Completing a Program

Lecture 7

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Software Development Notes

The Four Tenets of OO Paradigm

- Abstraction
  - Hidden Data
    - Implementation of Abstract Data Type (ADT) is irrelevant
    - *** Interdependent class data members are not (NEVER) accessed directly ***
      - No public interdependent class data members
      - It’s fine to expose independent class data members
- Encapsulation
  - Data and methods on that data are bundled together
    - A class defines the data implementation, access to the data elements, and methods that act on the data
- Inheritance
  - A class can take on the properties of another class
    - Creates the is-a relationship between the base type and the super-type (derived type)
- Polymorphism
  - Derived objects (those of a class inherited from another) can behave differently
    - Interface of inherited methods remain the same, but may function differently

The Five Principles of Type Design

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

- Other OO Design Principles
  - YAGNI
  - Once & Only Once

http://wiki.c2.com/?PrinciplesOfObjectOrientedDesign

1. Single Responsibility Principle

- Each responsibility should be a separate type, because each responsibility is an axis of change
- A type should have one, and only one, reason to change
- If a change to the business rules causes a type to change, then a change to the GUI, report format, or any other derived segment of the system should not force that type to change

http://wiki.c2.com/?SingleResponsibilityPrinciple

2. Open/Closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification
- Types (their interfaces) should be designed as if they will persist forever
- The motivation is to prevent the introduction of bugs

http://wiki.c2.com/?OpenClosedPrinciple

3. The Liskov Substitution Principle

- "If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T." – Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices, 23,5 (May, 1988).
- In other words, derived class objects must be substitutable for the base class objects.
- That means objects of the derived class must behave in a manner consistent with the promises made in the base class' contract.
- Types implementing an interface should not change the core semantics of that interface

http://wiki.c2.com/?LiskovSubstitutionPrinciple

4. The Interface Segregation Principle

- The dependency of one type to another one should depend on the smallest possible interface

https://wiki.c2.com/?InterfaceSegregationPrinciple

5. Dependency Inversion Principle

- High level modules should not depend upon low level module internals; both should depend upon abstractions (interfaces)
- Abstractions should not depend upon details; details should depend upon abstractions

http://wiki.c2.com/?DependencyInversionPrinciple

The YAGNI Principle

- "You aren't gonna need it"
- Avoid developing unless you have to:
  - The cheapest code is the code you don't write
- If it's in the requirements you probably do need it

http://c2.com/xp/YouArentGonnaNeedIt.html

Once and Only Once

- We never want to duplicate code
- What if there's an error in the code?
  - Now you have to change it everywhere
  - There is no way to ensure that code remains in sync

Best Practices

- Separate what changes from what stays the same
  - If something stays the same, you won't break it
  - Keeping change limited reduces the amount of analysis
- Coupling vs. Cohesion
  - Coupling is bad, cohesion is good
  - Classes should work together in small, cohesive clusters
  - You should have high cohesion within modules and low coupling between modules
- Program to an interface, not an implementation
  - Depend on an interface where practical (instead of a concrete type)
  - Allows types to be swapped out later
  - Even more pragmatic with duck-typing

Building a Calculator

Abstract

- Tokens and token streams
  - Structs and classes
- Cleaning up the code
  - Prompts
  - Program organization
    - constants
  - Recovering from errors
  - Commenting
  - Code review
  - Testing
- A word on complexity and difficulty
  - Variables

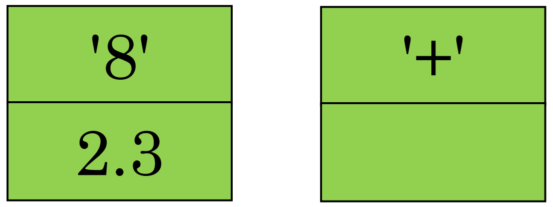
Token

- We want a type that can hold a “kind” and a value:

```
// Token stuff
struct token        // define a type called Token
{
    char kind;      // what kind of token
    double value;   // used for numbers (only): a value
};

token t;
t.kind = '8';    // . (dot) is used to access members
                 // (use '8' to mean "number")
t.value = 2.3;

token u = t;    // a token behaves much like a built-in type, such as int
                // so u becomes a copy of t
std::cout << u.value << "\n";    // will print 2.3
```



