A DISSECTION SERIES

A Dissection of Bitcoin

Paul Huang

A Dissection of Bitcoin

Paul Huang

1. Introduction

Bitcoin is the first digital currency in human history that does not require a central clearing authority. The technological implication of this invention is profound: it opens the possibility of building a completely distributed financial system where no centralized authorities are needed to conduct financial transactions. Some even venture to predict that such a system will eventually lead to the end of central banking and the cease of nation-states.

This book is the ultimate answer to the often asked, but even-more-often unsatisfactorily answered question: HOW DOES BITCOIN WORK?

Nowadays, you can easily find many answers to this question on the Web: from blogs to magazine articles, to books, to video lectures. The actual contents of these answers vary widely: from hand-waving introductions to conceptual accounts of the ideas behind Bitcoin, to painstaking illustrations of the nitty-gritty details of the bits-and-bytes layouts of the Bitcoin protocol.

After spending countless hours on reading many of these answers, somehow, I was still unsatisfied. If Bitcoin is a black-box, then the best among all these answers provide a blueprint of this black-box. A blueprint conceptually documents an engineering design. It is most useful for those who already understand the design, since it summarizes the ideas behind the design and the important inner mechanisms of the system. For novices who have little or no knowledge about the design, though of course helpful, it adds another layer of abstraction and filters out important implementation details, and thus hinders a thorough understanding of the system.

This book takes a different approach. Instead of presenting you yet another conceptual account of Bitcoin, I will dissect the Bitcoin source code: I will open the black-box, examine all its parts, and show you how to put them back together to build a complete system.

Code is the ultimate expression of design. It reflects the behavior of a software system as it is, and it reveals the inner-workings of the system in a plain and straightforward way. The Bitcoin source code is the ultimate answer to the question “how Bitcoin works”. Indeed, it is the ultimate answer to any question you can ever possibly ask about the Bitcoin system.

1.1. Bitcoin source code

The Bitcoin source code is freely available to everyone to download from GitHub (http://github.com). After Satoshi Nakamoto released the original code and disappeared from the public eye in 2010, the Bitcoin Foundation took over the control of the subsequent developments.

This books covers version **v0.1.5 ALPHA** of the Bitcoin source code. It is one of the original versions written by Satoshi and the earliest version available in GitHub. It contains about 16,000 lines of code (after removing all the empty lines), and thus a complete coverage of the whole code base in a book is a manageable task. The later versions developed by the Bitcoin Foundation did not change the original architecture. Understanding this original version will give you a solid foundation for further exploring of the later versions. Indeed, if you compare this version with the later ones, you will find that all the important classes are unchanged, and that many important functions are kept the same name.

Another reason to cover this version is that its coding style is classic: dense and concise. That is how Satoshi managed to implement such a complicated system in just 16,000 lines of code. This version keeps all the system states in global variables. Once you get familiar with these variables and understand what they do, which, of course, takes some time and efforts, reading the code becomes a fairly enjoyable endeavor. You will not suffer from the typical code-reading fatigue caused by a programming style in which "everything happens somewhere else" (a quote from Adele Goldberg). Everything happens right there in the place where it supposes to happen, cleanly and precisely.

1.2. Who should read this book

This book is intended for readers who want to acquire a thorough knowledge of Bitcoin. After finishing this book, you will not only appreciate the revolutionary ideas behind Bitcoin, but also acquire in-depth knowledge of the design principle of the Bitcoin software system, and master the implementation details of the system.

This book is also beneficial to professionals and students who want to improve their C++ skills. Reading good code is a great way to master a programming language. Of course, there are many excellent open-source C++ projects out there you can learn from, but why not study Satoshi's code so that you can master both the language and Bitcoin at the same time?

1.3. Prerequisite

You don't have to be a computer genius to understand this book. However, you do need to have some basic knowledge of the C++ programming Language to proceed. Understanding any basic C++ tutorial, for example, the "C++ Language Tutorial" at http://www.cplusplus.com, will be sufficient.

That being said, this is not a light book to read. This book is technical in nature. It covers many implement details. You will have to understand one detail before

moving on to the next one; and there seem to be endless of them. The best way to read this book is to be patient and persistent, take on one detail at a time. After working out enough details, you will reach a tipping point where a big picture will emerge. From that point on, everything will fall into its own place, and the whole structure of the system will become crystal clear to you.

1.4. Free e-book and full version

The first 4 chapters of this book is published as a free e-book at Amazon and Apple iBooks Store. You can find it by searching "A Dissection of Bitcoin" at one of these on-line stores.

The full version consists of 10 chapters. It can be purchased at

https://ebook.ubiqlink.com/.

1.5. How this book is organized

Before laying out the organization of this book, I'll have to first give a ten-thousand-feet overview on what a Bitcoin Application (thus abbreviated as BA throughout the book) compiled from the source code does. A BA needs to handle *four major tasks*:

- **Construct a transaction**. The primary function of a BA is to send bitcoins to someone. To do so, you need to first create a "transaction", which is just a fancier word for a record that records who paid how many bitcoins to whom. Let's say you want to pay 5 bitcoins to Bob. What you need to do is to first start a BA on you computer, then input Bob's id and the amount you want to send to him. Click a button, your BA will create a transaction for you.
- **Communicate with other BAs**. Next, your BA will broadcast this newly created transaction to all other BAs that are currently running on the Internet. The reason for doing that is to inform everyone that you are willing to pay Bob 5 bitcoins. To achieve that, all BAs employ a peer-to-peer protocol to communicate with each other. Peer-to-peer simply means that any BA can initiate a communication to any other BA directly; no intermediaries are needed. Just as your BA can send your transactions to any other BA, any other BA can send its transactions to you as well.
- **Mine a block**. After receiving enough numbers of transactions, a BA will pack them together to form a block, and then start "mining" it. Mining is process of hashing the block data into a certain format. The format is specified in such a way that it takes considerable amount of computational power to find a hash compatible with that format. Once such a hash is found, i.e., a block is mined, that block will be broadcast to all BAs.

- **Maintain the blockchain**. After receiving a new block, each BA will first verify that all the transactions in the received block are valid, and then check that the block hash is indeed compatible with the specified format. After that, this new block will be appended to the blocks mined previously: one after another to form a blockchain. Once included in the blockchain, a block is sealed and can not be changed anymore. Each BA has its own blockchain. They communicate via the Bitcoin peer-to-peer protocol to reach a consensus on which blockchain is the final ground-truth. And every BA then adjusts its own blockchain to converge to the ground-truth.

This book is organized along with the four major tasks listed above.

- *Chapters Transaction I* covers the first task: "construct a transaction". This chapter focuses on how to create a transaction in the computer's main memory.
- *Chapter Serialization* provides necessary details to understand *Chapter Transaction I*. This chapter explains how to serialize classes to disk and network.
- *Chapter Script* describes how to parse and execute Bitcoin scripts embedded in every transaction.
- *Chapter Block* covers all the data structures related to block and blockchain.
- *Chapter Database* covers the Bitcoin database.
- *Chapter Transaction II* continues the coverage of the first task. This chapter shows how to save the newly created transaction to the database.
- *Chapters Network* covers the second task: "communicate with other BAs". This chapter examines the Bitcoin peer-to-peer protocol in details.
- *Chapter Blockchain* covers the forth task: "maintain the blockchain".
- *Chapter Mining* covers the third task: "mine a block".

1.6. Compile the source code

This section provides a detailed guide on how to compile the Bitcoin source code. It is always a good practice to compile the code before analyzing it.

Bitcoin version "v0.1.15 ALPHA" compiles only on Windows. It does not work on Linux or MacOS. So you need a Windows machine to proceed. Please follow the steps listed below closely, it will save you a lot of time to fight through compilation errors.

- The `MinGW` tool-chain is used to compile the code. So the first thing to do is to install `MinGW` from http://www.mingw.org/. Download and run an installer

called “mingw-get-setup.exe”. It will subsequently download and start a program called “MinGW Installation Manager”, which will ask you to select `MinGW` packages to install.

  - Select “Basic Setup” on the left panel, then check packages “mingw32-base”, “mingwin32-gcc-g++” (the g++ compiler), and “msys-base” (a unix-like command-line shell) on the right panel.
  - Select “All Packages” → “MSYS” → “MinGW Developer Toolkit” on the left panel, then select “msys-perl” package on the right panel. The Perl package is needed later to compile the `openssl` library.
  - Click menu “Installation” → “Apply Changes” to install the selected packages.

- `MinGW` is installed into the default directory “C:\MinWG”. After the installation, add “C:\MinGW\bin” into the “Path” environment variable of Windows so that all commands in “C:\MinGW\bin” can be invoked from the command-line.
- Install `Git` from https://git-scm.com/.
- Run the `Git` program you just installed. This will bring up a command-line shell. In that shell, make a new directory “bc” under whatever directory you like. Get into directory “bc”, issue a “git clone” command to download the Bitcoin source code. This will create a sub-directory “bitcoin” under “bc”. Here are the commands you should issue in the `Git` command-line shell.

```
1: mkdir bc
2: cd bc
3: git clone https://github.com/bitcoin/bitcoin.git
```

- Now check out the earliest Bitcoin version available in the downloaded `Git` repository.

```
1: git checkout 4405b78d6059e536c36974088a8ed4d9f0f29898
```

- There should be a file named “readme.txt” in directory “bc/bitcoin”. Open that file. The first line should be “BitCoin v0.1.5 ALPHA”. This is the version number of the source code this book covers.
- Download `wxWidgets-2.8.12` (wxWidgets version 2.8.12) from https://www.wxwidgets.org/.
  Unzip the source files to directory “bc/wxWidgets-2.8.12”. Bring up the `MinGW` command-line shell by invoking a batch file at “C:\MinGW\msys\1.0\msys.bat”. Issue the following commands in the `MinGW` command-line shell to compile the `wxWidgets` library.

```
1: cd bc/wxWidgets-2.8.12
2: ./configure --with-msw --enable-debug --enable-debug_gdb --
```

```
disable-shared
    3: make
```

- Download `openssl-1.0.2d` (openssl version 1.0.2d) from https://www.openssl.org/.
  Unzip the source files to directory "bc/openssl-1.0.2d". In the `MinGW` command-line shell, go to directory "bc/openssl-1.0.2d" and issue the following command to compile the `openssl` library.

```
    1: cd bc/openssl-1.0.2d
    2: ./config
    3: make
```

- Download `db-4.8.30.NC` (Berkeley DB version 4.8.30) from http://www.oracle.com/.
  Unzip the source files to directory "bc/db-4.8.30.NC". In the `MinGW` command-line shell, go to directory "bc/db-4.8.30.NC/build_unix" and issue the following command to compile the `BerkeleyDB` library.

```
    1: cd bc/db-4.8.30.NC/build_unix
    2: ../dist/configure --enable-mingw --enable-cxx
    3: make
```

- Download `boost_1_35_0` (boost version 1.35.0) from http://www.boost.org/.
  Unzip the source files to directory "bc/boost_1_35_0".
- Go to directory "bc/bitcoin", modify the make file "makefile" to reflect the locations of the headers and compiled libraries needed to build Bitcoin. Three variables in "makefile" need to be changed, they are `INCLUDEPATHS`, `LIBPATHS`, and `LIBS`. The modified variables are shown below. Copy and paste the changes to "makefile". Don't touch the rest of the file.

```
    1: INCLUDEPATHS=-I"../boost_1_35_0" \
    2:              -I"../db-4.8.30.NC/build_unix" \
    3:              -I"../openssl-1.0.2d/include" \
    4:              -I"../wxWidgets-2.8.12/lib/wx/include/msw-ansi-
release-2.8" \
    5:              -I"../wxWidgets-2.8.12/include"
    6: LIBPATHS=-L"../db-4.8.30.NC/build_unix" \
    7:          -L"../openssl-1.0.2d" \
    8:          -L"../wxWidgets-2.8.12/lib"
    9: LIBS= \
   10:  -l db_cxx \
   11:  -l crypto \
   12:  -l wx_msw$(D)_richtext-2.8 \
   13:  -l wx_msw$(D)_html-2.8 \
   14:  -l wx_msw$(D)_core-2.8 \
   15:  -l wx_base$(D)-2.8 \
   16:  -l wxtiff$(D)-2.8 \
   17:  -l wxjpeg$(D)-2.8 \
   18:  -l wxpng$(D)-2.8 \
```

```
   19:  -l wxzlib$(D)-2.8 \
   20:  -l wxregex$(D)-2.8 \
   21:  -l wxexpat$(D)-2.8 \
   22:  -l kernel32 -l user32 -l gdi32 -l comdlg32 -l winspool -l winmm -
l shell32 \
   23:  -l comctl32 -l ole32 -l oleaut32 -l uuid -l rpcrt4 -l advapi32 -l
ws2_32
```

- Issue a “make” command in directory “bc/bitcoin” in the `MinGW` command-line shell. You will get an executable name “bitcoin.exe” in directory “bc/bitcoin”.

2. Transaction I

This chapter covers the first of the *four major tasks* laid out in *Chapter Introduction*: "construct a transaction". To send bitcoins to someone, you need to know the recipient's Bitcoin address. This chapter first examines how to generate such an address. Then it analyzes the code that constructs a new transaction.

2.1. Bitcoin address

In *Chapter Introduction*, a BA was compiled from the Bitcoin source code. Now start that BA, select menu "Options" → "Change Your Address", then click the "New Address" button, a dilog-box will pop up. This dialog-box is opened by function `CYourAddressDialog::OnButtonNew()` in `ui.cpp`. Here is the source code of this function.

```
    1: void CYourAddressDialog::OnButtonNew(wxCommandEvent& event)
    2: {
    3:     // Ask name
    4:     CGetTextFromUserDialog dialog(this, "New Bitcoin Address",
"Label", "");
    5:     if (!dialog.ShowModal())
    6:         return;
    7:     string strName = dialog.GetValue();
    8:     // Generate new key
    9:     string strAddress = PubKeyToAddress(GenerateNewKey());
   10:     SetAddressBookName(strAddress, strName);
   11:     // Add to list and select it
   12:     int nIndex = InsertLine(m_listCtrl, strName, strAddress);
   13:     SetSelection(m_listCtrl, nIndex);
   14:     m_listCtrl->SetFocus();
   15: }
```

The work-flow of this function is as follows:

- Show a dialog-box to accept a user-inputed string, and then put the string into local variable `strName` (lines 5-7)
- Call `GenerateNewKey()` in `main.cpp` to generate a new public-private key pair, and then calls `PubKeyToAddress()` in `base58.h` to generate a string representation `strAddress` of the newly generated public key (line 9)
- Add `strName` and `strAddress` as a pair to `m_llistCtrl`, which contains all the Bitcoin addresses you own, and will be shown on the GUI (line 12).

Let's take a look at `GenerateNewKey()`. Here is the source code of this function.

```
    1: bool AddKey(const CKey& key)
    2: {
    3:     CRITICAL_BLOCK(cs_mapKeys)
```

```
    4:     {
    5:         mapKeys[key.GetPubKey()] = key.GetPrivKey();
    6:         mapPubKeys[Hash160(key.GetPubKey())] = key.GetPubKey();
    7:     }
    8:     return CWalletDB().WriteKey(key.GetPubKey(),
key.GetPrivKey());
    9: }
   10: vector<unsigned char> GenerateNewKey()
   11: {
   12:     CKey key;
   13:     key.MakeNewKey();
   14:     if (!AddKey(key))
   15:         throw runtime_error("GenerateNewKey() : AddKey failed\n");
   16:     return key.GetPubKey();
   17: }
```

This function takes the following steps to generate a new public-private key pair.

