The Structure of a C++ Program

Steven Zeil

August 31, 2013

Contents

1 Separate Compilation **2**
1.1 Separate Compilation . . . 3

2 Pre-processing **5**
2.1 `#include` . . . 7
2.2 Other Pre-processing Commands . . . 10

3 Declarations and Definitions **13**
3.1 Decls&Defs: Variables . . . 15
3.2 Decls&Defs: Functions . . . 16
3.3 Decls&Defs: Data Types . . . 16

4 Modules **17**
4.1 Coupling and Cohesion . . . 19

5 Example of Modularization: the auction program **24**
5.1 Dividing Things Up . . . 48

1 Separate Compilation

Working with Large Programs

- C++ programs can range from a handful of statements to hundreds of thousands
- May be written by one person or by a team

. . . .

Single File Programs

Putting your entire program into a single file is

- OK for small programs (CS150)
- But with large programs
  - compilation would take minutes, hours, maybe days
    * might break compiler
  - Team members would interfere with one another's work.
    * "Are you still editing that file? You've had it all afternoon."
    * "What do you mean you're saving changes to the file? *I've* been editing it for the last 45 minutes!"

. . . .

Multiple File C++ Programs

By splitting a program up into multiple files that can be compiled separately,

- Team members can work in parallel on separate files
- Files are compiled separately
  - each individual compilation is fast

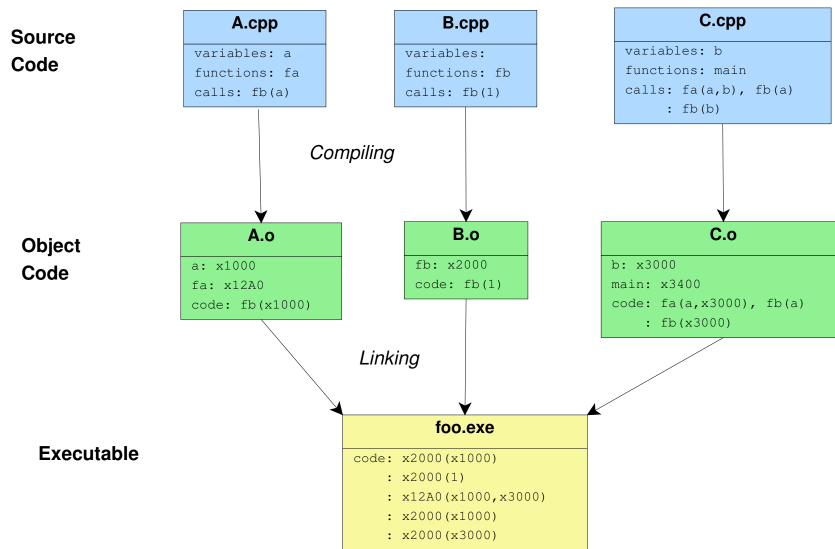
- Separately compiled code is *linked* to produce the executable
  - linking is much faster than compilation

. . . .

1.1 Separate Compilation

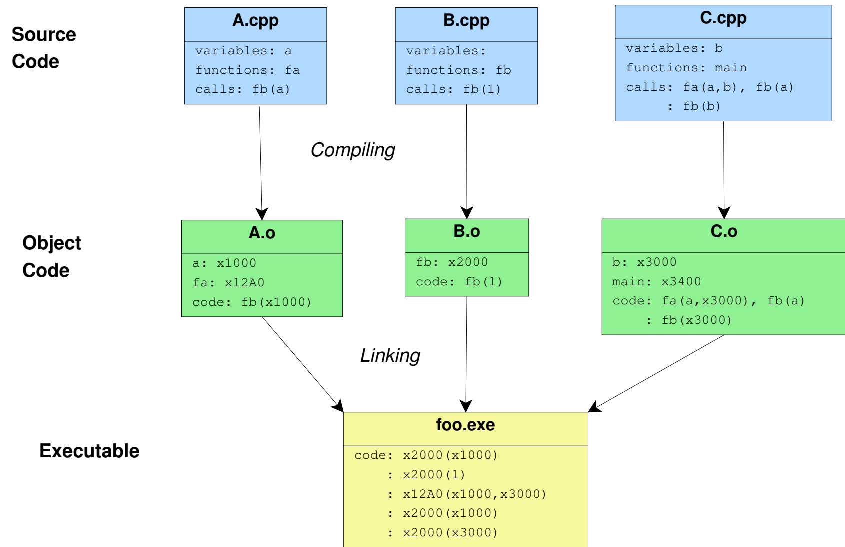
Separate Compilation

- Each file of *source code* (programming language text)
- is compiled to produce a file of *object code*.
- All object code files are *linked* to produce the executable



. . . .

