CS241 Principles and Practice of Problem Solving

Lecture 6: Completing a program

Yuye Ling, Ph.D.

Shanghai Jiao Tong University
John Hopcroft Center for Computer Science

September 26, 2019

Copyright Notice

A large portion of the contents in the following pages come from Prof. Bjarne Stroustrup's slides.
The original contents could be found here.

A revisit to the "simple" calculator

We think we are writing a calculator.

A revisit to the "simple" calculator

We think we are writing a calculator. But we are writing a compiler indeed.

A revisit to the "simple" calculator

We think we are writing a calculator. But we are writing a compiler indeed.

Potential steps

- Read in expressions: try to separate words, and put them into Tokens (lexical analysis)

A revisit to the "simple" calculator

We think we are writing a calculator. But we are writing a compiler indeed.

Potential steps

- Read in expressions: try to separate words, and put them into Tokens (lexical analysis)
- Organize the tokens: into a parse-tree (syntax analysis)

A revisit to the "simple" calculator

We think we are writing a calculator. But we are writing a compiler indeed.

Potential steps

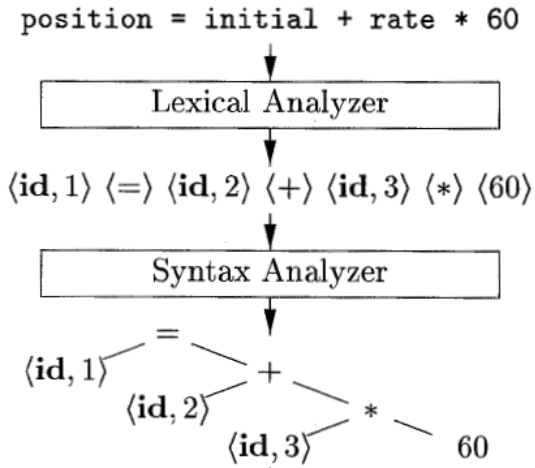
- Read in expressions: try to separate words, and put them into Tokens (lexical analysis)
- Organize the tokens: into a parse-tree (syntax analysis)
- Execute the operations: implement the specific arithmetic operations (semantic analysis)

A revisit to the "simple" calculator

We think we are writing a calculator. But we are writing a compiler indeed.

Potential steps

- Read in expressions: try to separate words, and put them into Tokens (lexical analysis)
- Organize the tokens: into a parse-tree (syntax analysis)
- Execute the operations: implement the specific arithmetic operations (semantic analysis)



Write a compiler (calculator)
- First attempt
- Debugging
- Improve the codes

Implement the Token and Token stream class
- Token
- Token stream

Improve the codes

Write a compiler (calculator)
- First attempt
- Debugging
- Improve the codes

Implement the Token and Token stream class
- Token
- Token stream

Improve the codes

Grammar: how can we realize it

We need functions to match the grammar rules

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

Grammar: how can we realize it

We need functions to match the grammar rules

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

- get(): read characters and compose tokens

Grammar: how can we realize it

We need functions to match the grammar rules

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

- get(): read characters and compose tokens
- expression(): deal with + and -

Grammar: how can we realize it

We need functions to match the grammar rules

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

- get(): read characters and compose tokens
- expression(): deal with + and -
- term(): deal with *, /, and %

Grammar: how can we realize it

We need functions to match the grammar rules

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

- get(): read characters and compose tokens
- expression(): deal with + and -
- term(): deal with *, /, and %
- primary(): deal with numbers and parentheses

Grammar: how can we realize it

We need functions to match the grammar rules

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

- get(): read characters and compose tokens, calls cin for input
- expression(): deal with + and -
- term(): deal with *, /, and %
- primary(): deal with numbers and parentheses

Grammar: how can we realize it

We need functions to match the grammar rules

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

- get(): read characters and compose tokens, calls cin for input
- expression(): deal with + and -, calls term() and get()
- term(): deal with *, /, and %
- primary(): deal with numbers and parentheses

Grammar: how can we realize it

We need functions to match the grammar rules

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

- 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

Grammar: how can we realize it

We need functions to match the grammar rules

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

- 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()

Grammar: how can we realize it

We need functions to match the grammar rules

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

- 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()

This is called *recursive descent parsing*

Determine the return types

What should the parser functions return?