- Generate a new object of type `CKey` (line 13).
- Call `addKey()` to save the newly generated key to 1) global map `mapKeys` (line 5), 2) global map `mapPubKeys` (line 6), and 3) wallet database `wallet.dat` (line 8), which will be covered in *Chapter Database*.
  - `mapKeys` maps the public key to the corresponding private key (line 5).
  - `mapPubKeys` maps the hash of the public key to the public key itself (line 6).
- Return the public key (line 16).

> The returned public key from `GenerateNewKey()` is a vector of unsigned chars of size 65. It has the following format:
>
> `(0x04 | PubKeyX(32 bits) | PubKeyY(32 bits))`.
>
> The prefix byte is used to distinguish between several encodings, with `0x04` denoting uncompressed `PubKeyX` and `PubKeyY`. This is a standard way of encoding the public key, managed by the `openssl` library. Bitcoin treats the returned public key as it is.

Important

> Macro `CRITICAL_BLOCK` at line 3 acquires a lock to access global variables `mapKeys` and `mapPubKeys`. A BA maintains multiple threads. To avoid corrupting a global variable that is shared among multiple threads, any thread must acquire a lock first before making any changes to the global variable. A lock is defined for each global variable. For example, lock `cs_mapKeys` is defined for `mapKeys` and `mapPubKeys`. Line 3-7 represents a general pattern to write to a global variable in the Bitcoin source code. You

> will see this pattern frequently throughout the book.

Now go back to function `CYourAddressDialog::OnButtonNew()`. After getting the returned public key from `GenerateNewKey()`, `CYourAddressDialog::OnButtonNew()` calls `PubKeyToAddress()` in `base58.h` and passes the return public key as the input. Here is the source code of `PubKeyToAddress()`.

```
    1: inline string Hash160ToAddress(uint160 hash160)
    2: {
    3:     // add 1-byte version number to the front
    4:     vector<unsigned char> vch(1, ADDRESSVERSION);
    5:     vch.insert(vch.end(), UBEGIN(hash160), UEND(hash160));
    6:     return EncodeBase58Check(vch);
    7: }
    8: inline string PubKeyToAddress(const vector<unsigned char>&
vchPubKey)
    9: {
   10:     return Hash160ToAddress(Hash160(vchPubKey));
   11: }
```

This function passes the 65 bits public key to `Hash160()` in `util.h` (line 10), which returns the hash of the public key. It then passes the returned hash to `Hash160ToAddress()` in `base58.h` to get a string representation of the hash (line 10). The final string is a Bitcoin address, which looks like a long string of random letters and numbers. To give an example, `171zNQDkKGpbvbLHHyJD4Csg393er4xnT6` is a Bitcoin address.

Summary

The procedure of generating a Bitcoin address is as follows: 1) generate a public-private key pair, 2) hash the public key, 3) encode the hash into a string, and the final string is a Bitcoin address.

2.2. `CTransaction` in `main.h`

`CTransaction` represents a Bitcoin transaction that transfers a certain amount of coins from one user to another. It is the one of the critical data structures of Bitcoin. The first task "construct a transaction" we are about to explore revolves around this class.

```
    1: class COutPoint
    2: {
    3: public:
    4:     uint256 hash;
    5:     unsigned int n;
    6:     //......
    7: };
    8: class CTxIn
    9: {
```

```
10: public:
11:     COutPoint prevout;
12:     CScript scriptSig;
13:     unsigned int nSequence;
14:     //......
15: };
16: class CTxOut
17: {
18: public:
19:     int64 nValue;
20:     CScript scriptPubKey;
21:     //......
22: };
23: class CTransaction
24: {
25: public:
26:     int nVersion;
27:     vector<CTxIn> vin;
28:     vector<CTxOut> vout;
29:     int nLockTime;
30:     //......
31: };
```

A `CTransaction` contains an input `vin`, and an output `vout`. Both are vectors. Each input slot of vector `vin` is of type `CTxIn`; and each output slot of vector `vout` is of type `CTxOut`.

Each input slot (of type `CTxIn`) of a transaction `Tx` contains a `COutPoint` object `prevout`, which refers to **an output slot of a source-transaction** of `Tx`. A source-transaction of `Tx` is a transaction from which `Tx` gets the coins it is about to spend. A transaction `Tx` can have an arbitrary number of source-transactions.

Any transaction is uniquely identified by its hash code, a 256-bit data structure `uint256`. To refer to a specific output slot of a specific source-transaction `TxSource` of a transaction `Tx`, two pieces of information are needed: 1) the hash code of `TxSource`, and 2) the index `n` that points to a specific output slot of `TxSource`. These two pieces of information are held in class `COutPoint`. And the specific output slot a `COutPoint` object refers to is `TxSource.vout[n]`. If this output slot of `TxSource` is referred by the `i`-th input slot of `tx` (i.e., `tx.vin[i].prevout`), we say that **the `i`-th input of `tx` spends the `n`-th output of `TxSource`**.

`COutPoint` holds a hash code of type `uint256`, which is the hash of the source-transaction. We cover this class next.

2.2.1. `uint256` and `uint160` in `uint.h`

An `uint256` holds a 256-bit hash code. It contains an `unsigned int` array of length `256/32=8` to hold the hash code. Another similar data structure `uint160`, defined in the same header file, holds a 160-bit hash code. It has an `unsigned`

`int` array of length `160/32=5` to hold the hash code. These two classes share the same base class `base_uint`.

```
    1: template<unsigned int BITS>
    2: class base_uint
    3: {
    4: protected:
    5:     enum { WIDTH=BITS/32 };
    6:     unsigned int pn[WIDTH];
    7:     //......
    8:     unsigned int GetSerializeSize(int nType=0, int
nVersion=VERSION) const
    9:     {
   10:         return sizeof(pn);
   11:     }
   12:     template<typename Stream>
   13:     void Serialize(Stream& s, int nType=0, int nVersion=VERSION)
const
   14:     {
   15:         s.write((char*)pn, sizeof(pn));
   16:     }
   17:     template<typename Stream>
   18:     void Unserialize(Stream& s, int nType=0, int nVersion=VERSION)
   19:     {
   20:         s.read((char*)pn, sizeof(pn));
   21:     }
   22:     //......
   23: }
   24: typedef base_uint< 160 > base_uint160;
   25: typedef base_uint< 256 > base_uint256;
   26: class uint160 : public base_uint160
   27: {
   28:     //......
   29: };
   30: class uint256 : public base_uint256
   31: {
   32:     //......
   33: };
```

`base_uint` overloads many operators. These operator overload functions are not shown in here. They all work on the inner data storage array `pn` to implement their functions.

One thing worth mentioning is that the three serialization member functions of `base_uint` (`GetSerializeSize()`, `Serialize()`, and `Unserialize()`) follow the same naming convention described in *Chapter Serialization*. So class `base_uint` and its derived classes can be integrated into the serialization/de-serialization chain explained in *Chapter Serialization*.

> Many classes in Bitcoin need to be serialized to a byte stream to be hashed, stored on disk, or sent to peer BA's via the Internet. These classes follow a clever scheme to implement their serialization/de-serialization functions. This scheme is elaborated in *Chapter Serialization*. It is an independent topic that

can be treated separately. If you wish to understand this scheme before proceeding further, you can now go to read *Chapter Serialization* and then come back. Or you can keep moving forward with a simple understanding that serialization writes an instance of a class to a byte stream, and de-serialization reads the byte stream to re-construct the instance.

2.3. Send bitcoins

Now start the BA, click the "Send Coins" button in the tool-bar (the first button in the tool-bar with a green arrow), the following dialog-box will show up.

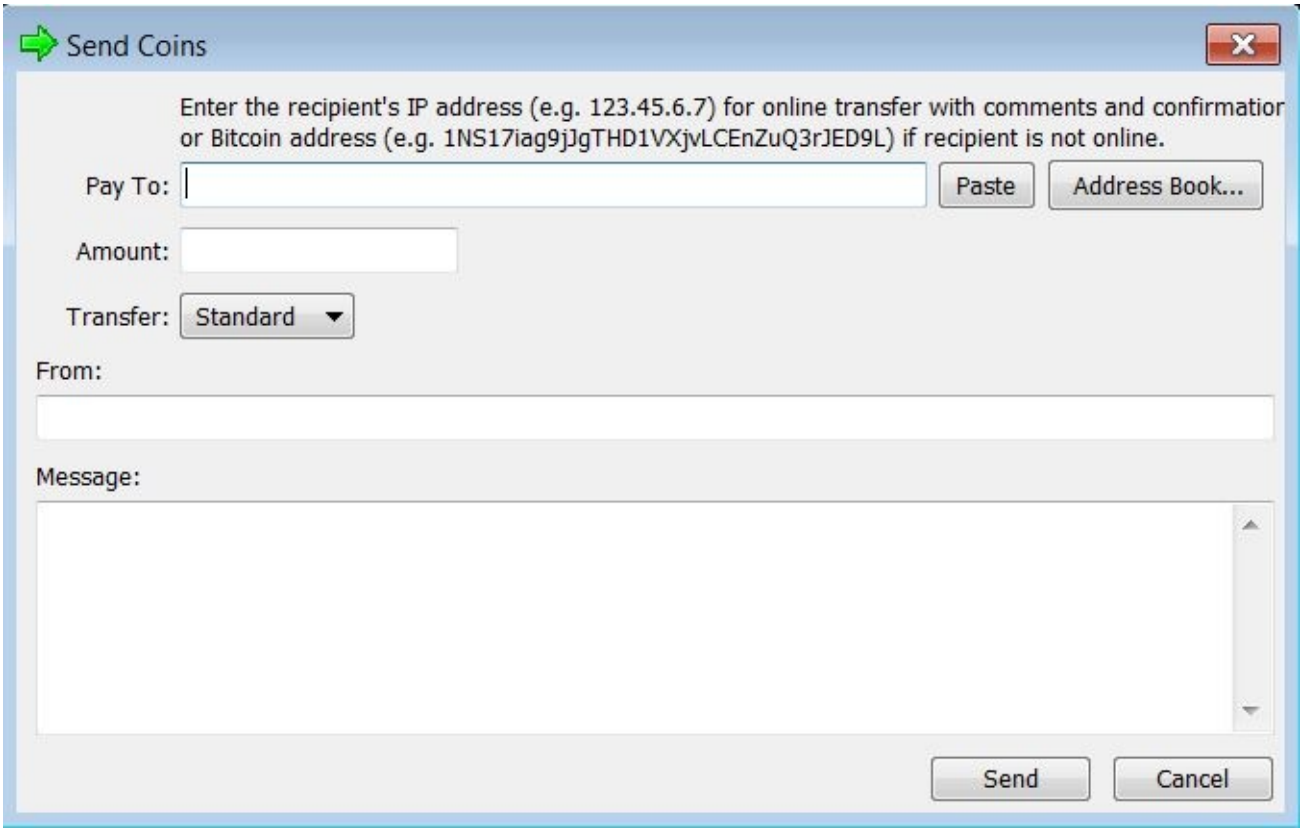

Figure 1. "Send Coins" dialog

This dialog-box is implemented in class `CSendDialog` in `ui.h`, which is derived from class `CSendDialogBase` in `uibase.h`. The latter contains the following 2 text input controls (there are other controls, but these 2 carry information required to construct a transaction):

```
1: wxTextCtrl* m_textCtrlAddress;
2: wxTextCtrl* m_textCtrlAmount;
```

The first control takes the Bitcoin address of a recipient as input. The second one takes the amount of coins to be sent as input. All you need are these two pieces of information to construct a transaction.

Click the "Send" button in the above dialog. Function `CSendDialog::OnButtonSend()` in `ui.cpp` will be called.

2.4. `CSendDialog::OnButtonSend()` in `ui.cpp`

Here is the source code of this function.

```
    1: void CSendDialog::OnButtonSend(wxCommandEvent& event)
    2: {
    3:     CWalletTx wtx;
    4:     string strAddress = (string)m_textCtrlAddress->GetValue();
    5:     // Parse amount
    6:     int64 nValue = 0;
    7:     if (!ParseMoney(m_textCtrlAmount->GetValue(), nValue) ||
nValue <= 0)
    8:     {
    9:         wxMessageBox("Error in amount  ", "Send Coins");
   10:         return;
   11:     }
   12:     if (nValue > GetBalance())
   13:     {
   14:         wxMessageBox("Amount exceeds your balance  ", "Send
Coins");
   15:         return;
   16:     }
   17:     if (nValue + nTransactionFee > GetBalance())
   18:     {
   19:         wxMessageBox(string("Total exceeds your balance when the
") + FormatMoney(nTransactionFee) + " transaction fee is included  ",
"Send Coins");
   20:         return;
   21:     }
   22:     // Parse bitcoin address
   23:     uint160 hash160;
   24:     bool fBitcoinAddress = AddressToHash160(strAddress, hash160);
   25:     if (fBitcoinAddress)
   26:     {
   27:         // Send to bitcoin address
   28:         CScript scriptPubKey;
   29:         scriptPubKey << OP_DUP << OP_HASH160 << hash160 <<
OP_EQUALVERIFY << OP_CHECKSIG;
   30:         if (!SendMoney(scriptPubKey, nValue, wtx))
   31:             return;
   32:         wxMessageBox("Payment sent  ", "Sending…");
   33:     }
   34:     else
   35:     {
   36:         // Parse IP address
   37:         CAddress addr(strAddress.c_str());
   38:         if (addr.ip == 0)
   39:         {
   40:             wxMessageBox("Invalid address  ", "Send Coins");
   41:             return;
   42:         }
   43:         // Message
   44:         wtx.mapValue["to"] = strAddress;
   45:         wtx.mapValue["from"] = m_textCtrlFrom->GetValue();
```

```
   46:         wtx.mapValue["message"] = m_textCtrlMessage->GetValue();
   47:         // Send to IP address
   48:         CSendingDialog* pdialog = new CSendingDialog(this, addr,
nValue, wtx);
   49:         if (!pdialog->ShowModal())
   50:             return;
   51:     }
   52:     if (!mapAddressBook.count(strAddress))
   53:         SetAddressBookName(strAddress, "");
   54:     EndModal(true);
   55: }
```

Here is the work-flow of this function:

- Get the recipient's address and put it into `strAddress` (line 4).
- Get the amount to be sent and put it into `nValue` (line 7).
- Check the balance to make sure that the amount of coins you own is greater than `nValue` plus `nTransactionFee` (line 17).
- Parse the recipient's address to `hash160` (line 24). If it is a hash address, then `fBitcoinAddress` will be `true` (line 25). If it is true, do the following:
    - Insert into object `scriptPubKey` (of type `CScript`) the following script code `OP_DUP OP_HASH160 hash160 OP_EQUALVERIFY OP_CHECKSIG` (line 29). The recipient's address `hash160` is included in this script. To emphasize `hash160` is the recipient's address, it will be referred as `<recipient_address_hash160>` from now on. So the above script code becomes `OP_DUP OP_HASH160 <recipient_address_hash160> OP_EQUALVERIFY OP_CHECKSIG`.
    - Call `SendMoney(scriptPubKey, nValue, wtx)` (line 30).
- If recipient's address is not a hash address (line 34), check if it is an IPv4 address (line 38). If it is, new a `CSendingDialog` class (line 48). You can safely ignore all the code after line 34. No one sends bitcoins to an IP address anymore. It is unsafe. All the code related to sending bitcoins to an IP has been removed from the later versions.

Now let's examine `SendMoney()`.