Object Code

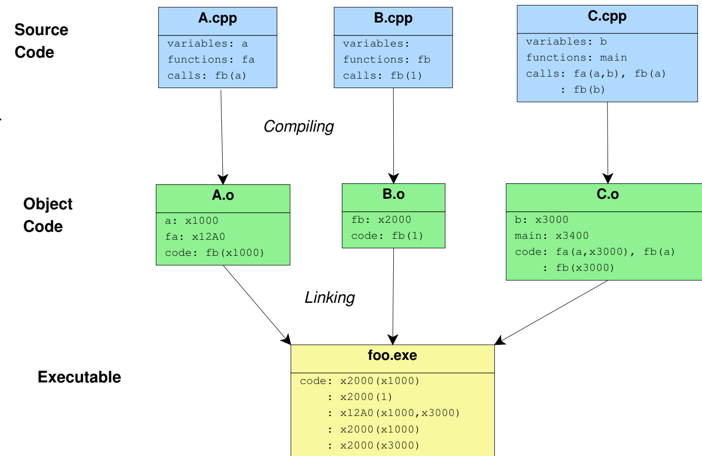


- binary code, almost executable
- exact addresses of variables and functions not known, represented by symbols

Linking

Linking mainly consists of replacing symbols by real addresses.
On large projects with hundreds to thousands of files,

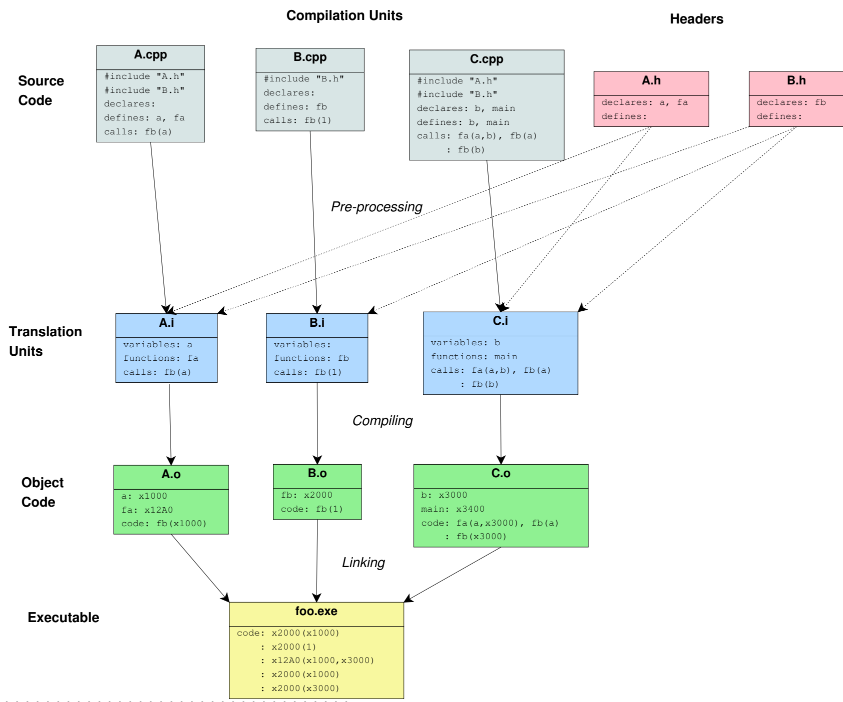
- Typically only a few files are changed on any one day
- Often only the changed files need to be recompiled
- Then link the changed and unchanged object code



. . . .

2 Pre-processing

The # Preprocessor



The preprocessor runs before the compiler proper. The preprocessor:

- modifies the source code
- processes preprocessor instructions
  - lines beginning with #
- strips out comments

Pre-Processor Instructions

The common pre-processor instructions are

- `#include`
  - insert a file
- `#define`
  - define a macro

- `#ifdef, #ifndef, #endif`
  - check to see if a macro has been defined

....

2.1 `#include`

`#include`

- Inserts a file or header into the current source code
- Two versions
  - **#include** <headerName>
    * inserts a system header file from a location defined when the compiler was installed
  - **#include** *"fileName"*
    * inserts a file from the current directory

....

Example: #include (simple case)

- Suppose we have three files:

```
// This is file A.h
code from A.h
```

```
//This is file B.h
#include "A.h"
code from B.h
more code from B.h
```

```
//This is file C.cpp
#include "A.h"
#include "B.h"
code from C.cpp
```

- We ask the compiler to only run the preprocessor and save the result:

```
g++ -E C.cpp > C.i
```

- The result is file `C.i`

```
# 1 "C.cpp"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "C.cpp"

# 1 "A.h" 1

code from A.h
# 3 "C.cpp" 2
# 1 "B.h" 1

# 1 "A.h" 1

code from A.h
```

```
# 3 "B.h" 2
code from B.h
more code from B.h
# 4 "C.cpp" 2
code from C.cpp
```

- Note the presence of content from all three files
  * includes markers telling where the content came from

. . . .

A more realistic example

Example: #include (realistic case)

In real programs, most of the code actually seen by the compiler may come from `#includes`

- From this source code:

```
#include <iostream>

using namespace std;

int main() {
  cout << "Hello World" << endl;
  return 0;
}
```

- the compiler sees this

. . . .

