Programming in C++

Factories

Curtis Larsen

Utah Tech University—Computing

Spring 2025

Objectives

Objectives:

- Create an enumeration.
- Create a class data member.
- Create a class method.
- Implement a factory pattern to create objects.

Enumerations

Example

```
int grade = compute_grade();

if(grade == 1) {
  std::cout << "Excellent work!" << std::endl;
} else if(grade == 2) {
  std::cout << "Good work!" << std::endl;
} else if(grade == 3) {
  std::cout << "Mid work." << std::endl;
} else if(grade == 4) {
  std::cout << "You can do it!" << std::endl;
} else if(grade == 5) {
  std::cout << "See you next semester." << std::endl;
} else {
  std::cout << "Woops!." << std::endl;
}
```

Issues

What issues do you see with the previous example code?

Purpose

`enum` defines a discrete set of named integer constants as possible values for a data type.

- Meaningful names for enumerated values
- Reduced risk of errors
- Easier refactoring
- Distinct type
- Limited range

Syntax

```
enum UserTypeName { SYMBOL_1, SYMBOL_2, ..., SYMBOL_N };

UserTypeName variable = SYMBOL_1;

if(variable == SYMBOL_2) {
  ...
}
```

Example

```
enum GradeType { GRADE_A, GRADE_B, GRADE_C, GRADE_D, GRADE_F };
GradeType grade = compute_grade();

if(grade == GRADE_A) {
  std::cout << "Excellent work!" << std::endl;
} else if(grade == GRADE_B) {
  std::cout << "Good work!" << std::endl;
} else if(grade == GRADE_C) {
  std::cout << "Mid work." << std::endl;
} else if(grade == GRADE_D) {
  std::cout << "You can do it!" << std::endl;
} else if(grade == GRADE_F) {
  std::cout << "See you next semester." << std::endl;
} else {
  std::cout << "Woops!." << std::endl;
}
```

Downloads

enum examples

Class Data Members

Instance Data Members

- Also called non-static data members.
- Each object has its own copy of each instance data member.
- Instance data members exist as long as their object exists.
- Declared in class declaration. `type var;`

Example Instance Data Members

```
class Chair {
public:
  Chair(int legs);
  int getLegCount();
private:
  int mLegCount; // declared in class
};
Chair::Chair(int legs)
  : mLegCount(legs) { // initialized in constructor
}
int Chair::getLegCount() { // accessed by method
  return mLegCount;
}
int main() {
  Chair stool(3); // initialized
  std::cout << stool.getLegCount() << std::endl; // accessed
  return 0;
}
```

Class Data Members

- Also called static data members.
- Only one copy of the data member, shared by all objects of the class.
- Static data members exist as long as the program is running.
- Static data members are stored in the static section of memory.
- Declared in class declaration. `static type var;`

Example Static Data Members

```
class Chair {
public:
  Chair(int legs);
  ~Chair();
private:
  static int sChairCount; // declared
};

int Chair::sChairCount = 0; // initialized in global scope
Chair::Chair(int legs) {
  sChairCount++;   // accessed in methods
}
Chair::~Chair() {
  sChairCount--;   // accessed in methods
}
int main() {
  Chair stool(3); // incremented
  return 0; // decremented
}
```

Downloads

static members examples

Class Methods

Instance Methods

- Also called member functions, non-static member functions.
- `this` pointer is available in all methods.
- Must be called on an object of the class.
- Accesses all instance data members and static data members.
- Declared in class declaration.

Example Instance Methods

```
class Chair {
public:
  Chair(int legs);   // declared in class
  int getLegCount(); // declared in class
private:
  int mLegCount;
};
Chair::Chair(int legs) // implemented in class:: scope
  : mLegCount(legs) {  // access to instance data members
}
int Chair::getLegCount() { // implemented in class:: scope
  return mLegCount; // access to instance data members
}
int main() {
  Chair stool(3); // called on instance
  std::cout << stool.getLegCount() << std::endl;
  return 0;
}
```

Class Methods

- Also called static member functions.
- There is no `this` pointer in static methods.
- No access to instance data members.
- Access to static data members.
- May be called on an object of the class.
- May be called on class scope.
- Declared in class declaration.

