Structured Data

Steven Zeil

August 24, 2013

Contents

Structured Data

Often we have data items that we would like to group together.

- Arrays can group *arbitrary* numbers of pieces of data of the *same* type
- *Structs* can group *fixed* numbers of pieces of data of *different* types
  - Structs are a.k.a. *records*

. . . . . .

1 Structs

Example: An Online Auction

A running example through this semester will be an on-line auction.
Key ideas:

- *Items* are offered for auction ending at a designated *time* and with a *reserve price*
- *Bidders* deposit money into an auction account as a sign of good faith.
  - They cannot win an auctions with bids larger than the amount deposited
- *Bids* are placed by bidders through the day for specific amounts for specific items

. . . . . .

Let's take a look at how we can represent some of these ideas as data:

Bidders: description

Bidders are described as

- "*bidder-name* is a login name for the bidders, and consists of a single word (no internal blanks).
- *account-balance* is the amount deposited by the bidder. It is a number and is specified to the penny (0.01)."

. . . . . .

Different Data Items Describing one "Thing"

So there are two critical components to each bidder:

```
struct Bidder {
    std::string name;
    Money balance;
};
```

A `struct` declares a data type as a collection of

- *fields* (a.k.a. *data members*)
- Each field has
  - a data type
  - a field name (a.k.a. member name)

. . . . . .

Using *std::string* for the first field type is pretty much a no-brainer.
The choice of data type to use for the account balance is trickier:

- Monetary amounts are normally written with decimal points, so we might think to sue *float* or *double*.

  But floating point does not really work well. The representation is not exact, nor are the calculations. And company auditors *really* don't ant to hear stories about round-off error.

- Integer types (*int* or *long*) work better. We could use the integer to represent the number of cents in the monetary amount.

  But integers don't read and write with decimal points, so we would need special functions for that purpose. It's not impossible, but it could be annoying.

If I *did* decide to use one of these types for money, I would still like to use the name "Money" because it is more descriptive than a generic type name like "int" or "double". I can do that with a `typedef`:

```
typedef double Money;
```

which creates a new type name "Money" as a synonym for *double*.

We'll return to the question of how to represent *Money*

Items: description

An `Item` is described as

- "*reserve-price* is the minimum price the seller will accept for an item. It is a number and is specified to the penny (0.01).
- *auction-end-time* is in 24 hour format of the form *XX:YY:ZZ* where *XX* is in hours from 00 to 23, *YY* is in minutes from 00 to 59, and *ZZ* is in seconds from 00 to 59.
- *item-name* is a string"

. . . . . .

A struct for Items

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

- This declares a new data type, `Item`.
- Each `Item` value will have 3 *fields* or *data members* of 3 different types

. . . . . .

Take Your Time

Item introduces a new concept, "Time". We could have dealt with it like this

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


```
  int auctionEndsAtHour;
  int auctionEndsAtMinute;
  int auctionEndsAtSecond;
};
```

but that's

- ugly
- and we will need the idea of "time" repeatedly
  - For example, bids occur at a specific time

.....

So What is Time?

The structure for *Time* is not complicated:

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

.....

Now, let's think about what making *Time* a `struct` gains for us.

Some Advantages of Using structs

```
struct Item {
  std::string name;
  Money reservedPrice;
  int auctionEndsAtHour;
  int auctionEndsAtMinute;
  int auctionEndsAtSecond;
};
```

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


The version on the right is

- simpler
- easier to read
- and, we will see, easier to work with

. . . . . .

structs Simplify Functions

One of the things we will need to know is whether a bid has been submitted after the official end of the auction. *With* structs, we can write a function for this purpose:

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

Without structs, we would have to pass all the components separately:

```
bool noLaterThan
  (int hours1, int minutes1, int seconds1,
   int hours2, int minutes2, int seconds2);
```

- Which function would you rather write?
- Which function would you rather call?

. . . . . .

Money

So, how would you design a struct to represent U.S. currency?

. . . . . .

Bids

Bids are described as

- "*bidder-name* is a login name for the bidders, and consists of a single word (no internal blanks).
- *amount-bid* is the amount bid. It is a number and is specified to the penny (0.01).
- *time-of-bid* is in 24 hour format of the form *XX:YY:ZZ* as described earlier.
- *item-name* is a string"

How would you design a struct for this?

