22. Classes

CPSC 120: Introduction to Programming
Pratishtha Soni~ CSU Fullerton

Agenda

0. Remind
    a. 120A Exam, Dec 11 and 13 (finals week)
    b. 120L Portfolio due Dec 6 (week 15)
1. Technical Q&A
2. Object-Oriented Concepts
3. Class Syntax
4. Encapsulation and Accessor Functions

1. Technical Q&A

Technical Q&A

Let’s hear your noted questions about...

- This week’s Lab
- Linux
- Any other technical issues

Reminder: write these questions in your notebook during lab

2. Object-Oriented Concepts

Concept of an Object

- Object
  - has data (variables)
  - has functions
  - represents a domain thing
- Example:
  - `std::string`: some text
  - `Card`: a playing card in a Blackjack hand
  - `Color`: a RGB color in an image

An Object Combines Data and Functions

- So far, these are separate
  - variables
  - functions
- An **object** has both
- Object **variables** = properties of a thing = **nouns**
- Object **member functions** = actions a thing can do = **verbs**

Classes and Instances

- **Class:** category of objects
  - Like class of ship
- A class is a data type
- An object is an **instance** of a class
- Each object has its own copy of all the class data members

Nimitz-class aircraft carrier,
image source: Wikipedia

```
int main(int argc, char* argv[]) {
 std::string word1{"the"};
 std::string word2{"of"};
 return 0;
}
```

word1

“the”

word2

“of”

Some Classes We Have Been Using

- std::vector
- std::string
- std::ifstream

3. Class Syntax

Modeling Domain Concepts with Objects

- A class should represent one domain concept
- Best to be focused/simple/small
- Designing classes is an art
- See *CPSC 462 - Software Design*
- A class has
  - A **member variable** for each domain property
  - A **member function** for each domain action

Domain Concept: Word Frequency

- **frequency:** how many times something occurs
- Ilya Semenov's frequency data from Wikipedia: https://github.com/IlyaSemenov/wikipedia-word-frequency
- Start of enwiki-2022-08-29.txt:

```
the 183212978
of 86859699
in 75290639
and 74708386
a 53698262
to 52250362
was 32540285
is 23812199
on 21691194
for 21634075
as 21126503
with 18605836
...
```

WordFrequency Class Declaration (.h file)

```
class WordFrequency {
public:
 WordFrequency(const std::string& word, int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

WordFrequency Class Usage

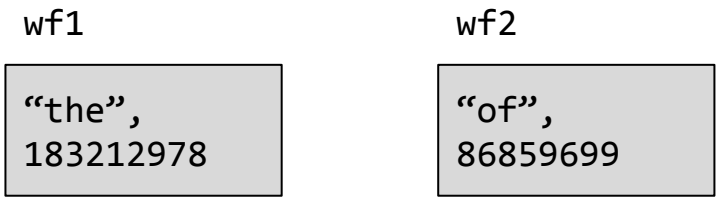
```
int main(int argc, char* argv[]) {
 WordFrequency wf1{"the", 183212978};
 WordFrequency wf2{"of", 86859699};

 std::cout << wf1.Word() << " " << wf1.Frequency() << "\n";
 std::cout << wf2.Word() << " " << wf2.Frequency() << "\n";

 return 0;
}
```

Output:

```
the 183212978
of 86859699
```



Syntax: Google-Style Class Declaration (.h file)

declaration:

class *identifier* **{**
public:
member-function-declaration...

private:
data-member-declaration...
};

Semantics:

- Creates *identifier* as a class data type
- Class has data members and member functions that are declared

```
class WordFrequency {
public:
 WordFrequency(const std::string& word,
               int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

Syntax: Data Member Declaration (.h file)

data-member-declaration:

data-type identifier;

(same pattern as a variable declaration)

Semantics:

- Each object has its own copy of all the class data members

```
class WordFrequency {
public:
 WordFrequency(const std::string& word,
               int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

Syntax: Member Function Declaration (.h file)

member-function-declaration:

return-type identifier(*parameter...*);

(same pattern as regular function declaration)

Semantics:

- The class has a member function with this prototype
- The function must be defined later, usually in a .cc file

```
class WordFrequency {
public:
 WordFrequency(const std::string& word,
               int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

Syntax: Member Function Definition (.cc file)

member-function-definition:

return-type class::*function*(*parameter...*) {
body-statement... }

(same as regular function definition, plus *class*::)

Semantics:

- The member function is defined

```
const std::string& WordFrequency::Word()
const {
 return word_;
}

int WordFrequency::Frequency() const {
 return frequency_;
}
```

Data Members are in Scope

- Data members are in scope in member functions
- The purpose of member functions is to manipulate data members

```
const std::string& WordFrequency::Word()
const {
 return word_;
}

int WordFrequency::Frequency() const {
 return frequency_;
}
```

Constructors

- **Constructor:** function that is called to initialize a class instance when it is created
- Function name is class name
- No return type
- Ex. the `WordFrequency` function is the constructor of the `WordFrequency` class

```
class WordFrequency {
public:
 WordFrequency(const std::string& word,
               int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

WordFrequency Class Usage (.cc file)

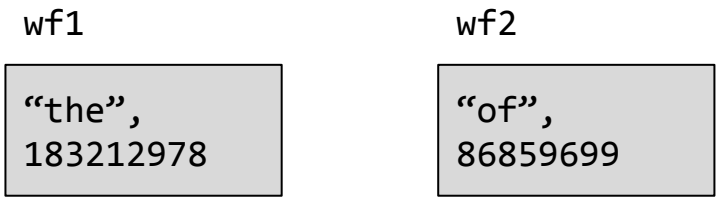
```
int main(int argc, char* argv[]) {
 WordFrequency wf1{"the", 183212978};
 WordFrequency wf2{"of", 86859699};

 std::cout << wf1.Word() << " " << wf1.Frequency() << "\n";
 std::cout << wf2.Word() << " " << wf2.Frequency() << "\n";

 return 0;
}
```

Output:

```
the 183212978
of 86859699
```



Syntax: Constructor Definition (.cc file)

constructor-definition:

class::*class*(*parameter...*) : *initializer...* {
body-statement... }

(same as regular function definition, plus *class*::, minus return type)

Semantics:

- The constructor is defined

```
WordFrequency::WordFrequency(
    const std::string& word,
    int frequency)
: word_(word), frequency_(frequency) { }
```

Syntax: Initializer (.cc file)

initializer:

member-variable(*expresssion*)

Semantics:

- *member-variable* is initialized to *expression*
- Happens **before** the body of the constructor
- Standard way to initialize the member variables

```
WordFrequency::WordFrequency(
    const std::string& word,
    int frequency)
: word_(word), frequency_(frequency) { }
```

Initializer Expression May Be a Constant

- Value in initializer is an **expression**
- Recall **expression** may be
  - variable name
  - literal value
  - arithmetic expression
- Initializer may initialize a variable to a constant literal value

```
// different definition
WordFrequency::WordFrequency(
    const std::string& word)
: word_(word), frequency_(0) { }
```

Valid vs Invalid State

- **State** (of an object): values in all data members
- **Valid state:** all data members initialized and object ready to go
- **Invalid state:** object uninitialized, or not ready
- Example: `ifstream::good()`

Resource Allocation Is Initialization (RAII)

- Resource Allocation Is Initialization (RAII): every variable is bound to a valid object
- Method for eliminating resource management bugs
- Means that objects
  - start valid
  - remain valid
- Constructor is responsible for initializing an object to a valid state
  - Usually means every data member must be initialized
- Every member function must preserve valid state
  - Throw an exception if impossible

Naming Conventions

- https://google.github.io/styleguide/cppguide.html#General_Naming_Rules
- class names: `CamelCase`
- member functions: `CamelCase`
- data members: `lower_case_with_underscore_at_end_`

Why Data Members End in Underscore

- **Problem**:
  - constructor needs to initialize every data member
  - constructor argument needs name
  - data member needs name
  - cannot be the same
- Either the constructor argument, or data member, needs to be different somehow
- Solution:
  - **be kind**
  - let your user have the nicer name
  - constructor parameter named normally
  - data member has extra underscore

Why Data Members End in Underscore