Determine the return types

What should the parser functions return?

```
1 Token get_token();
2 double expression();    //  return the sum (or difference)
3 double term();          //  return the product (or )
4 double primary(); // return the value
5
```

`get_token` should return a Token

Dealing with + and -

```
Expression:
    Term
    Expression '+' Term
    Expression '-' Term
```

Dealing with + and -

```
Expression:
    Term
    Expression '+' Term
    Expression '-' Term
```

Implementation

```
1  double expression() // read and evaluate: 1   1+2.5   1+2+3.14   etc.
2  {
3      double left = term();  // get the Term
4      while (true) {
5          Token t = get_token(); // get the next token
6          switch (t.kind) { //  and do the right thing with it
7              case '+':      left += term(); break;
8              case '-':      left -= term(); break;
9              default:      return left; // return the value of the expression
10         }
11     }
12 }
13
```

Dealing with *, /, and %

```
1  double term() // exactly like expression(), but for *, /, and %
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind)
7          {
8              case '*':    left *= primary(); break;
9              case '/':    left /= primary(); break;
10             case '%':    left %= primary(); break;
11             default:     return left; // return the value
12         }
13     }
14 }
15
```

Dealing with *, /, and %

```
1  double term() // exactly like expression(), but for *, /, and %
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind)
7          {
8              case '*':    left *= primary(); break;
9              case '/':    left /= primary(); break;
10             case '%':    left %= primary(); break;
11             default:     return left; // return the value
12         }
13     }
14 }
15
```

Will not compile.

Dealing with * and /

```
Term:
    Primary
    Term '*' Primary
    Term '\' Primary
```

Implementation

```
1  double term() // exactly like expression(), but for *, and /
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind) {
7              case '*':      left *= primary(); break;
8              case '/':      left /= primary(); break;
9              default:     return left; // return the value
10         }
11     }
12 }
13
```

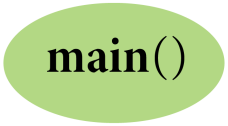
Dealing with divide by 0

```
1  double term() // exactly like expression(), but for * and  /
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind) {
7          case '*':
8              left *= primary();
9              break;
10         case '/':
11             {
12                 double d = primary();
13                 if  (d==0) error("divide by zero");
14                 left /= d;
15                 break;
16             }
17         default:
18             return left; // return the value
19         }
20     }
21 }
22
```

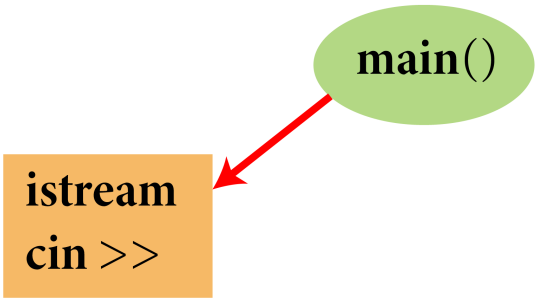
Dealing with numbers and parentheses

```
1  double primary() // Number or ( Expression )
2  {
3      Token t = get_token();
4      switch (t.kind) {
5      case '(':          // handle (expression )
6          {
7              double d = expression();
8              t = get_token();
9              if (t.kind != ')') error("')' expected");
10             return d;
11     }
12     case '8': // we use 8 to represent the kind of a number
13         return t.value; // return the numbers value
14     default:
15         error("primary expected");
16     }
17 }
18
```

