Program Organization Principles

CS 115

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

Last updated: January 28, 2025

Terminology concerning program organization, interface vs. implementation, data encapsulation, information hiding, modularity, layering, design by contract, abstract data types

Separation of Concerns

- is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern

Separation of Concerns

- is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern
- concern = a set of information that affects code

Separation of Concerns

- is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern
- concern = a set of information that affects code
- can be realized via layering and modularity

Separation of Concerns

- is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern
- concern = a set of information that affects code
- can be realized via layering and modularity
- Layering: use separate layers in the software, each of which addresses a different concern (e.g., presentation layer, business logic layer, data access layer, etc.)

Separation of Concerns

- is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern
- concern = a set of information that affects code
- can be realized via layering and modularity
- Layering: use separate layers in the software, each of which addresses a different concern (e.g., presentation layer, business logic layer, data access layer, etc.)
- **Modularity:** the degree to which a system's components can be separated and recombined

Separation of Concerns

- is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern
- concern = a set of information that affects code
- can be realized via layering and modularity
- Layering: use separate layers in the software, each of which addresses a different concern (e.g., presentation layer, business logic layer, data access layer, etc.)
- **Modularity:** the degree to which a system's components can be separated and recombined
- break system into parts and to hide the complexity of each part behind an abstraction and interface

Modularity

- Why bother?

Modularity

- Why bother?
- Simplifies development and maintenance of computer programs

Modularity

- Why bother?
- Simplifies development and maintenance of computer programs
- Promote software reuse

Modularity

- Why bother?
- Simplifies development and maintenance of computer programs
- Promote software reuse
- Modules can be developed and updated independently (can improve on one section of code without changing other sections)

Modularity

- Why bother?
- Simplifies development and maintenance of computer programs
- Promote software reuse
- Modules can be developed and updated independently (can improve on one section of code without changing other sections)
- How to realize modularity?

Modularity

- Why bother?
- Simplifies development and maintenance of computer programs
- Promote software reuse
- Modules can be developed and updated independently (can improve on one section of code without changing other sections)
- How to realize modularity?
    - procedural programming: via functions and top-down design

Modularity

- Why bother?
- Simplifies development and maintenance of computer programs
- Promote software reuse
- Modules can be developed and updated independently (can improve on one section of code without changing other sections)
- How to realize modularity?
    - procedural programming: via functions and top-down design
- OOP: via classes and objects

Modularity and Refactoring

- **Refactoring** is to rewrite code in order to improve its readability, reusability, or structure without affecting its meaning or behaviour

Modularity and Refactoring

- **Refactoring** is to rewrite code in order to improve its readability, reusability, or structure without affecting its meaning or behaviour
- Perhaps older version was poorly written due to time constraints etc.

Modularity and Refactoring

- **Refactoring** is to rewrite code in order to improve its readability, reusability, or structure without affecting its meaning or behaviour
- Perhaps older version was poorly written due to time constraints etc.
    - e.g., replace `306` with the constant `SK_CODE1`

Modularity and Refactoring

- **Refactoring** is to rewrite code in order to improve its readability, reusability, or structure without affecting its meaning or behaviour
- Perhaps older version was poorly written due to time constraints etc.
  - e.g., replace `306` with the constant `SK_CODE1`
  - replace long if-then-else branches with switch/case statements

Modularity and Refactoring

- **Refactoring** is to rewrite code in order to improve its readability, reusability, or structure without affecting its meaning or behaviour
- Perhaps older version was poorly written due to time constraints etc.
    - e.g., replace `306` with the constant `SK_CODE1`
    - replace long if-then-else branches with switch/case statements
    - divide overly complex implementation into smaller functions

Modularity and Refactoring

- **Refactoring** is to rewrite code in order to improve its readability, reusability, or structure without affecting its meaning or behaviour
- Perhaps older version was poorly written due to time constraints etc.
    - e.g., replace `306` with the constant `SK_CODE1`
    - replace long if-then-else branches with switch/case statements
    - divide overly complex implementation into smaller functions
    - replace with efficient code, etc.

