the two pointers are deallocated, but the `Action` objects are not deleted.
Why doesn't `vector` delete the `Action` objects?
The vector cannot know whether there are any other pointers to these objects that may later be used to delete the `Action` objects.

Deep Operation

```
{
  vector<Action*> labels;
  labels.push_back(new Action("produce"));
  labels.push_back(new Action("consume"));
  for (Action* action : labels) {
    delete action;
  }
  labels.clear();
  ...
}
```

This is a deep deletion of `labels`.

Shallow Operation

```
vector<Action*> labels;
labels.push_back(new Action("produce"));
labels.push_back(new Action("consume"));
vector<Action*> labels2;
labels2 = labels;
```

When `labels2` is assigned to `labels` the two vectors have the same list of pointers and the `Action` objects are the same.

The assignment operator for `vector` is shallow: it does not also assign the objects that the pointers point to.

Why doesn't `vector` assign the `Action` objects?

The objects could be very large and complex, so it would be inefficient to force them to be assigned.

Shallow Operation

```
vector<Action*> labels;
labels.push_back(new Action("produce"));
labels.push_back(new Action("consume"));
vector<Action*> labels2;
for (Action* action : labels) {
  labels.push_back(new Action(*action));
}
```

This is a deep assignment of `labels2` to `labels`. Now the two lists have independent copies of the `Action` objects.

The special members

- Default constructor
- Destructor
- Copy constructor
- Copy assignment
- Move constructor
- Move assignment

Default Constructor

- `Action action;`
- Constructor with no arguments
- Requirements for automatic generation
  - No user-declared constructors of any kind
- What the generated method does
  - Allocates space for the object
  - All fields are initialized as specified in the class declaration

Destructor

- Never explicitly called
  - Implicitly invoked when stack variable goes out of scope
  - Explicitly invoked when a free store object is deleted
- Requirements for automatic generation
  - No user-declared destructor
- What the generated method does
  - Shallow destruct of all fields
  - Deallocates space of the object

Copy Constructor

- `Action action2(action1);`
  - Constructs an object that is a copy of another one
- Requirements for automatic generation
  - No user-declared copy constructor
- What the generated method does
  - Allocates space for the object
  - Copy constructs all fields using the fields of the other object

Copy Assignment

- `action2 = action1;`
  - Replaces the contents of `action2` with the contents of `action1`
  - The `action2` object becomes a copy of `action1`
- Requirements for automatic generation
  - No user-declared copy assignment operator
- What the generated method does
  - Copy assigns all fields to the fields of the other object

Move Constructor

- `Action action(Action("produce"));`
  - Moves the contents of an unnamed temporary object to newly allocated object
  - The temporary object will be immediately destructed so it is inefficient to copy the contents only to have the originals be deallocated
  - It is more efficient to move the contents
- Many compilers use *Return Value Optimization* in a case like this
  - The address of the the action object will be the same as the address of the temporary object
  - This eliminates the need for moving the contents

Move Constructor

- Requirements for automatic generation
  - No user-declared copy constructor
  - No user-declared copy assignment operator
  - No user-declared move assignment operator
  - No user-declared destructor
  - Not marked as deleted
  - All members and bases are movable
- What the generated method does
  - Allocates space for the object
  - Move constructs all fields from the fields of the other object

Move Assignment

- `action = Action("produce");`
  - Moves the contents of an unnamed temporary object to the `action` object, replacing what was previously in the `action` object
  - The temporary object will be immediately destructed so it is inefficient to copy the contents only to have the originals be deallocated
  - It is more efficient to move the contents
- *Return Value Optimization* may also apply in this case
  - It depends on other optimizations that may have been applied

Move Assignment

- Requirements for automatic generation
  - No user-declared copy constructor
  - No user-declared copy assignment operator
  - No user-declared move constructor
  - No user-declared destructor
  - Not marked as deleted
  - All members and bases are movable
- What the generated method does
  - Move assigns all fields to the fields of the other object

The enroll package

Requirements

- Represent student enrollments in courses
- This is a many-to-many relationship between student records and courses
- A student record has a name and a map from course titles to enrollment records
- An enrollment record has the section and the grade
- The Student class implements all six special member functions
- Each constructor and special member function prints a message when it is called

Purpose

- The purpose of this example is to show the issues that a nontrivial class must address
- Pointers are used in the design even though these should not be used in their raw form
  - Nontrivial classes must deal with the allocation and deallocation of complex resources
  - The example shows what the issues are in a relatively simple case

Enrollment Header

```
#ifndef ENROLL_ENROLLMENT_H
#define ENROLL_ENROLLMENT_H

#include <string>

namespace enroll {

/**
 * A enrollment record.  It should have the
 * section of the course, the term it was taken,
 * all of the assignments, quizzes and exams of
 * the student in this course.  However, for
 * simplicity, it just has the section and the
 * grade.
 *
 * @author Ken Baclawski
 */
class Enrollment {
```

Enrollment is a database record

Enrollment Header

```
public:
  /**
   * Construct an enrollment specifying the
   * section and grade.
   */
  Enrollment(/** The section enrolled in. */
             const std::string& section,
             /** The grade in the section. */
             const std::string& grade = "unspecified");

  /**
   * Get the section enrolled in.
   * @return The section.
   */
  std::string getSection() const noexcept;

  /**
   * Set the section.
   */
  void setSection(const std::string& section) noexcept;
```

Enrollment Header

```
  /**
   * Get the grade.
   * @return The grade.
   */
  std::string getGrade() const noexcept;
  /**
   * Set the grade.
   */
  void setGrade(const std::string& grade) noexcept;
private:
  /**
   * The section enrolled in.
   */
  std::string section_;
  /**
   * The grade of the enrollment.
   */
  std::string grade_;
};
}
#endif
```

Enrollment Source

```
#include <string>
#include "Enrollment.h"
using namespace std;
using namespace enroll;

Enrollment::Enrollment(const std::string& section,
                       const std::string& grade)
  : section_(section), grade_(grade) {}

string Enrollment::getGrade() const noexcept {
  return grade_;
}
void Enrollment::setGrade(const string& grade) noexcept {
  grade_ = grade;
}
string Enrollment::getSection() const noexcept {
  return section_;
}
void Enrollment::setSection(const string& section) noexcept {
  section_ = section;
}
```

Student Header

```
#ifndef ENROLL_STUDENT_H
#define ENROLL_STUDENT_H

#include <map>
#include <string>
#include "Enrollment.h"

namespace enroll {

/**
 * A student record.  It includes the name of the
 * student and a map from courses to enrollments.
 *
 * @author Ken Baclawski
 */
class Student {
```

Student is more than just a database record because it also manages the enrollment records of the student

Student Header

```
public:
   /**
    * Construct a student record with default name
    * and no enrollments.
    */
   Student();

   /**
    * Construct a student record with a name and
    * one enrollment with an initial grade.
    */
   Student(/** The name of the student. */
           const std::string& name,
           /** The first course enrolled in. */
           const std::string& course,
           /** The section of the course. */
           const std::string& section = "1",
           /** The grade of the course. */
           const std::string& grade = "unspecified");
```

Default constructor

Ordinary constructor

Student Header

```
/**
 * Delete the student record.
 */
~Student();

/**
 * Construct a copy of this student record.
 */
Student(/** The student record to be copied. */
        const Student& otherStudent);

/**
 * Copy assign a student record to another.
 */
Student& operator=(/** The other student record to be
                       copy assigned to this one. */
                   const Student& otherStudent);
```

The destructor deletes the enrollment records

Copy constructor initializes a new student record to be a copy of another one

Copy assignment replaces one student record with a copy of another one

Student Header

```
/**
 * Move construct a student record.
 */
Student(/** The other student record
            to be moved to this one. */
        Student&& otherStudent);

/**
 * Move assign a student record to another.
 */
Student& operator=(/** The other student record to be
                       move assigned to this one. */
                   Student&& otherStudent);

/**
 * Get the name of the student.
 * @return The student name.
 */
std::string getName() const noexcept;
```

The double ampersand means “rvalue reference” **not** a reference to a reference

These are the move constructor and move assignment operator

Move constructor and assignment

- New feature of C++11
- Same as copy constructor and copy assignment for the left-hand side
- Right-hand side (rvalue reference) is an object that is *guaranteed* to be a temporary object that is about to be deleted
  - The contents of the right-hand side should be *moved* to the left-hand side, not copied

Student Header

```
/**
 * Set the name of the student.
 */
void setName(const std::string& name) noexcept;

/**
 * Enroll the student in a course.
 */
void enroll(/** The course that the student
                is enrolling in. */
            const std::string& course,
            /** The section of the course. */
            const std::string& section = "1",
            /** The grade of the course. */
            const std::string& grade = "unspecified")
              noexcept;
```

Student Header

```
/**
 * Get the enrollment record for this student in
 * a course.
 * @return The enrollment record or null if the
 * student is not enrolled in the specified
 * course.
 */
Enrollment* getEnrollment(/** The course enrolled in. */
                          const std::string& course)
  noexcept;

/**
 * Drop a course if the student is enrolled in it.
 */
void drop(/** The course the student is dropping. */
          const std::string& course) noexcept;
```

Student Header

```
private:
  /**
   * The name of the student.
   */
  std::string name_;
  /**
   * The map from courses to enrollment records.
   */
  std::map<std::string, Enrollment*> courseToEnrollment_;
  /**
   * Clear the enrollments.  All enrollments are
   * deleted and the map is cleared of all
   * entries.  This is used by the destructor and
   * also in other situations.
   */
  void clear() noexcept;
};
}
#endif
```

Student Source

```
#include <string>
#include <iostream>
#include "Student.h"
#include "Enrollment.h"
using namespace std;
using namespace enroll;

Student::Student() : name_("Default Student") {
  cout << "Constructing default student" << endl;
}

Student::Student(const string& name, const string& course,
                 const string& section, const string& grade)
  : name_(name) {
  cout << "Constructing student " << name_ << endl;
  courseToEnrollment_.insert({course,
    new Enrollment(section, grade)});
}
```

The constructors and assignment operators print information about what is being done

Student Source

```
Student::~Student() {
  cout << "Deleting student " << name_ << endl;
  clear();
}

Student::Student(const Student& otherStudent)
    : name_(otherStudent.name_) {
  cout << "Copy constructing student " << name_ << endl;

  // All of the course enrollment records are copy
  // constructed.  The default copy constructor
  // would only copy the pointers, not the records.

  for (auto courseEnrollment :
    otherStudent.courseToEnrollment_) {
      courseToEnrollment_.insert
        ({courseEnrollment.first,
            new Enrollment(*courseEnrollment.second)});
  }
}
```

Destructor and Copy Constructor

- The destructor deletes all of the enrollment objects
  - It must have its own copies for this to be valid
- The copy constructor copies the enrollment objects
  - The automatically generated copy constructor only copies the pointers, not the enrollment objects
  - The copy constructor must be defined for this class
  - If the automatically generated copy constructor were used, then the destructor would delete enrollment objects more than once

Student Source

```
Student& Student::operator=(const Student& otherStudent) {
  cout << "Copy assigning student " << name_
       << " to " << otherStudent.name_ << endl;

  // Always check for assigning an object to itself.

  if (this == &otherStudent) {
    return *this;
  }

  // Assign the name
  name_ = otherStudent.name_;

  // Clear all of the enrollments
  clear();
```

The copy constructor for the string class deletes the old string and copies the new one

This deletes all of the enrollment records

Student Source

```
  // All of the course enrollment records are copy
  // constructed.  The default copy assignment for
  // the courseToEnrollment_ map only copies the
  // pointers not the records.

  for (auto courseEnrollment :
    otherStudent.courseToEnrollment_) {
      courseToEnrollment_.insert
        ({courseEnrollment.first,
            new Enrollment(*courseEnrollment.second)});
  }

  // Return this student.

  return *this;
}
```

This is the same as in the copy constructor

Returning the object allows one to have a series of assignments: x = y = z = ...

Student Source

```
Student::Student(Student&& otherStudent)
  : name_(otherStudent.name_),
    courseToEnrollment_(otherStudent.courseToEnrollment_) {
  cout << "Move constructing student " << name_ << endl;

  // Clear the student map without deleting any
  // enrollment records because the enrollment
  // records are now in this student.

  otherStudent.courseToEnrollment_.clear();
  otherStudent.name_ = "Not Specified";
}
```

The move constructor uses the copy constructor of the map template which copies the pointers and does not copy the enrollment records (shallow copy). Then it clears the map of the student record being moved so that the enrollment records will not be deleted when the other student record is deleted.

Student Source

```
Student& Student::operator=(Student&& otherStudent) {
  cout << "Move assigning student " << name_
       << " to " << otherStudent.name_ << endl;

  // Always check for assigning an object to
  // itself, although this should never happen

  if (this == &otherStudent) {
    return *this;
  }

  // Assign the name

  name_ = otherStudent.name_;

  // Clear this student record

  clear();
```

This part of the method is the same as the copy assignment operator

Student Source

```
  // Assign the enrollment map

  courseToEnrollment_ = otherStudent.courseToEnrollment_;

  // Remove the map and name from the other
  // student record.

  otherStudent.courseToEnrollment_.clear();
  otherStudent.name_ = "Not Specified";

  // Return this student.

  return *this;
}
```

This part of the move assignment operator is not the same as the copy assignment operator. The copy assignment operator for map is used to copy the pointers to the enrollments of the other student map (shallow copy). Then the other student map is cleared so that the enrollment records will not be deleted twice.

Student Source

```
void Student::clear() noexcept {

  // Delete all of the enrollments

  for (auto courseEnrollment : courseToEnrollment_) {
    delete courseEnrollment.second;
  }

  // Clear the map

  courseToEnrollment_.clear();
}
```

The enrollment records are deleted. After this loop is complete, all of the pointers in the enrollment map are dangling pointers. They are removed when the map is cleared.

Student Source

```
string Student::getName() const noexcept {
  return name_;
}

void Student::setName(const string& name) noexcept {
  name_ = name;
}

void Student::enroll(const string& course,
    const string& section, const string& grade) {
  if (courseToEnrollment_.count(course) == 0) {
    courseToEnrollment_.insert(
      {course, new Enrollment(section, grade)});
  } else {
    throw domain_error("Attempt to enroll in a course "
      "that was not dropped");
  }
}
```

Student Source

```
Enrollment* Student::getEnrollment(const string& course)
    noexcept {
  if (courseToEnrollment_.count(course) == 1) {
    return courseToEnrollment_[course];
  } else {
    return nullptr;
  }
}

void Student::drop(const string& course) noexcept {
  if (courseToEnrollment_.count(course) == 1) {
    Enrollment* enrollment = courseToEnrollment_[course];
    courseToEnrollment_.erase(course);
    delete enrollment;
  }
}
```

Test Program

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Test of the enroll package
#include <boost/test/unit_test.hpp>

#include <stdexcept>
#include <iostream>
#include "Student.h"
#include "Enrollment.h"
using namespace std;
using namespace enroll;

/**
 * @namespace enroll Many-to-many relationship
 * between students and courses.
 *
 * @author Ken Baclawski
 */
```

Test Program

```
BOOST_AUTO_TEST_CASE(enroll_test) {

  // Construct a default student record

  Student unnamed;
  BOOST_CHECK_EQUAL(unnamed.getName(), "Default Student");
  BOOST_CHECK(!unnamed.getEnrollment("CS3520"));

  // Set the name

  unnamed.setName("Jane Doe");
  BOOST_CHECK_EQUAL(unnamed.getName(), "Jane Doe");

  // Enroll in some courses

  unnamed.enroll("CS3520", "1", "B");
  unnamed.enroll("CS3200", "2", "B+");
  BOOST_CHECK_EQUAL(unnamed.getEnrollment("CS3520")->getGrade(), "B");
  BOOST_CHECK_EQUAL(unnamed.getEnrollment("CS3200")->getGrade(), "B+");
}
```

Test Program