Deja-Vu

Compilation Units

Headers

Source Code

A.cpp
```
#include "A.h"
#include "B.h"
declares:
defines: a, fa
calls: fb(a)
```

B.cpp
```
#include "B.h"
declares:
defines: fb
calls: fb(1)
```

C.cpp
```
#include "A.h"
#include "B.h"
declares: b, main
defines: b, main
calls: fa(a,b), fb(a)
    : fb(b)
```

A.h
```
declares: a, fa
defines:
```

B.h
```
declares: fb
defines:
```

Pre-processing

Translation Units

A.i
```
variables: a
functions: fa
calls: fb(a)
```

B.i
```
variables:
functions: fb
calls: fb(1)
```

C.i
```
variables: b
functions: main
calls: fa(a,b), fb(a)
    : fb(b)
```

Compiling

Object Code

A.o
```
a: x1000
fa: x12A0
code: fb(x1000)
```

B.o
```
fb: x2000
code: fb(1)
```

C.o
```
b: x3000
main: x3400
code: fa(a,x3000), fb(a)
    : fb(x3000)
```

Linking

Executable

foo.exe
```
code: x2000(x1000)
    : x2000(1)
    : x12A0(x1000,x3000)
    : x2000(x1000)
    : x2000(x3000)
```

- Code that is in headers (.h files) may actually be compiled many times
- Code that is in compilation unit (.cpp) files will be compiled only once

This distinction will be important later.

2.2 Other Pre-processing Commands

#define

- Used to define macros (symbols that the preprocessor will later substitute for)
  - Sometimes used to supply constants

```
#define VersionNumber "1.0Beta1"

int main() {
   cout << "Running version "
        << VersionNumber
        << endl;
```

– Much more elaborate macros are possible, including ones with parameters

. . . .

#ifdef, #ifndef, #endif

Used to select code based upon whether a macro has been defined:

```
#ifdef __GNUG__
  /* Compiler is gcc/g++ */
#endif
#ifdef _MSC_VER
  /* Compiler is Microsoft Visual C++ */
#endif
```

. . . .

#if, #define, and #include

- All of these macros are used to reduce the amount of code seen by the actual compiler
- Suppose we have three files:

```
#ifndef A2_H
#define A2_H

// This is file A2.h
code from A2.h
```

```
#endif
```

,

```
#ifndef B2_H
#define B2_H

//This is file B2.h
#include "A2.h"
code from B2.h
more code from B2.h

#endif
```

,

```
//This is file C.cpp
#include "A2.h"
#include "B2.h"
code from C2.cpp
```

- We ask the compiler to only run the preprocessor and save the result:

```
g++ -E C2.cpp > C2.i
```

. . . .

Shorter Translation Unit

The result is file `C2.i`

```
# 1 "C2.cpp"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "C2.cpp"

# 1 "A2.h" 1





code from A2.h
# 3 "C2.cpp" 2
# 1 "B2.h" 1





code from B2.h
more code from B2.h
# 4 "C2.cpp" 2
code from C2.cpp
```

- Note that the code from `A2.h` is included only once
- Imagine now, how much we would have saved if if that were `iostream` instead of `A2.h`

. . . .

3 Declarations and Definitions

Common Errors

- Some of the most common error messages are
  - ... is undeclared
  - ... is undefined
  - ... is defined multiple times
- Fixing these requires that you understand the difference between declarations and definitions
  - and how they relate to the program structure

. . . .

Declarations

A *declaration* in C++

- introduces (or repeats) a name for something
- tells what "kind" of thing it is
- gives programmers enough information to use it

. . . .

Definitions

A *definition* in C++

- introduces (or repeats) a name for something
- tells what "kind" of thing it is
- tells what value it has and/or how it works
- gives the compiler enough information to generate this and assign it an address

. . . .

General Rules for Decls & Defs

- All definitions are also declarations.
  - But not vice versa
- A name must be declared before you can write any code that uses it.
- A name can be declared any number of times, as long as the declarations are identical.
- A name must be defined exactly once, somewhere within all the separately compiled files making up a program.

. . . .

3.1 Decls&Defs: Variables

Decls&Defs: Variables

- These are definitions of variables:

```
int x;
string s = "abc";
MyFavoriteDataType mfdt (0);
```

- These are declarations:

```
extern int x;
extern string s;
extern MyFavoriteDataType mfdt;
```

. . . .

3.2 Decls&Defs: Functions

Decls&Defs: Functions

- Declaration:

```
int myFunction (int x, int y);
```

- Definition

```
int myFunction (int x, int y)
{
   return x + y;
}
```

- The declaration provides only the header. The definition adds the body.

. . . .

3.3 Decls&Defs: Data Types

Decls&Defs: Data Types

- Data types in C++ are declared, but never defined.
  - Your textbook is often sloppy about this terminology
- These are declarations:

```
typedef float Weight;
typedef string* StringPointer;
enum Colors {red, blue, green};
struct Money {
```

```
    int dollars;
    int cents;
};
```

- Later we will look at these type declarations

```
class C { ... };
```

. . . .

4 Modules

Organizing Decls & Defs into Files

- A C++ program consists of declarations and definitions.
- These are arranged into files that are combined by
  - linking after separate compilation
  - #include'ing one file into another
- These arrangements must satisfy the general rules:
  - A name must be declared before you can write any code that uses it.
  - A name can be declared any number of times, as long as the declarations are identical.
  - A name must be defined exactly once, somewhere within all the separately compiled files making up a program.

. . . .

