Function Members

Steven Zeil

September 17, 2013

Contents

Structs can have Function Members

- Structs can group fixed numbers of pieces of data of different types
- Structs can also group related *functions*

. . . . . .

1 Function Members

Function Members

- *Function members* are functions declared inside a struct.
- We use this technique with functions that access or modify a value of this `struct` type.

. . . . . .

Example: Times

```
struct Time {
  int hours;
  int minutes;
  int seconds;
};


void read (std::istream& in, Time& time);

void print (std::ostream& out, const Time& time);

bool noLaterThan(const Time& time1, const Time& time2);
```

We can move these functions inside the struct...

. . . . . .

Example: Times

```
struct Time {
  int hours;
  int minutes;
  int seconds;

  void read (std::istream& in);

  void print (std::ostream& out);

  bool noLaterThan(const Time& time2);
};
```

Note that we remove a `Time` parameter from each function declaration.

. . . . . .

2 Calling Member Functions

Calling Member Functions

The parameter that we removed is now written to the left of the call:

- Stand-alone function:

  ```
  print (cout, myTime);
  ```

- Member function:

  ```
  myTime.print (cout);
  ```

  - The item on the left is the `struct` value that this call is accessing/modifying.
    - Think of the dot as selecting a function inside the `struct`, just as it can select a data member inside a struct.
  - Where have you seen this style of call before?

You've used this style of call with strings, I/O manipulators, and other data types that can only be accessed after `#include`'ing something from the `std` library.

3 Implementing Member Functions

Implementing Member Functions

```
void Time:: print (std::ostream& out)
{
  ⋮
}
```

- The `::` means "within the scope named on the left"
- You've seen it used like this: `std::string`, `std::istream`
  - In this instance, it means "the function named 'print' declared inside the class `Time`
- distinguishes this function from others elsewhere that might have the same name and parameter types

Fully Qualified Names

A *fully qualified name* of a C++ entity combines the name of the specific entity with the fully qualified names of any structs/classes/namespaces that contains it.

- *string* is a type name in C++
  - *std::string* is its fully qualified name
- *npos* is a constant provided by the *string* type
  - *std::string::npos* is its fully qualified name

Let's Say That Again

After

```
struct Time {
   ⋮
   void print (std::ostream& out);
   ⋮
};
```

we can write both

```
void print (std::ostream& out)
{
   ⋮
}

void Time::print (std::ostream& out)
{
   ⋮
}
```

- The first declares a new function, not a member of any struct, that takes a single parameter.
- The second one defines (provides a body for) the function declared in *Time*
  - We (and the compiler) know that because the definition uses the *fully qualified name* to tell us that.

. . . . . .

Implementing Member Functions

```
void Time::print (std::ostream& out)
{
    if ( hours < 10)
```

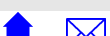

```
        out << '0';
    out << hours << ':';
    if ( minutes < 10)
        out << '0';
    out << minutes << ':';
    if ( seconds < 10)
        out << '0';
    out << seconds ;
}
```

- These are data member names
- Notice that there is nothing on the left, e.g., `myTime.hours`
- Within member function bodies, you can reference data and function members directly
  - implicitly refers to the value named on the left of the call

.....

Implicit Access to Members

```
void Time::print (std::ostream& out)
{
    if ( hours < 10)
        out << '0';
    out << hours << ':';
    if ( minutes < 10)
        out << '0';
    out << minutes << ':';
    if ( seconds < 10)
        out << '0';
```

```
    out << seconds;
}
```

- If we call `myTime.printTime(cout);`
- Then the highlighted items above refer to `myTime.hours`, `myTime.minutes`, and `myTime.seconds`.

.....

Multiple Struct Arguments

- If a function works on more than one value of the same struct type, when we turn it into a member function, we remove just one parameter.
- Before:

```
bool noLaterThan(const Time& time1, const Time& time2);
```

- After:

```
struct Time {
        ⋮
  bool noLaterThan(const Time& time2);
};
```

- Call it like this:

```
if (bidPlacedAt.noLaterThan(item.auctionEndsAt))
```

.....

Example: Time with Function Members

- Header:

```
#ifndef TIMES_H
#define TIMES_H

#include <iostream>

 * The time of the day, to the nearest second.


struct Time {
  int hours;
  int minutes;
  int seconds;


  /**
   * Read a time (in the format HH:MM:SS) after skipping any
   * prior whitespace.
   */
  void read (std::istream& in);


  /**
   * Print a time in the format HH:MM:SS (two digits each)
   */
  void print (std::ostream& out);

  /**
```


```
   *  Compare two times. Return true iff time1 is earlier than or equal to
   *  time2
   *
   *  Pre: Both times are normalized: seconds and minutes are in the range 0..59,
   *         hours are non-negative
   */
  bool noLaterThan(const Time& time2);

};

#endif // TIMES_H
```

- Compilation unit:

```
#include "times.h"

using namespace std;

/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respecitvely.
 */


/**
 * Read a time from the indicated stream after skipping any
 * leading whitespace
 */
void Time::read (istream& in)
{
  char c;
```


```
  in >> hours >> c >> minutes >> c >> seconds;
}


/**
 * Print a time in the format HH:MM:SS (two digits each)
 */
void Time::print (std::ostream& out)
{
    if (hours < 10)
        out << '0';
    out << hours << ':';
    if (minutes < 10)
        out << '0';
    out << minutes << ':';
    if (seconds < 10)
        out << '0';
    out << seconds;
}

/**
 *  Compare two times. Return true iff time1 is earlier than or equal to
 *  time2
 *
 *  Pre: Both times are normalized: sconds and minutes are in the range 0..59,
 *         hours are non-negative
 */
bool Time::noLaterThan(const Time& time2)
{
    // First check the hours
    if (hours > time2.hours)
```


```
        return false;
    if (hours < time2.hours)
        return true;
    // If hours are the same, compare the minutes
    if (minutes > time2.minutes)
        return false;
    if (minutes < time2.minutes)
        return true;
    // If hours and minutes are the same, compare the seconds
    if (seconds > time2.seconds)
        return false;
    return true;
}
```

.....

