Modular Programming with Abstraction

Humayun Kabir

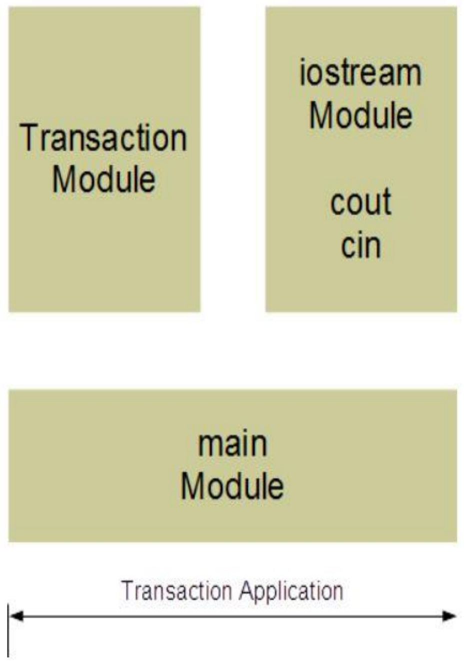
Professor, CS, Vancouver Island University, BC, Canada

Modular Programing with Abstraction

- A modular design consists of a set of ***modules***, which are developed and tested separately.
- C programming language supports modular design through library modules composed of functions.
- The **stdio** module provides input and output support, while hiding its implementation details; typically, the implementation for **scanf()**and **printf()** ships in binary form with the compiler.
- The **stdio.h** header file provides the abstraction, which is all that we need to complete our source code.
- We should practice the same modular design and abstraction in our own development.

Modular Programing with Abstraction

- The **main** module accesses the **Transaction** module.
- The **Transaction** module accesses the **iostream** module.
- The **Transaction** module defines the transaction functions used by the application.
- The **iostream** module defines the **cout** and **cin** objects used by the application.

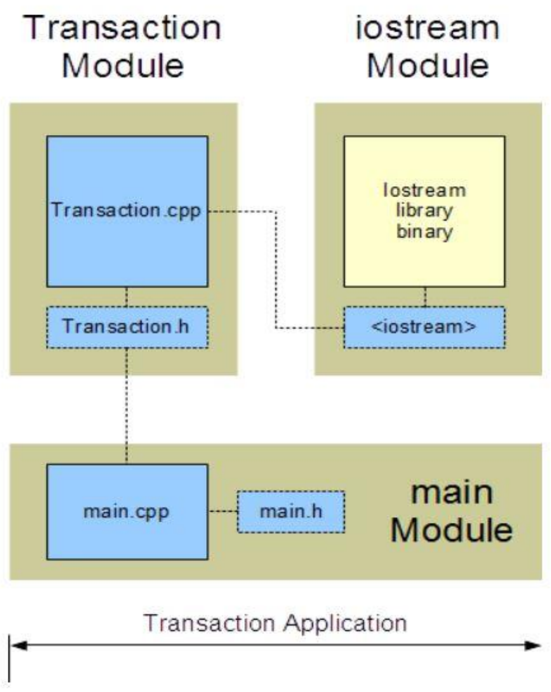


Modular Programing with Abstraction

- To translate the source code of any module the compiler only needs the names used within the module but defined outside the module.
- To enable this in C++, we store the source code for each module in two separate files:
  - a **header file** - declares the function or class prototypes
  - an **implementation file** - defines the functions and contains all of the logic
- The file extension **.h** (or **.hpp**) identifies the header file.
- The file extension **.cpp** identifies the implementation file.

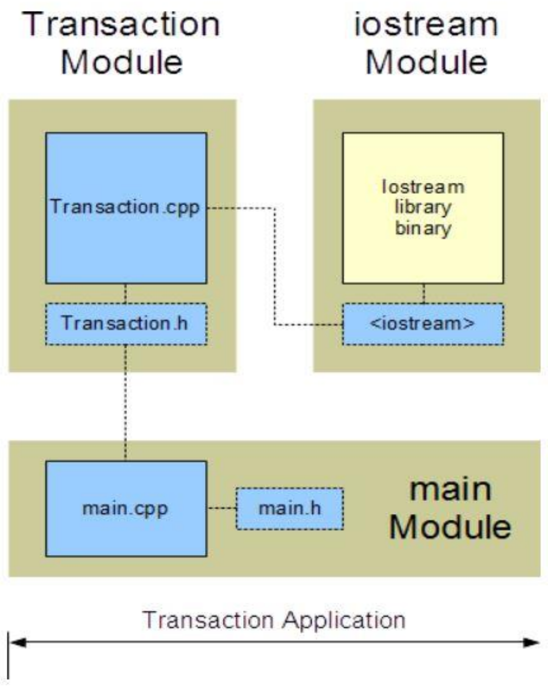
Modular Programing with Abstraction

- The **main.h** file contains definitions specific to the **main** module and the
- **Transaction.h** file contains definitions specific to the **Transaction** module.