2.5. `SendMoney()` in `main.cpp`

Here is the source code of this function.

```
    1: bool SendMoney(CScript scriptPubKey, int64 nValue, CWalletTx&
wtxNew)
    2: {
```

```
    3:     CRITICAL_BLOCK(cs_main)
    4:     {
    5:         int64 nFeeRequired;
    6:         if (!CreateTransaction(scriptPubKey, nValue, wtxNew,
nFeeRequired))
    7:         {
    8:             string strError;
    9:             if (nValue + nFeeRequired > GetBalance())
   10:                 strError = strprintf("Error: This is an oversized
transaction that requires a transaction fee of %s  ",
FormatMoney(nFeeRequired).c_str());
   11:             else
   12:                 strError = "Error: Transaction creation failed  ";
   13:             wxMessageBox(strError, "Sending…");
   14:             return error("SendMoney() : %s\n", strError.c_str());
   15:         }
   16:         if (!CommitTransactionSpent(wtxNew))
   17:         {
   18:             wxMessageBox("Error finalizing transaction  ",
"Sending…");
   19:             return error("SendMoney() : Error finalizing
transaction");
   20:         }
   21:         printf("SendMoney: %s\n",
wtxNew.GetHash().ToString().substr(0,6).c_str());
   22:         // Broadcast
   23:         if (!wtxNew.AcceptTransaction())
   24:         {
   25:             // This must not fail. The transaction has already
been signed and recorded.
   26:             throw runtime_error("SendMoney() :
wtxNew.AcceptTransaction() failed\n");
   27:             wxMessageBox("Error: Transaction not valid  ",
"Sending…");
   28:             return error("SendMoney() : Error: Transaction not
valid");
   29:         }
   30:         wtxNew.RelayWalletTransaction();
   31:     }
   32:     MainFrameRepaint();
   33:     return true;
   34: }
```

When this function is called, its 3 parameters take the following values:

- `scriptPubKey` contains script code `OP_DUP OP_HASH160 <recipient_address_hash160> OP_EQUALVERIFY OP_CHECKSIG`.
- `nValue` is the amount to send. Note that transaction fee `nTransactionFee` is not included. `nTransactionFee` is a global variable defined in `main.cpp`.
- `wtxNew` is `wtx`, a local variable of type `CWalletTx` declared in caller `CSendDialog::OnButtonSend()`. Variable `wtx` is an empty transaction waiting to be filled. Its type `CMerkleTx` is derived from class `CTransaction`

with some additional fields that you don't need to worry about at this moment. At for now, just treat `wtxNew` as if it is of type `CTransaction`.

What this function does is straightforward:

- First creates a transaction (`CreateTransaction(scriptPubKey, nValue, wtxNew, nFeeRequired)` at line 6).
- Then tries to commit the transaction into the database (`CommitTransactionSpent(wtxNew)` at line 16).
- If the transaction is accepted (`wtxNew.AcceptTransaction()` at line 23), sends the transaction to other peers (`wtxNew.RelayWalletTransaction()` at line 30).

All these 4 functions work on `wtxNew`. The first function `CreateTransaction()` is covered in this chapter. The other three will be covered in *Chapter Transaction II*.

2.6. `CreateTransaction()` in `main.cpp`

Here is the source code of `CreateTransaction()`.

```
    1: bool CreateTransaction(CScript scriptPubKey, int64 nValue,
CWalletTx& wtxNew, int64& nFeeRequiredRet)
    2: {
    3:     nFeeRequiredRet = 0;
    4:     CRITICAL_BLOCK(cs_main)
    5:     {
    6:         // txdb must be opened before the mapWallet lock
    7:         CTxDB txdb("r");
    8:         CRITICAL_BLOCK(cs_mapWallet)
    9:         {
   10:             int64 nFee = nTransactionFee;
   11:             loop
   12:             {
   13:                 wtxNew.vin.clear();
   14:                 wtxNew.vout.clear();
   15:                 if (nValue < 0)
   16:                     return false;
   17:                 int64 nValueOut = nValue;
   18:                 nValue += nFee;
   19:                 // Choose coins to use
   20:                 set<CWalletTx*> setCoins;
   21:                 if (!SelectCoins(nValue, setCoins)) //
   22:                     return false;
   23:                 int64 nValueIn = 0;
   24:                 foreach(CWalletTx* pcoin, setCoins)
   25:                     nValueIn += pcoin->GetCredit();
   26:                 // Fill vout[0] to the payee, recipient of the
bitcoin
```

```
   27:                 wtxNew.vout.push_back(CTxOut(nValueOut,
scriptPubKey));
   28:                 // Fill vout[1] back to self with any change
   29:                 if (nValueIn > nValue)
   30:                 {
   31:                     /// todo: for privacy, should randomize the
order of outputs,
   32:                     //        would also have to use a new key for
the change.
   33:                     // Use the same key as one of the coins
   34:                     vector<unsigned char> vchPubKey;
   35:                     CTransaction& txFirst = *(*setCoins.begin());
   36:                     foreach(const CTxOut& txout, txFirst.vout)
   37:                         if (txout.IsMine())
   38:                             if (ExtractPubKey(txout.scriptPubKey,
true, vchPubKey))
   39:                                 break;
   40:                     if (vchPubKey.empty())
   41:                         return false;
   42:                     // Fill vout[1] to ourself
   43:                     CScript scriptPubKey;
   44:                     scriptPubKey << vchPubKey << OP_CHECKSIG;
   45:                     wtxNew.vout.push_back(CTxOut(nValueIn -
nValue, scriptPubKey));
   46:                 }
   47:                 // Fill vin
   48:                 foreach(CWalletTx* pcoin, setCoins)
   49:                     for (int nOut = 0; nOut < pcoin->vout.size();
nOut++)
   50:                         if (pcoin->vout[nOut].IsMine())
   51:                             wtxNew.vin.push_back(CTxIn(pcoin-
>GetHash(), nOut));
   52:                 // Sign
   53:                 int nIn = 0;
   54:                 foreach(CWalletTx* pcoin, setCoins)
   55:                     for (int nOut = 0; nOut < pcoin->vout.size();
nOut++)
   56:                         if (pcoin->vout[nOut].IsMine())
   57:                             SignSignature(*pcoin, wtxNew, nIn++);
   58:                 // Check that enough fee is included
   59:                 if (nFee < wtxNew.GetMinFee(true))
   60:                 {
   61:                     nFee = nFeeRequiredRet =
wtxNew.GetMinFee(true);
   62:                     continue;
   63:                 }
   64:                 // Fill vtxPrev by copying from previous
transactions vtxPrev
   65:                 wtxNew.AddSupportingTransactions(txdb);
   66:                 wtxNew.fTimeReceivedIsTxTime = true;
   67:                 break;
   68:             }
   69:         }
   70:     }
   71:     return true;
   72: }
```

When this function is called, its 4 parameters take the follow values:

- `scriptPubKey` contains script code `OP_DUP OP_HASH160 <recipient_address_hash160> OP_EQUALVERIFY OP_CHECKSIG`.
- `nValue` is the amount to transfer, transaction fee `nTransactionFee` is not included.
- `wtxNew` is an empty transaction instance.
- `nFeeRequiredRet` is an output that will carry the actual transaction fee after this function returns.

Here is the work-flow of this function:

- Define a local variable `nValueOut = nValue` that holds the amount of coins to send out (line 17). Add transaction fee `nFee` to `nValue` so that now `nValue` includes the transaction fee (line 18).
- Call `SelectCoins(nValue, setCoins)` at line 21 to get a bunch of coins and put them into `setCoins`. Add up the values of these coins and then put the total amount into `nValueIn` (line 25). `setCoins` contains transactions that pay to your Bitcoin addresses (i.e., they are your coins). These transactions will become the source-transactions of `wtxNew`.
- Call `wtxNew.vout.push_back(CTxOut(nValueOut, scriptPubKey))` at line 27 to add an output entry to `wtxNew`. This entry pays to Bitcoin address `<recipient_address_hash160>` (contained in `scriptPubKey`) the amount of `nValueOut`.
- If there is any changes left (`nValueIn > nValue` at line 29), add another output entry to `wtxNew` to pay the leftover to yourself. A few steps are taken to achieve this:
    - Get the first transaction `txFirst` in `setCoin`, go through each slot in vector `txFirst.vout` and check if it is yours (line 37). If yes, extract the public key from this slot and put the extracted key to a local variable `vchPubKey`.
    - Put `vchPubKey` into the following script code: `vchPubKey OP_CHECKSIG`, and use this script code to add a payment output to `wtxNew` to pay to yourself (line 45).
    - Because `setCoin` contains transactions paid to you, so each transaction must have at least one payee address that belongs to you. So searching the first transaction `txFirst` should find such an address.
- At this point, the output vector `vout` of `wtxNew` has been setup. It is time to setup the input vector `vin`. Remember that each slot of `vin` refers to a

source-transaction, and `wtxNew`'s source-transactions are all in `setCoins`. For each source-transaction `pcoin` in `setCoins`, go through its output entries `pcoin→vout[nOut]` one by one. If the `nOut`-th entry is yours (meaning that `wtxNew` gets coins from this entry), add an input entry to `wtxNew.vin` (`wtxNew.vin.push_back(CTxIn(pcoin→GetHash(), nOut))` at line 51). This input entry points to the `nOut`-th output entry of `pcoin`. This connects an input slot of `wtxNew.vin` to the `nOut`-th output slot of `pcoin`.

- For each transaction `pcoin` in `setCoins`, go through its output entries `pcoin→vout[nOut]` one by one. If it is yours, call `SignSignature(*pcoin, wtxNew, nIn++)` (line 57) to **sign the `nIn`-th input entry of `wtxNew`**. Note that `nIn` tracks the input entry index of `wtxNew`.
- If the transaction fee `nFee` is less than `wtxNew.GetMinFee(true)`, set `nFee` equal to the latter, discard all the data filled into `wtxNew` (lines 13-14) and start all over again. In the first iteration of the big loop starting at line 11, `nFee` is a local copy of global variable `nTransactionFee = 0`.
- You may wonder why all this hassle of re-filling `wtxNew`. The source code of `GetMinFee()` gives out the answer: the minimum transaction fee of a transaction depends on the size of the transaction. The size of `wtxNew` is only known after it is fully constructed. If the calculated minimum transaction fee returned by `wtxNew.GetMinFee(true)` is greater than the assumed transaction fee `nFee` when `wtxNew` was constructed, there is no other way to go but to discard the current `wtxNew` and re-construct it.
- So there is a catch-22 situation: to construct a new transaction, you must know the transaction fee; but the transaction fee can only be determined after you have a fully constructed transaction. To break this cycle, a local variable `nFee` is used to hold an "estimated" transaction fee; and a new transaction is constructed based on the estimate. After the construction, the real transaction fee is calculated and compared with the estimate. If the estimate is less than the real, changed the estimate to the calculated transaction fee and start all over.

Here is the source code of `GetMinFee()`.

```
    1: class CTransaction
    2: {
    3:     //......
    4:     int64 GetMinFee(bool fDiscount=false) const
    5:     {
    6:         // Base fee is 1 cent per kilobyte
    7:         unsigned int nBytes = ::GetSerializeSize(*this,
SER_NETWORK);
```

```
    8:          int64 nMinFee = (1 + (int64)nBytes / 1000) * CENT;
    9:          // First 100 transactions in a block are free
   10:          if (fDiscount && nBytes < 10000)
   11:              nMinFee = 0;
   12:          // To limit dust spam, require a 0.01 fee if any output is
less than 0.01
   13:          if (nMinFee < CENT)
   14:              foreach(const CTxOut& txout, vout)
   15:                  if (txout.nValue < CENT)
   16:                      nMinFee = CENT;
   17:          return nMinFee;
   18:      }
   19:      //......
   20: }
```

- If the calculated transaction fee is greater than the estimate, break the big loop starting at line 11 and the whole function returns (line 67). Before doing that, the following two actions are taken:
  - Call `wtxNew.AddSupportingTransactions(txdb)` (line 65). This function will be covered in *Chapter Transaction II*.
  - Set `wtxNew.fTimeReceivedIsTxTime = true` (line 66).

Let's now examine how to sign the newly minted transaction `wtxNew` by calling `SignSignature()`.

2.6.1. `SignSignature()` in `script.cpp`

Here is the source code of `SignSignature()`.

```
    1: bool SignSignature(const CTransaction& txFrom, CTransaction& txTo,
unsigned int nIn, int nHashType, CScript scriptPrereq)
    2: {
    3:     assert(nIn < txTo.vin.size());
    4:     CTxIn& txin = txTo.vin[nIn];
    5:     assert(txin.prevout.n < txFrom.vout.size());
    6:     const CTxOut& txout = txFrom.vout[txin.prevout.n];
    7:     // Leave out the signature from the hash, since a signature
can't sign itself.
    8:     // The checksig op will also drop the signatures from its
hash.
    9:     uint256 hash = SignatureHash(scriptPrereq +
txout.scriptPubKey, txTo, nIn, nHashType);
   10:     if (!Solver(txout.scriptPubKey, hash, nHashType,
txin.scriptSig))
   11:         return false;
   12:     txin.scriptSig = scriptPrereq + txin.scriptSig;
   13:     // Test solution
   14:     if (scriptPrereq.empty())
   15:         if (!EvalScript(txin.scriptSig + CScript(OP_CODESEPARATOR)
+ txout.scriptPubKey, txTo, nIn))
   16:             return false;
   17:     return true;
```

```
   18: }
```

First notice that this function have 5 parameters while in `CreateTransaction()` it is called with 3. This is because in `script.h`, the default values of the last 2 parameters are provided.

```
    1: bool SignSignature(const CTransaction& txFrom, CTransaction& txTo,
unsigned int nIn, int nHashType=SIGHASH_ALL, CScript
scriptPrereq=CScript());
```

Here are the 3 arguments passed in by the caller `CreateTransaction()`.

- `txFrom` is `*pcoin`. It is one of the coins held by `setCoins` in `CreatTransaction()`. It is a source-transaction. One of its output slots contains the coins the new transaction is about to spend.
- `txTo` is `wtxNew` in `CreatTransaction()`. It is the new transaction that is about to spend coins in source-transaction `txFrom`. This new transaction needs to be signed.
- `nIn` is an index pointing to an input slot of `txTo`. This input slot contains a reference to an output slot of `txFrom`. To be more precise, `txin = txTo.vin[nIn]` (line 4) is the input slot of `txTo`; and `txout = txFrom.vout[txin.prevout.n]` (line 6) is the output slot of `txFrom` that `txin` refers to.

Here is what `SignSignature()` does:

- Calls `SignatureHash()` to generate a hash of `txTo`.
- Calls function `Solver()` to sign the hash generated.
- Calls `EvalScript()` to evaluate a piece of script to make sure the signature is correctly.

Let's examine these 3 functions.