Headers and Compilation Units

A typical C++ program is divided into many source code files

- Some are *headers*
  - Typically end in ".h"
  - May be `#included` from many different places
  - May `#include` other headers
  - Not directly compiled
- Some are *compilation units*
  - Typically end in ".cpp", ".cc", or ".C"
  - Should *never* be `#included` from elsewhere
  - May `#include` headers
  - Are directly compiled

. . . .

Can You See Me Now?...Now?...Now?

How often does the compiler process each line of code from a file?

- For headers, any number of times
- For compilation units, exactly once

Therefore a header file can *only* contain things that can legally appear multiple times in a C++ program – *declarations*

. . . .

Division: headers and compilation units

- Header files may contain only declarations
  - specifically, declarations that need to be shared with different parts of the code
- Compilation units may contain declarations and definitions
  - Definitions can *only* appear in a non-header.
- *Never, ever, ever* `#include` a non-header (.cpp) file

. . . .

4.1 Coupling and Cohesion

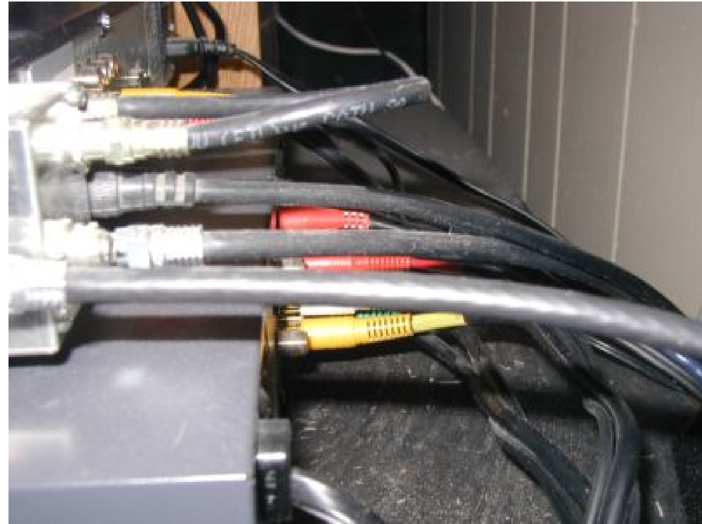
Coupling and Cohesion

- How do we divide things into *different* headers?
- Identify groups of declarations with
  - Low *coupling* - dependencies on other groups
  - High *cohesion* - dependencies within the group

. . . .

Coupling

Something with high coupling has many dependencies on external entities



Something with low coupling has few dependencies on external entities

Cohesion

In something with high cohesion, all the pieces contribute to a well-defined, common goal.

In something with low cohesion, the pieces have little relation to one another

. . . .

Extremely Low Cohesion

. . . .

Dividing into modules

- How do we divide up the compilation units?
- Usually pair them up with a header

  - one compilation unit for each header file
  - The compilation unit provide definitions for each declaration in the header that is it paired with.

- Such pairs are a one of the most common forms of *module*
  - a group of related source code files

. . . .

5 Example of Modularization: the auction program

Read this description of the auction program. It's an example that we will use over and over throughout the semester.

Online Auction

The overall algorithm is pretty simple

```
main (fileNames[]){
  readItems;
  readBidders;
  readBids;
  for each item {
    resolveAuction (item, bidders, bids)
  }
}
```

. . . .

We read in all the information about the auction, then resolve the bidding on each item, one at a time (in order by closing time of the bidding for the items).

The Auction Program before Modularization

We *could* implement that in one file:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

////////////////////////////////////////
// Basic Data Types
////////////////////////////////////////


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


/**
 * The time of the day, to the nearest second.
 */
struct Time {
  int hours;
  int minutes;
  int seconds;
};
```

```
/**
 * A Bid received during an auction
*/
struct Bid {
  std::string bidderName;
  Money amount;
  std::string itemName;
  Time bidPlacedAt;
};


/**
 * Someone registered to participate in an auction
 */
struct Bidder {
  std::string name;
  Money balance;
};


/**
 * An item up for auction
 */
struct Item {
    /// Name of the item
    std::string name;

    /// The minimum acceptable bid set by the seller
    Money reservedPrice;

    /// Time at which the auction ends
```

```
    Time auctionEndsAt;
};


/////////////////////////////////////
// Sequence Data Types
/////////////////////////////////////

/// A collection of bids
struct BidSequence {
    static const int capacity = 5000; ///< Max number allowed
    int size;                         ///< Number currently in array
    Bid data[capacity];               ///< The actual sequence data
};

/// A collection of bidders
struct BidderSequence {
    static const int capacity = 1000; ///< Max number allowed
    int size;                         ///< Number currently in array
    Bidder data[capacity];            ///< The actual sequence data
};

/// A collection of items
struct ItemSequence {
    static const int capacity = 500; ///< Max number allowed
    int size;                         ///< Number currently in array
    Item data[capacity];               ///< The actual sequence data
};


/////////////////////////////////////
```

```
// Global Variables
//////////////////////////////////////

/// The bidders participating in today's auction.
BidderSequence bidders;

/// The bids received for today's auction
BidSequence bids;

/**
 * The collection of all items being auctioned off for the day
 */
ItemSequence items;



//////////////////////////////////////
// Function Declarations
//////////////////////////////////////


/**
 * 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& left, const Money& right);
```