Example Static Methods

```
class Chair {
public:
  Chair(int legs);
  ~Chair();
  static int getChairCount(); // declared
private:
  static int sChairCount;
};


...
int Chair::getChairCount() { // implemented in class scope
  return sChairCount;
}
int main() {
  Chair stool(3);
  std::cout << Chair::getChairCount() << std::endl;
  std::cout << stool.getChairCount() << std::endl;
  return 0;
}
```

Downloads

static members examples

Factory Pattern

Example Object Creation (Ugly)

```
int user_choice;
std::string name;
std::cout << "Animal type? (1 == Dog, 2 == Fish) ";
std::cin >> user_choice;
std::cout << "Animal name? ";
std::cin >> name;

std::shared_ptr<Animal> animal;
if(user_choice == 1) {
  animal = std::make_shared<Dog>(name);
} else if(user_choice == 2) {
  animal = std::make_shared<Fish>(name);
} else {
  std::cout << "Unknown type: " << user_choice << std::endl;
}
```

Purpose

Positive Features of Factory Pattern

- Decoupling - client code is simpler
- Abstraction - client code doesn’t need details
- Flexibility - easily add new classes
- Reusability - can use same factory in different contexts

Purpose

Positive Features of Factory Pattern

- Decoupling - client code is simpler
- Abstraction - client code doesn’t need details
- Flexibility - easily add new classes
- Reusability - can use same factory in different contexts

Why Use Factory Pattern

- Instantiate from class hierarchy
- Hide complexity
- Centralize instantiation

Example Factory Use

```
std::string animal_type;
std::string animal_name;
std::cout << "Animal type? ";
std::cin >> animal_type;
std::cout << "Animal name? ";
std::cin >> animal_name;

std::shared_ptr<Animal> animal =
     AnimalFactory::create(animal_type, animal_name);
if(!animal) {
  std::cout << "Unknown animal type: "
      << animal_type << std::endl;
}
```

Example Factory Declaration

```
class AnimalFactory {
public:
  // enumerate unique named values
  enum AnimalId { A_DOG, A_FISH, A_ERROR };
  const static std::vector<std::string> AnimalName;

  static std::unique_ptr<Animal> create(const AnimalId id,
                                        const std::string& name);
  static std::unique_ptr<Animal> create(const std::string& id,
                                        const std::string& name);
  static AnimalId stringToAnimalId(const std::string& id);
  static bool validStringId(const std::string& id);

protected:
private:
};
```

Example Factory Implementation

```
const std::vector<std::string> AnimalFactory::AnimalName = {
  "dog", "fish", "error"
};
```

Example Factory Implementation (Hide the Ugly)

```
std::unique_ptr<Animal> AnimalFactory::create(const AnimalId id,
                                              const std::string& name) {
  std::unique_ptr<Animal> p;
  switch(id) {
  case A_DOG:
    p = std::make_unique<Dog>(name);
    break;
  case A_FISH:
    p = std::make_unique<Fish>(name);
    break;
  case A_ERROR:
    // fall through
  default:
    p = nullptr;
    std::cout << "Unknown animal id '" << id << "'." << std::endl;
    break;
  }
  return p;
}
```

Example Factory Implementation

```
AnimalFactory::AnimalId AnimalFactory::stringToAnimalId(
                                       const std::string& id) {
  int a_id;
  for(a_id = A_DOG; a_id < A_ERROR; a_id++) {
    if(id == AnimalName[a_id]) {
      break;
    }
  }
  return static_cast<AnimalId>(a_id);
}
```

Example Factory Implementation

```
bool AnimalFactory::validStringId(const std::string& id) {
  return stringToAnimalId(id) != A_ERROR;
}
```

Example Factory Implementation

```
std::unique_ptr<Animal> AnimalFactory::create(
      const std::string& id, const std::string& name) {
  AnimalId id_num = stringToAnimalId(id);
  return create(id_num, name);
}
```

Downloads

factory example

Summary

Review

- `enum`
- `static` members
- Factories