User defined Type: token

```
// user-defined type called ‘token’
struct token
{
     // data members
     // function members
};
```

- A **struct** is the simplest form of a class (type)
- “class” is C++’s term for “user-defined type”
- Defining types is the crucial mechanism for organizing programs in C++
  - as in most other modern languages
- a **class** (including **struct**s) can have
  - Data members (to hold information), and
  - Function members (providing operations on the data)
  - Member functions have implicit access to other members

User defined Type: token

```
struct token
{
    char kind;        // what kind of token
    double value;     // for numbers: a value

    // constructors
    token(char ch) : kind(ch), value(0) {}
    token(double val) : kind('8'), value(val) {}
};
```

- A constructor has the same name as its class and has no return value
- A constructor defines how an object of a class is initialized
  - Here **kind** is initialized with **ch**, and
  - **value** is initialized with **val** or **0**
  - `token t1('+');          // make a token t1 of “kind” '+'`
  - `token t2(4.5);          // make a token t2 of “kind” '8' and value 4.5`

User defined Type: token

```
class token
{
    char kind_;       // what kind of token
    double value_;    // for numbers: a value

public:
    // constructors
    token(char ch) : kind_(ch), value_(0) {}
    token(double val) : kind_('8'), value_(val) {}
    char kind() const { return kind_; }
    double value() const { return value_; }
};
```

- A constructor has the same name as its class and has no return value
- A constructor defines how an object of a class is initialized
  - Here **`kind_`** is initialized with **`ch`**, and **`value_`** is initialized with **`val`** or **`0`**
  - `token t1('+'); // make a token t1 of “kind” '+'`
  - `token t2(4.5); // make a token t2 of “kind” '8' and value 4.5`

token_stream

- A **`token_stream`** reads characters, producing **`token`**s on demand
- We can put a **`token`** 'back' into a **`token_stream`** for later use
- A **`token_stream`** uses a "buffer" to hold tokens we put back into it

token_stream buffer:	empty
Input stream:	1+2*3;

For **1+2*3;**, **`expression()`** calls **`term()`** which reads **1**, then reads **+**, decides that **+** is a job for **"someone else"** and puts **+** back in the **`token_stream`** (where **`expression()`** will find it)

token_stream buffer:	token('+')
Input stream:	2*3;

token_stream

- A **token_stream** reads characters, producing **token**'s
- We can put back a **token**

```
class token_stream {
    // representation: not directly accessible to users (private):
    bool full;       // is there a token in the buffer?
    token buffer;    // here is where we keep a token put back using putback()

public:
    // user interface:
    token get();            // get a token
    void putback(token);    // put a token back into the token_stream

    token_stream();         // constructor: make a token_stream
};
```

- A constructor
  - defines how an object of a class is initialized
  - has the same name as its class, and no return type

token_stream Implementation

```
class token_stream {
    // representation: not directly accessible to users:
    bool full;       // is there a token in the buffer?
    token buffer;    // here is where we keep a Token put back using putback()
public:
    // user interface:
    token get();              // get a token
    void putback(token);      // put a token back into the token_stream

    // constructor: make a token_stream, the buffer starts empty
    token_stream() : full(false), buffer('\0') {}
};

void token_stream::putback(token t) {
    if (full) throw std::runtime_error("putback() into a full buffer");
    buffer = t;
    full = true;
}
```

token_stream Implementation

```
token token_stream::get() {     // read a token from the token_stream
    if (full) {                 // check if we already have a Token ready
        full = false; return buffer;
    }

    char ch;
    std::cin >> ch;             // note that >> skips whitespace (space, newline, tab, etc.)

    switch (ch) {
    case '(': case ')': case ';': case 'q': case '+': case '-': case '*': case '/':
        return token(ch);           // let each character represent itself
    case '.': case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9': {
        std::cin.putback(ch);       // put digit back into the input stream
        double val;
        std::cin >> val;            // read a floating-point number
        return token(val);          // let '8' represent "a number"
    }
    default:
        throw std::runtime_error("Bad token");
    }
}
```

Streams

- Note that the notion of a stream of data is extremely general and very widely used
  - Most I/O systems
    - E.g., C++ standard I/O streams
  - With or without a putback/unget operation
    - We used putback for both token_stream and cin

