Abstract Data Types via Classes

CS 115

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

Last updated: January 21, 2025

Declaring ADT as classes, data representation, member functions, public vs. private functions, constructors

Counter Example Continued: Interface

- Classes are record types, and thus have fields, but can also declared member functions

```
// counter.h
class Counter {
public:
  // initialize
  void initialize(unsigned int value1, unsigned int upper1);
  // getValue
  unsigned int getValue();
  // increment
  void increment();
private:
  // Data representation to follow ...
};
```

Classes (cont’d)

- Public member functions can be used elsewhere

Classes (cont’d)

- Public member functions can be used elsewhere
    - how about public static ones?

Classes (cont’d)

- Public member functions can be used elsewhere
    - how about public static ones?
- Private member functions have class scope

Classes (cont'd)

- Public member functions can be used elsewhere
    - how about public static ones?
- Private member functions have class scope
    - (cf. file scope as in static or namespaces)

Classes (cont'd)

- Public member functions can be used elsewhere
    - how about public static ones?
- Private member functions have class scope
    - (cf. file scope as in static or namespaces)
- Note: member functions no longer take the counter as argument; why?

Classes (cont'd)

- Public member functions can be used elsewhere
  - how about public static ones?
- Private member functions have class scope
  - (cf. file scope as in static or namespaces)
- Note: member functions no longer take the counter as argument; why?
  - `void initialize(unsigned int value1, unsigned int upper1)`

Classes (cont'd)

- Public member functions can be used elsewhere
  - how about public static ones?
- Private member functions have class scope
  - (cf. file scope as in static or namespaces)
- Note: member functions no longer take the counter as argument; why?
  - `void initialize(unsigned int value1, unsigned int upper1)`
- Public vs. private fields/member functions of a class

Classes (cont'd)

- Public member functions can be used elsewhere
    - how about public static ones?
- Private member functions have class scope
    - (cf. file scope as in static or namespaces)
- Note: member functions no longer take the counter as argument; why?
    - `void initialize(unsigned int value1, unsigned int upper1)`
- Public vs. private fields/member functions of a class
    - how to call/invoke public member functions?

Classes (cont'd)

- Public member functions can be used elsewhere
  - how about public static ones?
- Private member functions have class scope
  - (cf. file scope as in static or namespaces)
- Note: member functions no longer take the counter as argument; why?
  - `void initialize(unsigned int value1, unsigned int upper1)`
- Public vs. private fields/member functions of a class
  - how to call/invoke public member functions?
  - how to define/implement a member function?

Client Code

Client Code

```
#include "counter.h"

int main( ) {
  Counter c, d;
  c.initialize(0, 3);
  d.initialize(0, 10);

  c.increment();
  c.increment();
  c.increment();
```

Client Code

```
#include "counter.h"

int main( ) {
  Counter c, d;
  c.initialize(0, 3);
  d.initialize(0, 10);

  c.increment();
  c.increment();
  c.increment();
  d.increment();
  d.increment();
  d.increment();

  cout << c.getValue() << endl;
  cout << d.getValue() << endl;

  return 0;
}
```

Data Representation

```
class Counter {
public:
  ... ... ...
private: // encapsulation
  unsigned int value; // current value of the counter
  unsigned int upper; // upper bound of valid counter values
};

int main() {
  Counter c;
  c.initialize(0, 3);
  c.value = 999; // can't access private data, error!
```

Implementing Methods

```cpp
// counter.cpp
#include "counter.h"

void Counter::initialize(unsigned int value1, unsigned int upper1) {
  assert(value1 < upper1);
  value = value1;
  upper = upper1;
}

unsigned int Counter::getValue() {
  return value;
}

void Counter::increment() {
  value++;
  if (value == upper)
    value = 0;
}
//not using Counter:: will make the
//declarations global!
```

Private Member Functions

```cpp
// counter.h

class Counter {
public:
  ... ... ...
private: // encapsulation
  // isInvariantTrue
  bool isInvariantTrue();
};

// counter.cpp
#include "counter.h"

void Counter::initialize(unsigned int value1, unsigned int upper1) {
  assert(value1 < upper1);
  value = value1;
  upper = upper1;
  assert(isInvariantTrue());
}
```

