DM560

Introduction to Programming in C++

Developing a Program

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Bjarne Stroustrup*]

Outline

Outline

1. Writing a Program

2. A First Version

3. Improvements

Overview

We focus on the task of designing a program through the example of a simple "desk calculator."

- Some thoughts on software development
- The idea of a calculator
- Using a grammar
- Expression evaluation
- Program organization

Developing a Program

- Analysis
  - Refine our understanding of the problem
  - Think of the final use of our program
- Design
  - Create an overall structure for the program
- Implementation
  - Write code
  - Debug
  - Test
- Go through these stages repeatedly

Reminder

- We learn by example
  - Not by just seeing explanations of principles
  - Not just by understanding programming language rules
- The more and the more varied examples the better
  - You won't get it right the first time
  - "You can't learn to ride a bike from a correspondence course"

Developing a Program: Example

We'll build a program in stages, making lot of "typical mistakes" along the way

- Even experienced programmers make mistakes
- Designing a good program is genuinely difficult
- It's often faster to let the compiler detect gross mistakes than to try to get every detail right the first time
- Concentrate on the important design choices
- Developing a simple, incomplete version allows us to experiment and get feedback
- Good programs are "grown"

A Simple Calculator

- Given expressions as input from the keyboard, evaluate them and write out the resulting value.

  For example:

Expression: 2+2	Result: 4
Expression: 2+2*3	Result: 8
Expression: 2+3-25/5	Result: 0

- Let's refine this a bit more ...

A Pseudo-Code

A first idea:

```
int main()
{
  variables                          // pseudo code
  while (get a line) {               // what is a line?
    analyze the expression           // what does that mean?
    evaluate the expression
    print the result
  }
}
```

- How do we represent 45+5/7 as data?
- How do we find 45 + 5 / and 7 in an input string?
- How do we make sure that 45+5/7 means 45+(5/7) rather than (45+5)/7?
- Should we allow floating-point numbers (sure!)
- Can we have variables? v=7; m=9; v*m (later)

A Simple Calculator

- Wait! What would the experts do?
  "Don't re-invent the wheel"

- Computers have been evaluating expressions for 50+ years
  There has to be a solution!
  What did the experts do?

- Reading is good for you
  Asking more experienced friends/colleagues can be far more effective, pleasant, and time-effective than slogging along on your own

Outline

1. Writing a Program

2. A First Version

3. Improvements

Expression Grammar

This is what the experts usually do: write a grammar:

```
Expression :
        Term
        Expression ‘+’ Term           e.g., 1+2,   (1-2)+3,   2*3+1
        Expression ‘-’ Term

Term :
        Primary
        Term ‘*’ Primary               e.g., 1*2,   (1-2)*3.5
        Term ‘/’ Primary
        Term ‘%’ Primary

Primary :
        Number                         e.g., 1,   3.5
        ’(’ Expression ’)’             e.g., (1+2*3)

Number :
        floating-point literal        e.g., 3.14, 0.274e1, or 42 - as defined for C++
```

A program is built out of Tokens (e.g., numbers and operators) = something we consider a unit.

Grammars

What's a grammar?

- A set of (syntax) rules for expressions.
- The rules say how to analyze ("parse") an expression.
- Some rules seem hard-wired into our brains
  Example, you know what this means:
  2*3+4/2
  birds fly but fish swim
- You know that this is wrong:
  2 * + 3 4/2
  fly birds fish but swim
- How can we teach what we know to a computer?
  Why is it right/wrong?
  How do we know?

Grammars - “English”

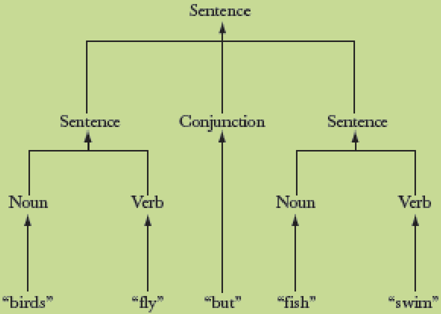
Parsing a simple English sentence

Sentence :
- Noun Verb
- Sentence Conjunction Sentence

Conjunction :
- “and”
- “or”
- “but”

Noun :
- “birds”
- “fish”
- “C++”