The calculator is primitive

- We can improve it in stages
  - Style – clarity of code
    - Comments
    - Naming
    - Use of functions
    - …
  - Functionality – what it can do
    - Better prompts
    - Recovery after error
    - Negative numbers
    - % (remainder/modulo)
    - Pre-defined symbolic values
    - Variables
    - …

Prompting

- Initially we said we wanted

```
Expression: 2+3; 5*7; 2+9;
Result : 5
Expression: Result: 35
Expression: Result: 11
Expression:
```

- But this is what we implemented

```
2+3; 5*7; 2+9;
5
35
11
```

- What do we really want?

```
> 2+3;
= 5
> 5*7;
= 35
>
```

Adding prompts and output indicators

```
double val = 0;
std::cout << "> ";     // print prompt
while (std::cin)
{
    token t = ts.get();
    if (t.kind == 'q')
        break;     // check for "quit"
    if (t.kind == ';')
        std::cout << "= " << val << "\n > ";     // print "= result" and prompt
    else
        ts.putback(t);
    val = expression();     // read and evaluate expression
}

> 2+3; 5*7; 2+9;    <--  the program doesn't see input before you hit "enter/return"
= 5
> = 35
> = 11
>
```

The code is getting messy

- Bugs thrive in messy corners
- Time to clean up!
  - Read through all of the code carefully
    - Try to be systematic ("have you looked at all the code?")
  - Improve comments
  - Replace obscure names with better ones
  - Improve use of functions
    - Add functions to simplify messy code
  - Remove "magic constants"
    - E.g. '8' ('8' what could that mean? Why '8'?)
- Once you have cleaned up, let a friend/colleague review the code ("code review")

Remove “magic constants”

```
// Token “kind” values:
char const number = '8';    // a floating-point number
char const quit = 'q';      // an exit command
char const print = ';';     // a print command

// User interaction strings:
std::string const prompt = "> ";
std::string const result = "= ";    // indicate that a result follows
```

Remove “magic constants”

```
// In token::token():
token(double val)
  : kind_(number)    // let ‘8’ represent “a number”
  , value_(val)
{
}

// In primary():
case number:                // rather than case '8':
    return t.value();       // return the number’s value
```

Remove “magic constants”

```
// In main():
while (std::cin)
{
    std::cout << prompt;          // rather than "> "
    token t = ts.get();

    while (t.kind == print)       // rather than == ';'
        t = ts.get();

    if (t.kind == quit)           // rather than =='q'
        break;

    ts.putback(t);
    cout << result << expression() << endl;
}
```

Remove "magic constants"

- But what's wrong with "magic constants"?
  - Everybody knows 3.14159265358979323846264, 12, -1, 365, 24, 2.7182818284590, 299792458, 2.54, 1.61, -273.15, 6.6260693e-34, 0.5291772108e-10, 6.0221415e23 and 42!
  - No; they don't.
- "Magic" is detrimental to your (mental) health!
  - It causes you to stay up all night searching for bugs
  - It causes space probes to self destruct (well … it can … sometimes …)
- If a "constant" could change (during program maintenance) or if someone might not recognize it, use a symbolic constant.
  - Note that a change in precision is often a significant change 3.14 != 3.14159265
  - 0 and 1 are usually fine without explanation, -1 and 2 sometimes (but rarely) are.
  - 12 can be okay (the number of months in a year rarely changes), but probably is not
- If a constant is used twice, it should probably be symbolic. That way, you can change it in one place

So why did we use "magic constants"?

- To make a point
  - Now you see how ugly that first code was
    - just look back to see
- Because we forget (get busy, etc.) and write ugly code
  - "Cleaning up code" is a real and important activity
    - Not just for students
    - Re-test the program whenever you have made a change
  - Ever so often, stop adding functionality and "go back" and review code
    - It saves time

Recover from errors

- Any user error terminates the program
  - That’s not ideal
  - Structure of code

```
int main()
try {
    // ... do “everything” ...
}
catch (std::exception const& e) {
    // catch errors we understand something about
    // ...
}
catch (...) {
    // catch all other errors
    // ...
}
```

Recover from errors

- Move code that actually does something out of main()
  - leave main() for initialization and cleanup only

```
int main()    // step 1
try
{
    calculate();
    return 0;
}
catch (std::exception const& e)
{    // errors we understand something about
    std::cerr << e.what() << std::endl;
    return 1;
}
catch (...)
{    // other errors
    std::cerr << "exception \n";
    return 2;
}
```