Constructors

- Can declare a class constructor

```
// counter.h
class Counter {
public:
  // Constructor
  // Purpose: Initialize a counter instance
  Counter(unsigned int value1, unsigned int upper1);
  ...
};
```

Constructors

- Can declare a class constructor
    - special kind of member function

```cpp
// counter.h
class Counter {
public:
  // Constructor
  // Purpose: Initialize a counter instance
  Counter(unsigned int value1, unsigned int upper1);
  ...
};
```

Constructors

- Can declare a class constructor
  - special kind of member function
  - automatically invoked when an instance of the class is created

```
// counter.h
class Counter {
public:
  // Constructor
  // Purpose: Initialize a counter instance
  Counter(unsigned int value1, unsigned int upper1);
  ...
};
```

Constructors

- Can declare a class constructor
  - special kind of member function
  - automatically invoked when an instance of the class is created
  - intended to perform initialization (forces to initialize when creating instances!)

```cpp
// counter.h
class Counter {
public:
  // Constructor
  // Purpose: Initialize a counter instance
  Counter(unsigned int value1, unsigned int upper1);
  ...
};
```

The Initialization Guarantee

```cpp
// counter.cpp

Counter::Counter(unsigned int value1, unsigned int upper1){

  assert(value1 < upper1);
  value = value1;
  upper = upper1;
  assert(isInvariantTrue());
}
// clientCode.cpp

int main( ) {

  Counter c(0, 3);
  Counter d(0, 10);
  c.increment();
  ...
    Counter x; // invalid!
}
```

Another example (time accumulator)

```
// time.h
Class Time{
 public:
 // Constructor
 Time(unsigned int hrs,
      unsigned int mins,
      unsigned int secs);
 // increment
 void increment(unsigned int hrs,
                unsigned int mins,
                unsigned int secs);
 // equals
 bool equals(const Time &t);
 // lessThan
 bool lessThan(const Time &t);
```

Interface ctd.

```
// getComponents
void getComponents(unsigned int &hrs,
                   unsigned int &mins,
                   unsigned int &secs);
// increment
void increment(unsigned int hrs,
               unsigned int mins,
               unsigned int secs);
// add
Time add(const Time &t);
// diff
Time diff(const Time &t);
private:
// Data representation to follow ...
};
```

Client Code

```cpp
#include "time.h"
int main( ) {
  unsigned int hrs, mins, secs;
  Time t1(0, 30, 45);
  t1.increment(0, 0, 15);
  Time t2(0, 30, 0);
  Time t3 = t1.add(t2);
  Time t4(0, 1, 0);
  Time t5 = t3.diff(t4);
  t5.getComponents(hrs, mins, secs);

  cout << hrs << ':' << mins << ':' << secs << endl;

  Return 0;
}
```

Data Representation and Private Constructor

```cpp
// time.h
class Time {
public:
  ...
private:
  // Another constructor
  Time(unsigned long int secs);



private:
  unsigned long int seconds;
};
```

Implementation

```
// time.cpp
#include <cassert>
#include "time.h"
namespace {
  const unsigned long int SECS_IN_MIN  = 60;
  const unsigned long int MINS_IN_HOUR = 60;
  const unsigned long int SECS_IN_HOUR = SECS_IN_MIN * MINS_IN_HOUR;

  unsigned long int convertToSecs(unsigned hrs, unsigned mins, unsigne
    return hrs * SECS_IN_HOUR + mins * SECS_IN_MIN + secs;
  }
}
```

Implementation

```cpp
// time.cpp
Time::Time(unsigned int hrs,
           unsigned int mins,
           unsigned int secs) {
  assert(mins < 60);
  assert(secs < 60);
  seconds = convertToSecs(hrs, mins, secs);
}
void Time::increment(unsigned int hrs,
                     unsigned int mins,
                     unsigned int secs) {
  assert(mins < 60);
  assert(secs < 60);
  seconds += convertToSecs(hrs, mins, secs);
}
```

Implementation

```cpp
// time.cpp
bool Time::equals(const Time &t) {
  return seconds == t.seconds;
}
bool Time::lessThan(const Time &t) {
  return seconds < t.seconds;
}
void Time::getComponents(unsigned int &hrs,
                         unsigned int &mins,
                         unsigned int &secs) {
  hrs  =  seconds / SECS_IN_HOUR;
  mins = (seconds / SECS_IN_MIN) % MINS_IN_HOUR;
  secs =  seconds % SECS_IN_MIN;
}
```

Implementation

```cpp
// time.cpp

Time Time::add(const Time &t) {
  Time result(seconds + t.seconds);
  return result;
}

Time Time::diff(const Time &t) {
  assert(!lessThan(t));
  Time result(seconds - t.seconds);
  return result;
}

// second constructor!
Time::Time(unsigned long int secs) {
  seconds = secs;
}
```

Remarks

- Note the second (private) constructor on slide 13 and 17

```cpp
Time Time::add(const Time &t) {
  return Time(seconds + t.seconds);
}
Time Time::diff(const Time &t) {
  assert(! lessThan(t));
  return Time(seconds - t.seconds);
}
```

Remarks

- Note the second (private) constructor on slide 13 and 17
    - used by `add( )` and `diff( )`

```
Time Time::add(const Time &t) {
  return Time(seconds + t.seconds);
}
Time Time::diff(const Time &t) {
  assert(! lessThan(t));
  return Time(seconds - t.seconds);
}
```

Remarks

- Note the second (private) constructor on slide 13 and 17
  - used by `add( )` and `diff( )`
  - in general, can have many

```
Time Time::add(const Time &t) {
  return Time(seconds + t.seconds);
}
Time Time::diff(const Time &t) {
  assert(! lessThan(t));
  return Time(seconds - t.seconds);
}
```

Remarks

- Note the second (private) constructor on slide 13 and 17
  - used by `add( )` and `diff( )`
  - in general, can have many
- Could have implemented `add( )` and `diff( )` differently

```cpp
Time Time::add(const Time &t) {
  return Time(seconds + t.seconds);
}
Time Time::diff(const Time &t) {
  assert(! lessThan(t));
  return Time(seconds - t.seconds);
}
```

More Remarks

- Above alternative implementation creates a temporary, anonymous instance of Time and returns it right away (more efficient)

```
Time t = Time(1, 0, 45).add(Time(0, 30, 15));
```

More Remarks

- Above alternative implementation creates a temporary, anonymous instance of Time and returns it right away (more efficient)
    - no intermediate variables are declared

```
Time t = Time(1, 0, 45).add(Time(0, 30, 15));
```

More Remarks

- Above alternative implementation creates a temporary, anonymous instance of Time and returns it right away (more efficient)
    - no intermediate variables are declared
- Another example (where 2 temporary instances are created):

```
Time t = Time(1, 0, 45).add(Time(0, 30, 15));
```

More Remarks

- Above alternative implementation creates a temporary, anonymous instance of Time and returns it right away (more efficient)
    - no intermediate variables are declared
- Another example (where 2 temporary instances are created):

```
Time t = Time(1, 0, 45).add(Time(0, 30, 15));
```

- Compilers can usually optimize your code to do this

Default constructor

- Can give default initial values

```
// time.h
class Time {
public:
  // Default Constructor
  Time( );
  ...
};
// time.cpp
Time::Time( ) {
  seconds = 0;
}

// client code in main
Time x;
Time y(13,13,13);
```

Default constructor

- Can give default initial values
- Constructor with no parameters

```
// time.h
class Time {
public:
  // Default Constructor
  Time( );
  ...
};
// time.cpp
Time::Time( ) {
  seconds = 0;
}

// client code in main
Time x;
Time y(13,13,13);
```