1.1 Accessing Fields

Accessing Fields

Fields are accessed by name, via the "." operator:

```
struct Time {
  int hours;
  int minutes;
  int seconds;
};
  ⋮
Time t1;
t1.hours = 12;
t1.minutes = 0;
t1.seconds = 1;

int hourOfDay = t1.hours;
bool t1IsPM = (t1.hours >= 12);
```

Dot Examples

```
struct Item {
  std::string name;
  Money reservedPrice;
  Time auctionEndsAt;
};
   ⋮
Item myItem;
myItem.name = "Tiffany Lamp";
bool inAM = (myItem.auctionEndsAt.hours < 12);
if (inAM)
{
   cout << myItem.name << " will sell before noon"
        << endl;
}
```

. . . . . .

"Chaining" dot Operations

```
bool inAM = (myItem.auctionEndsAt.hours < 12);
```

- *myItem.auctionEndsAt* is a *Time*
- *Time*s have hours, minutes, and seconds.
  - So we can apply *.hours* to *myItem.auctionEndsAt*

This is equivalent to

```
Time myItemTime = myItem.auctionEndsAt;
bool inAM = (myItemTime.hours < 12);
```

. . . . . .

1.2 Working with Structured Data

Things We Can Do With structs

- access and set field values
- Copy an entire structure

```
myItem = yourItem;
```

- Initialize a structure

```
Time noon = {12, 0, 0};
Money twoBits = {0, 25};
```

.....

With this structure for *Money*:

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

Example: Adding Monetary Amounts

```
Money add (const Money& left, const Money& right)
{
  Money result;
  result.dollars = left.dollars + right.dollars;
  result.cents = left.cents + right.cents;
  normalize (result);
  return result;
}
```


Most of this is pretty straightforward.

- But what does `normalize` do?

. . . . . .

normalize

This function makes sure that the number of cents is kept in the range 0...99:

```
void normalize(Money& m)
{
  while (m.cents > 99)
  {
    m.cents -= 100;
    ++m.dollars;
  }
  while (m.cents < 0)
  {
    m.cents += 100;
    --m.dollars;
  }
}
```

So adding `{1,50}` and `{2,75}` would initially yield `{3,125}`.

Then `normalize` corrects that to `{4,25}`.

. . . . . .

Things We Cannot Do With Structs

(At least, not by default.)

- compare structs

```
if (myItem == yourItem) // error
```


- read/write structs with the usual operators

```
cout << myItem; // error
```

If we want to do those, we need to craft custom functions

. . . . . .

Example: Comparing Money

```
bool equal (const Money& left, const Money& right)
{
  return (left.dollars == right.dollars)
    && (left.cents == right.cents);
}
```

. . . . . .

Example: Printing Money

(not counterfeiting!)

```
void print (std::ostream& out, const Money& money)
{
  out << money.dollars;
  out << '.';
  if (money.cents < 10)
    out << '0';
  out << money.cents;
}
```

. . . . . .

Example: Printing Time

```
void print (std::ostream& out, const Time& t)
{
  if (t.hours < 10)
     out << '0';
  out << t.hours << ':';
  if (t.minutes < 10)
     out << '0';
  out << t.minutes << ':';
  if (t.seconds < 10)
     out << '0';
  out << t.seconds;
}
```

E.g., noon would print as “12:00:00”.

. . . . . .

Overloading

Is it a problem that both functions are named “print”?

```
void print (std::ostream& out, const Money& money);
void print (std::ostream& out, const Time& t);
```

No!

- C++ allows a program to have multiple functions of the same name visible simultaneously
  - But their parameter types have to be different
- This is called *overloading* the function name

. . . . . .

2 Scope

The idea of “scope” is critical to understanding much of what goes on in a programming language.

Scope

- The *scope* of a declaration is the portion of the code within which that declared name can be accessed.

. . . . . .

Scope and Brackets

- A name whose declaration is *not* inside of any { } is usually visible from the point where it is declared to the end of the file.
- A name declared within { } is usually visible from the point where it is declared to the closing }
  - Exception: variables declared in `for` loop headers or in function headers are visible only to the end of the loop/function body.

. . . . . .