```
bool equal (const Money& left, const Money& right);



/**
 * Find the index of the bidder with the given name. If no such bidder exists,
 * return bidders.size.
 */
 int findBidder (std::string name, const BidderSequence& bidders);



 /**
 * 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& left, const Money& right);




/**
 *  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 noLaterThan(const Time& time1, const Time& time2);
```

```
/**
 * 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, const Money& money);

/**
 * 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
 */

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

/**
 * Read a bid from the indicated file
 */
void read (istream& in, Bid& bid);

/**
 * Read all bids from the indicated file
 */
```

```
 void read (istream& in, BidSequence& bids);

 /**
  * Read a bidder from the indicated file
  */
void read (istream& in, Bidder& bidder);

/**
 * Read all bidders from the indicated file
 */
void read (istream& in, BidderSequence& bidders);



/**
 * Read one item from the indicated file
 */
void read (istream& in, Item& item);



/**
 * Read all items from the indicated file
 */
void read (istream& in, ItemSequence& items);



/**
 * 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, Money& money);

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



/**
 * Determine the winner of the auction for item number i.
 * Announce the winner and remove money from winner's account.
 */
void resolveAuction (const BidSequence& bids, BidderSequence& bidders, const Item& item);


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

```
/////////////////////////////////////
// Function Bodies
/////////////////////////////////////


int main (int argc, char** argv)
{
  if (argc != 4)
    {
      cerr << "Usage: " << argv[0] << " itemsFile biddersFile bidsFile" << endl;
      return -1;
    }

  {
      ifstream itemInput (argv[1]);
      read (itemInput, items);
  }
  {
      ifstream bidderInput (argv[2]);
      read (bidderInput, bidders);
  }
  {
      ifstream bidInput (argv[3]);
      read (bidInput, bids);
```

```
  }


  for (int i = 0; i < items.size; ++i)
    {
      resolveAuction(bids, bidders, items.data[i]);
    }
  return 0;
}




/**
 * 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& left, const Money& right)
{
    Money result;
    result.dollars = left.dollars + right.dollars;
    result.cents = left.cents + right.cents;
    while (result.cents > 99)
    {
     result.cents -= 100;
     ++result.dollars;
    }
    while (result.cents < 0)
    {
```

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




/**
  * 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& left, const Money& right)
 {
    return (left.dollars == right.dollars)
     && (left.cents == right.cents);
 }




/**
 * Find the index of the bidder with the given name. If no such bidder exists,
 * return bidders.size.
 */
 int findBidder (std::string name, const BidderSequence& bidders)
 {
     int found = bidders.size;
     for (int i = 0; i < bidders.size && found == bidders.size; ++i)
```

```
    {
        if (name == bidders.data[i].name)
           found = i;
    }
    return found;
}



 /**
 * 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& left, const Money& right)
{
   if (left.dollars < right.dollars)
    return true;
   else if (left.dollars == right.dollars)
    return left.cents < right.cents;
   return false;
}




/**
 *  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 noLaterThan(const Time& time1, const Time& time2)
{
    // First check the hours
    if (time1.hours > time2.hours)
        return false;
    if (time1.hours < time2.hours)
        return true;
    // If hours are the same, compare the minutes
    if (time1.minutes > time2.minutes)
        return false;
    if (time1.minutes < time2.minutes)
        return true;
    // If hours and minutes are the same, compare the seconds
    if (time1.seconds > time2.seconds)
        return false;
    return true;
}



/**
 * 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, const Money& money){
   out << money.dollars;
```

```
    out << '.';
    if (money.cents < 10)
     out << '0';
    out << money.cents;
}



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




/**
 * Read a bid from the indicated file
 */
void read (istream& in, Bid& bid)
{
    in >> bid.bidderName;
```

```
    read (in, bid.amount);

    read (in, bid.bidPlacedAt);

    string word, line;
    in >> word; // First word of itemName
    getline (in, line); // rest of item name

    bid.itemName = word + line;
}



/**
 * Read all bids from the indicated file
 */
void read (istream& in, BidSequence& bids)
{
    int nBids;
    in >> nBids;
    bids.size = nBids;
    for (int i = 0; i < nBids; ++i)
    {
        Bid bid;
        read (in, bid);
        bids.data[i] = bid;
    }
}



/**
* Read a bidder from the indicated file
```

```
 */
void read (istream& in, Bidder& bidder)
{
    in >> bidder.name;
    read (in, bidder.balance);
}

/**
 * Read all bidders from the indicated file
 */
void read (istream& in, BidderSequence& bidders)
{
    int nBidders;
    in >> nBidders;
    bidders.size = nBidders;
    for (int i = 0; i < nBidders; ++i)
    {
     read (in, bidders.data[i]);
    }
}


/**
 * Read one item from the indicated file
 */
void read (istream& in, Item& item)
{
  read (in, item.reservedPrice);
  read (in, item.auctionEndsAt);

  // Reading the item name.
```

```
  char c;
  in >> c; // Skips blanks and reads first character of the name
  string line;
  getline (in, line); // Read the rest of the line
  item.name = string(1,c) + line;
}