Layers of Abstraction

- Each level represents an increasingly detailed model of the software system and its processes

Layers of Abstraction

- Each level represents an increasingly detailed model of the software system and its processes
- at each level, the model is described using concepts appropriate to a certain domain

Layers of Abstraction

- Each level represents an increasingly detailed model of the software system and its processes
- at each level, the model is described using concepts appropriate to a certain domain
- each higher, more abstract level builds on a lower, less abstract level

Layers of Abstraction

- Each level represents an increasingly detailed model of the software system and its processes
- at each level, the model is described using concepts appropriate to a certain domain
- each higher, more abstract level builds on a lower, less abstract level
- To understand levels of abstraction better, see optional slides on Layering

Interface vs. Implementation

- **Interface:** How to use your code (type signature, precondition, postcondition, description of return value)

Interface vs. Implementation

- **Interface:** How to use your code (type signature, precondition, postcondition, description of return value)
    - Modular programming: developing software where each section of code is a module with a carefully specified interface

Interface vs. Implementation

- **Interface:** How to use your code (type signature, precondition, postcondition, description of return value)
    - Modular programming: developing software where each section of code is a module with a carefully specified interface
    - makes the purpose of your code clear

Interface vs. Implementation

- **Interface:** How to use your code (type signature, precondition, postcondition, description of return value)
    - Modular programming: developing software where each section of code is a module with a carefully specified interface
    - makes the purpose of your code clear
    - client software can focus on the interface

Interface vs. Implementation

- **Interface:** How to use your code (type signature, precondition, postcondition, description of return value)
    - Modular programming: developing software where each section of code is a module with a carefully specified interface
    - makes the purpose of your code clear
    - client software can focus on the interface
        - *and ignore its implementation*

Interfaces ctd.

- A crucial aspect of modular programming is mentally separating the interface from the implementation

Interfaces ctd.

- A crucial aspect of modular programming is mentally separating the interface from the implementation
    - Do you know how `cin` and `cout` are implemented?

Interfaces ctd.

- A crucial aspect of modular programming is mentally separating the interface from the implementation
    - Do you know how `cin` and `cout` are implemented?
    - You don’t need to know to use them

Interfaces ctd.

- A crucial aspect of modular programming is mentally separating the interface from the implementation
    - Do you know how `cin` and `cout` are implemented?
    - You don’t need to know to use them
- We will specify the interfaces in .h files (as well-documented prototypes)

Interfaces ctd.

- A crucial aspect of modular programming is mentally separating the interface from the implementation
  - Do you know how `cin` and `cout` are implemented?
  - You don't need to know to use them
- We will specify the interfaces in .h files (as well-documented prototypes)
- We will specify the implementation in .cpp files (primarily as functions)

Interfaces ctd.

- A crucial aspect of modular programming is mentally separating the interface from the implementation
    - Do you know how `cin` and `cout` are implemented?
    - You don't need to know to use them
- We will specify the interfaces in .h files (as well-documented prototypes)
- We will specify the implementation in .cpp files (primarily as functions)
- Some functions and variables are not (directly) accessible!

Separating interface and implementation

- Two Approaches

Separating interface and implementation

- Two Approaches
    - via data encapsulation

Separating interface and implementation

- Two Approaches
    - via data encapsulation
        - hide variables describing state of the module inside the module

Separating interface and implementation

- Two Approaches
    - via data encapsulation
        - hide variables describing state of the module inside the module
        - (static variables/functions and namespaces)

Separating interface and implementation

- Two Approaches
    - via data encapsulation
        - hide variables describing state of the module inside the module
        - (static variables/functions and namespaces)
    - by defining new abstract data types (ADT) using records and classes

The Static Keyword

- On global variables and functions

The Static Keyword

- On global variables and functions

The Static Keyword

- On global variables and functions

The Static Keyword

- On global variables and functions

The Static Keyword

- On global variables and functions

```
// whatever.cpp

static int foo = 5;
int bar = 6;

static void doh(int var1) {
  // do something
}

void yay(char c){
  // do something
}
```