2.6.2. `SignatureHash()` in `script.cpp`

Here is the source code of `SignatureHash()`.

```
    1: uint256 SignatureHash(CScript scriptCode, const CTransaction&
txTo, unsigned int nIn, int nHashType)
    2: {
    3:     if (nIn >= txTo.vin.size())
    4:     {
    5:         printf("ERROR: SignatureHash() : nIn=%d out of range\n",
nIn);
    6:         return 1;
    7:     }
    8:     CTransaction txTmp(txTo);
    9:     // In case concatenating two scripts ends up with two
codeseparators,
```

```
   10:     // or an extra one at the end, this prevents all those
possible incompatibilities.
   11:     scriptCode.FindAndDelete(CScript(OP_CODESEPARATOR));
   12:     // Blank out other inputs' signatures
   13:     for (int i = 0; i < txTmp.vin.size(); i++)
   14:         txTmp.vin[i].scriptSig = CScript();
   15:     txTmp.vin[nIn].scriptSig = scriptCode;
   16:     // Blank out some of the outputs
   17:     if ((nHashType & 0x1f) == SIGHASH_NONE)
   18:     {
   19:         // Wildcard payee
   20:         txTmp.vout.clear();
   21:         // Let the others update at will
   22:         for (int i = 0; i < txTmp.vin.size(); i++)
   23:             if (i != nIn)
   24:                 txTmp.vin[i].nSequence = 0;
   25:     }
   26:     else if ((nHashType & 0x1f) == SIGHASH_SINGLE)
   27:     {
   28:         // Only lockin the txout payee at same index as txin
   29:         unsigned int nOut = nIn;
   30:         if (nOut >= txTmp.vout.size())
   31:         {
   32:             printf("ERROR: SignatureHash() : nOut=%d out of
range\n", nOut);
   33:             return 1;
   34:         }
   35:         txTmp.vout.resize(nOut+1);
   36:         for (int i = 0; i < nOut; i++)
   37:             txTmp.vout[i].SetNull();
   38:         // Let the others update at will
   39:         for (int i = 0; i < txTmp.vin.size(); i++)
   40:             if (i != nIn)
   41:                 txTmp.vin[i].nSequence = 0;
   42:     }
   43:     // Blank out other inputs completely, not recommended for open
transactions
   44:     if (nHashType & SIGHASH_ANYONECANPAY)
   45:     {
   46:         txTmp.vin[0] = txTmp.vin[nIn];
   47:         txTmp.vin.resize(1);
   48:     }
   49:     // Serialize and hash
   50:     CDataStream ss(SER_GETHASH);
   51:     ss.reserve(10000);
   52:     ss << txTmp << nHashType;
   53:     return Hash(ss.begin(), ss.end());
   54: }
```

Here are the values the input parameters of this function take:

- `txTo` is the transaction to be signed. It is `wtxNew` in `CreateTransaction()`. Its `nIn`-th input slot, i.e., `txTo.vin[nIn]`, is the target this function is about to work on.
- `scriptCode` is `scriptPrereq + txout.scriptPubKey`, where `txout`

is the output slot of source-transaction `txFrom` defined in `SignSignature()`. Since `scriptPrereq` is empty (see the default value of the 5-th parameter of `SignSignature()`), **`scriptCode` essentially is the script in the output slot of source-transaction `txFrom` that is referred by the `nIn`-th input slot of `txTo`**. There are two types of scripts that `txout.scriptPubKey` may contain:

- Script A: `OP_DUP OP_HASH160 <your_address_hash160> OP_EQUALVERIFY OP_CHECKSIG`.
  This script sends coins to you from source-transaction `txFrom`, where `<your_address_hash160>` is your Bitcoin address.
- Script B: `<your_public_key> OP_CHECKSIG`.
  This script sends the leftover to the creator of source-transaction `txFrom`. Since the new transaction `txTo/wtxNew` you created is about to spend coins from `txFrom`, you must be the creator of `txFrom` as well. That is to say, you're spending coins sent to you by yourself when you created `toForm`. Therefore, since `<your_public_key>` is a public key of the creator of `txFrom`, it is also a public key of you.

Let's pause for a moment and think about *Scripts A and B*. You may be wondering where they come from. Well, they come from the code we just covered. Specifically, Script A comes from line 29 of `CSendDialog::OnButtonSend()`, and Script B from line 44 of `CreateTransaction()`.

- When someone created transaction `txFrom`, function `CSendDialog::OnButtonSend()` was called. It inserted Script A at line 29 to an output slot of `txFrom`. Since this output slot paid bitcoins to you, `<recipient_address_hash160>`, the recipient's Bitcoin address embedded in the script, was `<your_address_hash160>`.
- If `txFrom` was created by yourself, then Script B was inserted into one of the output slots of `txFrom` in `CreateTransaction()`. In this case, the public key `vchPubKey` collected at line 44 of `CreateTransaction()` was a public key belonged to you.

With a good understanding of the inputs, let's go ahead and examine what `SignatureHash()` does.

`SignatureHash()` first copies `txTo` to `txTmp`, then it blanks out the `scriptSig` field of each input slot of `txTmp.vin`, except for slot `txTmp.vin[nIn]`, whose `scriptSig` field is set to `scriptCode` (lines 14 and 15).

Next, this function checks the value of `nHashType`. The caller passes in value `nHashType = SIGHASH_ALL`, which is an enum value defined as

```
    1: enum
    2: {
    3:     SIGHASH_ALL = 1,
    4:     SIGHASH_NONE = 2,
    5:     SIGHASH_SINGLE = 3,
    6:     SIGHASH_ANYONECANPAY = 0x80,
    7: };
```

Since `nHashType = SIGHASH_ALL`, none of the if-else conditions holds and the execution directly goes to the last 4 lines of codes.

In the last 4 lines, `txTmp` and `nHashType` are serialized to a `CDataStream` object, which holds data in a vector of characters (see *Chapter Serrialization*). The returned hash code is generated by applying `Hash()` on the serialized data.

A transaction has many input slots. `SignatureHash()` targets a single input slot. It does the following to generate a hash:

- Blanks out all the input slots except the targeted one.
- Copies the script from the source-transaction's output slot that is referred by the targeted input slot into the targeted input slot.
- Generates a hash based on the modified transaction.

Important

Modified transaction: It is important to remember that the hash is generated based on a modified transaction: the `scriptSig` field of the `nIn`-th input slot of the new transaction is filled with its source-transaction's output script `scriptPubKey`, and all other input slots are empty. The same modification must be done latter to verify the transaction later.

`Hash()` in `util.h`

Here is the source code of `Hash()` that is called to generate the hash.

```
    1: template<typename T1>
    2: inline uint256 Hash(const T1 pbegin, const T1 pend)
    3: {
    4:     uint256 hash1;
    5:     SHA256((unsigned char*)&pbegin[0], (pend - pbegin) *
sizeof(pbegin[0]), (unsigned char*)&hash1);
    6:     uint256 hash2;
    7:     SHA256((unsigned char*)&hash1, sizeof(hash1), (unsigned
char*)&hash2);
    8:     return hash2;
    9: }
```

This function applies `SHA256()` twice on the input data and returns the result. `SHA256()` is declared in `openssl/sha.h` with signature `unsigned char *SHA256(const unsigned char *d, size_t n,unsigned char *md)`.

2.6.3. `Solver()` in `script.cpp`

`Solver()` is called in `SignSignature()` right after `SignatureHash()`. It is the function that actually generates the signature based on the hash returned by `SignatureHash()`.

```
    1: bool Solver(const CScript& scriptPubKey, uint256 hash, int
nHashType, CScript& scriptSigRet)
    2: {
    3:     scriptSigRet.clear();
    4:     vector<pair<opcodetype, valtype> > vSolution;
    5:     if (!Solver(scriptPubKey, vSolution))
    6:         return false;
    7:     // Compile solution
    8:     CRITICAL_BLOCK(cs_mapKeys)
    9:     {
   10:         foreach(PAIRTYPE(opcodetype, valtype)& item, vSolution)
   11:         {
   12:             if (item.first == OP_PUBKEY)
   13:             {
   14:                 // Sign
   15:                 const valtype& vchPubKey = item.second;
   16:                 if (!mapKeys.count(vchPubKey))
   17:                     return false;
   18:                 if (hash != 0)
   19:                 {
   20:                     vector<unsigned char> vchSig;
   21:                     if (!CKey::Sign(mapKeys[vchPubKey], hash,
vchSig))
   22:                         return false;
   23:                     vchSig.push_back((unsigned char)nHashType);
   24:                     scriptSigRet << vchSig;
   25:                 }
   26:             }
   27:             else if (item.first == OP_PUBKEYHASH)
   28:             {
   29:                 // Sign and give pubkey
   30:                 map<uint160, valtype>::iterator mi =
mapPubKeys.find(uint160(item.second));
   31:                 if (mi == mapPubKeys.end())
   32:                     return false;
   33:                 const vector<unsigned char>& vchPubKey =
(*mi).second;
   34:                 if (!mapKeys.count(vchPubKey))
   35:                     return false;
   36:                 if (hash != 0)
   37:                 {
   38:                     vector<unsigned char> vchSig;
   39:                     if (!CKey::Sign(mapKeys[vchPubKey], hash,
vchSig))
```

```
40:                             return false;
41:                         vchSig.push_back((unsigned char)nHashType);
42:                         scriptSigRet << vchSig << vchPubKey;
43:                     }
44:                 }
45:             }
46:         }
47:         return true;
48: }
```

Here are the values its 4 parameters take:

- The caller function `SignSignature()` at line 10 passes in `txOut.scriptPubKey`, the output script from source-transaction `txFrom`, as the input value of the first parameter `scriptPubKey`. Remember it contains *either Script A or B*.
- The second parameter `hash` is the hash generated by `SignatureHash()`.
- The third parameter `nHashType` is `SIGHASH_ALL`.
- The forth parameter is the return value of this function, which is `txin.scriptSig` at line 12 in caller `SignSignature()`. Remember `txin` is the `nIn`-th input slot of the newly minted transaction `wtxNew` (referred as `txTo` in caller `SignSignature()`), therefore, **the the `scriptSig` field of the `nIn`-th input slot of `wtxNew` will hold the return signature after calling this function**.

This function first calls another `Solver()` function with 2 arguments. We have to examine it first.

2.6.4. `Solver()` in `script.cpp` with 2 parameters

Here is the source code of `Solver()` with 2 parameters.

```
    1: bool Solver(const CScript& scriptPubKey, vector<pair<opcodetype,
valtype> >& vSolutionRet)
    2: {
    3:     // Templates
    4:     static vector<CScript> vTemplates;
    5:     if (vTemplates.empty())
    6:     {
    7:         // Standard tx, sender provides pubkey, receiver adds
signature
    8:         vTemplates.push_back(CScript() << OP_PUBKEY <<
OP_CHECKSIG);
    9:         // Short account number tx, sender provides hash of
pubkey, receiver provides signature and pubkey
   10:         vTemplates.push_back(CScript() << OP_DUP << OP_HASH160 <<
OP_PUBKEYHASH << OP_EQUALVERIFY << OP_CHECKSIG);
   11:     }
   12:     // Scan templates
   13:     const CScript& script1 = scriptPubKey;
```

```
14:     foreach(const CScript& script2, vTemplates)
15:     {
16:         vSolutionRet.clear();
17:         opcodetype opcode1, opcode2;
18:         vector<unsigned char> vch1, vch2;
19:         // Compare
20:         CScript::const_iterator pc1 = script1.begin();
21:         CScript::const_iterator pc2 = script2.begin();
22:         loop
23:         {
24:             bool f1 = script1.GetOp(pc1, opcode1, vch1);
25:             bool f2 = script2.GetOp(pc2, opcode2, vch2);
26:             if (!f1 && !f2)
27:             {
28:                 // Success
29:                 reverse(vSolutionRet.begin(), vSolutionRet.end());
30:                 return true;
31:             }
32:             else if (f1 != f2)
33:             {
34:                 break;
35:             }
36:             else if (opcode2 == OP_PUBKEY)
37:             {
38:                 if (vch1.size() <= sizeof(uint256))
39:                     break;
40:                 vSolutionRet.push_back(make_pair(opcode2, vch1));
41:             }
42:             else if (opcode2 == OP_PUBKEYHASH)
43:             {
44:                 if (vch1.size() != sizeof(uint160))
45:                     break;
46:                 vSolutionRet.push_back(make_pair(opcode2, vch1));
47:             }
48:             else if (opcode1 != opcode2)
49:             {
50:                 break;
51:             }
52:         }
53:     }
54:     vSolutionRet.clear();
55:     return false;
56: }
```

The first parameter `scriptPubKey` contains *either Script A or B*. Again, it is the output script from source-transaction `txFrom` in `SignSignature()`.

The second parameter will hold the output. It is a vector of pairs, each pair consists of a script operator (of type `opcodetype`) and a script operand (of type `valtype`).

This function first defines two templates (lines 8-10):

- Template A: `OP_DUP OP_HASH160 OP_PUBKEYHASH OP_EQUALVERIFY OP_CHECKSIG`.

- Template B: `OP_PUBKEY OP_CHECKSIG`.

It is obvious that Templates A and B correspond to Scripts A and B respectively. For easy reference, a copy of Scripts A and B is shown below:

- Script A: `OP_DUP OP_HASH160 <your_address_hash160> OP_EQUALVERIFY OP_CHECKSIG`.
- Script B: `<your_public_key> OP_CHECKSIG`.

What this function does is to compare the input `scriptPubKey` with both templates:

- If the input is Script A, it pairs up `OP_PUBKEYHASH` from Template A and `<your_address_hash160>` from Script A, and puts the pair to `vSolutionnRet`.
- If the input is Script B, it extracts operator `OP_PUBKEY` from Template B, operand `<your_public_key>` from Script B, pairs them up, and puts the pair to `vSolutionnRet`.
- If the input script matches neither, this function fails and returns `false`.

2.6.5. Back to `Solver()`

Let's go back to `Solver()` with 4 parameters and continue our analysis of that function. What it does is clear now. It executes one of the two branches, depending if the pair it gets from `vSolution` is obtained from Script A or Script B. If it was from Script A, `item.first == OP_PUBKEYHASH`; and if from Script B, `item.first == OP_PUBKEY`.

- `item.first == OP_PUBKEY` (Script B).
  In this case, `item.second` contains `<your_public_key>`. Global variable `mapKeys` maps all your public keys to their corresponding private keys. If `mapKeys` does not contain such a public key, it is an error (line 16). Otherwise, use the private key extracted from `mapKeys` to sign the hash of the newly minted transaction `wtxNew` that is passed in as the second argument (`CKey::Sign(mapKeys[vchPubKey], hash, vchSig)` at line 23). The result is put into `vchSig`, which is then serialized to `scriptSigRet` as the returned value (`scriptSigRet << vchSig` at line 24).
- `item.first == OP_PUBKEYHASH` (Script A).
  In this case, `item.second` contains `<your_address_hash160>`. This Bitcoin address is used to find the corresponding public key from a global map `mapPubKeys` (line 30), which maps your Bitcoin addresses to your public keys that were used to generate those address (check function

`AddKey()` in *Section Bitcoin address* in this chapter). Next, the public key found is used to find the corresponding private key from `mapKeys`. The private key is then used to sign the second argument `hash`. Both the signature and the public key are serialized to `scriptSigRet` and returned (`scriptSigRet << vchSig << vchPubKey` at line 42).

> Keys are generated in the following order: First generate a private key (a random number), then generate a public key from a private key. Hash the public key to get a hash code of 160 bits. This hash code is of type `uint160`. It can be represented by an ASCII string using base58 encoding. And this string is a Bitcoin address.
> A hash code of type `uint160` and its base58 encoding are equivalent. They can be transfered to each other easily.
> To find a public key from its 160 bit hash, BA maintains a global map `mapPubKeys` in `main.cpp`.
> To find a private key from the corresponding public key, BA maintains another global map `mapKeys` in `main.cpp`.

2.6.6. `EvalScript()` in `script.cpp`

Now let's go back to `SignSignature()`. After line 12 of this function, `txin.scriptSig`, the `scriptSig` field `wtxNew`'s `nIn`-th input slot, is filled with a signature that is

- either `vchSig vchPubKey` (Signature A for Script A)
- or `vchSig` (Signature B for Script B)

`vchSig` will be referred as `<your_signature_vchSig>`, and `vchPubKey` as `<your_pubKey_vchPubKey>` to emphasize that they are your signature and your public key.

Important

> Now it is clear that the `scriptSig` field of the `nIn`-th input slot of a transaction will be filled with a signature that is generated based on the hash of a *modified transaction*.