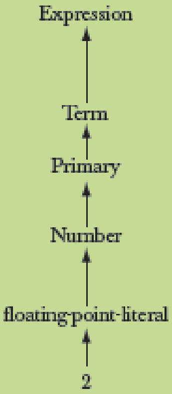
Verb :
- “rules”
- “fly”
- “swim”



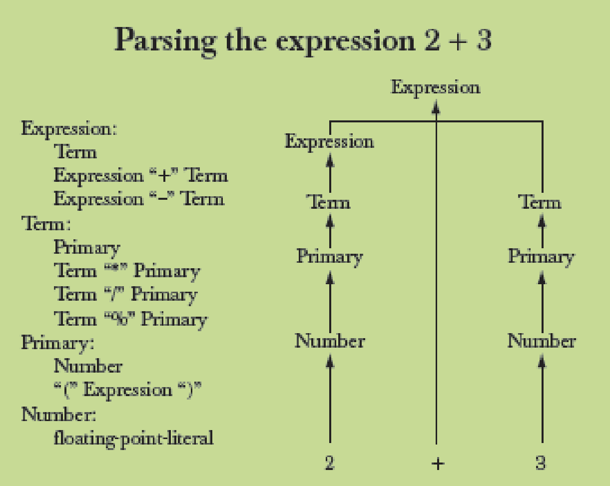
Grammars - Expressions

Parsing the number 2

Expression:
  Term
  Expression "+" Term
  Expression "–" Term
Term:
  Primary
  Term "*" Primary
  Term "/" Primary
  Term "%" Primary
Primary:
  Number
  "(" Expression ")"
Number:
  floating-point-literal



Grammars - Expressions



Grammars - Expressions

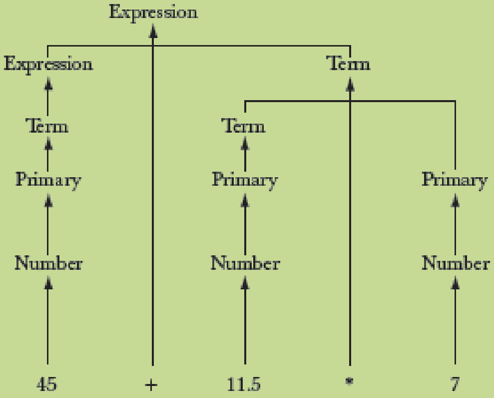
Parsing the expression 45 + 11.5 * 7

Expression:
- Term
- Expression "+" Term
- Expression "–" Term

Term:
- Primary
- Term "*" Primary
- Term "/" Primary
- Term "%" Primary

Primary:
- Number
- "(" Expression ")"

Number:
- floating-point-literal



Functions for Parsing

We need functions to match the grammar rules

```
get()   // read characters and compose tokens
        // calls cin for input

expression()    // deal with + and -
                // calls term() and get()

term()  // deal with *, /, and %
        // calls primary() and get()

primary()       // deal with numbers and parentheses
                // calls expression() and get()
```

- Note: each function deals with a specific part of an expression and leaves everything else to other functions – this radically simplifies each function.
- Analogy: a group of people can deal with a complex problem by each person handling only problems in his/her own specialty, leaving the rest for colleagues.

Function Return Types

What should the parser functions return? How about the result?

```
Token get_token();          // read characters and compose tokens
double expression();        // deal with + and -
                            //      return the sum (or difference)
double term();  // deal with *, /, and %
                //      return the product (or ...)
double primary();           // deal with numbers and parentheses
                            //      return the value
```

What is a Token?

What is a Token?

- We want to see input as a stream of tokens
  - We read characters `1 + 4*(4.5-6)` (That's 13 characters incl. 2 spaces)
  - 9 tokens in that expression: `1 + 4 * ( 4.5 - 6 )`
  - 6 kinds of tokens in that expression: number `+ * ( - )`
- We want each token to have two parts
  - A "kind"; e.g., number
  - A value; e.g., 4
- We need a type to represent this "Token" idea
  - We need to define a class (Chp. 7). For now:
    - `get_token()` gives us the next token from input
    - `t.kind` gives us the kind of the token
    - `t.value` gives us the value of the token

Dealing with + and -

```
Expression:
        Term
        Expression '+' Term // Note: every Expression starts with a Term
        Expression '-' Term
```

```
double expression()     // read and evaluate: 1   1+2.5   1+2+3.14   etc.
{
  double left = term();                    // get the Term
  while (true) {
    Token t = get_token();         // get the next token...
    switch (t.kind) {    // ... and do the right thing with it
      case '+':      left += term(); break;
      case '-':      left -= term(); break;
      default:       return left;          // return the value of the expression
    }
  }
}
```

Dealing with *, / and %

```
Term :
        Primary
        Term '*' Primary   // Note: every Term starts with a Primary
        Term '/' Primary
```

```
double term()    // exactly like expression(), but for *, /, and  %
{
  double left = primary();                    // get the Primary
  while (true) {
    Token t = get_token();                    // get the next Token...
    switch (t.kind) {
      case '*':      left *= primary(); break;
      case '/':      left /= primary(); break;
      case '%':      left %= primary(); break;
      default:       return left;                    // return the value
    }
  }
}  // Oops: doesn't compile: % isn't defined for floating-point numbers
```

Dealing with * and /

```
Term :
        Primary
        Term '*' Primary  // Note: every Term starts with a Primary
        Term '/' Primary
```

```
double term()   // exactly like expression(), but for *, and /
{
  double left = primary();                    // get the Primary
  while (true) {
    Token t = get_token();                    // get the next Token
    switch (t.kind) {
      case '*':      left *= primary(); break;
      case '/':      left /= primary(); break;
      default:       return left;             // return the value
    }
  }
}
```

Dealing with Divide by 0

```
double term()   // exactly like expression(), but for * and  /
{
  double left = primary();                   // get the Primary
  while (true) {
    Token t = get_token();          // get the next Token
    switch (t.kind) {
      case '*':      left *= primary();   break;
      case '/':
      {
        double d = primary();
        if  (d==0) error("divide by zero");
        left /= d;
        break;
      }
      default:       return left;           // return the value
    }
  }
}
```

Note: in `switch` you need a block `{}` if you want to declare variables in a `case`

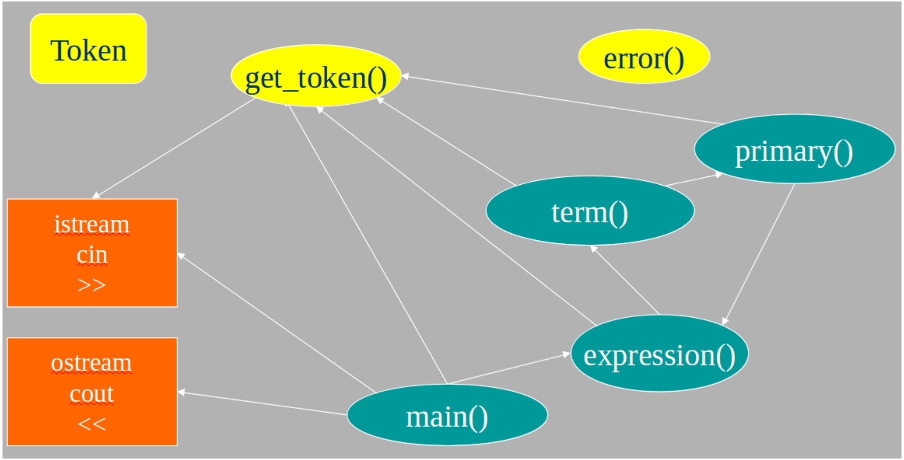
Dealing with Numbers and Parentheses

```
Primary :
        Number
        '(' Expression ')'
Number :
        floating-point literal
```

```
double primary()          // Number or '(' Expression ')'
{
  Token t = get_token();
  switch (t.kind) {
    case '(':                          // handle '('expression ')'
    {
      double d = expression();
      t = get_token();
      if (t.kind != ')') error("')' expected");
      return d;
    }
    case '8':             // we use '8' to represent the ``kind'' of a number
    return t.value;       // return the number's value
    default:
    error("primary expected");
  }
}
```

Program Organization

Who calls whom? (note the loop)



The Program

```
#include "std_lib_facilities.h"

// Token stuff (explained in the next lecture)

double expression(); // declaration so that primary() can call  expression()

double primary() { /* ... */ }  // deal with numbers and parentheses
double term() { /* ... */ }               // deal with * and / (pity about %)
double expression() { /* ... */ }         // deal with + and -

int main() { /* ... */ }                  // on next slide
```

The Program - main()

```
int main()
try {
    while (cin)
    cout << expression() << '\n';
    // keep_window_open();                    // for some Windows versions
}
catch (runtime_error& e) {
    cerr << e.what() << endl;
    // keep_window_open ();
    return 1;
}
catch (...) {
    cerr << "exception \n";
    // keep_window_open ();
    return 2;
}
```

Find the code at: http://www.stroustrup.com/Programming/calculator00.cpp

Execution

```
2

3
4
2                        // an answer
5+6
5                        // an answer
X
Bad token                // an answer (finally, an expected answer)
```

A Detective Job

- Expect “mysteries”
- Your first try rarely works as expected
  - That’s normal and to be expected even for experienced programmers
  - If it looks as if it works be suspicious and test a bit more
  - Now comes the debugging finding out why the program misbehaves
- We have to understand what our code is doing and explain why it does the right thing
- Analyzing our errors is often also the best way to find a correct solution

```
1 2 3 4+5 6+7 8+9 10 11 12
1                      // an answer
4                      // an answer
6                      // an answer
8                      // an answer
10                     // an answer
```

Aha! Our program "eats" two out of three inputs.
How come?
Let's have a look at `expression()`

Dealing with + and -

```
Expression:
        Term
        Expression '+' Term // Note: every Expression starts with a Term
        Expression '-' Term
```

```
double expression()      // read and evaluate: 1   1+2.5   1+2+3.14   etc.
{
  double left = term();                    // get the Term
  while (true) {
    Token t = get_token();        // get the next token...
    switch (t.kind) {   // ... and do the right thing with it
      case '+':      left += term(); break;
      case '-':      left -= term(); break;
      default:       return left;          //  <= does not use "next Token"
    }
  }
}
```

Dealing with + and -

So, we need a way to "put back" a token!

- Put back into what?
- "the input," of course: we need an input stream of tokens, a "token stream"

```
double expression()      // deals with '+' and '-'
{
  double left = term();                   // get the Term
  while (true) {
    Token t = get();     // get the next token from a token stream
    switch (t.kind) {    // ... and do the right thing with it
      case '+':      left += term(); break;
      case '-':      left -= term(); break;
      default:       ts.putback(t); return left; // put the unused token back
    }
  }
}
```

Dealing with * and /

Now make the same change to term()

```
double term()   // deal with * and  /
{
  double left = primary();
  while (true) {
    Token t = ts.get(); // get the next Token from input
    switch (t.kind) {
      case '*':
      // deal with *
      case '/':
      // deal with /
      default:
        ts.putback(t);  // put unused token back into input stream
        return left;
    }
  }
}
```

The Program

- Now the program sort of work
- We get feedback and it starts the fun

Another Case for our Detective

```
2 3 4 2+3 2*3
2                       an answer
3                       an answer
4                       an answer
5                       an answer
```

What!? No “6” ?

- The program looks ahead one token. It’s waiting for the user
- So, we introduce a “print result” command. Let it be `;`
- While we’re at it, we also introduce a “quit” command. Let it be `q`

The main() Program

```
int main()
{
  double val = 0;
  while (cin) {
    Token t = ts.get(); // rather than get_token()
    if (t.kind == 'q') break;            // 'q' for ``quit''
    if (t.kind == ';')                   // ';' for ``print now''
      cout << val << '\n';       // print result
    else
      ts.putback(t);     // put a token back into the input stream
    val = expression(); // evaluate
  }
  keep_window_open();
}
// ... exception handling ...
```

Execution

```
2;
2                        an answer
2+3;
5                        an answer
3+4*5;
23                       an answer
q
```

Completing the Program

Now wee need to complete the implementation

- `Token` and `Token_stream`; `struct` and `class`
- Get the calculator to work better
- Add features based on experience
- Clean up the code:
  After many changes code often becomes a bit of a mess
  We want to produce maintainable code
  - Prompts
  - Program organization
    constants
  - Recovering from errors
  - Commenting
  - Code review
  - Testing

Token

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

'+'

'8'
2.3

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

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
cout << u.value;            // will print 2.3
```

Token

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

Token{'+'};               // make a Token of ``kind''  '+'
Token{'8',4.5}; // make a Token of ``kind'' '8' and value 4.5
```

- A struct is the simplest form of a class
- 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 structs) can have
  - data members (to hold information), and
  - function members (providing operations on the data)

Token_stream

- A Token_stream reads characters, producing Tokens on demand
- We can put a Token into a Token_stream for later use
- A Token_stream uses a "buffer" to hold tokens we put back into it

Example:

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 Tokens. We can put back a Token.

Definition:

```
class Token_stream {
public: // user interface:
        Token get();              // get a Token
        void putback(Token); // put a Token back into the Token_stream
private: // representation: not directly accessible to users:
        bool full {false};        // is there a Token in the buffer?
        Token buffer;    // here is where we keep a Token put back using putback()
};
// the Token_stream starts out empty: full==false
```

Implementation:

```
void Token_stream::putback(Token t)     // note void when nothing returned
{
        if (full) error("putback() into a full buffer");
        buffer=t;
        full=true;
}
```

Token_stream

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

   char ch;
   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':
     {  cin.putback(ch);          // put digit back into the input stream
        double val;
        cin >> val;               // read a floating-point number
        return Token{'8',val};  // let '8' represent "a number"
     }
     default:  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`

Outline

1. Writing a Program

2. A First Version

3. Improvements

Improvements

We can improve the calculator in stages

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

⇝ 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

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;
cout << "> ";                          // print prompt
while (cin) {
  Token t = ts.get();
  if (t.kind == 'q') break;            // check for "quit"
  if (t.kind == ';')
     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
>
```

But my Window Disappeared!

Test case: +1;

```
cout << "> ";                              // prompt
while (cin) {
    Token t = ts.get();
    while (t.kind == ';') t=ts.get();      // eat all semicolons
    if (t.kind == 'q') {
        keep_window_open("~~");
        return 0;
    }
    ts.putback(t);
    cout << "= " << expression() << "\n > ";
}
keep_window_open("~~");
return 0;
```

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' (What could that mean? Why '8'?)
- Once you have cleaned up, let a friend/colleague review the code ("code review")
  Typically, do the review together

Remove Magic Constants

- If a “constant” could change (during program maintenance) or if someone might not recognize it, use a symbolic constant
- If a constant is used twice, it should probably be symbolic

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

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

Remove Magic Constants

```
// In  Token_stream::get():

case '.':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
        {       cin.putback(ch);                    // put digit back into the input
                double val;
                cin >> val;                     // read a floating-point number
                return Token{number,val}; // rather than Token{'8',val}
        }

// In primary():

case number:    // rather than case '8':
                return t.value; // return the number's value
```

Re-test the program whenever you have made a change

Remove Magic Constants

```
// In main():

        while (cin) {
                cout << prompt;                         // rather than "> "
                Token t = ts.get();
                while (t.kind == print) t=ts.get();      // rather than ==';'
                if (t.kind == quit) {                   // rather than =='q'
                        keep_window_open();
                        return 0;
                }
                ts.putback(t);
                cout << result << expression() << endl;
        }
```

Recover from Errors

Currently, any user error terminates the program: That's not ideal!
Structure of code

```
int main()
try {
        // ... do "everything" ...
}
catch (exception& 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();
        keep_window_open();      // cope with Windows console mode
        return 0;
}
catch (exception& e) {           // errors we understand something about
        cerr << e.what() << endl;
        keep_window_open("~~");
        return 1;
}
catch (...) {                    // other errors
        cerr << "exception \n";
        keep_window_open("~~");
        return 2;
}
```

Recover from Errors

Separating the read and evaluate loop out into `calculate()` allows us to simplify it no more ugly `keep_window_open()` !

```
void calculate()
{
    while (cin) {
        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);
        cout << result << expression() << endl;
    }
}
```

Recover from Errors

Move code that handles exceptions from which we can recover from `error()` to `calculate()`

```
int main()        // step 2
try {
        calculate();
        keep_window_open();      // cope with Windows console mode
        return 0;
}
catch (...) {                    // other errors (don't try to recover)
        cerr << "exception \n";
        keep_window_open("~~");
        return 2;
}
```

Recover from Errors

```
void calculate()
{
    while (cin) try {
        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);
        cout << result << expression() << endl;
    }
    catch (exception& e) {
        cerr << e.what() << 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 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 {
    public:
        Token get();              // get a Token
        void putback(Token t);    // put back a Token
        void ignore(char c);      // discard tokens up to and including a c
    private:
        bool full {false};                // is there a Token in the buffer?
        Token buffer;  // here is where we keep a Token put back using putback()
};
```

Recover from Errors

```
void Token_stream::ignore(char c)
        // skip characters until we find a c; also discard that 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 (cin>>ch)
                if (ch==c) return;
}
```

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

Summary

1. Writing a Program
2. A First Version
3. Improvements