The Static Keyword

- On global variables and functions

```cpp
// whatever.cpp

static int foo = 5;
int bar = 6;

static void doh(int var1) {
  // do something
}

void yay(char c){
  // do something
}
```

The Static Keyword

- On global variables and functions

```cpp
// whatever.cpp

static int foo = 5;
int bar = 6;

static void doh(int var1) {
  // do something
}

void yay(char c){
  // do something
}
```

```cpp
// main.cpp

int main ( ){

  extern int foo; // invalid
  extern int bar; // works!

  doh(13); // invalid
  yay('a'); // works!

}
```

Local Variables and static

Local Variables and static

Local Variables and static

Local Variables and static

```
void fun(int var1) {
  int x1=0;
  x1+=var1;
  cout << x1 << endl;
}
void funS(int var1) {
  static int x2=0;
  x2+=var1;
  cout << x2 << endl;
}
int main ( ){
  fun(5);
  fun(5);
  fun(7);

  funS(5);
  funS(5);
  funS(7);
}
```

```
5
5
7
5
10
17
```

Local Variables and static

```
void fun(int var1) {
  int x1=0;
  x1+=var1;
  cout << x1 << endl;
}
void funS(int var1) {
  static int x2=0;
  x2+=var1;
  cout << x2 << endl;
}
int main ( ){
  fun(5);
  fun(5);
  fun(7);

  funS(5);
  funS(5);
  funS(7);
}
```

- Variable value persists across multiple calls to the function

```
5
5
7
5
10
17
```

Local Variables and static

```
void fun(int var1) {
  int x1=0;
  x1+=var1;
  cout << x1 << endl;
}
void funS(int var1) {
  static int x2=0;
  x2+=var1;
  cout << x2 << endl;
}
int main ( ){
  fun(5);
  fun(5);
  fun(7);

  funS(5);
  funS(5);
  funS(7);
}
```

- Variable value persists across multiple calls to the function
  - Like a global, but can only be accessed from inside the function

```
5
5
7
5
10
17
```

Local Variables and static

```cpp
void fun(int var1) {
  int x1=0;
  x1+=var1;
  cout << x1 << endl;
}
void funS(int var1) {
  static int x2=0;
  x2+=var1;
  cout << x2 << endl;
}
int main ( ){
  fun(5);
  fun(5);
  fun(7);

  funS(5);
  funS(5);
  funS(7);
}
```

- Variable value persists across multiple calls to the function
  - Like a global, but can only be accessed from inside the function
  - So other things can't mess it up!

```
5
5
7
5
10
17
```

Namespaces

Namespaces

Namespaces

- Scope for identifiers

Namespaces

- Scope for identifiers
- Avoids name collisions

Namespaces

- Scope for identifiers
- Avoids name collisions
- Makes it clear where a name is coming from

Namespaces

- Scope for identifiers
- Avoids name collisions
- Makes it clear where a name is coming from

Namespaces

- Scope for identifiers
- Avoids name collisions
- Makes it clear where a name is coming from

```
// myProg.h

#pragma once

namespace myNSpace{
  void Foo();
  int Bar();
}
```

Namespaces

- Scope for identifiers
- Avoids name collisions
- Makes it clear where a name is coming from

```
// myProg.h

#pragma once

namespace myNSpace{
  void Foo();
  int Bar();
}
```

Namespaces

- Scope for identifiers
- Avoids name collisions
- Makes it clear where a name is coming from

```
// myProg.h

#pragma once

namespace myNSpace{
  void Foo();
  int Bar();
}
```

```
#include "myProg.h"
using namespace myNSpace;

// use fully-qualified name here
void myNSpace::Foo(){
  // no qualification needed for Bar()
  Bar();
}

int ContosoDataServer::Bar(){
  return 0;
}
```

Anonymous namespaces

Anonymous namespaces

Anonymous namespaces

- Used for hiding identifiers

Anonymous namespaces

- Used for hiding identifiers

Anonymous namespaces

- Used for hiding identifiers

```
// myProg.h

#pragma once

namespace {
  float foo;
  double pi(){
    return 3.141592653;
  }
}

char bar;
```

