Calculator

Zhengwei QI

Most slides are from Bjarne Stroustrup

www.stroustrup.com/Programming

Recap

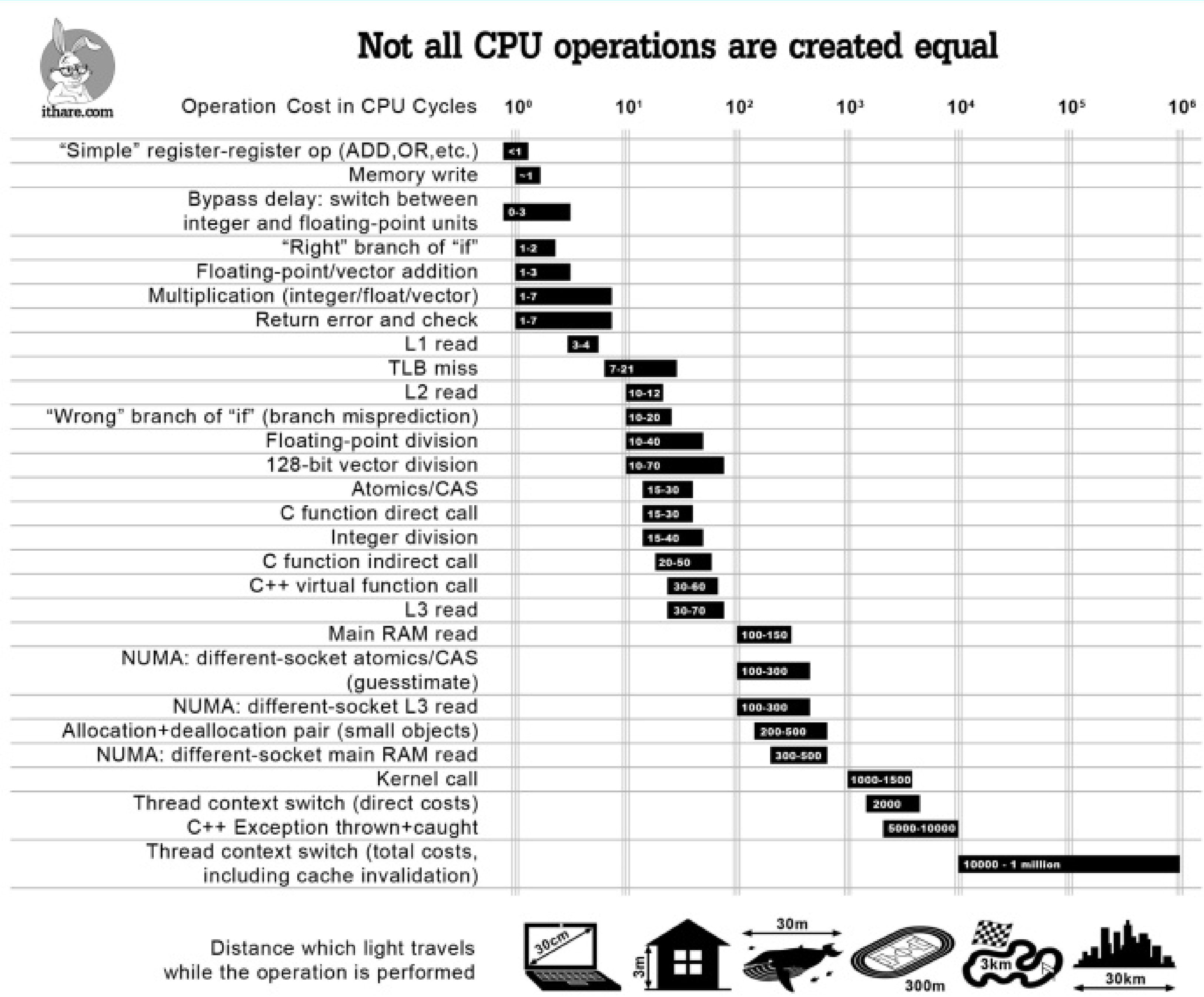
- Report an error by throwing an exception

```
class Bad_area { };  // a class is a user defined type
                     // Bad_area is a type to be used as an exception

int area(int length, int width)
{
   if (length<=0 || width<=0) throw Bad_area{};   // note the {} – a value
   return length*width;
}
```

- Catch and deal with the error (e.g., in **main()**)

```
try {
   int z = area(x,y);        //  if area() doesn't throw an exception
}                            // make the assignment and proceed
catch(Bad_area) {            // if area() throws Bad_area{}, respond
   cerr << "oops! Bad area calculation – fix program\n";
}
```



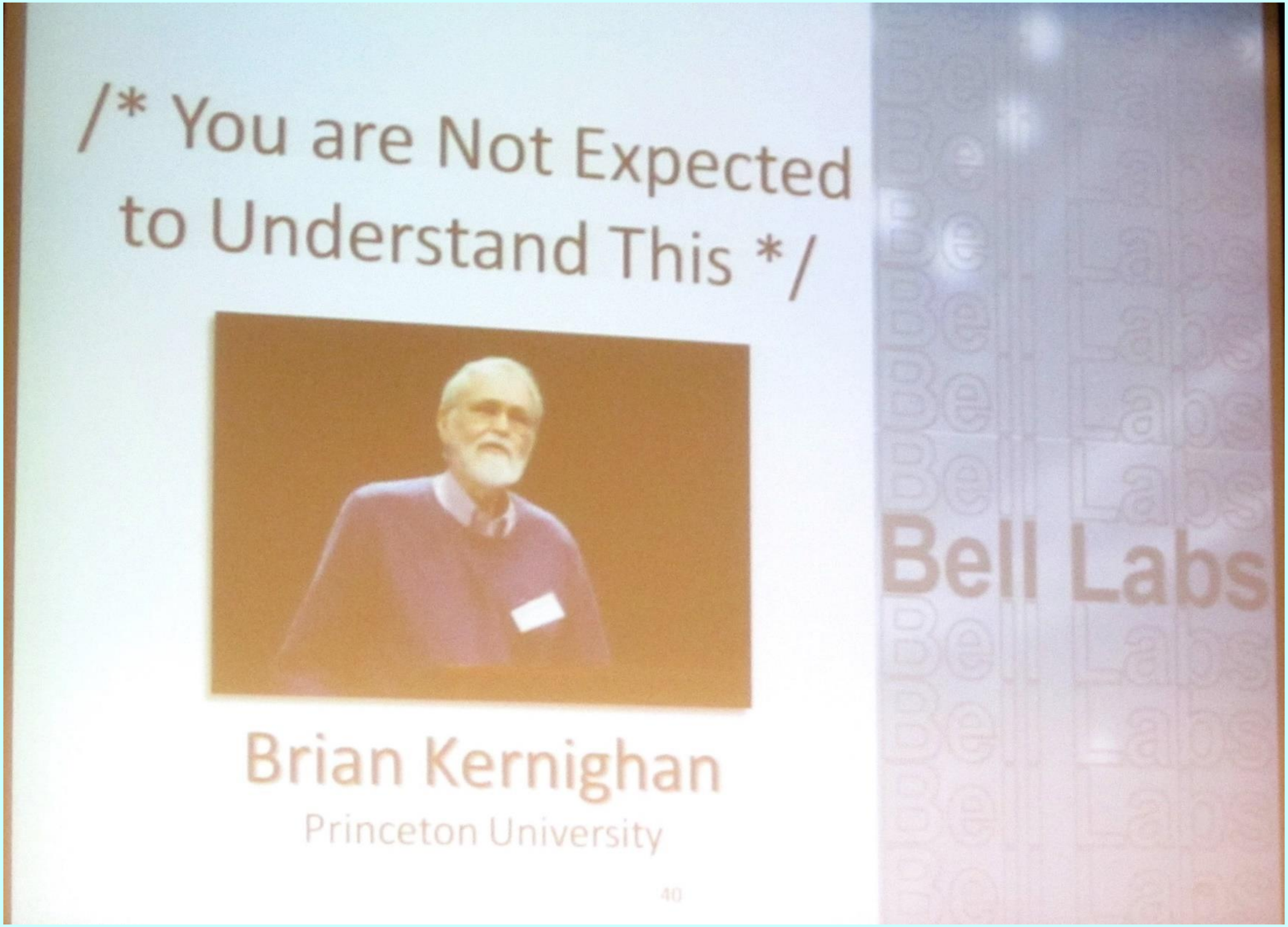
Not all CPU operations are created equal
ithare.com
Operation Cost in CPU Cycles
10^0 10^1 10^2 10^3 10^4 10^5 10^6
"Simple" register-register op (ADD,OR,etc.) <1
Memory write ~1
Bypass delay: switch between integer and floating-point units 0-3
"Right" branch of "if" 1-2
Floating-point/vector addition 1-3
Multiplication (integer/float/vector) 1-7
Return error and check 1-7
L1 read 3-4
TLB miss 7-21
L2 read 10-12
"Wrong" branch of "if" (branch misprediction) 10-20
Floating-point division 10-40
128-bit vector division 10-70
Atomics/CAS 15-30
C function direct call 15-30
Integer division 15-40
C function indirect call 20-50
C++ virtual function call 30-60
L3 read 30-70
Main RAM read 100-150
NUMA: different-socket atomics/CAS (guesstimate) 100-300
NUMA: different-socket L3 read 100-300
Allocation+deallocation pair (small objects) 200-500
NUMA: different-socket main RAM read 300-500
Kernel call 1000-1500
Thread context switch (direct costs) 2000
C++ Exception thrown+caught 5000-10000
Thread context switch (total costs, including cache invalidation) 10000 - 1 million
Distance which light travels while the operation is performed
30cm
3m
30m
300m
3km
30km

Overview

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

Overview-Cont.

- 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



A famous quote in the original Unix source code. You can't understand everything – certainly not at first and in detail. BWK giving a memorial lecture for Dennis Ritchie, the designer of the C programming language.

Building 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

Writing a program: Strategy

- What is the problem to be solved?
  - Is the problem statement clear?
  - Is the problem manageable, given the time, skills, and tools available?
- Try breaking it into manageable parts
  - Do we know of any tools, libraries, etc. that might help?
    - Yes, even this early: **iostream**s, **vector**, etc.
- Build a small, limited version solving a key part of the problem
  - To bring out problems in our understanding, ideas, or tools
  - Possibly change the details of the problem statement to make it manageable
- If that doesn't work
  - Throw away the first version and make another limited version
  - Keep doing that until we find a version that we're happy with
- Build a full scale solution
  - Ideally by using part of your initial version

Programming is also a practical still

- 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"

Writing a program: Example

- I'll build a program in stages, making lot of "typical mistakes" along the way
  - Even experienced programmers make mistakes
    - Lots of mistakes; it's a necessary part of learning
  - 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
  - Building 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 …

Pseudo Code

- A first idea:

```
int main()
{
   variables                       // pseudo code
   while (get a line) {            // what's 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!
  - We are just about to reinvent the wheel!
  - Read Chapter 6 for more examples of dead-end approaches
- What would the experts do?
  - 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
    - "Don't re-invent the wheel"

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).

A side trip: 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"

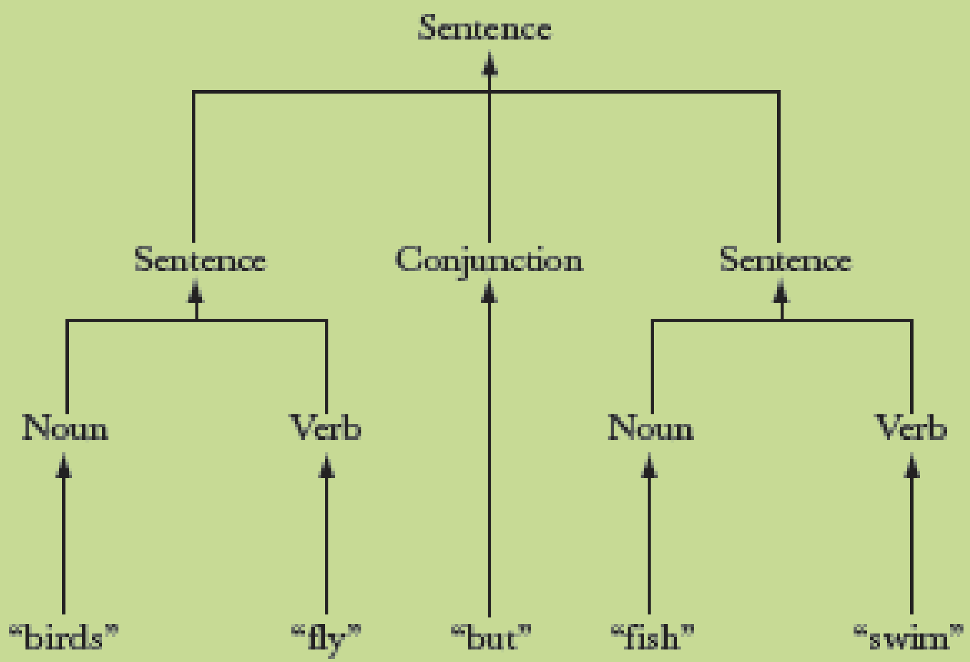
Parsing a simple English sentence

Sentence :
    Noun Verb
    Sentence Conjunction Sentence

Conjunction :
    "and"
    "or"
    "but"

Noun :
    "birds"
    "fish"
    "C++"

Verb :
    "rules"
    "fly"
    "swim"



Grammars - expression

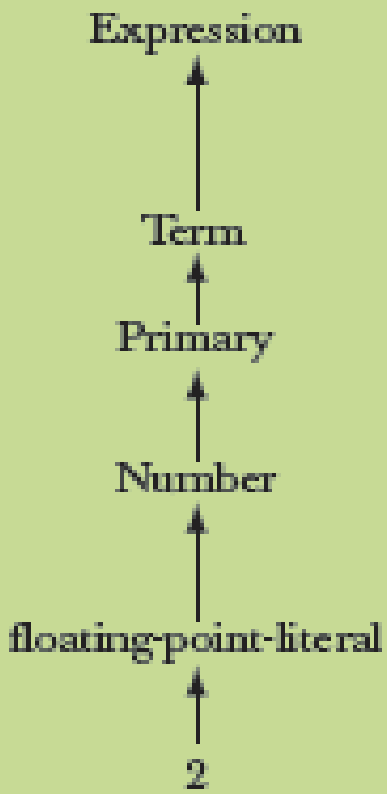
Parsing the number 2

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

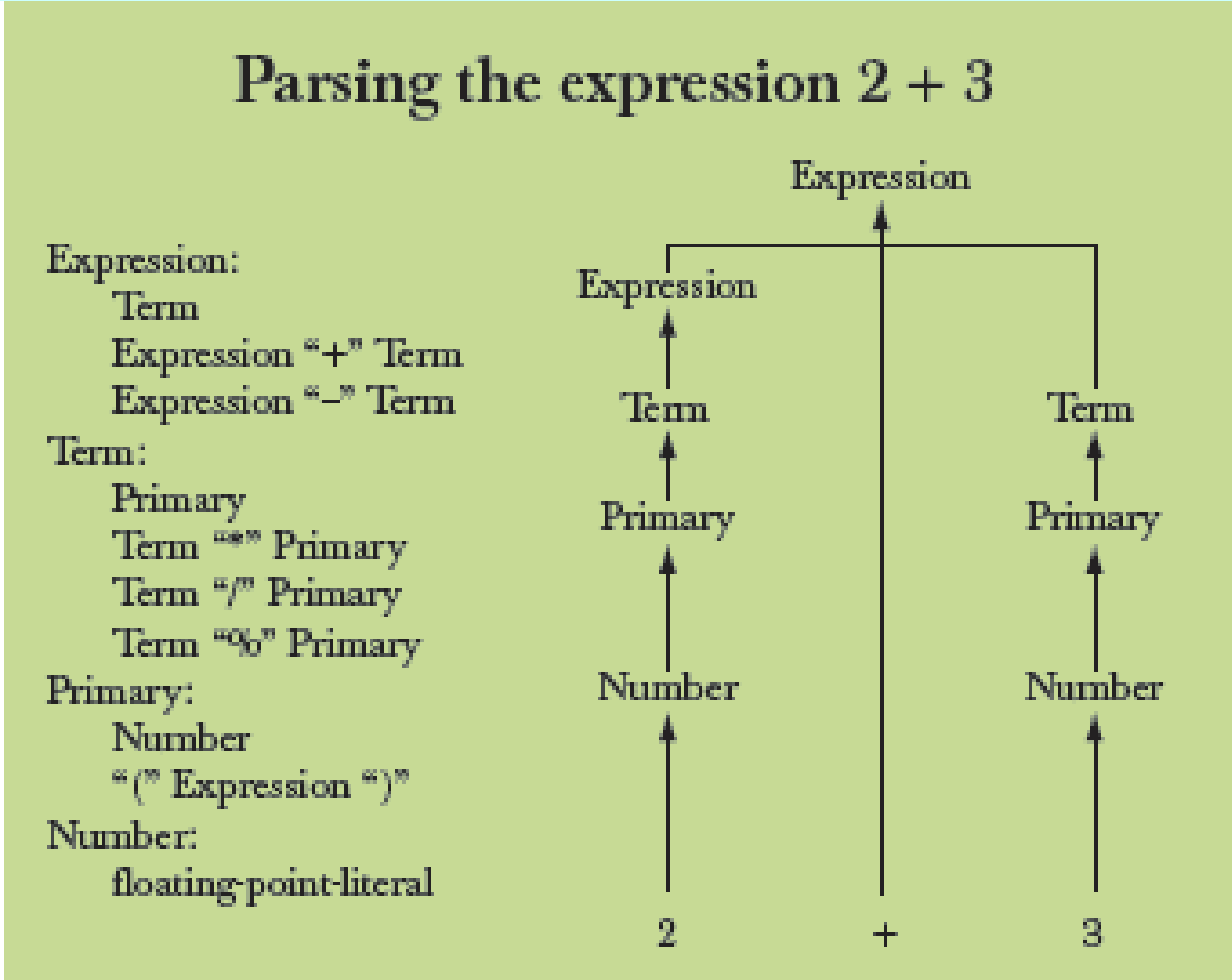
Term:
- Primary
- Term "*" Primary
- Term "/" Primary
- Term "%" Primary

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

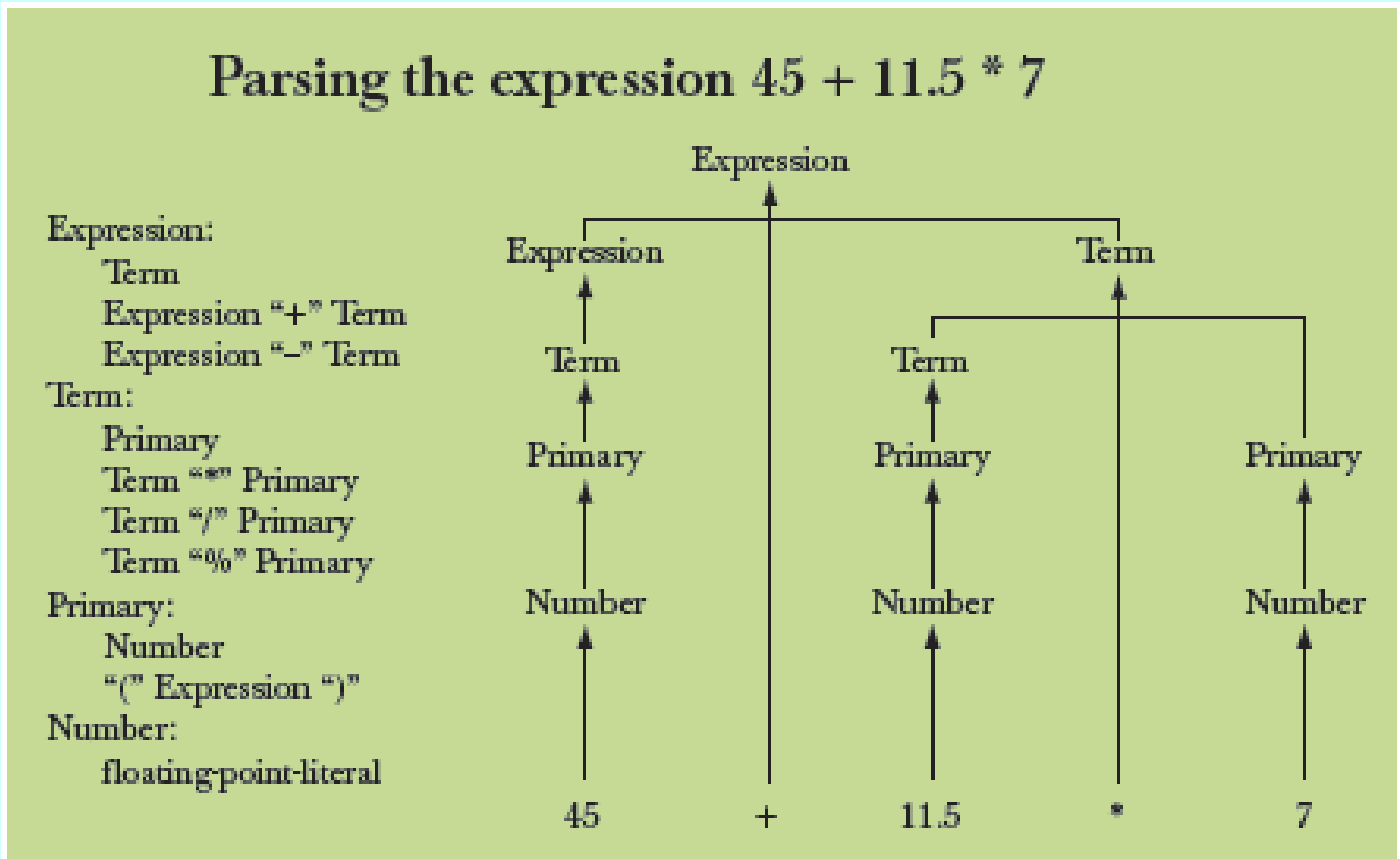
Number:
- floating-point-literal



Grammars - expression



Grammars - expression



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.

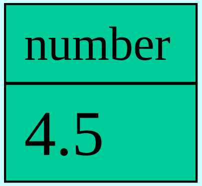
Analogy: a group of people can deal with a complex problem by each person handling only problems in his/her own specialty, leaving the rest for colleagues.

Function Return Types

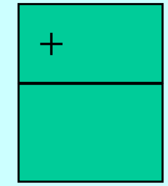
- What should the parser functions return?
  - How about the result?

```
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'll build that in the next lecture, but 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 %

```
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
        }
    }
}
```

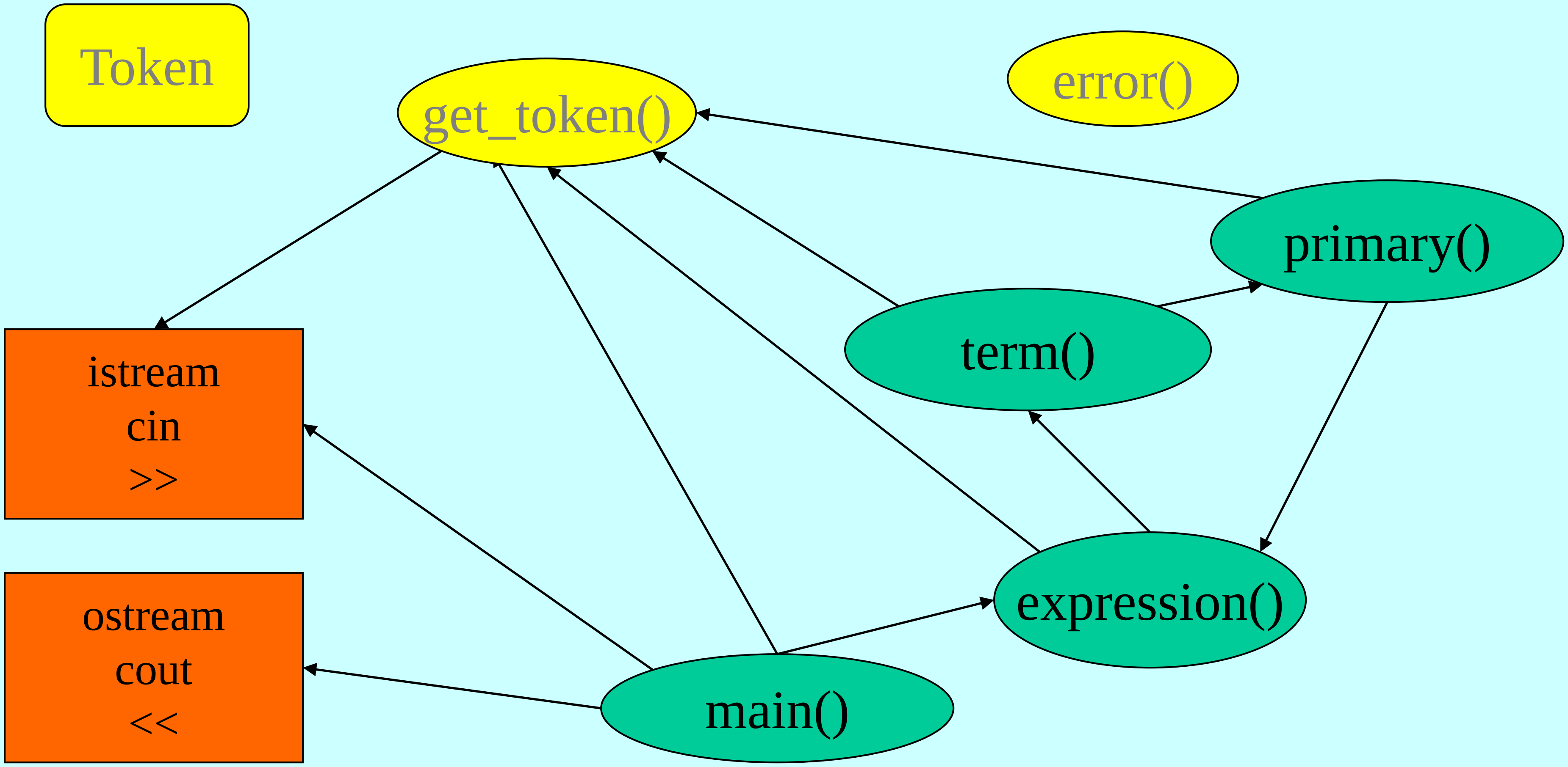
Dealing with divide by 0

```
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
        }
    }
}
```

Dealing with numbers and parentheses

```
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;
}
```

A mystery

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

A mystery

- 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
  - And don't expect your second try to work either

A mystery

- 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;              // <<< doesn't 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()       // deal with + and -
{
    double left = term();
    while (true) {
        Token t = ts.get();            // get the next token from a "token
    stream"
        switch (t.kind) {
        case '+':              left += term(); break;
        case '-':     left -= term(); break;
        default:      ts.putback(t);  // put the unused token back
                      return left;
        }
    }
}
```

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

- It "sort of works"
  - That' s not bad for a first try
    - Well, second try
    - Well, really, the fourth try; see the book
  - But "sort of works" is not good enough
  - When the program "sort of works" is when the work (and fun) really start
- Now we can get feedback!

Another mystery

- 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
  - While we're at it, we also introduce a "quit" command

The main() program

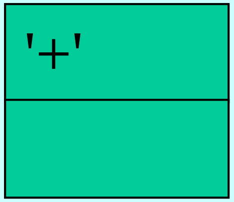
```
int main()
{
    double val = 0;
    while (cin) {
        Token t = ts.get();          // rather than get_token()
        if (t.kind == 'q') break;          // 'q' for "quit"
        if (t.kind == ';')                 // ';' for "print now"
            cout <<  val << '\n';    // print result
 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

Now the calculator is minimally useful

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

Completing the calculator

- Now wee need to
  - Complete the implementation
    - Token and Token_stream
  - Get the calculator to work better
  - Add features based on experience
  - Clean up the code
    - After many changes code often become a bit of a mess
    - We want to produce maintainable code



Token

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

```
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 **struct**s) can have
  - data members (to hold information), and
  - function members (providing operations on the data)

Token_stream

- A **Token_stream** reads characters, producing **Token**s 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

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 {
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
```

Token_stream implementation

```
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()

};

void Token_stream::putback(Token t)
{
    if (full) 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) { full=false; return buffer; }  // check if we already have a Token ready

    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

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;
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"

```
// 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 stream
      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
```

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;
    }
```

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 (see Chapter 10).
- 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
  - Every 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 (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 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 {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

Features

- We did not (yet) add
  - Negative numbers
  - % (remainder/modulo)
  - Pre-defined symbolic values
  - Variables
- Read about that in Chapter 7
  - % and variables demonstrate useful techniques
- 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

Readings

- PPP: Chapter 6, 7

Next

- PPP Chapter 8: Functions
  - we'll take a more systematic look at the language features we have used so far. In particular, we need to know more about classes, functions, statements, expressions, and types