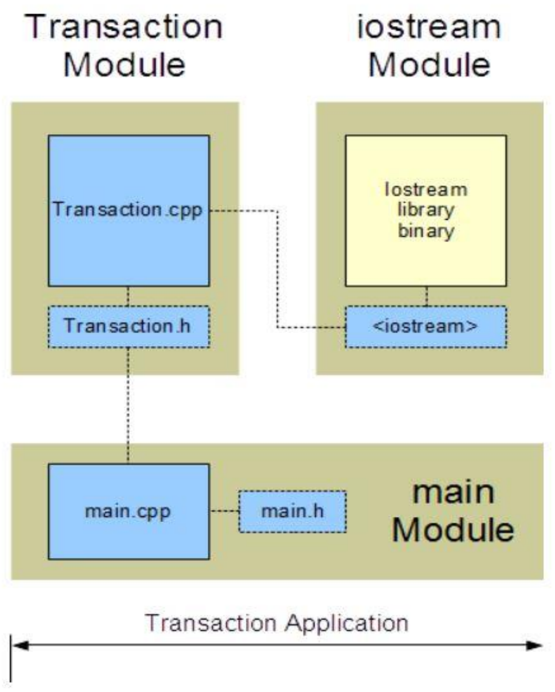
Modular Programing with Abstraction

- The implementation file for the **main** module includes the header files for itself (**main.h**) and the **Transaction**module (**Transaction.h**).
- The implementation file for the **Transaction** module includes the header files for itself (**Transaction.h**) and the **iostream** module.
- An implementation file can include several header files but DOES NOT include any other implementation file.



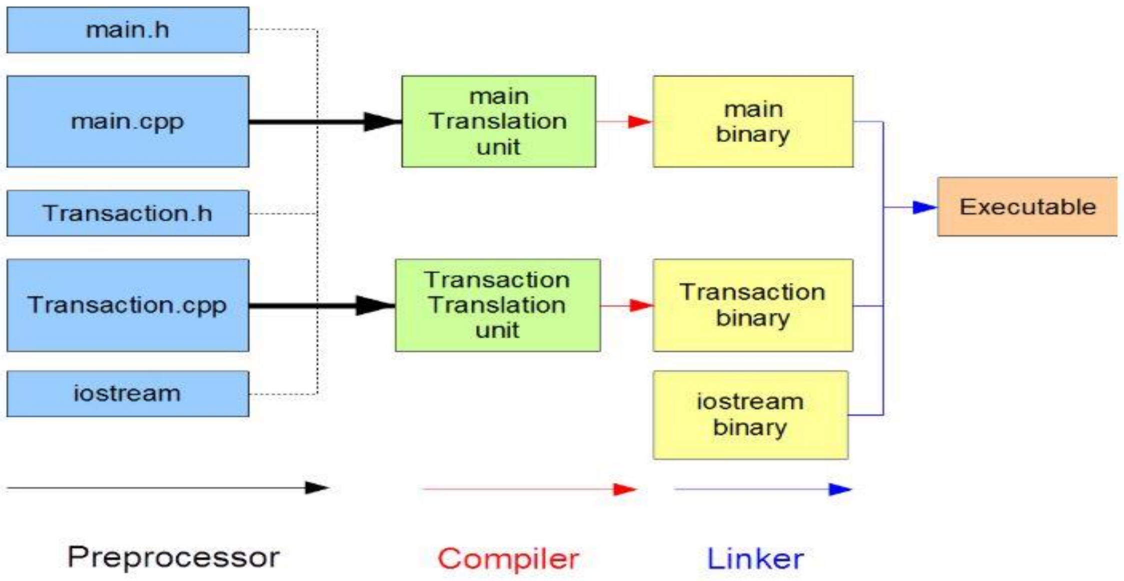
Modular Programing with Abstraction

- We compile each implementation (***.cpp**) file separately and only once.
- We do not compile header (***.h**) files.
- A compiled version of **iostream**'s implementation file is part of the system library.