Default constructor

- Can give default initial values
- Constructor with no parameters
- Invoked by compiler if the client did not invoke another constructor

```
// time.h
class Time {
public:
  // Default Constructor
  Time( );
  ...
};
// time.cpp
Time::Time( ) {
  seconds = 0;
}

// client code in main
Time x;
Time y(13,13,13);
```

C++ classes are records with encapsulated fields

C++ classes are records with encapsulated fields

```
struct Time {
  unsigned long int seconds;
};
```

C++ classes are records with encapsulated fields

```
struct Time {
  unsigned long int seconds;
};
```

```
class Time {
public:
  ...
private:
  unsigned long int seconds;
};
```

Structs with Functions

Structs with Functions

Structs with Functions

- Only difference: by default, fields are public in structures and private in classes

Structs with Functions

- Only difference: by default, fields are public in structures and private in classes

```
struct Time {
public:
  Time();
  Time(unsigned int hrs,
       unsigned int mins,
       unsigned int secs);
  void increment(unsigned int hrs,
                 unsigned int mins,
                 unsigned int secs);
  Time add(const Time &t);
  Time diff(const Time &t);
  bool equals(const Time &t);
  bool lessThan(const Time &t);
  void getComponents(unsigned int &hrs,
                     unsigned int &mins,
                     unsigned int &secs);
private:
  Time(unsigned long int secs);
  unsigned long int seconds;
};
```

Initializing, Assignment, Copying

```
class A { ... };
void func1(A z) { ... }

A x, y;
...
x = y;

...
func1(x);

A func2( ) {
  A x;
  ...
    return x;
}

A z = func2( );
```

Default Initialization

- Just like structures, no initialization is performed by default (unless a constructor is provided)

```
class A {
  // no constructor declared here
  ...
};
A x; // initialization will not be performed
```

Default Initialization

- Just like structures, no initialization is performed by default (unless a constructor is provided)
- If no constructors are provided, the compiler supplies a dummy one that does nothing!

```
class A {
  // no constructor declared here
  ...
};
A x; // initialization will not be performed
```

Passing objects as arguments

- Can be costly

```
int f(const Time &t) {
  if (t.lessThan(Time(0, 30, 0))) // valid: lessThan is const
    t.increment(0, 30, 0);             // invalid: increment is not c
}
```

Passing objects as arguments

- Can be costly
- better to pass by reference

```
int f(const Time &t) {
  if (t.lessThan(Time(0, 30, 0))) // valid: lessThan is const
    t.increment(0, 30, 0);                 // invalid: increment is not c
}
```

Passing objects as arguments

- Can be costly
- better to pass by reference
- sometimes want to ensure that the passed object is not modified via the `const` keyword

```
int f(const Time &t) {
  if (t.lessThan(Time(0, 30, 0))) // valid: lessThan is const
    t.increment(0, 30, 0);             // invalid: increment is not c
}
```

const member functions

```
Time add(const Time &t); // in Time class
Time t3 = t1.add(t2);        // in main function
```

- How to ensure that member function add doesn’t accidentally modify the reference object t1?

```
Time add(const Time &t) const; // in Time.h

Time Time::add(const Time &t) const {  // in Time.cpp
  increment(1,15,30); // invalid!
  ...
    }
```

const member functions

```
Time add(const Time &t); // in Time class
Time t3 = t1.add(t2);        // in main function
```

- How to ensure that member function add doesn’t accidentally modify the reference object t1?
- Use the following declaration instead

```
Time add(const Time &t) const; // in Time.h

Time Time::add(const Time &t) const {  // in Time.cpp
  increment(1,15,30); // invalid!
  ...
    }
```

const member functions

```
Time add(const Time &t); // in Time class
Time t3 = t1.add(t2);        // in main function
```

- How to ensure that member function add doesn’t accidentally modify the reference object t1?
- Use the following declaration instead
  - Note `const` keyword *after* parameter list

```
Time add(const Time &t) const; // in Time.h

Time Time::add(const Time &t) const {  // in Time.cpp
  increment(1,15,30); // invalid!
  ...
    }
```