Recover from errors

- Separating the read and evaluate loop out into calculate() allows us to simplify it

```
void calculate() {
    while (std::cin) {
        std::cout << prompt;
        token t = ts.get();
        while (t.kind == print)
            t = ts.get();    // first discard all “prints”
        if (t.kind == quit)
            return;          // quit
        ts.putback(t);
        std::cout << result << expression() << std::endl;
    }
}
```

Recover from errors

- Move code that handles exceptions from which we can recover from main() to calculate()

```
int main()    // step 2
try
{
    calculate();
    return 0;
}
catch (...)
{    // other errors (don’t try to recover)
    std::cerr << "exception \n";
    return 2;
}
```

Recover from errors

```
void calculate() {
    while (std::cin) {
        try {
            std::cout << prompt;
            token t = ts.get();
            while (t.kind == print)
                t = ts.get();    // first discard all “prints”
            if (t.kind == quit)
                return;    // quit
            ts.putback(t);
            std::cout << result << expression() << std::endl;
        }
        catch (std::exception const& e) {
            std::cerr << e.what() << std::endl;    // write error message
            clean_up_mess();                       // <<< The tricky part!
        }
    }
}
```

Recover from errors

- First try

```
void clean_up_mess()
{
    while (true) {    // skip until we find a print (';')
        token t = ts.get();
        if (t.kind == print)
            return;
    }
}
```

- Unfortunately, that doesn't work all that well. Why not? Consider the input **1@$z; 1+3;**
  - When you try to **clean_up_mess()** from the bad token **@**, you get a **"Bad token"** error trying to get rid of **$**
  - We always try not to get errors while handling errors

Recover from errors

- Classic problem: the higher levels of a program can't recover well from low-level errors (i.e., errors with bad tokens).
  - Only `token_stream` knows about characters
- We must drop down to the level of characters
  - The solution must be a modification of `token_stream`:

```
class token_stream {
    bool full;        // is there a Token in the buffer?
    token buffer;     // here is where we keep a Token put back using
                      // putback()
public:
    token get();                 // get a Token
    void putback(Token t);       // put back a Token
    token_stream();              // make a token_stream that reads from std::cin
    void ignore(char c);         // <<< discard tokens up to and including a c
};
```

Recover from errors

```
// skip characters until we find a c; also discard that c
void token_stream::ignore(char c)
{
    // first look in buffer:
    if (full && c == buffer.kind()) {    // && means ‘and’
        full = false;
        return;
    }
    full = false;    // discard the contents of buffer

    // now search input:
    char ch = 0;
    while (std::cin >> ch) {
        if (ch == c)
            break;
    }
}
```

Recover from errors

- clean_up_mess() now is trivial
  - and it works

```
void clean_up_mess()
{
    ts.ignore(print);
}
```

- Note the distinction between what we do and how we do it:
  - clean_up_mess() is what users see; it cleans up messes
    - The users are not interested in exactly how it cleans up messes
  - ts.ignore(print) is the way we implement clean_up_mess()
    - We can change/improve the way we clean up messes without affecting users

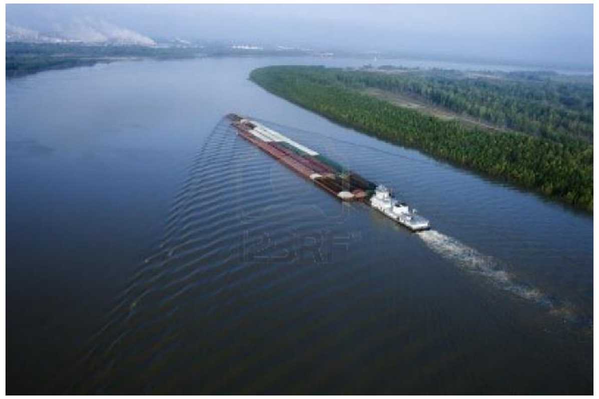
Features

- We did not (yet) add
  - Negative numbers
  - % (remainder/modulo)
  - Pre-defined symbolic values
  - Variables

$\Big\}$ Exercise: implement any (or all) of those

- Major Point
  - Providing "extra features" early causes major problems, delays, bugs, and confusion
  - "Grow" your programs
    - First get a simple working version
    - Then, add features that seem worth the effort

LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP