Constructors

Steven Zeil

September 17, 2013


Outline


Outline I


Constructors

- A *constructor* is a special member function used to initialize structured data.
- A constructor function returns a value of the structured type.
  - although no return statement is written in the body


naming Constructors

- The name of the constructor function is the same as the name of the struct.
  - The normal function declaration syntax would look a bit odd:

```
struct MyStruct {
  MyStruct MyStruct(); // Not legal C++!
     ⋮
```

because the struct name, the return type, and the function name would all be the same.

  - So C++ omits the return type from declarations of constructors:

```
struct MyStruct {
  MyStruct(); // a Constructor!
     ⋮
```


Calling Constructors

A constructor can be called like a normal function:

- Given
  auction/times.h, we can write:

```
if (myTime.noLaterThan(Time(16,15,00)))
    cout << "Class is not yet over" << endl;
```


Calling Constructors in Declarations

More often, a constructor is called within a declaration:

```
Time midnight; // calls Time()
Time endOfClass (16, 15, 00); // calls Time(h,m,s)
```


Some Special Syntax for Declarations

A few special rules:

- If a constructor takes zero parameters, we call it in a declaration like this:

```
Time t;
```

not like this:

```
Time t();
```

  - Otherwise the parentheses would turn this into a declaration of a new function named "t" with a return type of "Time"


Some Special Syntax for Declarations (cont.)

- If a constructor takes one parameter, then it can be called like this

```
string s ("abc");
```

or like this:

```
string s = "abc";
```

They mean *exactly* the same thing.


What You've Been Doing All Along

You've written or read declarations like the ones below.

```
string s;
string s1 (s);
string s2 = "abc";
string s3 (5, z); // "zzzzz"
ifstream in ("myFileName.txt");
```

- What do these tell you about the constructors used with these types?


Implementing Constructors

The primary purpose of a constructor is to initialize a structured data value.

```
struct BidderSequence {
  static const int capacity = 1000;
  int size;
  Bidder data[capacity];

  // Initialize a sequence with size 0
  BidderSequence();
};
   ⋮
BidderSequence::BidderSequence()
{
  size = 0;
}
```

- Implementation is much like ordinary member functions
  - Remember to omit the return type


Implementing Constructors (cont.)

Another example:

```
struct Money {
  int dollars;
  int cents;

  /**
  * Initialize a monetary amount. Default is $0.0
  */
  Money (int dollarPart = 0, int centsPart = 0);
    ⋮
};
    ⋮
Money::Money (int dollarPart, int centsPart)
{
  dollars = dollarPart;
  cents = centsPart;
}
```

Data Must be Initialized Before Use

Forgetting to initialize data is one of the most common mistakes that programmers make.
Without constructors, it's all too easy to do something like

```
BidderSequence seq1, seq2;
  ⋮
seq1.size = 0;
// code that uses seq1
  ⋮
// code that uses seq2
  ⋮
```

forgetting that *seq2* has not yet been properly initialized.


Constructors Help to Protect You

Once we introduce constructor functions, however, our variables get initialized the moment we declare them.

```
BidderSequence::BidderSequence ()
{
  size = 0;
}
  :
BidderSequence seq1, seq2;
// Both variables are immediately
// initialized with size = 0
```

- As a general rule, *every* structured type should have at least one constructor.


Overloading Constructors

Like other functions, constructors can be overloaded, providing multiple functions with the same name but different parameter types.

```
struct Item {
  std::string name;
  Money reservedPrice;
  Time auctionEndsAt;

  /**
  * Initialize an item to empty name, no reserve,
  * and with auction ending at midnight.
  */
  Item();

  /** Initialize an item */
  Item(std::string iName, Money reserved, Time endsAt);
    :
};
```


Overloading Constructors (implem.)

```
/**
 * Initialize an item to empty name, no reserve, with auction
 * ending at midnight.
 */
Item::Item()
{
  name = "";
  reservedPrice = Money(0,0);
  auctionEndsAt = Time(0,0,0);
}

/**  Initialize an item  */
Item::Item (std::string iName, Money reserved, Time endsAt)
{
  name = iName;
  reservedPrice = reserved;
  auctionEndsAt = endsAt;
}
```


Outline I


The Default Constructor

The *default constructor* is a constructor that takes no arguments. This is the constructor you are calling when you declare an object with no parameters. E.g.,

```
std::string s;
```


Declaring a Default Constructor

It might be declared like this

```
class Time {
public:
    Time();
     ⋮
```

or with defaults:

```
namespace std {
class string {
public:
     ⋮
    string(char* s = "");
    ⋮
```

Either way, we can *call* it with no parameters.


Why 'default'?

- It's just an ordinary constructor
- But it is used (implicitly) to initialize elements of an array.
- It is also used (implicitly) in other ADTs' constructors when they do not explicitly initialize a data member.


Implicit Use: Arrays

For example, if we declared:

```
std::string words[5000];
```

then each of the 5000 elements of this array will be initialized using the default constructor for `string`


Implicit Use: Other Constructors

Earlier, we had:

```
/**
 * Initialize an item to empty name, no reserve, with au
 * ending at midnight.
 */
Item::Item()
{
   name = "";
   reservedPrice = Money(0,0);
   auctionEndsAt = Time(0,0,0);
}
```

But, consider that


Implicit Use: Other Constructors

Earlier, we had:

```
/**
 * Initialize an item to empty name, no reserve, with au
 * ending at midnight.
 */
Item::Item()
{
   name = "";
   reservedPrice = Money(0,0);
   auctionEndsAt = Time(0,0,0);
}
```

But, consider that

- The default constructor for *std::string* sets a new string to ""


Implicit Use: Other Constructors

Earlier, we had:

```
/**
 * Initialize an item to empty name, no reserve, with au
 * ending at midnight.
 */
Item::Item()
{
   name = "";
   reservedPrice = Money(0,0);
   auctionEndsAt = Time(0,0,0);
}
```

But, consider that

- The default constructor for *std::string* sets a new string to ""
- The default constructor for Money initializes to $0.0


Implicit Use: Other Constructors

Earlier, we had:

```
/**
 * Initialize an item to empty name, no reserve, with au
 * ending at midnight.
 */
Item::Item()
{
   name = "";
   reservedPrice = Money(0,0);
   auctionEndsAt = Time(0,0,0);
}
```

But, consider that

- The default constructor for *std::string* sets a new string to ""
- The default constructor for Money initializes to $0.0
- The default constructor for Time initializes to midnight


Implicit Use: Other Constructors (cont.)

So we might as well write:

```
Item::Item()
{
}
```

rather than

```
Item::Item()
{
  name = "";
  reservedPrice = Money(0,0);
  auctionEndsAt = Time(0,0,0);
}
```

It's shorter *and* faster!


Think About This

For our type

```
struct BidderSequence {
    static const int capacity = 1000;
    int size;
    Bidder data[capacity];

    BidderSequence();
      ⋮
};
```

we had the constructor

```
BidderSequence::BidderSequence ()
{
  size = 0;
}
```

Is the *data* initialized?


Outline I


Avoiding the Default

What if we have data members that we want to initialize with a non-default value?

```
struct Item {
    std::string name;
    Money reservedPrice;
    Time auctionEndsAt;


    /**
     * Initialize an item to empty name, no reserve,
     * with auction ending at midnight.
     */
    Item();
     ⋮
```

Suppose that we wanted a default item's reserve price to be one dollar and the auction to end a second before midnight?


One Approach: Re-assignment

We could do

```
Item::Item()
{
   reservedPrice = Money(1,0);
   auctionEndsAt = Time(23,59,59);
}
```

But this is inefficient...


One Approach: Re-assignment

We could do

```
Item::Item()
{
   reservedPrice = Money(1,0);
   auctionEndsAt = Time(23,59,59);
}
```

But this is inefficient...

- First the data members are initialized to default values


One Approach: Re-assignment

We could do

```
Item::Item()
{
   reservedPrice = Money(1,0);
   auctionEndsAt = Time(23,59,59);
}
```

But this is inefficient...

- First the data members are initialized to default values
- Then we construct new time and money values


One Approach: Re-assignment

We could do

```
Item::Item()
{
   reservedPrice = Money(1,0);
   auctionEndsAt = Time(23,59,59);
}
```

But this is inefficient...

- First the data members are initialized to default values
- Then we construct new time and money values
- and then copy over the original values of the data members


Initialization List

Instead we can tell the compiler how we want to initially construct the data members:

```
Item::Item()
   : reservedPrice(1,0), auctionEndsAt(23,59,59)
{
}
```

- This is an *initialization list*
  - Special syntax, only within constructors
- A comma-separated list of "constructor calls"


Interpeting Initialization Lists

```
Item::Item()
   : reservedPrice(1,0), auctionEndsAt(23,59,59)
{
}
```

- Each item in the list shows a constructor we want invoked
- As if we had written

```
Money reservedPrice (1,0);
Time auctionEndsAt (23, 59, 59);
```

  - The parameters in the initialization list entries are the same as the parameters we would supply to a constructor.


Outline I


The Helpful Compiler

If we create no constructors at all for a class, the compiler generates a default constructor for us.

- Initializes each data member using their data types' default constructors
- For primitives such as `int`, `double`, pointers, etc., this does nothing at all

Example: Name I

```
class Name {
public:
    string getGivenName();
    void setGivenName (string);

    string getSurName();
    void setSurName (string);
private:
    string givenName;
    string surName;
};
```

- Compiler will generate a default constructor `Name()`
- `givenName` and `surName` will be initialized using the default constructor of `string`
  - Probably just fine


Example: Name 2 I

```
class Name {
public:
    Name (string gName, string sName)
      : givenName(gName), surName(sName) {}

    string getGivenName();
    void setGivenName (string);

    string getSurName();
    void setSurName (string);
private:
    string givenName;
    string surName;
};
```


Example: Name 2 II

- Compiler will *not* generate a default constructor `Name()` because we provided a different constructor
- If we want one, we have to write our own
  - If we don't, we cannot have arrays of `Names`