Anonymous namespaces

- Used for hiding identifiers

```cpp
// myProg.h

#pragma once

namespace {
  float foo;
  double pi(){
    return 3.141592653;
  }
}

char bar;
```

Anonymous namespaces

- Used for hiding identifiers

```
// myProg.h

#pragma once

namespace {
  float foo;
  double pi(){
    return 3.141592653;
  }
}

char bar;
```

```
// myProg.cpp

#include "myProg.h"

int main(){
  foo = 2.718281828; // invalid!
  double y = pi();          // invalid!
  char c = bar;             // works

  return 0;
}
```

Other Namespace Issues

- Can declare the same namespace over multiple sections

Other Namespace Issues

- Can declare the same namespace over multiple sections
- Have to be careful about usage of identifiers

Other Namespace Issues

- Can declare the same namespace over multiple sections
- Have to be careful about usage of identifiers
- Can have nested namespaces, inline namespaces, namespace aliases, etc.

Other Namespace Issues

- Can declare the same namespace over multiple sections
- Have to be careful about usage of identifiers
- Can have nested namespaces, inline namespaces, namespace aliases, etc.
- Also check out the global namespace

Data encapsulation

- to place a barrier around the variables that represent the internal state of a software component so that these variables cannot be accessed directly by client code

Data encapsulation

- to place a barrier around the variables that represent the internal state of a software component so that these variables cannot be accessed directly by client code
- can be achieved via static variables

Data encapsulation

- to place a barrier around the variables that represent the internal state of a software component so that these variables cannot be accessed directly by client code
- can be achieved via static variables
- (restricts variable/function scope to file)

Data encapsulation

- to place a barrier around the variables that represent the internal state of a software component so that these variables cannot be accessed directly by client code
- can be achieved via static variables
- (restricts variable/function scope to file)
- hides implementation details

Data encapsulation

- to place a barrier around the variables that represent the internal state of a software component so that these variables cannot be accessed directly by client code
- can be achieved via static variables
- (restricts variable/function scope to file)
- hides implementation details
- clients are forced to use interface to access data

Data encapsulation

- to place a barrier around the variables that represent the internal state of a software component so that these variables cannot be accessed directly by client code
- can be achieved via static variables
- (restricts variable/function scope to file)
- hides implementation details
- clients are forced to use interface to access data
- similar effects can be achieved using namespaces

Separating interface and implementation

- e.g. A Bounded Counter

Separating interface and implementation

- e.g. A Bounded Counter
- Start by specifying the interface of the module

Separating interface and implementation

- e.g. A Bounded Counter
- Start by specifying the interface of the module

Separating interface and implementation

- e.g. A Bounded Counter
- Start by specifying the interface of the module

```
// initializeCounter
//
// Purpose: Initialize the bounded counter module.
// Parameter(s):
//  <1> value1: Initial value for the counter
//     expressed as an unsigned integer.
//  <2> upper1: Upper bound for counter value
//     expressed as an unsigned integer.
// Precondition(s): value1 < upper1
// Returns: N/A
// Side effect: The counter is initialized, with value 1
//    the current counter value, and upper1 as the
// upper bound of counter values.
```

Separating interface and implementation

Separating interface and implementation

```
// getCounterValue
//
// Purpose: Retrieve the current value of
// the counter.
// Parameter(s): N/A
// Precondition(s): N/A
// Returns: The unsigned integer value of
// the counter.
// Side effect: N/A

// incrementCounter
//
// Purpose: Increment the value of the
// counter.
// Parameter(s): N/A
// Precondition(s): N/A
// Returns: N/A
// Side effect: The counter value is
// incremented by one. If the incremented
// value reaches the upper bound, then the
```

Complete Interface

Complete Interface

```
// encapsulated_counter.h
//
// This module provides ...
// Data encapsulation is used to
// protect the state of the bounded
// counter from manipulation by client
// code, except via the functions in
// the interface.


#pragma once
//initializeCounter
//...
void initializeCounter(unsigned int value1, unsigned int upper1);
// getCounterValue
//...
unsigned int getCounterValue();
// incrementCounter
//...
void incrementCounter();
```