Scope Examples

Look at each of the declared names in the following code.
What are the scopes of those names?

```
int array1[MaxData]; // error
const int MaxData = 1000;
int array1[MaxData]; // OK

int k;

int foo (int i)
{
  bar(i-1); // error
  int x = i - 1; // OK
```


```
  return x;
}

void bar (int j)
{
  double k; // OK, not the same k
  i = j; // error
  int x = j; // OK, this is not the same x
  cout << foo(x) << endl; // OK
  int k = j-1; // error
  int x; // OK
}
```

```
int main (int argc, char** argv)
{
  int nData;
  {
    istringstream in (argv[1]);
    in >> nData;  // "converts" a string to an int
  }
  double value;
  {
    istringstream in (argv[2]); // OK - a different "in"
    in >> value;  // "converts" a string to a double
  }
  int* data = new int[nData+1];
  data[0] = 0;
  for (int i = 0; i < nData && data[i] >= 0; ++i)
    {
     cin >> data[i+1];
    }
```


```
  if (i < nData) // error!
    cout << "We ended early" << endl;
    for (int i = 0; i < nData && data[i] >= 0; ++i) // OK
    {
      cout << data[i+1] << "\n";
    }
}
```

.....

The Scope of Data Members

- Field/data member delcarations obey the { } rule
- But the dot operator also temporarily "opens up" the struct scope when looking for the name on the right.

.....

Data Member Scope

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

struct Bidder {
  std::string name; // OK, not the same name
  double balance;
};
```

- Is there a problem having two fields called “name”?

- No, because their scopes do not overlap
  - So there's never any ambiguity

.....

Dot Opens Up a Scope

```
Item item;
Bidder bidder;
    ⋮
cout << name << endl; // error: neither "name"
                      //        is visible
cout << "This is "
     <<  (bidder.name + "'s " + item.name); // OK
```

.....

- In the first line, the scope of the "name" fields does not extend past the { } that enclosed them,
  - So "name" cannot be used here.
- In the second statement, the "." temporarily opens the scope of the struct representing the type of the thing on the left.
  - We know *which* `name` is meant by looking at the data type of the variable on the left of the dot.

3 Combining Data Structures

Building New Types

We now have two ways to build new data types from existing ones:

1. Create an array of an existing type.
2. Create a struct with existing types for the data members

.....

Combining Data Structures

We can combine structs and arrays with one another.
The challenge in working with these is

- Always be aware of the "outermost" data type.
  - If it's an array, you can only do array-like things to it, such as `[i]`
    - Result might be a primitive type, an array, or a struct
  - If it's a struct, you can only do struct-like things, such as = or `.`

. . . . . .

3.1 Structs Within Structs

structs Within structs

- We can use structs as data members of other structs
- We've already seen this using *Time* and *Money* as fields of *Item*, *Bid*, and *Bidder*
- Leads to "chains" of dots like

```
if (latestBid.bidPlacedAt.minutes
        < item.auctionEndsAt.minutes)
{
```

. . . . . .

Interpreting Chains of Operations

```
Item myItem;
myItem.auctionEndsAt.hours = 13;
```

Move from left to right, being aware at all times of the data type that you are working with.

- *myItem* has data type *Item*.
- *Item* is a struct.

  So we can use `.` to access its fields
- `myItem.auctionEndsAt` has type *Time*
  - We know this from the declaration of *Item*
- *Time* is a struct

  So we can use `.` to access its fields
- `myItem.auctionEndsAt.hours` has type *int*
  - We know this from the declaration of *Time*

.....

3.2 Structs Within Arrays

Structs Within Arrays

Arrays of structs follow the same general principles

```
Bidder bidders[100];
```

- *Bidder* is a struct
- *bidders* is an array of *Bidder*, so we can use `[ ]`
- So we could write

  ```
  bidders[bidderNum].balance = highestBidSoFar;
  ```

  Working from left to right:

- *bidders* is an array of *Bidder*
- So `bidders[bidderNum]` has type *Bidder*
- *Bidder* is a struct, so we can use dot
- `bidders[bidderNum].balance` has type *Money*

. . . . . .

4 Arrays Within Structs

Data members of structs can themselves be of any legal data type, including arrays.

Arrays Within Structs

We can also place arrays within a struct, e.g.:

```
const int MaxDailyBids = 5000;
struct DailyBids {
   Bid bidsReceived[MaxDailyBids];
   int numberOfBidsReceived; // must be <= MaxDailyBids
};
```

. . . . . .

Arrays Within Structs

Given

```
DailyBids todaysBids;
```

Which of the following expressions is legal?

- *todaysBids[0]*
- *todaysBids.bidsReceived[1]*

- *todaysBids[0].bidsReceived*
- *todaysBids.bidsReceived[1].amount.cents*
- *todaysBids.bidsReceived.amount.cents[2]*

. . . . . .

And how do you tell?