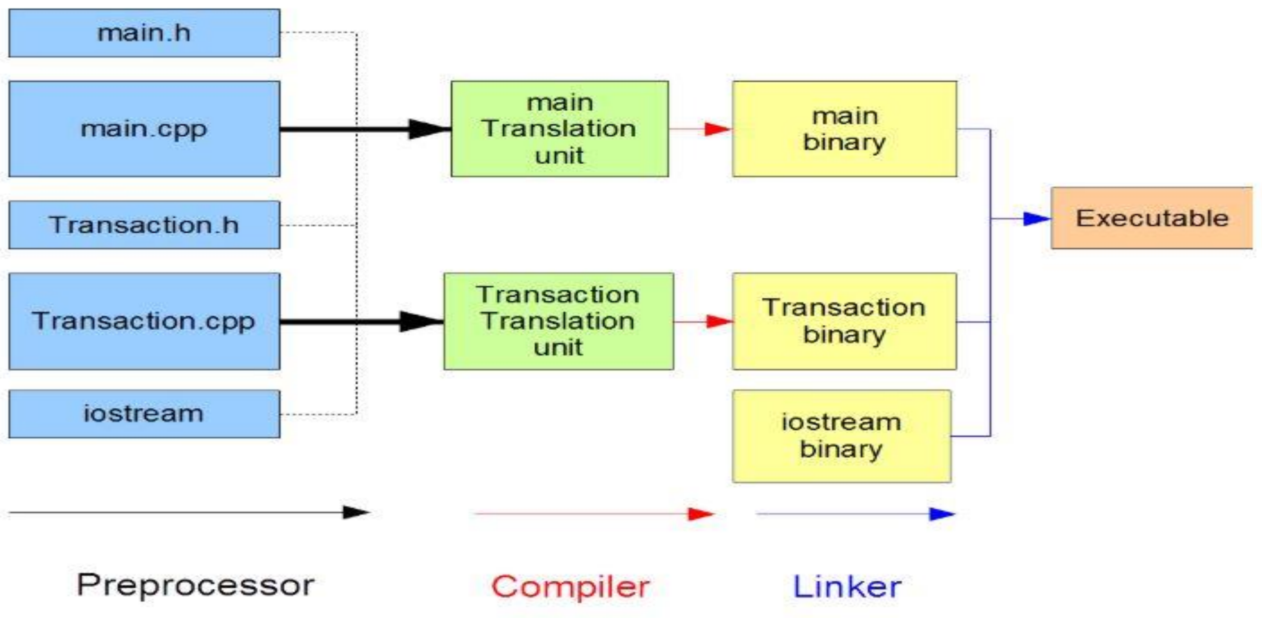
Modular Programing with Abstraction

- Preprocessor
  - interprets all directives creating a single translation unit for the compiler
  - inserts the contents of all **#include** header files
  - substitutes all **#define** macros



Modular Programing with Abstraction

- Compiler - compiles each translation unit separately and creates a corresponding binary version
- Linker - assembles the various binary units along with the system binaries to create one complete executable binary



Modular Programing with Abstraction

```
// Modular Example
// Transaction.h

struct Transaction {
    int acct;       // account number
    char type;      // credit 'c' debit 'd'
    double amount; // transaction amount
};

void enter(struct Transaction* tr);
void display(const struct Transaction* tr);
```

```
// Modular Example
// Transaction.cpp

#include <iostream> // for cout, cin
#include "Transaction.h"  // for Transaction
using namespace std;

// prompts for and accepts Transaction data
//
void enter(struct Transaction* tr) {

    cout << "Enter the account number : ";
    cin  >> tr->acct;
    cout << "Enter the account type (d debit, c credit) : ";
    cin  >> tr->type;
    cout << "Enter the account amount : ";
    cin  >> tr->amount;
}

// displays Transaction data
//
void display(const struct Transaction* tr) {

    cout << "Account " << tr->acct;
    cout << ((tr->type == 'd') ? " Debit $" : " Credit $") << tr->amount;
    cout << endl;
}
```

Modular Programing with Abstraction

```
// Modular Example
// main.h

#define NO_TRANSACTIONS 3
```

```
// Modular Example
// main.cpp

#include "main.h"
#include "Transaction.h"

int main() {
     int i;
     struct Transaction tr;

     for (i = 0; i < NO_TRANSACTIONS; i++) {
          enter(&tr);
          display(&tr);
     }
}
```

Modular Programing with Abstraction

- Separate Compiling
  - ***g++ -Wall –c Transaction.cpp***   //Creates ***Transaction.o***
  - ***g++ -Wall –c main.cpp***   //Creates ***main.o***
- Linking object files together
  - ***g++ -Wall –o accounting main.o Transaction.o***
- Executing
  - ***./accounting***

Modular Programing with Abstraction

Header Guards

```
#ifndef __TRANSACTION_HEADER__
#define __TRANSACTION_HEADER__

// contents of Transaction.h
//...

#endif
```

- To ensure it is safe to include a file more than once.

Modular Programing with Abstraction

Header Guards

```
#ifndef __TRANSACTION_HEADER__
#define __TRANSACTION_HEADER__

// contents of Transaction.h
//...

#endif
```

If this variable is not defined…

Define it.

End of guarded area.

Modular Programing with Abstraction

- Enables individual module development, compilation, and testing
- Easy to develop the software
- Easy to test the software
- Easy to maintain the software