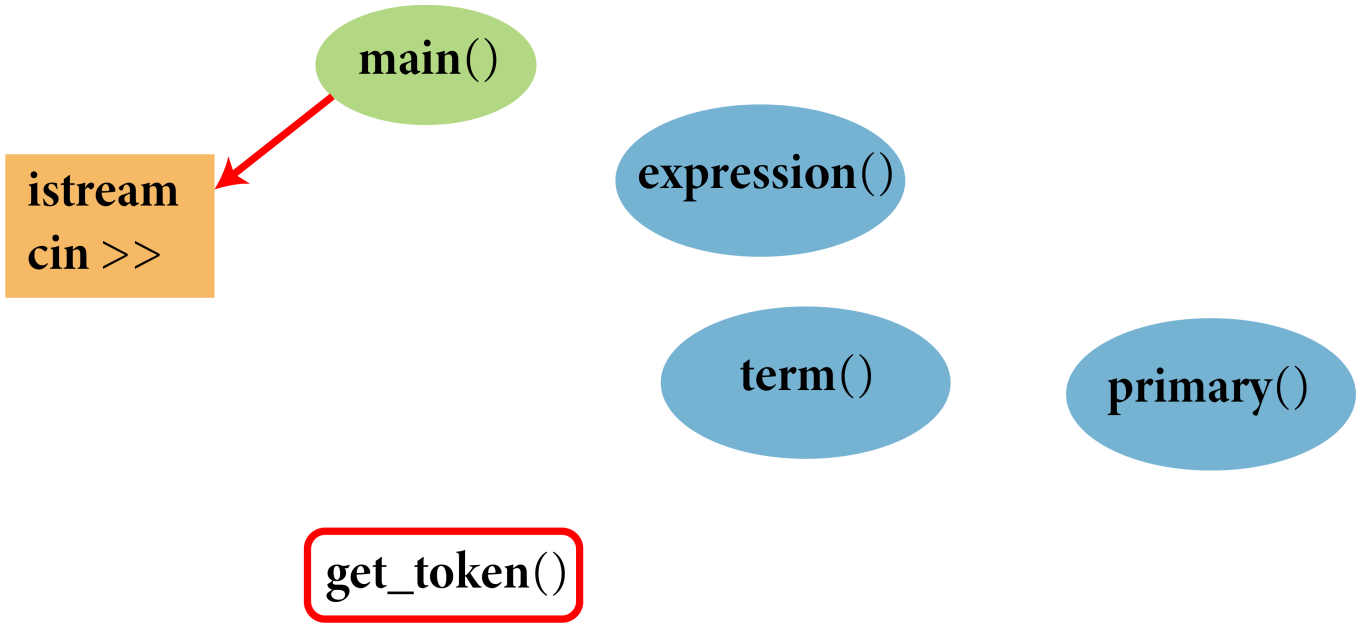
The structure of the program



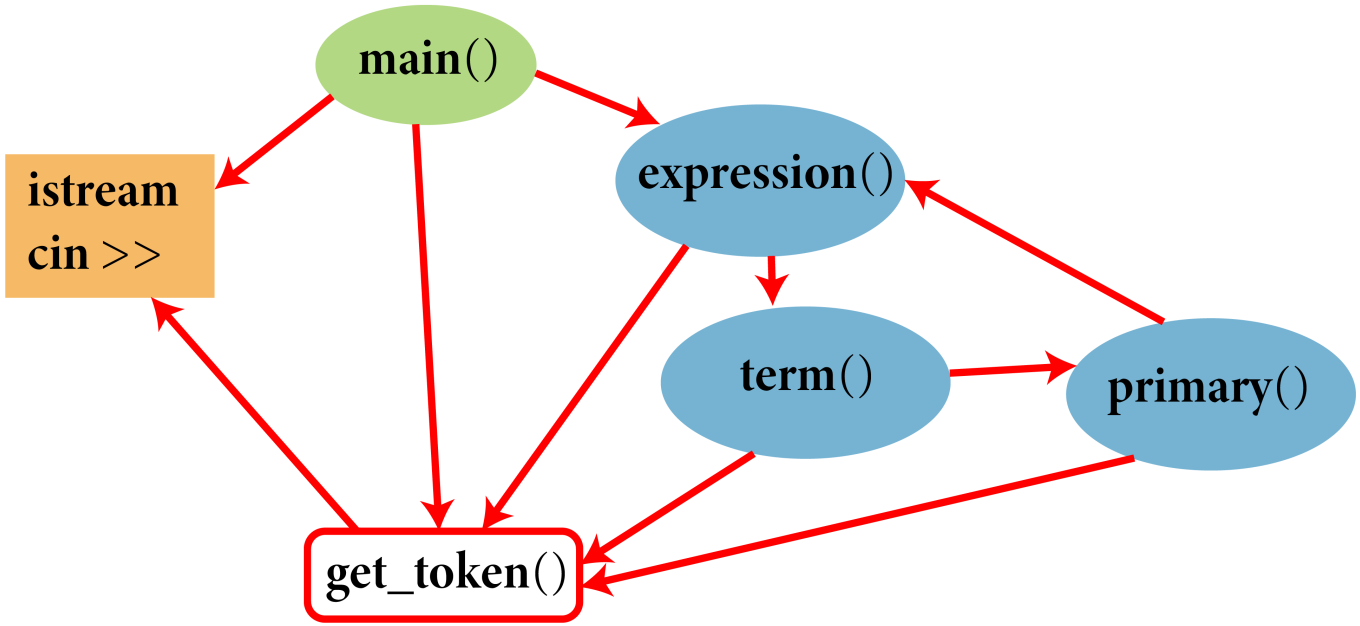
The structure of the program



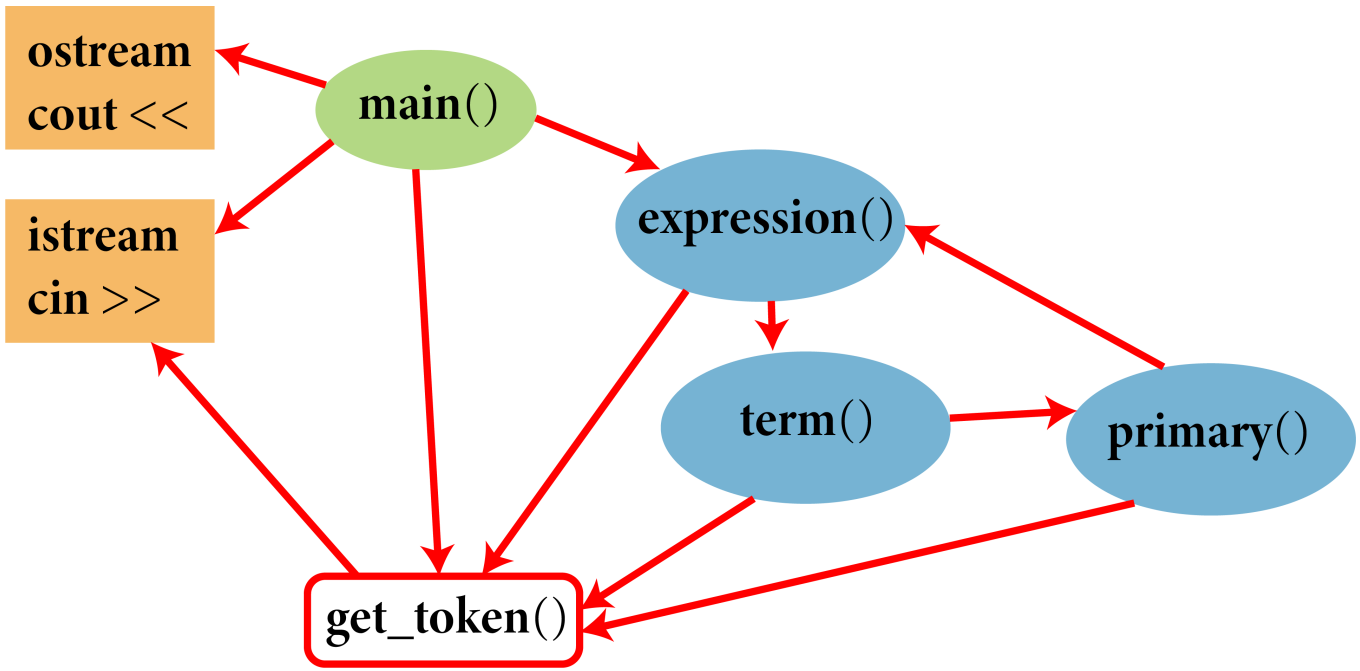
The structure of the program



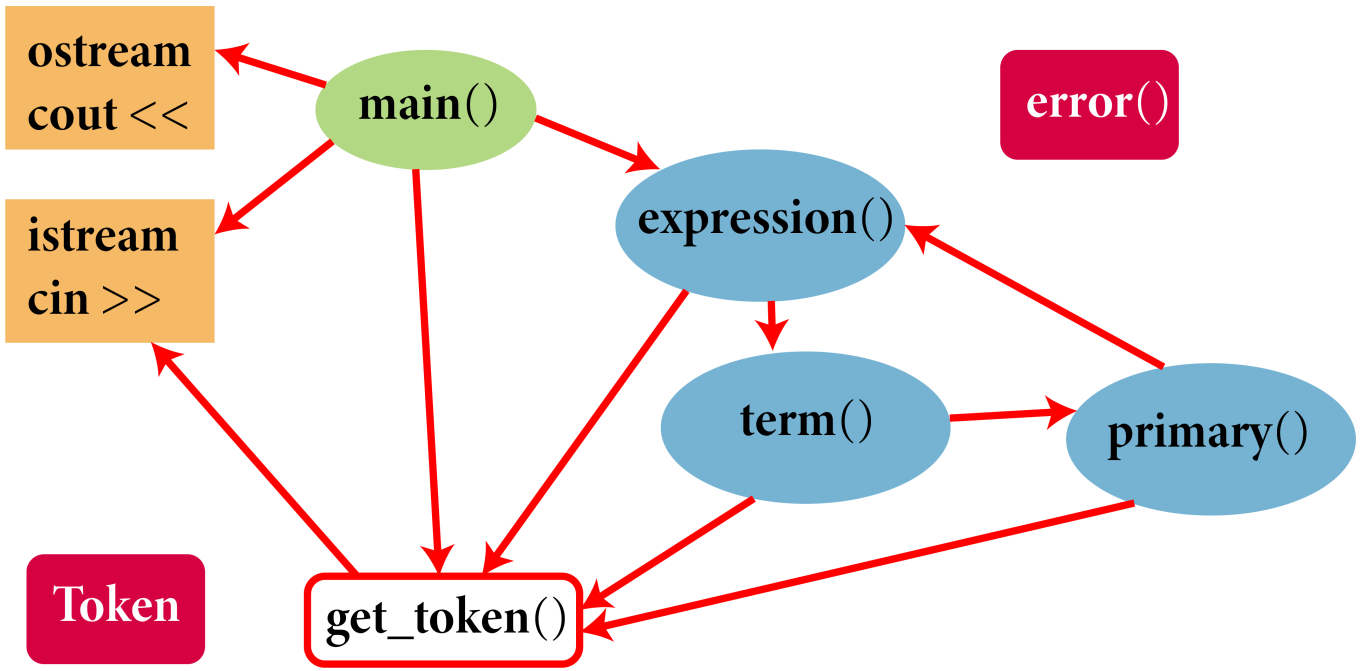
The structure of the program



The structure of the program



The structure of the program



Forward declaration

Recall the loop between `expression()`, `term()`, and `primary()`

Forward declaration

Recall the loop between `expression()`, `term()`, and `primary()`

```
1 double expression(); // declaration so that primary() can call  expression()
2
3 double primary() { /*  */ } // deal with numbers and parentheses
4 double term() { /*  */ } // deal with * and / (pity about %)
5 double expression() { /*  */ } // deal with + and
6
7 int main() { /*  */ } // on next slide
8
```

How does the main function look like

```
1  int main()
2  try {
3      while (cin)
4          cout << expression() << '\n';
5  }
6  catch (runtime_error& e) {
7      cerr << e.what() << endl;
8      return 1;
9  }
10 catch () {
11     cerr << "exception \n";
12     return 2;
13 }
14
```

How does the main function look like

```
1  int main()
2  try {
3      while (cin)
4          cout << expression() << '\n';
5  }
6  catch (runtime_error& e) {
7      cerr << e.what() << endl;
8      return 1;
9  }
10 catch () {
11     cerr << "exception \n";
12     return 2;
13 }
14
```

Now, everything is ready. Do you think the program will work?

A mystery

Our program eats two out of three inputs

A mystery

Our program eats two out of three inputs

Why is that?

Debugging I

Let's start with the case when we are only inputing "1"

Debugging I

Let's start with the case when we are only inputing "1"

1. `main()` calls `expression()`

```
1 int main()
2 try {
3     while (cin)
4         cout << expression() << '\n';
5 }
6
```

Debugging I

Let's start with the case when we are only inputing "1"

1. `main()` calls `expression()`

```
1  int main()
2  try {
3      while (cin)
4          cout << expression() << '\n';
5  }
6
```

2. Evaluate `expression()`: `expression()` calls `term()`

```
1  double expression() // read and evaluate: 1   1+2.5   1+2+3.14   etc.
2  {
3      double left = term();  // get the Term
4      while (true) {
5          Token t = get_token(); // get the next token
6          switch (t.kind) { //  and do the right thing with it
7              case '+':      left += term(); break;
8              case '-':      left -= term(); break;
9              default:      return left; // return the value of the
       expression
10         }
11     }
12 }
13
```

Debugging II

3. Evaluate `term()`: `term()` calls `primary()`

```
1  double term() // exactly like expression(), but for * and  /
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind) {
7          case '*':
8              left *= primary();
9              break;
10         case '/':
11             {
12                 double d = primary();
13                 if  (d==0) error("divide by zero");
14                 left /= d;
15                 break;
16             }
17         default:
18             return left; // return the value
19         }
20     }
21 }
22
```

Debugging III

4. Evaluate `primary()`: Finally! "1" is retrieved by `get_token()`

```
1  double primary() // Number or ( Expression )
2  {
3      Token t = get_token();
4      switch (t.kind) {
5      case '(':          // handle (expression )
6          {
7              double d = expression();
8              t = get_token();
9              if (t.kind != ')') error("')' expected");
10             return d;
11     }
12     case '8': // we use 8 to represent the kind of a number
13         return t.value; // return the numbers value
14     default:
15         error("primary expected");
16     }
17 }
18
```

Debugging III

4. Evaluate `primary()`: Finally! "1" is retrieved by `get_token()`

```
1  double primary() // Number or ( Expression )
2  {
3      Token t = get_token();
4      switch (t.kind) {
5      case '(':          // handle (expression )
6          {
7              double d = expression();
8              t = get_token();
9              if (t.kind != ')') error("')' expected");
10             return d;
11     }
12     case '8': // we use 8 to represent the kind of a number
13         return t.value; // return the numbers value
14     default:
15         error("primary expected");
16     }
17 }
18
```

What's next?

Debugging III

4. Evaluate `primary()`: Finally! "1" is retrieved by `get_token()`

```
1  double primary() // Number or ( Expression )
2  {
3      Token t = get_token();
4      switch (t.kind) {
5      case '(':          // handle (expression )
6          {
7              double d = expression();
8              t = get_token();
9              if (t.kind != ')') error("')' expected");
10             return d;
11     }
12     case '8': // we use 8 to represent the kind of a number
13         return t.value; // return the numbers value
14     default:
15         error("primary expected");
16     }
17 }
18
```

What's next?

This is a recursive process. We have to go all the way back up.

Debugging IV

5. We go back to `term()`

```
1  double term() // exactly like expression(), but for * and  /
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind) {
7          case '*':
8              left *= primary();
9              break;
10         case '/':
11             {
12                 double d = primary();
13                 if  (d==0) error("divide by zero");
14                 left /= d;
15                 break;
16             }
17         default:
18             return left; // return the value
19         }
20     }
21 }
22
```

Debugging IV

5. We go back to `term()`

```
1  double term() // exactly like expression(), but for * and  /
2  {
3      double left = primary();  // get the Primary
4      while (true) {
5          Token t = get_token(); // get the next Token
6          switch (t.kind) {
7          case '*':
8              left *= primary();
9              break;
10         case '/':
11             {
12                 double d = primary();
13                 if  (d==0) error("divide by zero");
14                 left /= d;
15                 break;
16             }
17         default:
18             return left; // return the value
19         }
20     }
21 }
22
```

Line 5.

Token stream

The token should be put back if it is not used!

Token stream

The token should be put back if it is not used!

Where to put back?

Token stream

The token should be put back if it is not used!

Where to put back?

`token_stream`.

Token stream

The token should be put back if it is not used!

Where to put back?

`token_stream`. We will discuss this later.

```
1  double expression() // deal with + and -
2  {
3      double left = term();
4      while (true) {
5          Token t = ts.get();        // get the next token from a token stream
6          switch (t.kind) {
7          case '+':      left += term(); break;
8          case '-':      left -= term(); break;
9          default:      ts.putback(t);  // put the unused token back
10             return left;
11         }
12     }
13 }
14
```

Token stream

The token should be put back if it is not used!

Where to put back?

`token_stream`. We will discuss this later.

```
1  double expression() // deal with + and -
2  {
3      double left = term();
4      while (true) {
5          Token t = ts.get();       // get the next token from a token stream
6          switch (t.kind) {
7          case '+':      left += term(); break;
8          case '-':      left -= term(); break;
9          default:      ts.putback(t);  // put the unused token back
10             return left;
11         }
12     }
13 }
14
```

We have to fix `term()` as well.

Token stream

The token should be put back if it is not used!

Where to put back?

`token_stream`. We will discuss this later.

```
1  double expression() // deal with + and -
2  {
3      double left = term();
4      while (true) {
5          Token t = ts.get();       // get the next token from a token stream
6          switch (t.kind) {
7          case '+':      left += term(); break;
8          case '-':      left -= term(); break;
9          default:      ts.putback(t);  // put the unused token back
10             return left;
11         }
12     }
13 }
14
```

We have to fix `term()` as well.

Question: what about `primary()`?

Adding prompts and output indicators

Let's assume all the details about Token has been taken care of. Time for making up.

Adding prompts and output indicators

Let's assume all the details about Token has been taken care of. Time for making up.

```
1  double val = 0;
2  cout << "> "; // print prompt
3  while (cin) {
4      Token t = ts.get();
5      if (t.kind == 'q') break; // check for quit
6      if (t.kind == ';')
7          cout << "= " << val << "\n > "; // print = result and prompt
8      else
9          ts.putback(t);
10     val = expression(); // read and evaluate expression
11 }
12
```

Adding prompts and output indicators

Let's assume all the details about Token has been taken care of. Time for making up.

```
1  double val = 0;
2  cout << "> "; // print prompt
3  while (cin) {
4      Token t = ts.get();
5      if (t.kind == 'q') break; // check for quit
6      if (t.kind == ';')
7          cout << "= " << val << "\n > "; // print = result and prompt
8      else
9          ts.putback(t);
10     val = expression(); // read and evaluate expression
11 }
12
```

Using "q" for quit; using ";" for the end of expression

Remove "magic constants"

During the coding, we have used several "magic numbers" as indicators

Remove "magic constants"

During the coding, we have used several "magic numbers" as indicators

Need to encapsulate them to increase the reliability of the codes.

```
1 // Token kind values:
2 const char number = '8'; // a floating-point number
3 const char quit = 'q'; // an exit command
4 const char print = ';'; // a print command
5
6 // User interaction strings:
7 const string prompt = "> ";
8 const string result = "= "; // indicate that a result follows
9
```

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12, 24

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12, 24, 2.71828

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12, 24, 2.71828, 299792458

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12, 24, 2.71828, 299792458, 2.54

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12, 24, 2.71828, 299792458, 2.54, 2.2046

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12, 24, 2.71828, 299792458, 2.54, 2.2046
2. It might cause a space probe to self destruct. (Mars Polar Lander, NASA, $110 million)

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12, 24, 2.71828, 299792458, 2.54, 2.2046
2. It might cause a space probe to self destruct. (Mars Polar Lander, NASA, $110 million)

LISA GROSSMAN 11.10.10 07:00 AM

Nov. 10, 1999: Metric Math Mistake Muffed Mars Meteorology Mission

A NASA review board found that the problem was in the software controlling the orbiter's thrusters. The software calculated the force the thrusters needed to exert in *pounds* of force. A separate piece of software took in the data assuming it was in the metric unit: *newtons*.

Magic constants

It is extremely important to remove magic constants in your code.

1. Not everyone knows these constants!
   - 3.1415926, 12, 24, 2.71828, 299792458, 2.54, 2.2046
2. It might cause a space probe to self destruct. (Mars Polar Lander, NASA, $110 million)

LISA GROSSMAN 11.10.10 07:00 AM

Nov. 10, 1999: Metric Math Mistake Muffed Mars Meteorology Mission

A NASA review board found that the problem was in the software controlling the orbiter's thrusters. The software calculated the force the thrusters needed to exert in *pounds* of force. A separate piece of software took in the data assuming it was in the metric unit: *newtons*.

3. If a constant is used multiple times in a large project, use a symbol could save you huge amount of time.

Write a compiler (calculator)
- First attempt
- Debugging
- Improve the codes

Implement the Token and Token stream class
- Token
- Token stream

Improve the codes

Token

Recall how we design the Token

Token

Recall how we design the Token
We want a container that holds

1. type
2. value

Token

Recall how we design the Token

We want a container that holds

1. type
2. value

```
1 struct Token { // define a type called Token
2     char kind; // what kind of token
3     double value; // used for numbers (only): a value
4 }; // semicolon is required
5
```

Token

Recall how we design the Token

We want a container that holds

1. type
2. value

```
1 struct Token { // define a type called Token
2     char kind; // what kind of token
3     double value; // used for numbers (only): a value
4 }; // semicolon is required
5
```

A struct is the simplest form of a class

Token

Recall how we design the Token

We want a container that holds

1. type
2. value

```
1 struct Token { // define a type called Token
2     char kind; // what kind of token
3     double value; // used for numbers (only): a value
4 }; // semicolon is required
5
```

A struct is the simplest form of a class

It was there before C++

Token operations

```
1 Token t;
2 t.kind = number; // . (dot) is used to access members
3 t.value = 2.3;
4 Token u = t;  // a Token behaves much like a built-in type, such as int
5 // so u becomes a copy of t
6 cout << u.value; // will print 2.3
7
```

Token stream

What do we want from the `Token_stream`?

Token stream

What do we want from the `Token_stream`?

Design goals

- reads characters, producing Tokens on demand
- put back a Token into a `Token_stream` for later use

Token stream

What do we want from the `Token_stream`?

Design goals

- reads characters, producing Tokens on demand
- put back a Token into a `Token_stream` for later use

So we need a buffer to temporarily hold tokens

Token stream: Example

We use 1+2*3 as an example to illustrate the use of buffer

Token stream: Example

We use 1+2*3 as an example to illustrate the use of buffer

Left	**empty**
Token_stream buffer	**empty**
Input stream	**1+2*3**

Token stream: Example

We use `1+2*3` as an example to illustrate the use of buffer

Left	**empty**
Token_stream buffer	**empty**
Input stream	**1+2*3**

`expression()` calls `term()` which reads 1

Token stream: Example

We use `1+2*3` as an example to illustrate the use of buffer

Left	**empty**
Token_stream buffer	**empty**
Input stream	**1+2*3**

`expression()` calls `term()` which reads `1`

Left	**1**
Token_stream buffer	**empty**
Input stream	**+2*3**

Token stream: Example (Cont'd)

`term()` will keep read + and then realize that it is not its job.

Token stream: Example (Cont'd)

`term()` will keep read + and then realize that it is not its job.

Why?

Token stream: Example (Cont'd)

`term()` will keep read + and then realize that it is not its job.

Why?

So it will will put + back to the buffer

Token stream: Example (Cont'd)

`term()` will keep read + and then realize that it is not its job.

Why?

So it will will put + back to the buffer

Left	**1**
Token_stream buffer	+
Input stream	**2*3**

Token stream: Example (Cont'd)

`term()` will keep read + and then realize that it is not its job.

Why?

So it will will put + back to the buffer

Left	**1**
Token_stream buffer	+
Input stream	**2*3**

The control will now be transferred to `expression()`, it will first check buffer before reading the isream

Token stream: Example (Cont'd)

`term()` will keep read + and then realize that it is not its job.

Why?

So it will will put + back to the buffer

Left	**1**
Token_stream buffer	+
Input stream	2*3

The control will now be transferred to `expression()`, it will first check buffer before reading the isream

Left	**1**
Operator	+
Right	
Token_stream buffer	
Input stream	2*3

Token stream: Interface

Two public functions:

- Obtain a token
- Put back a token (to a buffer)

Token stream: Interface

Two public functions:

- Obtain a token
- Put back a token (to a buffer)

```
1  class Token_stream {
2  public:
3      // user interface:
4      Token get(); // get a Token
5      void putback(Token); // put a Token back into the Token_stream
6  private:
7      // representation: not directly accessible to users:
8      bool full {false}; // is there a Token in the buffer?
9      Token buffer;       // here is where we keep a Token put back using putback
         ()
10 };
11
```

Token stream: Implementation

```
1  Token Token_stream::get()  // read a Token from the Token_stream
2  {
3      if (full) { full=false; return buffer; }  // check if we already have a
        Token ready
4      char ch;
5      cin >> ch; // note that >> skips whitespace (space, newline, tab, etc.)
6      switch (ch) {
7      case '(': case ')': case ';': case 'q': case '+': case '-': case '*': case '
        /':
8          return Token{ch}; // let each character represent itself
9      case '.':
10     case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '
        7': case '8': case '9':
11         { cin.putback(ch); // put digit back into the input stream
12         double val;
13         cin >> val; // read a floating-point number
14         return Token{number ,val};
15         }
16     default:
17         error("Bad token");
18     }
19 }
20
```

Token stream: Implementation

```
1 void Token_stream::putback(Token t)
2 {
3     if (full) error("putback() into a full buffer");
4     buffer=t;
5     full=true;
6 }
7
```

Use functions

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

Use functions

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

```
1  int main() // step 1
2  try {
3      calculate();
4      return 0;
5  }
6  catch (exception& e) { // errors we understand something about
7      cerr << e.what() << endl;
8      return 1;
9  }
10 catch (...) { // other errors
11     cerr << "exception \n";
12     return 2;
13 }
14
```

Put the read and evaluate loop into calculate()

Recover from errors

Currently, all the error handling are performed on the same level as `main()`

Recover from errors

Currently, all the error handling are performed on the same level as `main()`

What's the potential problem for this practice?

Recover from errors

Currently, all the error handling are performed on the same level as `main()`

What's the potential problem for this practice?

Whenever an error is detected, the main program will be terminated.

Recover from errors

Currently, all the error handling are performed on the same level as `main()`

What's the potential problem for this practice?

Whenever an error is detected, the main program will be terminated.

What can we do? What if we want to give the user another chance to correct?

Recover from errors

Currently, all the error handling are performed on the same level as `main()`

What's the potential problem for this practice?

Whenever an error is detected, the main program will be terminated.

What can we do? What if we want to give the user another chance to correct?

We can move the known errors to `calculate()`

Recover from errors (Cont'd)

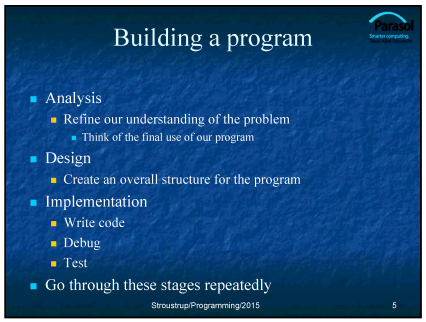
```
1  void calculate()
2  {
3      while (cin) try {
4          cout << prompt;
5          Token t = ts.get();
6          while (t.kind == print) t=ts.get(); // first discard all prints
7          if (t.kind == quit) return; // quit
8          ts.putback(t);
9          cout << result << expression() << endl;
10     }
11     catch (exception& e) {
12         cerr << e.what() << endl; // write error message
13         clean_up_mess(); // <<< The tricky part!
14     }
15 }
16
```

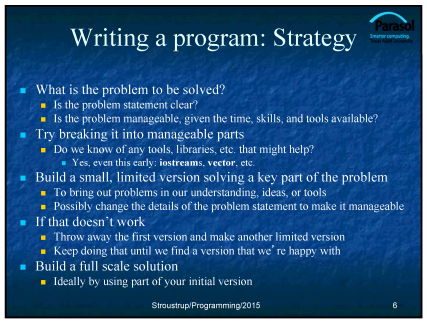
Summary

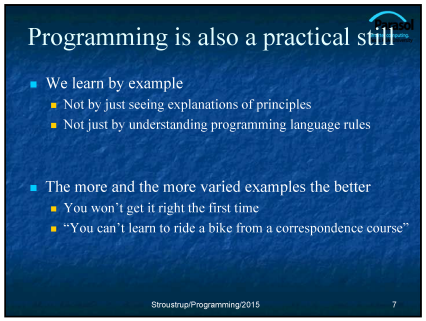
Yay, we end up with a primitive calculator (or compiler)!

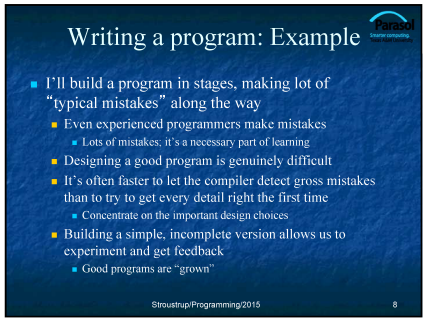
Summary

Yay, we end up with a primitive calculator (or compiler)!
It is about the time to revisit the first couple slides









Building a program

Parasol
Smarter computing.
Texas A&M University

- 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

Stroustrup/Programming/2015 5

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

Stroustrup/Programming/2015 6

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"

Stroustrup/Programming/2015 7

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"

Stroustrup/Programming/2015 8