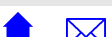
Example: Money with Function Members

- Header:

```
/*
 * money.h
 *
 *  Created on: Aug 23, 2013
 *      Author: zeil
 */

#ifndef MONEY_H_
#define MONEY_H_

#include <iostream>
```



```
/**
 * An amount of U.S. currency
 */
struct Money {
    int dollars;
    int cents; //< @invariant should be in the range 0..99, inclusive


  /**
   * Read a money value from the input.  Acceptable formats are
   *
   *        ddd.cc or ddd
   *
   * where ddd is any positive/negative integer of
   * one or more digits denoting dollars, and cc, if
   * supplied, is a two-digit integer.
   *
   * @param in   stream from which to read
   * @param money the value read in. Result is unpredictable if an
   *              I/O error occurs
   */
  void read (std::istream& in);

  /**
   * Print a monetary amount.The output format will always
   * include a decimal point and a two-digit cents amount.
   *
   * @param out the stream to which to print
   * @param money the value to be printed
   */
```

```
void print (std::ostream& out);

/**
 * Compare two Money amounts to see if they are equal
 *
 * @param left  1st value to be compared
 * @param right 2nd value to be compared
 * @return true iff the two amounts are equal
 */
bool equal (const Money& right);


/**
 * Compare two Money amounts to see if the 1st is smaller
 * than the second
 *
 * @param left  1st value to be compared
 * @param right 2nd value to be compared
 * @return true iff left is a smaller amount than right
 */
bool lessThan (const Money& right);

/**
 * Adds two Money amounts together
 *
 * @param left  1st value to be added
 * @param right 2nd value to be added
 * @return sum of the two amounts
 */
Money add (const Money& right);
```


```
  /**
   * Subtract one Money amount from another
   *
   * @param left  the minuend
   * @param right the subtrahend
   * @return difference of the two amounts
   */
  Money subtract (const Money& right);

};


#endif /* MONEY_H_ */
```

- Compilation unit:

```
/*
 * money.h
 *
 *  Created on: Aug 23, 2013
 *      Author: zeil
 */

#include "money.h"
#include <iostream>


using namespace std;
```


```
/**
 * Read a money value from the input.  Acceptable formats are
 *
 *        ddd.cc or ddd
 *
 * where ddd is any positive/negative integer of
 * one or more digits denoting dollars, and cc, if
 * supplied, is a two-digit integer.
 *
 * @param in   stream from which to read
 * @param money the value read in. Result is unpredictable if an I/O error occurs
 */
void Money::read (std::istream& in)
{
  if (!in) return;
  in >> dollars;
  if (!in) return;
  if (in.peek() == '.') // if next input is a '.'
    {
      char decimal;
      in >> decimal;
      in >> cents;
    } else
    cents = 0;
}

/**
 * Print a monetary amount.The output format will always
 * include a decimal point and a two-digit cents amount.
 *
```



```
 * @param out the stream to which to print
 * @param money the value to be printed
 */
void Money::print (std::ostream& out)
{
  out << dollars;
  out << '.';
  if (cents < 10)
    out << '0';
  out << cents;
}

/**
 * Compare two Money amounts to see if they are equal
 *
 * @param left  1st value to be compared
 * @param right 2nd value to be compared
 * @return true iff the two amounts are equal
 */
bool Money::equal (const Money& right)
{
  return (dollars == right.dollars)
    && (cents == right.cents);
}


/**
 * Compare two Money amounts to see if the 1st is smaller
 * than the second
 *
 * @param left  1st value to be compared
```


```
 * @param right 2nd value to be compared
 * @return true iff left is a smaller amount than right
 */
bool Money::lessThan (const Money& right)
{
  if (dollars < right.dollars)
    return true;
  else if (dollars == right.dollars)
    return cents < right.cents;
  return false;
}

/**
 * Adds two Money amounts together
 *
 * @param left  1st value to be added
 * @param right 2nd value to be added
 * @return sum of the two amounts
 */
Money Money::add (const Money& right)
{
  Money result;
  result.dollars = dollars + right.dollars;
  result.cents = cents + right.cents;
  while (result.cents > 99)
    {
      result.cents -= 100;
      ++result.dollars;
    }
  while (result.cents < 0)
    {
```


```
      result.cents += 100;
      --result.dollars;
    }
  return result;
}


/**
 * Subtract one Money amount from another
 *
 * @param left  the minuend
 * @param right the subtrahend
 * @return difference of the two amounts
 */
Money Money::subtract (const Money& right)
{
  Money result;
  result.dollars = dollars - right.dollars;
  result.cents = cents - right.cents;
  while (result.cents > 99)
    {
      result.cents -= 100;
      ++result.dollars;
    }
  while (result.cents < 0)
    {
      result.cents += 100;
      --result.dollars;
    }
  return result;
}
```


. . . . . .