```
BOOST_AUTO_TEST_CASE(enroll_copy_test) {

  // Construct a student record

  Student zhang("Zhang Wei", "CS3520", "2", "A-");
  BOOST_CHECK_EQUAL(zhang.getName(), "Zhang Wei");
  BOOST_CHECK_EQUAL(zhang.getEnrollment("CS3520")
                    ->getGrade(), "A-");

  // Copy construct the student record

  Student zhang2(zhang);
  BOOST_CHECK_EQUAL(zhang2.getName(), "Zhang Wei");
  BOOST_CHECK_EQUAL(zhang2.getEnrollment("CS3520")
                    ->getGrade(), "A-");
```

Test Program

```
  // Construct another student record

  Student jose("Jose Garcia", "CS3520", "1", "B+");
  BOOST_CHECK_EQUAL(jose.getName(), "Jose Garcia");
  BOOST_CHECK_EQUAL(jose.getEnrollment("CS3520")
                    ->getGrade(), "B+");

  // Copy assign the second student record to the first

  jose = zhang;
  BOOST_CHECK_EQUAL(jose.getName(), "Zhang Wei");
  BOOST_CHECK_EQUAL(jose.getEnrollment("CS3520")
                    ->getGrade(), "A-");
}
```

Test Program

```
BOOST_AUTO_TEST_CASE(enroll_move_test) {

  // Construct an anonymous student record
  // The record is anonymous, not the student!
  // Then move construct a named student record

  Student zhang(Student("Zhang Wei", "CS3520",
                        "2", "A-"));
  BOOST_CHECK_EQUAL(zhang.getName(), "Zhang Wei");
  BOOST_CHECK_EQUAL(zhang.getEnrollment("CS3520")
                    ->getGrade(), "A-");

  // Construct an anonymous student record
  // Then move assign to the previous named
  // student record

  zhang = Student("Jose Garcia", "CS3520", "1", "B+");
  BOOST_CHECK_EQUAL(zhang.getName(), "Jose Garcia");
  BOOST_CHECK_EQUAL(zhang.getEnrollment("CS3520")
                    ->getGrade(), "B+");
}
```

Unit Test Output

```
Running 3 test cases...
Constructor Test
Constructing default student
Deleting student Jane Doe
Copy Test
Constructing student Zhang Wei
Copy constructing student Zhang Wei
Constructing student Jose Garcia
Copy assigning student Jose Garcia to Zhang Wei
Deleting student Zhang Wei
Deleting student Zhang Wei
Deleting student Zhang Wei
Move Test
Constructing student Zhang Wei
Move constructing student Zhang Wei
Deleting student Not Specified
Constructing student Jose Garcia
Move assigning student Zhang Wei to Jose Garcia
Deleting student Not Specified
Deleting student Jose Garcia

*** No errors detected
```

Assignment #9

Requirements

- Develop generic tree traversal facility
- Develop a templates for the Node class
  - A node contains an element
  - A node has a list of child nodes
  - A node is the root of its descendant nodes
- Develop a Marker class
  - These will be the elements in the tree
- Develop the Main program as a Boost test program

Node Template

- The template parameter is the type of the element
- Constructor has one parameter that is a constant reference to a marker
- Destructor destroys all descendant nodes
- `addChild` method adds a node to the list of child nodes
  - Parameter is a pointer to the node
  - Throw a `domain_error` if the node is null
- `sort` method sorts the child nodes recursively
  - Parameter is a comparison functor object that compares elements
- `append` method appends a lisp-style string representation of the subtree to a string parameter
  - The same parameter is returned as the value of the method

Marker Class

- Consists of a string and an identifier
- Constructor has a constant string reference parameter
- The `operator<` method compares the marker strings
- `str` method displays the marker as a string
  - If marker is the first one constructed and its string is "`A`" then `str()` returns "`A1`"
  - If marker the tenth one constructed and its string is "`Q`" then `str()` returns "`Q10`"

Main Program

- Test program using Boost
- Construct a tree with two child nodes, one of which has two child node
- Check that `str` returns the expected value
  - If the markers have strings “A”, “B”, ... constructed in this order, then `str` should produce

```
“ (A0 (B1 D3 E4) C2)”
```

- Sort the tree and check that `str` returns the expected value
- Check that an exception is thrown if null is added as a child node