/**
 * Read all items from the indicated file
 */
void read (istream& in, ItemSequence& items)
{
  int nItems;
  in >> nItems;
  items.size = nItems;
  for (int i = 0; i < nItems; ++i)
    read (in, items.data[i]);
}



/**
 * 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, Money& money)
{
    if (!in) return;
    in >> money.dollars;
    if (!in) return;
    if (in.peek() == '.') // if next input is a '.'
    {
     char decimal;
     in >> decimal;
     in >> money.cents;
    } else
     money.cents = 0;
}




/**
 * Determine the winner of the auction for item number i.
 * Announce the winner and remove money from winner's account.
 */
void resolveAuction (const BidSequence& bids, BidderSequence& bidders, const Item& item)
{
  Money zero = {0, 0};
  Money highestBidSoFar = zero;
  string winningBidderSoFar;
  for (int bidNum = 0; bidNum < bids.size; ++bidNum)
    {
      Bid bid = bids.data[bidNum];
```

```
      if (noLaterThan(bid.bidPlacedAt, item.auctionEndsAt))
        {
          if (bid.itemName == item.name
            && lessThan(highestBidSoFar, bid.amount)
            && !lessThan(bid.amount, item.reservedPrice)
            )
            {
              int bidderNum = findBidder(bid.bidderName, bidders);
              Bidder bidder = bidders.data[bidderNum];
              // Can this bidder afford it?
              if (!lessThan(bidder.balance, bid.amount))
                {
                  highestBidSoFar = bid.amount;
                  winningBidderSoFar = bid.bidderName;
                }
            }
        }
    }

    // If highestBidSoFar is non-zero, we have a winner
    if (lessThan(zero, highestBidSoFar))
      {
        int bidderNum = findBidder(winningBidderSoFar, bidders);
        cout << item.name
             << " won by " << winningBidderSoFar
             << " for $";
        print (cout, highestBidSoFar);
        cout << endl;
        bidders.data[bidderNum].balance =
     subtract (bidders.data[bidderNum].balance, highestBidSoFar);
      }
```

```
    else
      {
        cout << item.name
             << " reserve not met"
             << endl;
      }
}



/**
 * Read a time from the indicated stream after skipping any
 * leading whitespace
 */
void read (istream& in, Time& t)
{
  char c;
  in >> t.hours >> c >> t.minutes >> c >> t.seconds;
}




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

```
    result.dollars = left.dollars - right.dollars;
    result.cents = left.cents - right.cents;
    while (result.cents > 99)
    {
     result.cents -= 100;
     ++result.dollars;
    }
    while (result.cents < 0)
    {
     result.cents += 100;
     --result.dollars;
    }
    return result;
}
```

... but it would not be a very good idea.

....

The details of the code for each function body really aren't important right now (with a slight exception that I'll get into later). The most important thing during modularization is to know what the functions and data *are* and what role they play in the program.

A Possible Modularization

From the description of the program and from a glance through the code, we might guess that the key modules would be

Items Data and functions related to the items up for auction

Bidders Data and functions related to the people bidding in the auction

Bids Data and functions related to the bids placed by those people

Money Data and functions related to money

Time Data and functions related to time.

. . . .

(In later lessons we'll talk about other ways to identify good modules.)

Module Files

And we would then expect to divide the program into files corresponding to those modules:

- Group everything describing the items into `items.h` and `items.cpp`.
- Group everything describing the bidders into `bidders.h` and `bidders.cpp`.
- Group everything describing the bids into `bids.h` and `bids.cpp`.
- Group everything for manipulating times into `times.h` and `times.cpp`.
- Group everything for manipulating money into `money.h` and `money.cpp`.
- Put the main program and the core auction algorithm into `auctionMain.cpp`.

. . . .

Making a List. . .

If we then make a list of the data. . .

Types
Money
Time
Bid
Bidders
Item
BidSequence
BidderSequence
ItemSequence

Data
bidders
bids
items

. . . .

Making a List...

...and of a list of the functions in this program...

Functions
add
equal
findBidder
lessThan
noLaterThan
print (money)
print (time)
read (bid)
read (bid sequence)
read (bidder)
read (bidder sequence)
read (item)
read (item sequence)
read (money)
read (time)
resolveAuction
subtract

...and Checking It (at least twice)

...then we can pretty much assign these to our modules just by

- reading their names and, in a few cases,
- looking at the comments in the code that describe them:

Bids *Bid*, *BidSequence*, *bids*, `read` bid, `read` bid seq.

Bidders *Bidder*, *BidderSequence*, *bidders*, `findBidder`, `read` bidder, `read` bidder seq.

Items *Item*, *ItemSequence*, *items*, `read` item, `read` item seq.

Money *Money*, `add`, `equal`, `lessThan`, `print`, `read`, `subtract`

Time *Time*, `noLaterThan`, `print`, `read`

?? `resolveAuction`, `main`

The “application”

The final two functions, `resolveAuction` and `main`, constitute the core algorithm, the "main application" for this particular program, and so can be kept in the main cpp file.

. . . .

5.1 Dividing Things Up

Dividing Things Up

We can now prepare the modular version of this program.

- By default, each function or variable that we have identified gets
  - declared in its module’s header file and
  - defined in its module’s implementation file.

. . . .

Headers are for Sharing

We can reduce possible coupling, though, by looking at the actual function bodies and

- checking to see if any of these declarations would only be used internally within a single module.
  - A declaration only needs to appear in the header file if some code outside the module is using it.
  - If it is only used from within its own module. it can be kept "hidden" inside that module’s .cpp file.

. . . .

Not Shared

For example, in the *Items* module, the function `read` that reads a single item is only called from inside the function `read` that reads an entire sequence of items.

- Because that first `read` is only called by a function within its own module, it does not need to be listed in `items.h`.

. . . .

Possible Division

- Bids

```
#ifndef BIDS_H
#define BIDS_H

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


/**
 * A Bid received during an auction
*/
struct Bid {
  std::string bidderName;
  Money amount;
  std::string itemName;
  Time bidPlacedAt;
};


/// A collection of bids
struct BidSequence {
    static const int capacity = 5000; ///< Max number allowed
    int size;                         ///< Number currently in array
    Bid data[capacity];               ///< The actual sequence data
};
```

```
/// The bids received for today's auction
extern BidSequence bids;




/**
 * Read all bids from the indicated input
 */
void read (std::istream& in, BidSequence& bids);



#endif
```

```
#include <string>
#include <fstream>

using namespace std;

//
//  Bids Received During Auction
//

#include "bids.h"



// Bids received
BidSequence bids;
```

```
/**
 * Read a bid from the indicated file
 */
void read (istream& in, Bid& bid)
{
    in >> bid.bidderName;
    read (in, bid.amount);

    read (in, bid.bidPlacedAt);

    string word, line;
    in >> word; // First word of itemName
    getline (in, line); // rest of item name

    bid.itemName = word + line;
}


/**
 * Read all bids from the indicated file
 */
void read (istream& in, BidSequence& bids)
{
    int nBids;
    in >> nBids;
    bids.size = nBids;
    for (int i = 0; i < nBids; ++i)
    {
        Bid bid;
```

```
        read (in, bid);
        bids.data[i] = bid;
    }
}
```

- Bidders

```
#ifndef BIDDERS_H
#define BIDDERS_H

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

//
//  Bidders Registered for auction
//



/**
 * Someone registered to participate in an auction
 */
struct Bidder {
  std::string name;
  Money balance;
};

/// A collection of bidders
struct BidderSequence {
    static const int capacity = 1000; ///< Max number allowed
```

```
    int size;                                ///< Number currently in array
    Bidder data[capacity];                   ///< The actual sequence data
};


/// The bidders participating in today's auction.
extern BidderSequence bidders;


/**
 * Read all bidders from the indicated file
 */
void read (std::istream& in, BidderSequence& bidders);


/**
 * Find the index of the bidder with the given name. If no such bidder exists,
 * return bidders.size.
 */
 int findBidder (std::string name, const BidderSequence& bidders);



#endif
```

```
#include <string>
#include <fstream>
#include <iostream>
//
//  Bidders Registered for auction
//
```

```
#include "bidders.h"

using namespace std;



BidderSequence bidders;



/**
 * Read a bidder from the indicated file
 */
void read (istream& in, Bidder& bidder)
{
    in >> bidder.name;
    read (in, bidder.balance);
}

/**
 * Read all bidders from the indicated file
 */
void read (istream& in, BidderSequence& bidders)
{
    int nBidders;
    in >> nBidders;
    bidders.size = nBidders;
    for (int i = 0; i < nBidders; ++i)
    {
     read (in, bidders.data[i]);
```

```
    }
}



/**
 * Find the index of the bidder with the given name. If no such bidder exists,
 * return bidders.size.
 */
 int findBidder (std::string name, const BidderSequence& bidders)
 {
     int found = bidders.size;
     for (int i = 0; i < bidders.size && found == bidders.size; ++i)
     {
         if (name == bidders.data[i].name)
            found = i;
     }
     return found;
 }
```

- Items

```
#ifndef ITEMS_H
#define ITEMS_H

#include <string>

#include "money.h"
#include "times.h"

/**
 * An item up for auction
```

```
 */
struct Item {
    /// Name of the item
    std::string name;

    /// The minimum acceptable bid set by the seller
    Money reservedPrice;

    /// Time at which the auction ends
    Time auctionEndsAt;
};

/// A collection of items
struct ItemSequence {
    static const int capacity = 500; ///< Max number allowed
    int size;                         ///< Number currently in array
    Item data[capacity];               ///< The actual sequence data
};


/**
 * The collection of all items being auctioned off for the day
 */
extern ItemSequence items;


/**
 * Read all items from the indicated input
 */
void read (std::istream& in, ItemSequence& items);
```

```
#endif
```

```
#include <iostream>
#include <fstream>

#include "items.h"

//
//  Items up for auction
//

using namespace std;


ItemSequence items;



/**
 * Read one item from the indicated file
 */
void read (istream& in, Item& item)
{
  read (in, item.reservedPrice);
  read (in, item.auctionEndsAt);

  // Reading the item name.
  char c;
  in >> c; // Skips blanks and reads first character of the name
```

```
  string line;
  getline (in, line); // Read the rest of the line
  item.name = string(1,c) + line;
}



/**
 * Read all items from the indicated file
 */
void read (istream& in, ItemSequence& items)
{
  int nItems;
  in >> nItems;
  items.size = nItems;
  for (int i = 0; i < nItems; ++i)
    read (in, items.data[i]);
}
```

- Money

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

#ifndef MONEY_H_
#define MONEY_H_
```

```cpp
#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, Money& money);

/**
 * 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, const Money& money);

/**
 * 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& left, 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& left, 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& left, 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& left, const Money& right);


#endif /* MONEY_H_ */
```

```
/*
 * 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 read (std::istream& in, Money& money)
{
    if (!in) return;
    in >> money.dollars;
    if (!in) return;
    if (in.peek() == '.') // if next input is a '.'
    {
     char decimal;
     in >> decimal;
     in >> money.cents;
    } else
     money.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 print (std::ostream& out, const Money& money){
    out << money.dollars;
    out << '.';
    if (money.cents < 10)
     out << '0';
    out << money.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 equal (const Money& left, const Money& right)
{
    return (left.dollars == right.dollars)
     && (left.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 lessThan (const Money& left, const Money& right)
```

```
{
    if (left.dollars < right.dollars)
     return true;
    else if (left.dollars == right.dollars)
     return left.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 add (const Money& left, const Money& right)
{
    Money result;
    result.dollars = left.dollars + right.dollars;
    result.cents = left.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 subtract (const Money& left, const Money& right)
{
    Money result;
    result.dollars = left.dollars - right.dollars;
    result.cents = left.cents - right.cents;
    while (result.cents > 99)
    {
     result.cents -= 100;
     ++result.dollars;
    }
    while (result.cents < 0)
    {
     result.cents += 100;
     --result.dollars;
    }
    return result;
}
```

- Time

```
#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, Time& time);


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

/**
 *  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 noLaterThan(const Time& time1, const Time& time2);

#endif // TIMES_H
```

```
#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 read (istream& in, Time& t)
{
  char c;
  in >> t.hours >> c >> t.minutes >> c >> t.seconds;
}



/**
 * Print a time in the format HH:MM:SS (two digits each)
```

```
 */
void print (std::ostream& out, 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;
}

/**
 *  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 noLaterThan(const Time& time1, const Time& time2)
{
    // First check the hours
    if (time1.hours > time2.hours)
        return false;
    if (time1.hours < time2.hours)
        return true;
    // If hours are the same, compare the minutes
    if (time1.minutes > time2.minutes)
        return false;
```

```
    if (time1.minutes < time2.minutes)
        return true;
    // If hours and minutes are the same, compare the seconds
    if (time1.seconds > time2.seconds)
        return false;
    return true;
}
```

. . . .

Main Auction Program

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

#include "items.h"
#include "bidders.h"
#include "bids.h"
#include "times.h"
#include "sequence.h"


/**
 * Determine the winner of the auction for item number i.
 * Announce the winner and remove money from winner's account.
 */
void resolveAuction (const BidSequence& bids, BidderSequence& bidders, const Item& item);
```

```
int main (int argc, char** argv)
{
  if (argc != 4)
    {
      cerr << "Usage: " << argv[0] << " itemsFile biddersFile bidsFile" << endl;
      return -1;
    }

  {
      ifstream itemInput (argv[1]);
      read (itemInput, items);
  }
  {
      ifstream bidderInput (argv[2]);
      read (bidderInput, bidders);
  }
  {
      ifstream bidInput (argv[3]);
      read (bidInput, bids);
  }


  for (int i = 0; i < items.size; ++i)
    {
      resolveAuction(bids, bidders, items.data[i]);
    }
  return 0;
}
```

```
/**
 * Determine the winner of the auction for item number i.
 * Announce the winner and remove money from winner's account.
 */
void resolveAuction (const BidSequence& bids, BidderSequence& bidders, const Item& item)
{
  Money zero = {0, 0};
  Money highestBidSoFar = zero;
  string winningBidderSoFar;
  for (int bidNum = 0; bidNum < bids.size; ++bidNum)
    {
      Bid bid = bids.data[bidNum];
      if (noLaterThan(bid.bidPlacedAt, item.auctionEndsAt))
        {
          if (bid.itemName == item.name
            && lessThan(highestBidSoFar, bid.amount)
            && !lessThan(bid.amount, item.reservedPrice)
            )
            {
              int bidderNum = findBidder(bid.bidderName, bidders);
              Bidder bidder = bidders.data[bidderNum];
              // Can this bidder afford it?
              if (!lessThan(bidder.balance, bid.amount))
                {
                  highestBidSoFar = bid.amount;
                  winningBidderSoFar = bid.bidderName;
                }
            }
        }
    }
```

```
    // If highestBidSoFar is non-zero, we have a winner
    if (lessThan(zero, highestBidSoFar))
      {
        int bidderNum = findBidder(winningBidderSoFar, bidders);
        cout << item.name
             << " won by " << winningBidderSoFar
             << " for $";
        print (cout, highestBidSoFar);
        cout << endl;
        bidders.data[bidderNum].balance =
     subtract (bidders.data[bidderNum].balance, highestBidSoFar);
      }
    else
      {
        cout << item.name
             << " reserve not met"
             << endl;
      }
}
```

. . . .