Client Code

Client Code

```cpp
#include "encapsulated_counter.h"

int main() {
  initializeCounter(0, 3);
  cout << getCounterValue() << endl;
  incrementCounter();
  cout << getCounterValue() << endl;
  incrementCounter();
  incrementCounter();
  cout << getCounterValue() << endl;
  return 0;
}
```

- Output:

Client Code

```
#include "encapsulated_counter.h"

int main() {
  initializeCounter(0, 3);
  cout << getCounterValue() << endl;
  incrementCounter();
  cout << getCounterValue() << endl;
  incrementCounter();
  incrementCounter();
  cout << getCounterValue() << endl;
  return 0;
}
```

- Output:
  - `0`

Client Code

```cpp
#include "encapsulated_counter.h"

int main() {
  initializeCounter(0, 3);
  cout << getCounterValue() << endl;
  incrementCounter();
  cout << getCounterValue() << endl;
  incrementCounter();
  incrementCounter();
  cout << getCounterValue() << endl;
  return 0;
}
```

- Output:
    - `0`
    - `1`

Client Code

```
#include "encapsulated_counter.h"

int main() {
  initializeCounter(0, 3);
  cout << getCounterValue() << endl;
  incrementCounter();
  cout << getCounterValue() << endl;
  incrementCounter();
  incrementCounter();
  cout << getCounterValue() << endl;
  return 0;
}
```

- Output:
  - `0`
  - `1`
  - `0`

Implementing the Interface

Implementing the Interface

```cpp
// encapsulated_counter.cpp
//
static unsigned int counter_value;
static unsigned int counter_upper;

void initializeCounter(unsigned int value1, unsigned int upper1) {
  counter_value = value1;
  counter_upper = upper1;
}

unsigned int getCounterValue(){
  return counter_value;
}
void incrementCounter(){
  ++counter_value;
  if (counter_value == counter_upper)
    counter_value = 0;
}
```

Design by contract

- Allows for clean division of labour

Design by contract

- Allows for clean division of labour
- Specifies the usage convention for a module is captured in a contract between the supplier (the developer of the module) and the client (the user of the module)

Design by contract

- Allows for clean division of labour
- Specifies the usage convention for a module is captured in a contract between the supplier (the developer of the module) and the client (the user of the module)
- Protects all parties by specifying

Design by contract

- Allows for clean division of labour
- Specifies the usage convention for a module is captured in a contract between the supplier (the developer of the module) and the client (the user of the module)
- Protects all parties by specifying
    - supplier's POV: how little is acceptable

Design by contract

- Allows for clean division of labour
- Specifies the usage convention for a module is captured in a contract between the supplier (the developer of the module) and the client (the user of the module)
- Protects all parties by specifying
    - supplier’s POV: how little is acceptable
    - Client’s POV: how much is expected

Design by contract

- Allows for clean division of labour
- Specifies the usage convention for a module is captured in a contract between the supplier (the developer of the module) and the client (the user of the module)
- Protects all parties by specifying
    - supplier’s POV: how little is acceptable
    - Client’s POV: how much is expected
- Usually specified using

Design by contract

- Allows for clean division of labour
- Specifies the usage convention for a module is captured in a contract between the supplier (the developer of the module) and the client (the user of the module)
- Protects all parties by specifying
  - supplier’s POV: how little is acceptable
  - Client’s POV: how much is expected
- Usually specified using
  - preconditions

Design by contract

- Allows for clean division of labour
- Specifies the usage convention for a module is captured in a contract between the supplier (the developer of the module) and the client (the user of the module)
- Protects all parties by specifying
    - supplier’s POV: how little is acceptable
    - Client’s POV: how much is expected
- Usually specified using
    - preconditions
    - postconditions

Design by contract

- Allows for clean division of labour
- Specifies the usage convention for a module is captured in a contract between the supplier (the developer of the module) and the client (the user of the module)
- Protects all parties by specifying
    - supplier’s POV: how little is acceptable
    - Client’s POV: how much is expected
- Usually specified using
    - preconditions
    - postconditions
    - invariants

Design By Contract in our Counter

Design By Contract in our Counter

```
// initializeCounter
//
// Purpose: Initialize the bounded counter module.
// Parameter(s):
//  <1> value1: Initial value for the counter
//    expressed as an unsigned integer.
//  <2> upper1: Upper bound for counter value
//    expressed as an unsigned integer.
// Precondition(s):
//  <1>: value1 < upper1
// Returns: N/A
// Side Effect: The global counter is initialized, with value1 as
//              the current counter value, and upper1 as the upper
//              bound of counter values.
```

Preconditions and Posconditions

Preconditions and Posconditions

```
// encapsulated_counter.cpp
#include <cassert>

void initializeCounter(unsigned int value1, unsigned int upper1){
  assert(value1 < upper1);  // encapsulated_counter.cpp
  counter_value = value1;
  counter_upper = upper1;
}
```

Invariants

Invariants

```
// initializeCounter
//
// Module invariant: Current counter value is
//  always strictly less than the upper bound
//

static bool isInvariantTrue(){
  return counter_value < counter_upper;
}
```

Invariants ctd.

Invariants ctd.

```
void initializeCounter(unsigned int value1, unsigned int upper1){
  assert(value1 < upper1);
  counter_value = value1;
  counter_upper = upper1;
  assert(isInvariantTrue());
}
unsigned int getCounterValue(){
  assert(isInvariantTrue());
  return counter_value;
}
void incrementCounter(){
  assert(isInvariantTrue());
  ++counter_value;
  if (counter_value == counter_upper)
    counter_value = 0;
  assert(isInvariantTrue());
}
```

Another Example (see the notes)

- Consider designing a timer that represents the accumulated time in [hh:mm:ss] format

Another Example (see the notes)

- Consider designing a timer that represents the accumulated time in [hh:mm:ss] format
- Internally can be implemented in many ways

Another Example (see the notes)

- Consider designing a timer that represents the accumulated time in [hh:mm:ss] format
- Internally can be implemented in many ways
- e.g., only store seconds

Another Example (see the notes)

- Consider designing a timer that represents the accumulated time in [hh:mm:ss] format
- Internally can be implemented in many ways
- e.g., only store seconds
- e.g., store all hours, minutes, and seconds

Another Example (see the notes)

- Consider designing a timer that represents the accumulated time in [hh:mm:ss] format
- Internally can be implemented in many ways
- e.g., only store seconds
- e.g., store all hours, minutes, and seconds
- But if interface remains the same, changing implementation does not require changing client code

Abstract data types (ADT)

- Motivation: returning to our example, we want to have multiple counters

Abstract data types (ADT)

- Motivation: returning to our example, we want to have multiple counters
- ADT: data type defined by its possible values and operations, e.g.: counters

Abstract data types (ADT)

- Motivation: returning to our example, we want to have multiple counters
- ADT: data type defined by its possible values and operations, e.g.: counters

Abstract data types (ADT)

- Motivation: returning to our example, we want to have multiple counters
- ADT: data type defined by its possible values and operations, e.g.: counters

```
// counter.h
//
// This module defines an abstract data type named Counter.
// A counter value is maintained by
// each instance of the Counter type.
// Users may increment or retrieve the value of the counter.
// Data type invariant: Current value of a counter instance
//  must be strictly smaller than its
// upper bound
struct Counter{
  // ... details to be filled out later
};
```

Abstract data types (ADT)

Abstract data types (ADT)

```
// counterInitialize
//
// Purpose: Initialize a counter instance.
// Parameter(s):
//  <1> counter: A counter instance to be initialized.
//  <2> value1: Initial value for the counter
//    specified as an unsigned integer.
//  <3> upper1: Upper bound for counter value
//    specified as an unsigned integer.
// Precondition:
//  <1> value1 < upper1
// Side Effect: The counter instance is initialized, with value1 as
//              the current counter value, and upper1 as the upper
//              bound of counter values.
//
void counterInitialize(Counter& counter,
                       unsigned int value1,
                       unsigned int upper1);
```