Let's examine `EvalScript()` now, which is the last function called by `SignSignature()` at line 15. The three parameters of `EvalScript()` take the following values:

- The first argument passed in is `txin.scriptSig + CScript(OP_CODESEPARATOR) + txout.scriptPubKey`. It is
  - either Verification Case A:
    `<your_signature_vchSig> <your_pubKey_vchPubKey>`

`OP_CODESEPARATOR OP_DUP OP_HASH160 <your_address_hash160> OP_EQUALVERIFY OP_CHECKSIG`, i.e., Signature A + `OP_CODESEPARATOR` + Script A;

- or Verification Case B: `<your_signature_vchSig> OP_CODESEPARATOR <your_pubKey_vchPubKey> OP_CHECKSIG`, i.e., Signature B + `OP_CODESEPARATOR` + Script B.

- The second parameter is the new transaction `txTo`, which is `wtxNew` in `CreatTransaction()`.
- The third parameter is `nIn`, the index of `txTo`'s input slot to be verified.

An elaborate description on how verification is done presented in *Chapter Script*. For now, it is sufficient to say that `EvalScript()` verifies that the newly created transaction `wtxNew` contains the right signature in its `nIn`-th input slot.

2.7. Summary

At this point, we have finished analyzing function `CreatTransaction()`. We know that what it does is to construct a new transaction and sign it, and we know **exactly** how that is done.

We did not complete analyzing `SendMoney()` yet, there are three more functions left un-covered. These three functions contain involve new data structures and database operations. We will cover them after providing an explanation of these data structures in *Chapter Block* and Bitcoin database in *Chapter Database*.

3. Serialization

This chapter covers Bitcoin serialization functions. It provides necessary information to fill in the details left out in *Chapter Transaction I*.

All Bitcoin serialization functions are implemented in a single header file `serialize.h`. Class `CDataStream` is the central data structure of this file.

3.1. CDataStream

Class `CDataStream` holds a `vector <char>` that stores the serialized data. It combines a vector and a stream interface to serve the data. It does so by maintaining the following 6 member variables:

```
 1: class CDataStream
 2: {
 3: protected:
 4:     typedef vector<char, secure_allocator<char> > vector_type;
 5:     vector_type vch;
 6:     unsigned int nReadPos;
 7:     short state;
 8:     short exceptmask;
 9: public:
10:     int nType;
11:     int nVersion;
12:     //......
13: };
```

- `vch` stores the serialized data. It is of type `vector_type`, which is just a `vector<char>` with a customized allocator. An allocator is called by the implementation of `vector` to allocate/de-allocate memory when necessary. This customized allocator clears the memory content before releasing the memory to the OS to prevent other processes running in the same machine to read the data. This ensures the security of the data it stores. The source code of this allocator is not shown in here. Refer to `serialize.h` for the implementation.
- `nReadPos` is the starting position in `vch` to read data from.
- `state` is an error flag. It is set to a value to indicate some errors occur during the serialization/de-serialization process.
- `exceptmask` is an error mask. It is initialized to `ios::badbit | ios::failbit`. Together with `state`, it is used to find out what kinds of errors happened.
- `nType` takes a value of `SER_NETWORK, SER_DISK, SER_GETHASH,`

SER_SKIPSIG, SER_BLOCKHEADERONLY, which informs CDataStream the kind of serialization to carry out. These 5 symbols are defined in an enum. Each symbol is an int type (4 bytes) and their values are all powers of 2.

```
 1: enum
 2: {
 3:     // primary actions
 4:     SER_NETWORK         = (1 << 0),
 5:     SER_DISK            = (1 << 1),
 6:     SER_GETHASH         = (1 << 2),
 7:     // modifiers
 8:     SER_SKIPSIG         = (1 << 16),
 9:     SER_BLOCKHEADERONLY = (1 << 17),
10: };
```

- nVersion is the version number.

3.1.1. CDataStream::read() and CDataStream::write()

Member functions CDataStream::read() and CDataStream::write() are low-level functions to serialize and de-serialize data a CDataStream object.

```
 1: class CDataStream
 2: {
 3: //......
 4:     CDataStream& read(char* pch, int nSize)
 5:     {
 6:         // Read from the beginning of the buffer
 7:         assert(nSize >= 0);
 8:         unsigned int nReadPosNext = nReadPos + nSize;
 9:         if (nReadPosNext >= vch.size())
10:         {
11:             if (nReadPosNext > vch.size())
12:             {
13:                 setstate(ios::failbit, "CDataStream::read() : end
of data");
14:                 memset(pch, 0, nSize);
15:                 nSize = vch.size() - nReadPos;
16:             }
17:             memcpy(pch, &vch[nReadPos], nSize);
18:             nReadPos = 0;
19:             vch.clear();
20:             return (*this);
21:         }
22:         memcpy(pch, &vch[nReadPos], nSize);
23:         nReadPos = nReadPosNext;
24:         return (*this);
25:     }
26:     CDataStream& write(const char* pch, int nSize)
27:     {
28:         // Write to the end of the buffer
29:         assert(nSize >= 0);
30:         vch.insert(vch.end(), pch, pch + nSize);
31:         return (*this);
```

```
32:     }
33: //......
34: }
```

`CDataStream::read()` copies `nSize` chars from `CDataStream` to a pre-allocated piece of memory pointed by `char* pch`. Here is how it works:

- Calculates the ending position of the data to be read from `vch`, `unsigned int nReadPosNext = nReadPos + nSize`.
- If the ending position is greater than the size of `vch`, there is no enough data to read. In the case, sets `state` to `ios::failbit` by calling function `setstate()`, and copies all zeros to `pch`.
- Otherwise, calls `memcpy(pch, &vch[nReadPos], nSize)` to copy `nSize` chars, starting at position `nReadPos` in `vch`, to a pre-allocated piece of memory pointed by `pch`. Then moves forward `nReadPos` to the next starting position `nReadPosNext` (line 22).

This implementation reveals that 1) once a piece of data has been read from the stream, it is gone, you can not read it again; and 2) `nReadPos` is the first valid reading position to read the data from.

`CDataStream::write()` is very simple. It appends `nSize` chars pointed by `pch` to the end the of `vch`.

3.1.2. Macros `READDATA()` and `WRITEDATA()`

Functions `CDataStream::read()` and `CDataStream::write()` are used to serialize and de-serialize **primitive** types (`int`, `bool`, `unsigned long`, etc). To serialize data of these types, pointers of these types are casted to `char*`. Since the size of these types are known, they can be read/written from/to the char buffer in `CDataStream`. Two macros are defined as helpers to call these two functions.

```
1: #define WRITEDATA(s, obj)   s.write((char*)&(obj), sizeof(obj))
2: #define READDATA(s, obj)    s.read((char*)&(obj), sizeof(obj))
```

Here is an example of using these macros. The following function serializes a `unsinged long` type,

```
1: template<typename Stream> inline void Serialize(Stream& s,
unsigned long a,  int, int=0) { WRITEDATA(s, a); }
```

Replacing `WRITEDATA(s, a)` by its definition, here is the expanded function

```
1: template<typename Stream> inline void Serialize(Stream& s,
unsigned long a,  int, int=0) {s.write((char*)&(a), sizeof(a)); }
```

This function takes a `unsigned long` argument `a`, gets its address, casts the pointer to `char*` and calls function `s.write()`.

3.1.3. Operators << and >> of CDataStream

CDataSteam overloads operators << and >> for serialization and de-serialization.

```
    1: class CDataStream
    2: {
    3: //......
    4:     template<typename T>
    5:         CDataStream& operator<<(const T& obj)
    6:         {
    7:             // Serialize to this stream
    8:             ::Serialize(*this, obj, nType, nVersion);
    9:             return (*this);
   10:         }
   11:         template<typename T>
   12:         CDataStream& operator>>(T& obj)
   13:         {
   14:             // Unserialize from this stream
   15:             ::Unserialize(*this, obj, nType, nVersion);
   16:             return (*this);
   17:         }
   18: //......
   19: }
```

The implementations of these two operators are pretty simple, they call global functions ::Serialize(*this, obj, nType, nVersion) and ::Unserialize(*this, obj, nType, nVersion) respectively.

3.1.4. Global serialization functions

Global functions ::Serialize(*this, obj, nType, nVersion) and ::Unserialize(*this, obj, nType, nVersion) are overloaded with many different versions, one for each type of obj. For instance, if obj is of type unsigned long, the serialization function is implemented as the follows (it is used as an example to illustrate how to use macro WRITEDATA(s, a)):

```
    1: template<typename Stream> inline void Serialize(Stream& s,
unsigned long a,  int, int=0){ WRITEDATA(s, a); }
```

For type vector, the overloaded function is

```
    1: template<typename Stream, typename T, typename A> inline void
Serialize(Stream& os, const std::vector<T, A>& v, int nType, int
nVersion=VERSION);
```

Header file serialize.h contains 14 overloaded versions of these two global functions for 14 primitive types (signed and unsigned versions of char, short, int, long and long long, plus char, float, double and bool) and 6 overloaded versions for 6 composite types (string, vector, pair, map, set and CScript). So for these types, you can simple use the following code to serialize/de-serialize the data:

```
    1: CDataStream ss(SER_GETHASH);
```

```
    2: ss<<obj1<<obj2; //serialize
    3: ss>>obj3>>obj4; //de-serialize
```

If none of the implemented types matches the type of the second argument `obj`, the following global function with a generic type `T` will be called.

```
    1: template<typename Stream, typename T>
    2: inline void Serialize(Stream& os, const T& a, long nType, int
nVersion=VERSION)
    3: {
    4:     a.Serialize(os, (int)nType, nVersion);
    5: }
```

In this generic version, type `T` is expected to implement a member function with signature `T::Serialize(Stream, int, int)`. It is called via `a.Serialize()`.

Note

Indeed, besides the two that serialize and de-serialize data, there is another global function that returns the size of the data. It is named `GetSerializeSize()`.

Note

Note the third argument in the above generic serialization function is `long nType` instead of `int nType`. According to Satoshi Nakamoto: *"int nType" is changed to "long nType" to keep from getting an ambiguous overload error. The compiler will only cast int to long if none of the other templates matched. Thanks to Boost serialization for this idea.*

3.2. How to implement serialization for a class

As mentioned in the preceding section, a generic class `T` needs to implement the following three member functions for serialization.

```
    1: unsigned int GetSerializeSize(int nType=0, int nVersion=VERSION)
const;
    2: template<typename Stream> void Serialize(Stream& s, int nType=0,
int nVersion=VERSION) const;
    3: template<typename Stream> void Unserialize(Stream& s, int nType=0,
int nVersion=VERSION);
```

These three functions will be called by their corresponding global counterparts with a generic parameter of type `T`. These global functions in turn will be called by the overloaded operators << and >> of class `CDataStream`.

A macro `IMPLEMENT_SERIALIZE(statements)` is defined to help an arbitrary class implement these three functions.

```
    1: #define IMPLEMENT_SERIALIZE(statements)    \
    2:     unsigned int GetSerializeSize(int nType=0, int
nVersion=VERSION) const  \
```

```
    3:     {                                                    \
    4:         CSerActionGetSerializeSize ser_action;  \
    5:         const bool fGetSize = true;             \
    6:         const bool fWrite = false;              \
    7:         const bool fRead = false;               \
    8:         unsigned int nSerSize = 0;              \
    9:         ser_streamplaceholder s;                \
   10:         s.nType = nType;                        \
   11:         s.nVersion = nVersion;                  \
   12:         {statements}                            \
   13:         return nSerSize;                        \
   14:     }                                           \
   15:     template<typename Stream>                   \
   16:     void Serialize(Stream& s, int nType=0, int nVersion=VERSION)
const  \
   17:     {                                           \
   18:         CSerActionSerialize ser_action;         \
   19:         const bool fGetSize = false;            \
   20:         const bool fWrite = true;               \
   21:         const bool fRead = false;               \
   22:         unsigned int nSerSize = 0;              \
   23:         {statements}                            \
   24:     }                                           \
   25:     template<typename Stream>                   \
   26:     void Unserialize(Stream& s, int nType=0, int nVersion=VERSION)
\
   27:     {                                           \
   28:         CSerActionUnserialize ser_action;       \
   29:         const bool fGetSize = false;            \
   30:         const bool fWrite = false;              \
   31:         const bool fRead = true;                \
   32:         unsigned int nSerSize = 0;              \
   33:         {statements}                            \
   34:     }
```

Here is an example that illustrates how to use this macro.

```
    1: #include <iostream>
    2: #include "serialize.h"
    3: using namespace std;
    4: class AClass {
    5: public:
    6:         AClass(int xin): x(xin){};
    7:         int x;
    8:         IMPLEMENT_SERIALIZE(READWRITE(this->x);)
    9: };
   10: int main() {
   11:         CDataStream astream2;
   12:         AClass aObj(200); //an AClass object with x=200
   13:         cout<<"aObj=" << aObj.x<<endl;
   14:         astream2<<aObj;
   15:         AClass a2(1); //another object with x=1
   16:         astream2>>a2;
   17:         cout<<"a2="<<a2.x<<endl;
   18:         return 0;
   19: }
```

This program serializes and de-serializes an `AClass` object. It prints out the following on the screen.

```
    1: aObj=200
    2: a2=200
```

All three serialize/de-serialization member functions of `AClass` is implemented in a single line of code: `IMPLEMENT_SERIALIZE(READWRITE(this→x);)`.

Macro `READWRITE()` is defined as

```
    1: #define READWRITE(obj)      (nSerSize += ::SerReadWrite(s, (obj),
nType, nVersion, ser_action))
```

The expansion of this macro is placed in all three functions in macro `IMPLEMENT_SERIALIZE(statements)`. Therefore, it needs to accomplish three things at once: 1) return the size of serialized data, 2) serialize (write) data to the stream; and 3) de-serialize (read) data from the stream. Refer to the three functions defined in macro `IMPELMENT_SERIALIZE(statements)`.

To understand how macro `READWRITE(obj)` works, you first need to understand where symbols `nSerSize`, `s`, `nType`, `nVersion` and `ser_action` in its expansion come from. They all come from the the body of the three functions in macro `IMPELMENT_SERIALIZE(statements)`:

- `nSerSize` is an `unsigned int` initialized to `0` in all three functions;
- `ser_action` is an object declared in all three functions, but it has three different types. It is of type `CSerActionGetSerializeSize`, `CSerActionSerialize` and `CSerActionUnserialize` respectively in three functions;
- `s` is defined as of type `ser_streamplaceholder` in the first function. It is the first passed-in argument in the other two functions, with parametric type `Stream`.
- `nType` and `nVersion` are passed-in arguments in all three functions.

So once macro `READWRITE()` expands into macro `IMPELMENT_SERIALIZE()`, all it symbols will be resolved, as they already exist in the body of macro `IMPELMENT_SERIALIZE()`. +READ

The expansion of macro `READWRITE(obj)` calls a global function `::SerReadWrite(s, (obj), nType, nVersion, ser_action)`. Here are the three versions of this function.

```
    1: template<typename Stream, typename T>
    2: inline unsigned int SerReadWrite(Stream& s, const T& obj, int
nType, int nVersion, CSerActionGetSerializeSize ser_action)
    3: {
    4:     return ::GetSerializeSize(obj, nType, nVersion);
    5: }
```

```
    6: template<typename Stream, typename T>
    7: inline unsigned int SerReadWrite(Stream& s, const T& obj, int
nType, int nVersion, CSerActionSerialize ser_action)
    8: {
    9:     ::Serialize(s, obj, nType, nVersion);
   10:     return 0;
   11: }
   12: template<typename Stream, typename T>
   13: inline unsigned int SerReadWrite(Stream& s, T& obj, int nType, int
nVersion, CSerActionUnserialize ser_action)
   14: {
   15:     ::Unserialize(s, obj, nType, nVersion);
   16:     return 0;
   17: }
```

As you can see, function `::SerReadWrite()` is overloaded with 3 versions. Depending on the type of the last argument, it will call global functions `::GetSerializeSize()`, `::Serialize()` and `::Unserialize()` respectively; and these are the three global functions covered in *Section Global serialization functions*.

