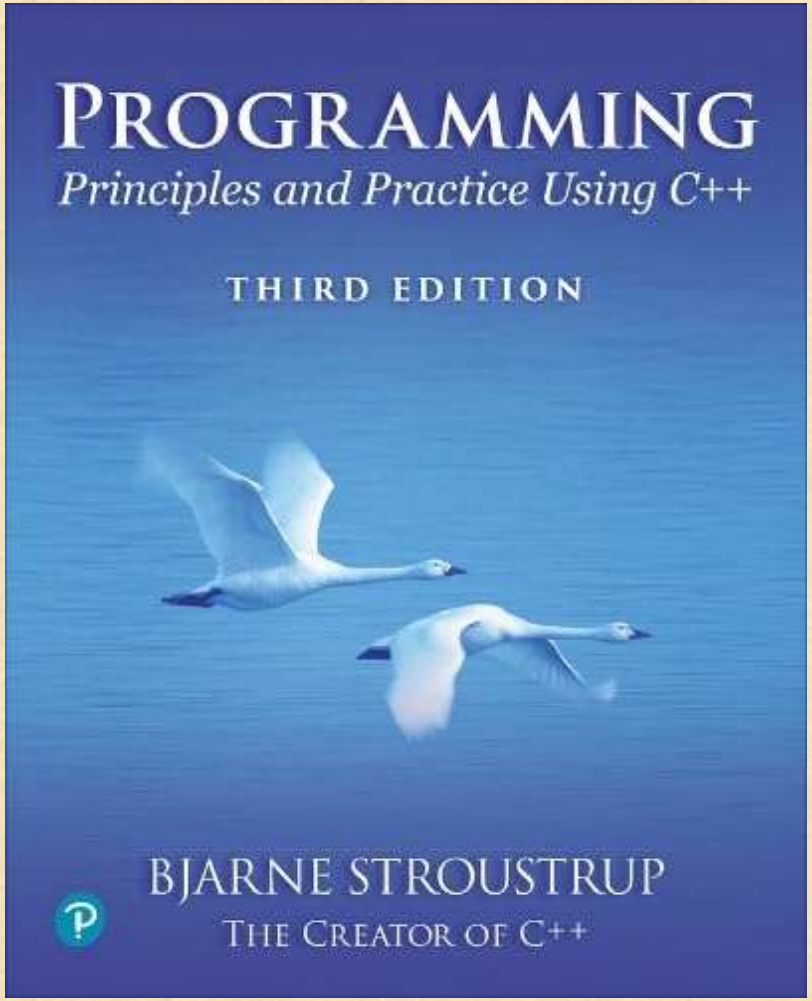
Chapter 6 – Completing a Program



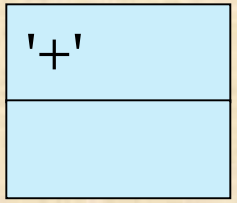
Keep it simple:
as simple as possible,
but no simpler.
– Albert Einstein

Abstract

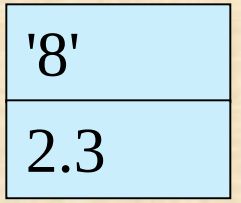
- Tokens and token streams (from Chapter 5)
  - Structs and classes
- Cleaning up the code
  - Prompts
  - Program organization
    - constants
  - Recovering from errors
  - Commenting
  - Code review
  - Testing
- A word on complexity and difficulty
  - Variables

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 = 0;  // a value; zero is not explicitly set
};

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
- **W**e 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 {
  // ….
};

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) {                  // check if we already have a Token ready
       full=false;
       return buffer;
  }

  char ch;
  if (!(cin>>ch))              // note that >> skips whitespace (space, newline, tab, etc.)
       error("no input");

  switch (ch) {                // compose a Token
       // …
  }
}
```

Token_stream implementation

```
Token Token_stream::get()   // read a Token from the Token_stream
{
  // …
  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? How about?

```
> 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; =
5
> = 35
> = 11
>
```

- *Oops! The program doesn't see input before you hit*

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();
  return 0;
}
catch (exception& e) {          // errors we understand something
  about
  cerr << e.what() << '\n';
  return 1;
}
catch (...) {                   // other errors
  cerr << "exception \n";
  return 2;
}
```

Recover from errors

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

```
void calculate()
{
    while (cin) {
        cout << prompt;
        Token t = ts.get();
        while (t.kind == print)        // first discard all "prints"
            t=ts.get();
        if (t.kind == quit)
            return;                    // quit
        ts.putback(t);
        cout << result << expression() << '\n';
    }
}
```

Recover from errors

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

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

Recover from errors

```
void calculate()
{
  while (cin) try {
        // …
  }
  catch (exception& e) {
        cerr << e.what() << '\n';              // 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


  char ch = 0;
  while (cin>>ch)                    // search input for the sentinel character
       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 those in Chapter 6
  - They demonstrate useful programming 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
    - At the very beginning you don’t know
      - What the effort will be
      - What’s worth that effort

Next lecture

- In the next two lectures, 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.