Canonical forms for C++ classes

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: March 21, 2025

Canonical Form/Standard Form

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
    - default constructor

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
    - default constructor
    - copy constructor

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
    - default constructor
    - copy constructor
    - destructor

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
  - default constructor
  - copy constructor
  - destructor
  - assignment operator

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
  - default constructor
  - copy constructor
  - destructor
  - assignment operator
- These are otherwise provided by default

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
    - default constructor
    - copy constructor
    - destructor
    - assignment operator
- These are otherwise provided by default
    - unless some constructor other than the copy constructor is provided for a class, then the default constructor is not provided by default!

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
  - default constructor
  - copy constructor
  - destructor
  - assignment operator
- These are otherwise provided by default
  - unless some constructor other than the copy constructor is provided for a class, then the default constructor is not provided by default!
- The reason for putting a class in canonical form

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
    - default constructor
    - copy constructor
    - destructor
    - assignment operator
- These are otherwise provided by default
    - unless some constructor other than the copy constructor is provided for a class, then the default constructor is not provided by default!
- The reason for putting a class in canonical form
    - to avoid memory leaks

Canonical form (definition)

- A C++ class is in canonical form if it provides the following four member functions/operators:
  - default constructor
  - copy constructor
  - destructor
  - assignment operator
- These are otherwise provided by default
  - unless some constructor other than the copy constructor is provided for a class, then the default constructor is not provided by default!
- The reason for putting a class in canonical form
  - to avoid memory leaks
  - to make call by value and return from functions work as expected

How to write a copy constructor

How to write a copy constructor

```
MyClass::MyClass(const MyClass &original)
    : MyBaseClass(
          original), // delegate copying of base class fields
                     // to its own cc
      // delegate copying of field1 to its own cc, etc.
      field1(original.field1),
      field2(original.field2), field3(original.field3)
// ...
{
  // do everything that is required
  // to perform a deep copy of original fields to
  // the reference object fields
}
```

How to write a destructor

How to write a destructor

```
MyClass::~MyClass()
   {
     // usually empty (unless code performed dynamic allocation)

     // free/deallocate all dynamically allocated memory
     // in reverse allocation order
   }
```

- A virtual destructor should be used for any class that is involved in inheritance, i.e., for any base class or any derived class

How to write a destructor

```
MyClass::~MyClass()
  {
    // usually empty (unless code performed dynamic allocation)

    // free/deallocate all dynamically allocated memory
    // in reverse allocation order
  }
```

- A virtual destructor should be used for any class that is involved in inheritance, i.e., for any base class or any derived class
- if in doubt, make it virtual

How to write a destructor

```
MyClass::~MyClass()
   {
     // usually empty (unless code performed dynamic allocation)

     // free/deallocate all dynamically allocated memory
     // in reverse allocation order
   }
```

- A virtual destructor should be used for any class that is involved in inheritance, i.e., for any base class or any derived class
- if in doubt, make it virtual

How to write a destructor

```
MyClass::~MyClass()
   {
     // usually empty (unless code performed dynamic allocation)

     // free/deallocate all dynamically allocated memory
     // in reverse allocation order
   }
```

- A virtual destructor should be used for any class that is involved in inheritance, i.e., for any base class or any derived class
- if in doubt, make it virtual

```
virtual ~MyClass();
```

- never make it purely virtual, however (provide implementation regardless)

How to write a destructor

```
MyClass::~MyClass()
   {
     // usually empty (unless code performed dynamic allocation)

     // free/deallocate all dynamically allocated memory
     // in reverse allocation order
   }
```

- A virtual destructor should be used for any class that is involved in inheritance, i.e., for any base class or any derived class
- if in doubt, make it virtual

```
virtual ~MyClass();
```

- never make it purely virtual, however (provide implementation regardless)
- don’t attempt to invoke base-class destructor (will be done automatically)

How to write an assignment operator

How to write an assignment operator

```
MyClass &MyClass::operator=(const MyClass &original){
  if(&original != this) // don't assign to itself
    {
      // 1. everything in destructor
        //(get rid of the existing value of this reference instance)
      // 2. Everything in the copy constructor
        // (to copy original's fields to reference instance)
      // 2a. if this is a derived class, add this line
      MyBaseClass::operator=(original);
      // 2b. if the copy constructor copies fields
      //using the ":" syntax (i.e., an initializer list),
      // add these lines
      field1 = original.field1;
      field2 = original.field2;
      field3 = original.field3;
      // 2c. everything in copy constructor body
    }
  return *this;
}
```

Example: the Committee class

Example: the Committee class

```
class Committee { // a class with dynamic allocation
private:
  float *pbudget;
  string *pmembers[10];

public:
  // Default Constructor
  Committee();

  // Copy Constructor
  Committee(const Committee &original);

  // Destructor
  ~Committee();

  // Assignment Operator
  Committee &operator=(const Committee &original);
};
```

The Committee class (default constructor)

The Committee class (default constructor)

```
// Default Constructor

Committee::Committee() {

  pbudget = new float(0.0f);

  for (int i = 0; i < 10; i++) {
    pmembers[i] = new string;
  }

}
```

The Committee class (copy constructor)

The Committee class (copy constructor)

```
// Copy Constructor
Committee::Committee(const Committee &original)
    : pbudget(new float(*(
          original.pbudget))) // (delegate copying to cc of float)
{
  // OR these 2 lines (i.e. copy manually)
  // pbudget = new float;
  // *pbudget = *(original.pbudget);

  // OR this one line (again, copy manually)
  // pbudget = new float(*(original.pbudget));

  for (int i = 0; i < 10; i++) {
    pmembers[i] = new string(*(original.pmembers[i]));
  }
}
```

The Committee class (destructor)

The Committee class (destructor)

```cpp
// Destructor
Committee::~Committee() {
  // optional for tracing execution:
  // cout << "Destructor for Committee class" << endl;

  delete pbudget;
  for (int i = 0; i < 10; i++) {
    delete pmembers[i];

    // optional for tracing execution:
    // cout << "Deleting array... " << 10 - i << endl;
  }
}
```

The Committee class (assignment operator)

The Committee class (assignment operator)

```
Committee &Committee::operator=(const Committee &original) {
  if (&original != this) {
    // from destructor
    delete pbudget;
    for (int i = 0; i < 10; i++)
      delete pmembers[i];

    // no base class from which to call operator

    // from copy constructor
    pbudget = new float(*(original.pbudget));
    for (int i = 0; i < 10; i++)
      pmembers[i] = new string(*(original.pmembers[i]));
  }
  return *this;
}
```

New C++ features to support canonical classes

New C++ features to support canonical classes

```
Class A {
  // default constructor has default implementation
  //  (i.e. does nothing)
  A() = default;
  // copy const. has default impl. (i.e. shallow copies)
  A(const A &original) = default;
  // destructor has default implementation (i.e. does nothing)
  ~A() = default;
  A &operator=(const A &other) = default; // shallow copy again
  ...
};
```

Defaults for Canonical Classes (ctd)

Defaults for Canonical Classes (ctd)

```
// tells compiler no implementation is desired
// usually a bad idea
Class B{
  // e.g., prevents arrays from being declared
  B() = delete;
  // prevent instances from being passed by value
  B(const B &original) = delete;
   // prevents instances from being deallocated
  // (can't use delete on B)
  ~B() = delete;
  // can't use assignment operator
  B &operator=(const B &other) = delete;
  ...
};
```