If you check the type of the last argument in the three versions of `::SerReadWrite()`, you can see that they are all empty classes. The sole purpose of these types is to be used to differentiate three versions of `::SerReadWrite()`, so it can be used in all these functions defined in macro `IMPELMENT_SERIALIZE()`.

Summary

Let's recap what we just covered. To serialize an arbitrary class `AClass`, you use macro `IMPLEMENT_SERIALIZE(READWRITE(this→x);)` to serialize its member variables `x`. If the type of `x` is any one of the 20 types (14 primitive and 6 composite) for which the corresponding global serialization functions have been implemented, these functions will be called. Otherwise, if `x` is an instance of some class, then a generic version of the overloaded global serialization function is called, which in turn calls the serialization function `x.Serilaize()` for object `x`. This member function presumably is implemented using macro `IMPLEMENT_SERIALIZE(READWRTIE(…);)` as well, and the calling chain recursively goes down until hits one of the 20 implemented types.

Comment

This is a nice scheme that cleverly takes advantage of 1) macro substitution and 2) type matching mechanism of template functions, to build a recursive calling chain for serialization functions. All serialization functions are resolved at compiling time, making the code very efficient.

Note

`const int*, const int* const, int const*`. Split the statement at `*` sign, if the `const` keyword appears on the left part (like in `const int *`), the pointed data cannot be

> changed; if it's on the right part (`int * const bar`), the pointer itself is a constant.

4. Script

Bticoin has a stack-based scripting language. A script is embedded in each output slot of a transaction. To spend the coins a transaction carries, a recipient must provide his/her public key so that the script can be executed successfully.

This chapter covers this scripting language. After finishing this chapter, you will understand how Signatures A and B shown at the end of *Chapter Transaction I* are verify.

All the classes and functions covered in this chapter are located in `script.h` or `script.cpp`.

The Bitcoin scripting language defines a list of operators like `OP_FALSE`, `OP_RIPEMD160`, `OP_SHA256`, etc. Operators work on data held on a stack. Output data generated after applying an operator to data held on the stack are again pushed back to the stack. There are two types of objects in the scripting language: operators and operands. Operators are listed in `enum opcodetype`, one entry for an operator. Operands are input data of operators. They are of type `valtype`, which is a vector of `unsigned char` (`typedef vector<unsigned char> valtype`). An operator and its operand, if there is any, together are called an instruction.

Note

There are three distinct char types `char`, `signed char`, `unsigned char` in C++. If you use char types for text, use the unqualified `char`. Chars can also be used as number values, but it is unspecified whether that value is treated as signed or unsigned. `signed char` gives values from -128 to 127, and `unsigned char`, from 0 to 255.

4.1. `enum opcodetype`

```
 1: enum opcodetype
 2: {
 3:     // push value
 4:     OP_0=0,
 5:     OP_FALSE=OP_0,
 6:     OP_PUSHDATA1=76,
 7:     OP_PUSHDATA2,
 8:     OP_PUSHDATA4,
 9:     OP_1NEGATE,
10:     OP_RESERVED,
11:     OP_1, // this is decimal 81
12:     OP_TRUE=OP_1,
13:     //......
14:     OP_IF,
```

```
    OP_NOTIF,
    OP_VERIF,
    OP_VERNOTIF,
    OP_ELSE,
    OP_ENDIF,
    //......
    OP_CHECKMULTISIG,
    OP_CHECKMULTISIGVERIFY, //this is decimal 175
    // multi-byte opcodes
    OP_SINGLEBYTE_END = 0xF0,
    OP_DOUBLEBYTE_BEGIN = 0xF000,
    // template matching params
    OP_PUBKEY,
    OP_PUBKEYHASH,
    OP_INVALIDOPCODE = 0xFFFF,
};
```

There are 106 distinct operators, plus `OP_FALSE` and `OP_TRUE`, which are alias of `OP_0` and `OP_1`, respectively. Operator codes are not consecutive. Starting from `OP_PUSHDATA1 = 76 = 0x4C`, code values increase consecutively to `OP_CHECKMULTISIGVERIFY = 175 = 0xAF`. Then the next code value is `OP_SINGLEBYTE_END = 0xF0`. And the one after that is `OP_DOUBLEBYTE_BEGIN = 0xF000`. All code values after that occupy two bytes.

4.2. CScript

Class `CScript` holds a script to be parsed and executed. A script is nothing but a stream of chars, so `CScript` is just a `vector<unsigned char>`. Scripts are inserted into `CScript` using the overloaded operator `<<`. It takes many different input types (don't confuse it with the operators of the Bitcoin scripting language).

4.2.1. Operator << of CScript

```
class CScript : public vector<unsigned char>
{
protected:
    CScript& push_int64(int64 n)
    {
        if (n == -1 || (n >= 1 && n <= 16))
        {
            push_back(n + (OP_1 - 1));
        }
        else
        {
            CBigNum bn(n);
            *this << bn.getvch();
        }
        return (*this);
    }
    //......
```

```
   18: public:
   19:     //......
   20:     explicit CScript(char b)           { operator<<(b); }
   21:     explicit CScript(short b)          { operator<<(b); }
   22:     explicit CScript(int b)            { operator<<(b); }
   23:     explicit CScript(long b)           { operator<<(b); }
   24:     explicit CScript(int64 b)          { operator<<(b); }
   25:     //......
   26:     explicit CScript(opcodetype b)     { operator<<(b); }
   27:     explicit CScript(const uint256& b) { operator<<(b); }
   28:     explicit CScript(const CBigNum& b) { operator<<(b); }
   29:     explicit CScript(const vector<unsigned char>& b) {
operator<<(b); }
   30:     CScript& operator<<(char b)           { return
(push_int64(b)); }
   31:     CScript& operator<<(short b)          { return
(push_int64(b)); }
   32:     CScript& operator<<(int b)            { return
(push_int64(b)); }
   33:     CScript& operator<<(long b)           { return
(push_int64(b)); }
   34:     CScript& operator<<(int64 b)          { return
(push_int64(b)); }
   35:     CScript& operator<<(opcodetype opcode)
   36:     {
   37:         if (opcode <= OP_SINGLEBYTE_END)
   38:         {
   39:             insert(end(), (unsigned char)opcode);
   40:         }
   41:         else
   42:         {
   43:             assert(opcode >= OP_DOUBLEBYTE_BEGIN);
   44:             insert(end(), (unsigned char)(opcode >> 8));
   45:             insert(end(), (unsigned char)(opcode & 0xFF));
   46:         }
   47:         return (*this);
   48:     }
   49:     CScript& operator<<(const uint160& b)
   50:     {
   51:         insert(end(), sizeof(b));
   52:         insert(end(), (unsigned char*)&b, (unsigned char*)&b +
sizeof(b));
   53:         return (*this);
   54:     }
   55:     CScript& operator<<(const uint256& b)
   56:     {
   57:         insert(end(), sizeof(b));
   58:         insert(end(), (unsigned char*)&b, (unsigned char*)&b +
sizeof(b));
   59:         return (*this);
   60:     }
   61:     CScript& operator<<(const CBigNum& b)
   62:     {
   63:         *this << b.getvch();
   64:         return (*this);
   65:     }
```

```
   66:     CScript& operator<<(const vector<unsigned char>& b)
   67:     {
   68:         if (b.size() < OP_PUSHDATA1)
   69:         {
   70:             insert(end(), (unsigned char)b.size());
   71:         }
   72:         else if (b.size() <= 0xff)
   73:         {
   74:             insert(end(), OP_PUSHDATA1);
   75:             insert(end(), (unsigned char)b.size());
   76:         }
   77:         else
   78:         {
   79:             insert(end(), OP_PUSHDATA2);
   80:             unsigned short nSize = b.size();
   81:             insert(end(), (unsigned char*)&nSize, (unsigned
char*)&nSize + sizeof(nSize));
   82:         }
   83:         insert(end(), b.begin(), b.end());
   84:         return (*this);
   85:     }
   86:     //......
   87: }
```

Overloaded operator `<<` of `CScript` takes `char`, `short`, `int`, `long`, implicitly converses them to `int64` (an alias of `long long`), and calls `push_int64(int64 n)` to put them into `CScript`'s internal storage. (Since `CSCript` extends `vector<unsigned char>`, its internal storage is that of the latter. You can think of it as a dynamic array of unsigned characters.) Examining the body of `push_int64(int64 n)` reveals that if `n` is `-1` or an number between `1` and `16`, `n` is pushed in as `n + (OP_1 - 1)`, which is `n + (81 -1) = n + 80` (`OP_1 = 81 = 0x51`). So if `n` is -1+, `n` is pushed in as a single byte value `79`, which is the code value of operator `OP_1NEGATE`. If `n` is a number from `1` to `16`, it is pushed in as a single byte value between `81` and `96`, which is the code value of an operator between `OP_1` and `OP_16`. For other values of `n`, it is treated as a `CBigNum`. To summarize:

- If `n` is `-1`, `n` is treated as operator `OP_1NEGATE (79)`.
- If `n` is a number between `1` to `16`, `n` is treated as an operator between `OP_1 (81)` and `OP_16 (96)`.
- Otherwise, `n` is treated as a `CBigNum`, and the return value of `bn.getvch()`, which is a vector of chars, is pushed into the internal storage (line 12). For more details about `CBigNum`, see *Section CBigNum* of this chapter.

Overloaded operator `<<` also accepts the `unsigned` versions of `char`, `short`, `int`, `long`, and `uint64` as inputs. Their corresponding operator overload functions follow the same logic as their signed counterparts do (not shown in the source code above).

Let's examine the operator overload functions for other input types.

- For input type `uint160`, the size of the input data is first pushed into the internal storage (line 51), then the input data itself (line 52). The same procedure is applied to input type `uint256` (line 55). These two types are used to held hash codes.
- For input type `opcodetype`, i.e., the operator type, the code value is directly pushed into the internal storage (line 39). If the code value has 2 bytes, they are pushed in one after another (lines 44-45). Note that the most significant byte is pushed in first (line 44). So code `0xF001` will be pushed in as `0xF001`, not `0x01F0`. This is important, see the following *Section CScript::GetOp()* for details.
- For input type `CBigNum`, a vector of chars that represents the big number is pushed in (line 63). This is consistent with the way `push_int64()` handles `CBigNum`.
- For input type `vector <unsigned char>`, the procedure is more involving:
  - If the size of input `b` is less than 76 (code value of operator `OP_PUSHDATA1`), the size of `b` is pushed in as a single byte value first (line 70), then data `b` itself (line 83).
  - If the size of `b` is between 76 and 255 (includsive), operator `OP_PUSHDATA1` is pushed in first (line 74), then the size of `b` as a single byte value (line 75), then data `b` itself (line 83).
  - Otherwise, operator `OP_PUSHDATA2` is pushed in first (line 79), then the size of `b` as a two-byte (`short`) value (lines 80-81), then data `b` itself (line 83).

Overloaded operator << pushes instructions into `CScript`, function `CScript::GetOp()` extracts them from `CScript`.

Note

Keyword `explicit`. In C++, the compiler makes implicit conversion to resolve arguments to find the right function to call. For example, if you call `fun1( 2 )`, where argument `2` is an `int`, but only `fun1(long)` is defined, then the compiler will convert `2` to `long` to call `fun1()`. Keyword `explicit` tells the compiler NOT to do that.

4.2.2. `CScript::GetOp()`

Here is the source code of `CScript::GetOp()`.

```
    1: class CScript : public vector<unsigned char>
```

```
    2: {
    3: //......
    4:     bool GetOp(const_iterator& pc, opcodetype& opcodeRet,
vector<unsigned char>& vchRet) const
    5:     {
    6:         opcodeRet = OP_INVALIDOPCODE;
    7:         vchRet.clear();
    8:         if (pc >= end())
    9:             return false;
   10:         // Read instruction
   11:         unsigned int opcode = *pc++;
   12:         if (opcode >= OP_SINGLEBYTE_END)
   13:         {
   14:             if (pc + 1 > end())
   15:                 return false;
   16:             opcode <<= 8;
   17:             opcode |= *pc++;
   18:         }
   19:         // Immediate operand
   20:         if (opcode <= OP_PUSHDATA4)
   21:         {
   22:             unsigned int nSize = opcode;
   23:             if (opcode == OP_PUSHDATA1)
   24:             {
   25:                 if (pc + 1 > end())
   26:                     return false;
   27:                 nSize = *pc++;
   28:             }
   29:             else if (opcode == OP_PUSHDATA2)
   30:             {
   31:                 if (pc + 2 > end())
   32:                     return false;
   33:                 nSize = 0;
   34:                 memcpy(&nSize, &pc[0], 2);
   35:                 pc += 2;
   36:             }
   37:             else if (opcode == OP_PUSHDATA4)
   38:             {
   39:                 if (pc + 4 > end())
   40:                     return false;
   41:                 memcpy(&nSize, &pc[0], 4);
   42:                 pc += 4;
   43:             }
   44:             if (pc + nSize > end())
   45:                 return false;
   46:             vchRet.assign(pc, pc + nSize);
   47:             pc += nSize;
   48:         }
   49:         opcodeRet = (opcodetype)opcode;
   50:         return true;
   51:     }
   52: //......
   53: }
```

This function reads an instruction (an operator and its operand if necessary) from `pc` and put them into `opcodeRet` and `vchRet` respectively. Iterator `pc` points to

a position in the internal storage of `CScript` where data should be read from. Here is what this function does:

- It differentiates 1-byte and 2-byte operators. If the first byte is greater than `0xF0` (line 14), the next byte is extracted; and these two bytes together form a two-byte operator (lines 16-17). This is the reason why the most significant byte of a two byte operator is pushed in first. The first byte of all two-byte operators is greater than `0xF0`, while all single byte operators have code values less than `0xF0`.
- At line 20, if the first byte is less than or equal to 78 (the code value `OP_PUSHDATA4`), then depending on the value, this function differentiates 4 cases:
    - If the code value is 76 (code value of `OP_PUSHDATA1`), it reads the next byte as the size of the data into `nSize` (line 27), and then reads the number of bytes indicated by `nSize` into `vchRet` (line 46). So if `nSize=100`, it reads the next 100 bytes into `vchRet`.
    - If the code value is 77 (code value of `OP_PUSHDATA2`), it reads the next two bytes as the size of the data (line 34), and then reads the data itself into `vchRet` (line 46).
    - If the code value is 78 (code value of `OP_PUSHDATA4`), it reads the next four bytes as the size of the data (line 41), and then reads the data itself into `vchRet` (line 46).
    - Otherwise, i.e., if the code value is between 0 and 75 (inclusive), this value is treated as the size of data (line 22). It reads the number of bytes indicated by this value into `vchRet` (line 46). You may be wondering what the returned operator is then. The returned operator code is the extracted value that lies in between 0 and 75, which is a `UNKNOWN_OPCODE` (execept for value 0, which `OP_0` or `OP_FALSE`). This will not cause any problem once you see how this `UNKNOWN_OPCODE` is handled when executing the instruction.

Summary

depending on its value, the first byte of an instruction stored in `CScript` is interpreted differently: 1) If it is `0`, it is operator `OP_0` or `OP_FALSE`. 2) If it is a value between `1` and `75` (inclusive), it is a `UNKNOWN_OPCODE`; and the number of bytes indicated by that value is returned in `vchRet`. 3) If it is `76`, `77`, or `78`, it is a `OP_PUSHDATA#` operator, and whatever number of bytes that should be read are returned in `vchRet`. 4) If it is a value between `79` and `175` (`0xAF`), it is treated as an operator with such a code value. 5) If it is value greater than `0xF0`, it is the first byte of a two-byte operator.