```
...
private:
 std::string word_;
 int frequency_;
};

WordFrequency::WordFrequency(
   const std::string& word,
   int frequency)
: word_(word), frequency_(frequency) { }
```

```
...
private:
 std::string word;
 int frequency;
};

WordFrequency::WordFrequency(
   const std::string& word_,
   int frequency_)
: word(word_), frequency(frequency_) { }
```

3. Encapsulation and Accessor Functions

Encapsulation

- Encapsulation: restricting access to class members
- Prevent bugs from class users tampering with data members
- Make it easy to change class later
- Principle of least privilege: only allow data members to be used in approved ways

Access Specifiers

- Access specifier: defines accessibility of subsequent class members
- **public**
- **private**

```
class WordFrequency {
public:
 WordFrequency(const std::string& word,
               int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

public

- **public** access specifier: member is available everywhere
- in member functions and constructors
- outside of member functions

```
class WordFrequency {
public:
 WordFrequency(const std::string& word,
               int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

private

- **private** access specifier: member is available **only** in member functions and constructors
- Not available outside of class functions

```
class WordFrequency {
public:
 WordFrequency(const std::string& word,
               int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

private members are inaccessible

```
class WordFrequency {
public:
 WordFrequency(const std::string& word,
               int frequency);

 const std::string& Word() const;
 int Frequency() const;

private:
 std::string word_;
 int frequency_;
};
```

```
int main(int argc, char* argv[]) {
 WordFrequency wf1{"the", 183212978};

 wf1.frequency_ = 0;

 return 0;
}
```

```
23.cc:37:7: error: 'frequency_' is a private member
of 'WordFrequency'
  wf1.frequency_ = 0;
      ^
23.cc:14:7: note: declared private here
  int frequency_;
      ^
1 error generated.
```

Accessors and Mutators

- **Accessor:** member function that gives **access** to a data member (read)
- **Mutator:** member function that **changes** a data member (write)
- Necessary because data members are private

Accessors

- **Accessor**: returns read-only version of a data member
- Only exists for data members that users ought to see
- Return type is
  - same for primitive types (`int`, `double`, `bool`)
  - `const&` reference for class types (`string`, `vector`, classes)

```
const std::string& WordFrequency::Word()
const {
 return word_;
}

int WordFrequency::Frequency() const {
 return frequency_;
}
```

Mutators

- **Mutator:** function that changes a data member
- `Set`*`Variable`*`(...)`
- Parameter data type works like accessor return type
  - primitive types use same type
  - compound types use `const&` reference

```
void WordFrequency::SetWord(const std::string& word) {
 word_ = word;
}

void WordFrequency::SetFrequency(int frequency) {
 frequency_ = frequency;
}
```

Review: Resource Allocation Is Initialization (RAII)

- Resource Allocation Is Initialization (RAII): every variable is bound to a valid object
- Method for eliminating resource management bugs
- Means that objects
  - start valid
  - remain valid
- Constructor is responsible for initializing an object to a valid state
  - Usually means every data member must be initialized
- Every member function must preserve valid state
  - Throw an exception if impossible

Mutators Maintain Valid State

- Recall RAII rule: objects must maintain valid state
- Mutator function body can prevent invalid data member values
- Would not be possible if data members were public

```
void WordFrequency::SetFrequency(int frequency) {
 if (frequency <= 0) {
   throw std::invalid_argument("frequency must be positive");
 }
 frequency_ = frequency;
}
```

Prevents something like:

```
WordFrequency wf1{"the", 183212978};
wf1.frequency = -1;
```