Abstract data types (ADT)

Abstract data types (ADT)

```cpp
// counterGetValue
//
// Purpose: Retrieve the current value of a
// counter instance.
// Parameter(s):
//   <1> counter: A counter instance
// Returns: The unsigned integer value of the
// counter instance.

unsigned counterGetValue(const Counter& counter);
```

Abstract data types (ADT)

Abstract data types (ADT)

```
// counterIncrement
//
// Purpose: Increment a given counter
// instance.
// Parameter(s):
//   <1> counter: counter instance to be
//    incremented
// Side Effect: The counter value of the
// parameter is incremented by one. If the
// incremented value reaches the upper
// bound, then the counter value is reset to
// zero.
void counterIncrement(Counter& counter);
```

Client Code

Client Code

```
int main( ){
  Counter c, d;
  counterInitialize(c, 0, 3);
  counterInitialize(d, 0, 10);
  counterIncrement(c);  counterIncrement(c);  counterIncrement(c);
  counterIncrement(d);  counterIncrement(d);  counterIncrement(d);
  cout << counterGetValue(c) << endl;
  cout << counterGetValue(d) << endl;
  return 0;
}
```

- Outputs

Client Code

```
int main( ){
  Counter c, d;
  counterInitialize(c, 0, 3);
  counterInitialize(d, 0, 10);
  counterIncrement(c);  counterIncrement(c);  counterIncrement(c);
  counterIncrement(d);  counterIncrement(d);  counterIncrement(d);
  cout << counterGetValue(c) << endl;
  cout << counterGetValue(d) << endl;
  return 0;
}
```

- Outputs
  - 0

Client Code

```
int main( ){
  Counter c, d;
  counterInitialize(c, 0, 3);
  counterInitialize(d, 0, 10);
  counterIncrement(c);  counterIncrement(c);  counterIncrement(c);
  counterIncrement(d);  counterIncrement(d);  counterIncrement(d);
  cout << counterGetValue(c) << endl;
  cout << counterGetValue(d) << endl;
  return 0;
}
```

- Outputs
  - 0
  - 3

Data Representation, Implementation, Issues

Data Representation, Implementation, Issues

Data Representation, Implementation, Issues

Data Representation, Implementation, Issues

```c
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before

Data Representation, Implementation, Issues

```
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:

Data Representation, Implementation, Issues

```
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:
  - no data encapsulation

Data Representation, Implementation, Issues

```
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:
    - no data encapsulation
    - no initialization guarantees

Data Representation, Implementation, Issues

```c
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:
  - no data encapsulation
  - no initialization guarantees

Data Representation, Implementation, Issues

```
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:
  - no data encapsulation
  - no initialization guarantees

- No encapsulation

Data Representation, Implementation, Issues

```
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:
    - no data encapsulation
    - no initialization guarantees

- No encapsulation

Data Representation, Implementation, Issues

```
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:
  - no data encapsulation
  - no initialization guarantees

- No encapsulation

```
Counter c;
counterInitialize(c, 0, 3);
c.value = 999; // allowed!
```

- No initialization guarantees

Data Representation, Implementation, Issues

```
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:
  - no data encapsulation
  - no initialization guarantees

- No encapsulation

```
Counter c;
counterInitialize(c, 0, 3);
c.value = 999; // allowed!
```

- No initialization guarantees

Data Representation, Implementation, Issues

```
struct Counter {
  unsigned int value;
  unsigned int upper;
};
```

- Can implement as before
- Problems:
  - no data encapsulation
  - no initialization guarantees

- No encapsulation

```
Counter c;
counterInitialize(c, 0, 3);
c.value = 999; // allowed!
```

- No initialization guarantees

```
// Precondition:
//  <1> The counter module must
// have been properly initialized
Counter c;
cout << counterGetValue(c) << endl;
```