4.3. EvalScript()

This function evaluates the instructions in a given script one after another. The final result is either `true` or `false`. It extracts the instructions in the script one by one in a while-loop (line 13). Within the loop, depending on the operator of the current instruction, the execution flow switches to the corresponding branch to execute the instruction. Here is the source code of this function (some switch branches are omitted to save spaces).

```
    1: bool EvalScript(const CScript& script, const CTransaction& txTo,
unsigned int nIn, int nHashType,
    2:                 vector<vector<unsigned char> >* pvStackRet)
    3: {
    4:     CAutoBN_CTX pctx;
    5:     CScript::const_iterator pc = script.begin();
    6:     CScript::const_iterator pend = script.end();
    7:     CScript::const_iterator pbegincodehash = script.begin();
    8:     vector<bool> vfExec;
    9:     vector<valtype> stack;
   10:     vector<valtype> altstack;
   11:     if (pvStackRet)
   12:         pvStackRet->clear();
   13:     while (pc < pend)
   14:     {
   15:         bool fExec = !count(vfExec.begin(), vfExec.end(), false);
   16:         //
   17:         // Read instruction
   18:         //
   19:         opcodetype opcode;
   20:         valtype vchPushValue;
   21:         if (!script.GetOp(pc, opcode, vchPushValue))
   22:             return false;
   23:         if (fExec && opcode <= OP_PUSHDATA4)
   24:             stack.push_back(vchPushValue);
   25:         else if (fExec || (OP_IF <= opcode && opcode <= OP_ENDIF))
   26:         switch (opcode)
   27:         {
   28:             //
   29:             // Push value
   30:             //
   31:             case OP_1NEGATE:
   32:             case OP_1:
   33:             case OP_2:
   34:             case OP_3:
   35:             case OP_4:
   36:             case OP_5:
   37:             case OP_6:
   38:             case OP_7:
   39:             case OP_8:
   40:             case OP_9:
   41:             case OP_10:
   42:             case OP_11:
   43:             case OP_12:
```

```
   44:             case OP_13:
   45:             case OP_14:
   46:             case OP_15:
   47:             case OP_16:
   48:             {
   49:                 // (–value)
   50:                 CBigNum bn((int)opcode - (int)(OP_1 - 1));
   51:                 stack.push_back(bn.getvch());
   52:             }
   53:             break;
   54:             //......
   55:             case OP_IF:
   56:             case OP_NOTIF:
   57:             case OP_VERIF:
   58:             case OP_VERNOTIF:
   59:             {
   60:                 // <expression> if [statements] [else
[statements]] endif
   61:                 bool fValue = false;
   62:                 if (fExec)
   63:                 {
   64:                     if (stack.size() < 1)
   65:                         return false;
   66:                     valtype& vch = stacktop(-1);
   67:                     if (opcode == OP_VERIF || opcode ==
OP_VERNOTIF)
   68:                         fValue = (CBigNum(VERSION) >=
CBigNum(vch));
   69:                     else
   70:                         fValue = CastToBool(vch);
   71:                     if (opcode == OP_NOTIF || opcode ==
OP_VERNOTIF)
   72:                         fValue = !fValue;
   73:                     stack.pop_back();
   74:                 }
   75:                 vfExec.push_back(fValue);
   76:             }
   77:             break;
   78:             case OP_ELSE:
   79:             {
   80:                 if (vfExec.empty())
   81:                     return false;
   82:                 vfExec.back() = !vfExec.back();
   83:             }
   84:             break;
   85:             case OP_ENDIF:
   86:             {
   87:                 if (vfExec.empty())
   88:                     return false;
   89:                 vfExec.pop_back();
   90:             }
   91:             break;
   92:             case OP_VERIFY:
   93:             {
   94:                 // (true–) or
   95:                 // (false–false) and return
```

```
   96:                 if (stack.size() < 1)
   97:                     return false;
   98:                 bool fValue = CastToBool(stacktop(-1));
   99:                 if (fValue)
  100:                     stack.pop_back();
  101:                 else
  102:                     pc = pend;
  103:             }
  104:             break;
  105:             //......
  106:             case OP_EQUAL:
  107:             case OP_EQUALVERIFY:
  108:             //case OP_NOTEQUAL: // use OP_NUMNOTEQUAL
  109:             {
  110:                 // (x1 x2 - bool)
  111:                 if (stack.size() < 2)
  112:                     return false;
  113:                 valtype& vch1 = stacktop(-2);
  114:                 valtype& vch2 = stacktop(-1);
  115:                 bool fEqual = (vch1 == vch2);
  116:                 // OP_NOTEQUAL is disabled because it would be too
easy to say
  117:                 // something like n != 1 and have some wiseguy
pass in 1 with extra
  118:                 // zero bytes after it (numerically, 0x01 ==
0x0001 == 0x000001)
  119:                 //if (opcode == OP_NOTEQUAL)
  120:                 //    fEqual = !fEqual;
  121:                 stack.pop_back();
  122:                 stack.pop_back();
  123:                 stack.push_back(fEqual ? vchTrue : vchFalse);
  124:                 if (opcode == OP_EQUALVERIFY)
  125:                 {
  126:                     if (fEqual)
  127:                         stack.pop_back();
  128:                     else
  129:                         pc = pend;
  130:                 }
  131:             }
  132:             break;
  133:             //......
  134:             //
  135:             // Crypto
  136:             //
  137:             case OP_RIPEMD160:
  138:             case OP_SHA1:
  139:             case OP_SHA256:
  140:             case OP_HASH160:
  141:             case OP_HASH256:
  142:             {
  143:                 // (in—hash)
  144:                 if (stack.size() < 1)
  145:                     return false;
  146:                 valtype& vch = stacktop(-1);
  147:                 valtype vchHash(opcode == OP_RIPEMD160 || opcode
== OP_SHA1 || opcode == OP_HASH160 ? 20 : 32);
```

```
  148:                 if (opcode == OP_RIPEMD160)
  149:                     RIPEMD160(&vch[0], vch.size(), &vchHash[0]);
  150:                 else if (opcode == OP_SHA1)
  151:                     SHA1(&vch[0], vch.size(), &vchHash[0]);
  152:                 else if (opcode == OP_SHA256)
  153:                     SHA256(&vch[0], vch.size(), &vchHash[0]);
  154:                 else if (opcode == OP_HASH160)
  155:                 {
  156:                     uint160 hash160 = Hash160(vch);
  157:                     memcpy(&vchHash[0], &hash160,
sizeof(hash160));
  158:                 }
  159:                 else if (opcode == OP_HASH256)
  160:                 {
  161:                     uint256 hash = Hash(vch.begin(), vch.end());
  162:                     memcpy(&vchHash[0], &hash, sizeof(hash));
  163:                 }
  164:                 stack.pop_back();
  165:                 stack.push_back(vchHash);
  166:             }
  167:             break;
  168:             case OP_CODESEPARATOR:
  169:             {
  170:                 // Hash starts after the code separator
  171:                 pbegincodehash = pc;
  172:             }
  173:             break;
  174:             case OP_CHECKSIG:
  175:             case OP_CHECKSIGVERIFY:
  176:             {
  177:                 // (sig pubkey—bool)
  178:                 if (stack.size() < 2)
  179:                     return false;
  180:                 valtype& vchSig    = stacktop(-2);
  181:                 valtype& vchPubKey = stacktop(-1);
  182:                 ////// debug print
  183:                 //PrintHex(vchSig.begin(), vchSig.end(), "sig:
%s\n");
  184:                 //PrintHex(vchPubKey.begin(), vchPubKey.end(),
"pubkey: %s\n");
  185:                 // Subset of script starting at the most recent
codeseparator
  186:                 CScript scriptCode(pbegincodehash, pend);
  187:                 // Drop the signature, since there's no way for a
signature to sign itself
  188:                 scriptCode.FindAndDelete(CScript(vchSig));
  189:                 bool fSuccess = CheckSig(vchSig, vchPubKey,
scriptCode, txTo, nIn, nHashType);
  190:                 stack.pop_back();
  191:                 stack.pop_back();
  192:                 stack.push_back(fSuccess ? vchTrue : vchFalse);
  193:                 if (opcode == OP_CHECKSIGVERIFY)
  194:                 {
  195:                     if (fSuccess)
  196:                         stack.pop_back();
  197:                     else
```

```
198:                             pc = pend;
199:                         }
200:                     }
201:                     break;
202:                     //......
203:                     default:
204:                         return false;
205:                 }
206:             }
207:             if (pvStackRet)
208:                 *pvStackRet = stack;
209:             return (stack.empty() ? false : CastToBool(stack.back()));
210: }
```

Here are a few key observations about this function:

- Iterator `pc` always points to the next byte to read.
- `stack` is the main stack this function works on. All operands extracted from input `script` will be pushed into this stack. And all results generated by applying an operator to its operand held in this stack will be pushed back to this stack.
- If `script.GetOp(pc, opcode, vchPushValue)` returns `true` (line 21), `opcode` will contains the extracted operator and `vchPushValue` will contain the operand, if there is any.
- At line 23, if `opcode` is less than or equal to 78 (code value of `OP_PUSHDATA4`), the data held in `vchPushValue` is pushed into `stack` at line 24 (don't worry about boolean variable `fExec` for now, it will be addressed later). Therefore, if `opcode` is a value between 1 and 75, i.e., a `UNKNOWN_OPCODE`, the returned data in `vchPushValue` will be pushed into `stack`. The same thing will happen for `opcode` values 0, 76, 77, or 78. So, the Bitcoin scripting language is optimized for short operands: For short operands less than or equal to 75 bytes, you just need to push the size of the data, and then the data itself into a `CScript`. The interpretor will handle it correctly. For long operands, you have to push in an `OP_PUSHDATA#` first.
- Operator `OP_1NEGATE` and operators between `OP_1` and `OP_16` push their corresponding data values into `stack` (line 51). So, `OP_1NEGATE` (code value 79) pushes -1 into `stack`, `+OP_2` (code value 82) pushes 2 into `stack`, etc. This is consistent with the way `push_int64()` handles its input data.

The structure of `EvalScript()` is simple and clear: `script.GetOP()` extracts an instruction from `script` (line 21). Depending on the extracted operator `opcode`, each case is executed in a branch of the `switch(opcode)` statement (line 26). The only puzzling codes are those involve variable `fExec` (lines 23 and 25). Let's examine them more closely.

4.3.1. Nested `OP_IF`

Variable `fExec` is assigned a new value at the beginning of each iteration (line 15). If vector `vfExec` contains a single `false`, it will be `false`. Otherwise, if every entry of `vfExec` is `true`, it will be `true`.

Vector `vfExec` is a stack that records the position of the current instruction within a nested `OP_IF` statement. This position determines whether the current instruction should be executed or not. This may sound very abstract. Let me give you an example to illustrate what I mean. Consider the following script:

```
1: OP_TRUE OP_IF
2:     OP_FALSE OP_IF
3:         statement1
4:     OP_ELSE
5:         statement2
6:     OP_ENDIF
7: OP_ELSE
8:     statement3
9: OP_ENDIF
```

Let's assume `statement1-3` contain no operators between `OP_IF` and `OP_ENDIF`, i.e., there is no more nested `OP_IF` in these statements and the whole script has a nested `OP_IF` statement with 2 levels. Let's follow the execution of this script to understand how it is interpreted and executed.

- First operator `OP_TURE`:
  - At the beginning, `stack` is empty. `vfExec` is empty. `fExec` is `true`.
  - At line 21, operator `OP_FALSE` is extracted and held in `opcode`.
  - At line 23, since `OP_TURE > OP_PUSHDATA4`, the execution flow goes to line 25.
  - At line 25, since `fExec` is `true`, the flow continues to line 26.
  - At line 26, `OP_TURE` held in `opcode` matches line 32 (`OP_TURE` is an alias of `OP_1`).
  - At lines 50-51, value 1 (represented by a `vector <unsigned char>`) is pushed into `stack`.
  - Now the first operator `OP_TURE` has been processed. The execution flow goes to line 53, and then starts the next loop at line 13.
- Second operator `OP_IF`:
  - In this loop, `vfExec` is still empty, and `fExec` is still `true` (line 15). `opcode` is `OP_IF` (line 21).
  - At line 23, since `OP_IF > OP_PUSHDATA4`, the execution flow goes to line 25.

    - At line 25, since `fEexc` is `true` and `OP_IF` does lie between `OP_IF` and `OP_ENDIF`, the execution goes to line 26.
    - At line 26, `OP_IF` matches line 55.
    - At line 61, `fValue` is `false`. Since `fEexc` is true, line 66 is executed. `stacktop(-1)` returns the value `1` pushed into `stack` in the first iteration.
    - At line 70, value `1` is casted to bool and assigned to `fValue`. `CastToBool` casts any non-zero values to `true` and zero to `false` (not showed in the source code).
    - At line 75, `fvalue`,i.e., value `true`, is pushed into `vfExec`.
    - Now the second operator `OP_IF` has been processed. The execution flow goes to line 77, and then starts the next loop at line 13.
- Third operator `OP_FASLE`:
    - In this loop, `vfExec` contains a single `true`, so `fEexc` is still `true` (line 15). `opcode` is `OP_FALSE`.
    - At line 23, since `fExec` is `true` and `OP_FALSE` is less than `OP_PUSHDATA4`, `OP_FALSE` is pushed into `stack` (line 24).
    - Now the third operator `OP_FALSE` has been processed. The execution flow starts the next loop at line 13.
- Forth operator `OP_IF`:
    - In this loop, `vfExec` still contains a single `true` and `fExec` is `true` (line 15). `opcode` is `OP_IF`.
    - At line 23, the execution flow follows the same path as the second operator (which is also `OP_IF`) did. But since this time `stocktop(-1)` will return value `0` at line 66 (the operator code of `OP_FASLE` that was pushed into `stack` in the third iteration), `fValue` will be `false` (line 70).
    - This `false` will be pushed into `vfExec` (line 75).
    - Now the forth operator `OP_IF` has been processed. The execution flows starts the next loop at line 13.
- Fifth operator `statement1`:
    - In this loop, `vfExec` contains `[true, false]`. So `fExec` is `false` (line 15).
    - No matter what the current operator `opcode` is extracted from `statement1`, the condition at line 23 will be not satisfied.

    - As for the condition at line 25, since `statement1` contains no operator between `OP_IF` and `OP_ENDIF` by assumption, it will not be satisfied either.
    - At this point, the execution flows will go back to line 13 and starts a new loop.
    - So basically whatever it is in `statement1`, it is read out and then discarded. And this is the right thing to do, since the condition to execute this branch is not satisfied (the condition is `OP_FALSE`).
- Sixth operator `OP_ELSE`:
    - In this loop, `vfExec` contains `[true, false]`. So `fExec` is `false` (line 15).
    - At line 23, since `fExec` is false, the execution flow goes to line 25.
    - At line 25, since `OP_ELSE` lies in between `OP_IF` and `OP_ENDIF`, the execution goes to line 26.
    - At line 26, `opcode` matches line 78.
    - At line 82, the top element in `vfExec` is flipped. So now `vfExec` contains `[true, true]`. The second `true` is flipped from the previous value `false`.
    - At line 84, the execution goes back to line 13 and starts the next loop.
- Seventh operator `statment2`:
    - In this loop, `vfExec` contains `[true, true]`. So `fExec` is `true` (line 15).
    - At line 23, if `opcode` extracted is less than `OP_PUSHDATA4`, the operand data will be pushed into `stack` and the execution will go back to line 13 and start the next loop.
    - Otherwise, since `fExec` is `true`, the `switch` statement at line 26 will always be executed. This will lead to the execution of `statement2` for what it is.
    - So `statement2` will always be executed, no matter what kind of instruction it is. And this is the right thing to do, since the condition to execute this branch is satisfied.
- Eight operator `OP_ENDIF`:
    - In this loop, `vfExec` contains `[true, true]`. So `fExec` is `true` (line 15).
    - At line 23, since `OP_ENDIF > OP_PUSHDATA4`, the execution flow goes

to line 25.
    - At line 25, since `OP_ENDIF` lies in between `OP_IF` and `OP_ENDIF`, the execution goes to line 26.
    - At line 26, `OP_ENDIF` matches line 85.
    - At line 89, the top element of `vfExec` is popped out. So now `vfExec` contains `[true]`.
    - At line 91, the execution goes back to line 13 and starts the next loop.
- Ninth operator `OP_ELSE`:
    - In this loop, `vfExec` contains `[true]` and so `fExec` is `true`.
    - At line 23, since `OP_ELSE > OP_PUSHDATA4`, the execution flow goes to line 25.
    - At line 25, since `OP_ELSE` lies in between `OP_IF` and `OP_ENDIF`, the execution goes to line 26.
    - At line 26, `OP_ELSE` matches line 78.
    - At line 82, the value of top element in `vfExec` is flipped. So now `vfExec` contains `[false]`.
    - At line 84, the execution goes back to line 13 and starts the enxt loop.
- Tenth operator `statement3`:
    - In this loop, `vfExec` contains `[false]` and so `fExec` is `false`.
    - Again, not matter what kind of instruction `statemen3` is, it will be read out and then discarded.
- Eleventh operator `OP_ENDIF`:
    - This operator will pop the last element of `vfExec` out at line 89. `vfExec` will become empty again, just like it was at the very beginning.

Therefore, every time the execution flow encounters: 1) an `OP_IF`, a bool value at the top of `stack` is pushed into `vfExec`; 2) an `OP_ELSE`, the bool value at the top of stack `vfExec` is flipped; 3) an `OP_ENDIF`, the bool at the top of `vfExec` is popped out.

At any point of time, elements in `vfExec` represent the position of the current instruction within the nested `OP_IF` statement. For example, `[true, false]` means that the current instruction is located in the `false` branch of the inner statement, which in turn is located in the `true` branch of the outer statement. This is the reason why `fExec` is `false` if there is any `false` in `vfExec`. Because any `false` in `vfExec` indicates that the current instruction is in some `false`

branch of the nested `OP_IF` statement and thus should not be executed.

4.3.2. A few `switch` branches

- `OP_EQUALVERIFY` branch is pretty simple (lines 106-107), it pops up the top two items from `stack` and compares them (lines 113-114).
- Hash operators are grouped together (lines 137-141). Outputs for operators `OP_RPEMD160`, `OP_SHA1` and `OP_HASH160` are 160 bits, and for `OP_SHA256` and `OP_HASH256` are 256 bits. Data to be hashed are extracted from the top of `stack` (line 146), the result is pushed back to `stack` (line 165).
- `OP_CHECKSIG` branch is more interesting (lines 174-175). This branch checks the signature of transaction `txTo` (the second argument). Let's use Verification Cases A and B shown at the end of *Chapter Transaction I* as an example to examine how this operator is executed to verify the signature.

4.3.3. Execute `OP_CHECKSIG`

For easy reference, Let's copy Verification Cases A and B in *Chapter Transaction I* below:

- Verification Case A:
  `<your_signature_vchSig> <your_pubKey_vchPubKey> OP_CODESEPARATOR OP_DUP OP_HASH160 <your_address_hash160> OP_EQUALVERIFY OP_CHECKSIG`,
  i.e., Signature A + `OP_CODESEPARATOR` + Script A;
- Verification Case B:
  `<your_signature_vchSig> OP_CODESEPARATOR <your_pubKey_vchPubKey> OP_CHECKSIG`,
  i.e., Signature B + `OP_CODESEPARATOR` + Script B.

I'll work out Verification Case B. You can work out Verification Case A by yourself.

- First notice that Signatures B + `OP_CODESEPARATOR` + Script B is passed in as the first argument `script` to function `EvalScript()`.
- The first instruction to read is `<your_signature_vchSig>`, which is a signature of type `vector <unsigned char>`. Therefore, it will be pushed to `stack` at line 24.
- The second instruction to read is `OP_CODESEPARATOR`. At line 171, a local variable `pbegincodehash` is assigned value `pc`, which points to the next instruction to read, which is `<your_pubKey_vchPubKey>`.
- The third instruction to read is `<your_pubKey_vchPubKey>`, which is a 65-bit public key, and thus will be pushed into `stack` at line 24.

- The forth instruction to read is `OP_CHECKSIG`. The execution goes to the `OP_CHECKSIG` branch (lines 174-175).
- At line 180, `<your_signature_vchSig>` is extracted from `stack` as `vchSig = stacktop(-2)`.
- At line 181, `<your_pubKey_vchPubKey>` is extracted from `stack` as `vchPubKey = stacktop(-1)`.
- At line 186, instructions between `pbegincodehash` and `pend` are read to a `CScript` object `scriptCode`. Remember `pbegincodehash` points to `<your_pubKey_vchPubKey>`, and `pend` points to the end of `script`. So all instructions starting at `<your_pubKey_vchPubKey>` (inclusive) are read into `scriptCode`, which is `<your_pubKey_vchPubKey> OP_CHECKSIG`, i.e., it is Script B.
- Line 188 will have no effect on `scriptCode`, since it does not contain any signature.
- At line 189, function `CheckSig()` is called with arguments `CheckSig(vchSig, vchPubKey, scriptCode, txTo, nIn, nHashType)`. Here is the source code of this function.

```
    1: bool CheckSig(vector<unsigned char> vchSig, vector<unsigned char>
vchPubKey, CScript scriptCode,
    2:                 const CTransaction& txTo, unsigned int nIn, int
nHashType)
    3: {
    4:     CKey key;
    5:     if (!key.SetPubKey(vchPubKey))
    6:         return false;
    7:     // Hash type is one byte tacked on to the end of the signature
    8:     if (vchSig.empty())
    9:         return false;
   10:     if (nHashType == 0)
   11:         nHashType = vchSig.back();
   12:     else if (nHashType != vchSig.back())
   13:         return false;
   14:     vchSig.pop_back();
   15:     if (key.Verify(SignatureHash(scriptCode, txTo, nIn,
nHashType), vchSig))
   16:         return true;
   17:     return false;
   18: }
```

Let's examine this function:

- Its first argument `vchSig` is `<your_signature_vchSig>`;
- The second argument `vchPubKey` is `<your_pubKey_vchPubKey>`;
- The third one `scriptCode` is `<your_pubKey_vchPubKey>`

`OP_CHECKSIG`.

Function `CheckSig()` calls function `SignatureHash(scriptCode, txTo, nIn, nHashType)` at line 15. If you compare function *SignatureHash()* covered in *Chapter Transaction I*, you can see that the input arguments are exactly the same. So you will get exactly the same hash code.

Now go back to continue tracking the execution of Verification Case B:

- At line 190-191, the top 2 elements of `stack`, i.e., `<your_signature_vchSig>` and `<your_pubKey_vchPubKey>`, are popped out.
- At line 192, the return value of `CheckSig()`, which is `true`, will be pushed into `stack`. So `stack` now contains a single `true`, meaning that the execution is successful.

4.4. CBigNum

Class `CBigNum` is a wrapper around type `BIGNUM` that is defined in `openssl` library. Public key cryptography needs to handle very large integers. Standard data types are not enough for that purpose. `BIGNUM` can hold arbitrarily large integers.

The structure of class `CBigNum` is not complicated. It has a bunch of constructors that construct big numbers from various data types, including `char`, `short`, `int`, `long`, `int64`, `int256`, their `unsigned` versions, and `vector<unsigned char>`, etc. It also overloads operators like addition, subtraction, multiplication, division, bit-shifting, etc. All real work is delegated to base type `BIGNUM`. Most of the codes in `CBigNum` merely prepare data for `BIGNUM` functions. Some of the data preparation codes look puzzling if you don't know the `BIGNUM` functions they are calling. Here is an example.

4.4.1. CBigNum::setuint64()

```
 1: class CBigNum : public BIGNUM
 2: {
 3: public:
 4:     //......
 5:     CBigNum(uint64 n)           { BN_init(this); setuint64(n); }
 6:     //......
 7:     void setuint64(uint64 n)
 8:     {
 9:         unsigned char pch[sizeof(n) + 6];
10:         unsigned char* p = pch + 4;
11:         bool fLeadingZeroes = true;
12:         for (int i = 0; i < 8; i++)
```

```
13:         {
14:             unsigned char c = (n >> 56) & 0xff;
15:             n <<= 8;
16:             if (fLeadingZeroes)
17:             {
18:                 if (c == 0)
19:                     continue;
20:                 if (c & 0x80)
21:                     *p++ = 0;
22:                 fLeadingZeroes = false;
23:             }
24:             *p++ = c;
25:         }
26:         unsigned int nSize = p - (pch + 4);
27:         pch[0] = (nSize >> 24) & 0xff;
28:         pch[1] = (nSize >> 16) & 0xff;
29:         pch[2] = (nSize >> 8) & 0xff;
30:         pch[3] = (nSize) & 0xff;
31:         BN_mpi2bn(pch, p - pch, this);
32:     }
33:     //......
34: }
```

Function `setuint64(uint64 n)` sets up a big number equal to `n`. It is called in a constructor that takes an input argument of type `uint64`. If you examine the body of this function, it is not immediately clear what it is doing. At the end of this function, it calls `BN_mpi2bn()`. Here is the document of `BN_mpi2bn()`:

"BN_bn2mpi() and BN_mpi2bn() convert BIGNUMs from and to a format that consists of the number's length in bytes represented as a 4-byte big-endian number, and the number itself in big-endian format, where the most significant bit signals a negative number (the representation of numbers with the MSB set is prefixed with null byte)".

Like the parking signs in Manhattan, you have to double-read this description before you think you understand it. Put it in plain English: These two functions convert a `BIGNUM` from and to another format called mpi. That format consists of two parts. The first part (length-part) has 4 bytes; and it contains a number that is the length of the second part. The second part (data-part) is the actual big number. Both numbers, the big number and its length, are represented in the big-endian format (the most significant byte in the lowest address).

Now let's go back to function `setuint64()`. Here is what this function does:

- Allocates a buffer `pch` of `8+6=14` bytes (line 9). Note `n` is of type `uint64` and has `8` bytes.
- Pointer `p` points to the 4-th byte of `pch`, which is the starting point of the data-part (line 10).
- For each byte of `n`, starting from the most significant byte (`(n >> 56) & 0xff` extracts the most significant byte), converts it to a `unsigned char c`

(line 14).

- Moves the next byte of `n` to the most significant byte position (`n <<=8`) so it is ready for the next extraction (line 15).
- Now if you just ignore lines 16-23, the next step `*p++ =c` simply puts `c` into the position pointed by `p`, and increases `p` by one so it is ready for the next move (line 24).
- Therefore what this function does is to extract the byte of `n` one by one and re-arrange them in the big-endian format. The reason for doing that is because `n` is little-endian (Windows is little-endian).
- After that, this function calculates the size of the re-arrangeed data . Clearly, `p-(pch + 4)` is the length of the re-arrangeed data since `p` started at `pch + 4` (line 26).
- It then extracts the bytes of `nSize` one by one and puts them in the big-endian format from `pch[0]` to `pch[3]` (lines 27-30).
- Now the mpi format is ready and it calls `BN_mpi2bn()`.

So what does the if-block at lines 16-23 do then? It turns out the that the mpi format requires that the data-part to ignore leading zeros. A number `0x7E` takes 1 byte. It can also takes 2 bytes and represented as `0x007E` with the most significant byte be `0x00`. The second format is not allowed in mpi, you have to remove leading zeros when filling in the data-part. This is part of what the if-block does. If condition `fLeadingZeros` is `true`. The extracted char `c` is checked if it is zero (line 18). If yes, `c` is ignored and the procedure continues to extract the next byte. This ignores the leading zeros of `n`.

But then what does the mysterious conditional statement `if (c & 0x80) *p++ = 0` at line 20 do then ? Well, literally, it checks if `c` is greater than `0x80`. If yes, the byte pointed by `p` is filled with `0` and `p` is increased by one. So if the most significant bit of `c` is one (i.e., `c` is greater than `0x80`), an extract null byte is inserted in front of the byte that will hold `c`.

You may be wondering “Why all this hassle”? I was wondering too. A quick experiment reveals that if you don’t do that, a `0x8F` input will be interpreted as a negative number `-0x0F`. So here is the reason: In the mpi format, the most significant bit of the first non-null byte is interpreted as the sign of the number. If you wish not to do so, put a zero byte before that byte. So `fLeadingZeros` is initialized to `true` (line 11). Once the first non-null byte is extracted, this byte is checked against `0x80` to see if its most significant bit is `1` (line 20). After that, `fLeadingZeros` is set to false. And the if-block will never be activated again no matter what.

Now if you go back and triple-read the quoted document text, you will be amazed to realize that all the details I have just painstakingly explained is already in that short paragraph.

Now let's examine two member functions of `CBigNum` that import/export big numbers from/to `vector<unsigned char>`.

4.4.2. `CBigNum::setvch()` and `CBigNum::getvch()`

```
 1: class CBigNum : public BIGNUM
 2: {
 3: //......
 4:     void setvch(const std::vector<unsigned char>& vch)
 5:     {
 6:         std::vector<unsigned char> vch2(vch.size() + 4);
 7:         unsigned int nSize = vch.size();
 8:         vch2[0] = (nSize >> 24) & 0xff;
 9:         vch2[1] = (nSize >> 16) & 0xff;
10:         vch2[2] = (nSize >> 8) & 0xff;
11:         vch2[3] = (nSize >> 0) & 0xff;
12:         reverse_copy(vch.begin(), vch.end(), vch2.begin() + 4);
13:         BN_mpi2bn(&vch2[0], vch2.size(), this);
14:     }
15:     std::vector<unsigned char> getvch() const
16:     {
17:         unsigned int nSize = BN_bn2mpi(this, NULL);
18:         if (nSize < 4)
19:             return std::vector<unsigned char>();
20:         std::vector<unsigned char> vch(nSize);
21:         BN_bn2mpi(this, &vch[0]);
22:         vch.erase(vch.begin(), vch.begin() + 4);
23:         reverse(vch.begin(), vch.end());
24:         return vch;
25:     }
26: //......
27: }
```

`setvch()` finds the size of input vector `vch` and reverses its order to construct a `CBigNum`. So `vch[0] = 0x34` and `vch[1]=0x12` is interpreted as `0x1234` (the most significant byte is the last byte of `vch`). So `vch` is interpreted as in the little-endian format, just like input `uint64 n` in `setuint64()`. Little-endian is a unnatural format for human to read. But that is what the Bitcoin system adopts.

`getvch()` exports a `CBigNum` to a vector of chars in the little-endian format.

5. Block

The full version of this book can be purchased at
https://ebook.ubiqlink.com/.

6. Database

The full version of this book can be purchased at
https://ebook.ubiqlink.com/.

7. Transaction II

The full version of this book can be purchased at
https://ebook.ubiqlink.com/.

8. Network

The full version of this book can be purchased at https://ebook.ubiqlink.com/.

9. Blockchain

The full version of this book can be purchased at
https://ebook.ubiqlink.com/.

10. Mining

Last updated 2015-12-11 09:00:21 EST