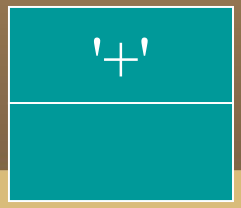
Completing a simple calculator. Structures. Classes. Streams.



Token.

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

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

Token t;              // create an object
t.kind = '8';         // . (dot) is used to access members.
                      // (let's here '8' is a "tag" and means "number")
t.value = 2.3;        // store value

Token u = t;          // Token behaves much like a built-in type (e.g. int)
                      // so u becomes a copy of t
cout << u.value;      // will print 2.3
```

Structure. Class.

```
struct Token {        // user-defined type called Token
    // data members
    // function members
};
```

- A **struct** is the simplest form of a class
- "class" in C++ is a term for "user-defined type"
- Defining types is the crucial mechanism for organizing programs in C++
  - … as in many other modern languages
- a **class** (and also **struct**) can have
  - data members (to hold information), and
  - function members (for operations on the data)

Structure, Class.

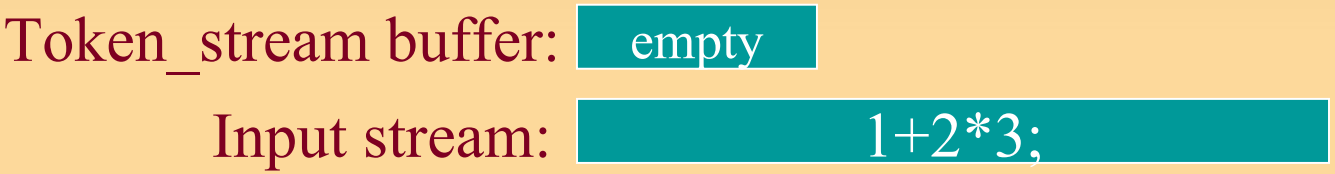
```
struct Token {
    char kind;          // what kind of token
    double value;       // for numbers: a value

    Token(char ch): kind(ch), value(0) { }                  // constructor
    Token(char ch, double val): kind(ch), value(val) { }    // constructor
};
```

- A constructor has the same name as its class and no return type
- A constructor defines how an object of a class is initialized
  - Here **kind** is initialized with **ch,** and
  - **value** is initialized with **val** or **0**
  - **Token t('+');** // make an object **t** of type **Token** with "kind" = **'+'**
  - **Token t('8', 4.5);** // make an object **t of** type **Token** with "kind" = **'8'**
    // and "value" = **4.5**

Token_stream class functionality we need.

- A **Token_stream** class can read characters, producing **Token**s on demand
- **Token** could be put back into a **Token_stream** for later use

A **Token_stream** uses a "buffer" to hold tokens we put back into it



For **1+2*3** input **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 class declaration

```
class Token_stream {

    // representation: not directly accessible to users (private data):
    bool full;          // "is there a Token in the buffer"?
    Token buffer;       // here is where we keep a Token
                        // could be put back using void PutBack(Token t);
public:
    // user interface (functions which could be called by user):
    Token Get();                        // get a Token from the input
    void PutBack(Token t); // put a Token t back into the Token_stream

    Token_stream();             // constructor: make and init a Token_stream
};
```

Token_stream implementation

```
class Token_stream {
    bool full;          // "is there a Token in the buffer"?
    Token buffer;       // here is where we keep a Token
                        // which could be put back by PutBack()
public:
    Token Get();                        // get a Token object (structure)
    void PutBack(Token t);              // put back a Token (declaration)
    Token_stream(): full(false) { }     // default constructor:  buffer starts empty
};

void Token_stream::PutBack(Token t) // method implementation (outside class)
{
    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; } // some token was already in the buffer. Return it.

    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);       // parentheses, ";", "q", operators are returned "as is"
     case '0': case '1': case '2': case '3': case '4': case '5': case '6':
            case '7': case '8': case '9':  // numbers handling
      {
            cin.putback(ch);        // put digit back into the input stream for later use
                                    // here "putback" is method of cin stream class from <iostream>
            double val;
            cin >> val;             // re-read what was "putback" as a floating-point number
            return Token('8',val); // '8' is just a tag that token has a number in "val" data member
      }
     default:
            error("Bad token");
    }   // end of "switch" block
}
```

Streams

- Note that the concept of a stream of data is extremely general and very widely used
  - *Most I/O systems* (e.g., C++ standard I/O streams)

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

- This is what we implemented

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

- What do we really want? Something better looking:

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

Adding prompts and output indicators

```
double val = 0;
cout << "> ";          // print prompt
while (cin) {          // infinite loop while input stream exists
    Token t = ts.Get();          // ts is an object of class Token_stream
    if (t.kind == 'q') break;    // check for "quit"
    if (t.kind == ';') {
        cout << "= " << val << "\n > ";          // print "= result" and prompt
    } else {
        ts.PutBack(t);                          // put t back to stream if it's not ';' or 'q'
        val = expression();                     // read and evaluate an expression
    }
}
```

> 2+3; 5*7; 2+9; the program doesn't see an input before you hit <enter> key
= 5
> = 35
> = 11

The code is getting messy

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

Remove “magic constants”

```
// Token’s “kind” values:
const char number = '8';        // a floating-point number label
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 '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'
    return 0;
  }
  ts.PutBack(t);
  cout << result << expression() << endl;
}
```

Remove “magic constants”

- But what’s wrong with “magic constants”?
  - Everybody knows numbers. E.g. **3.14159265358979323846264**, **12**, **-1**, **365**, **24**, **2.7182818284590**, **299792458**, **2.54**, **1.61**, **-273.15**, **6.6260693e-34**, **0.5291772108e-10**, **6.0221415e23**, **42**
- “Magic” is detrimental to your (mental) health!
  - It causes you to stay up all night searching for bugs
- 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 number of months
- If a constant is used twice or more, it should probably be symbolic
  - Then, if it is needed, you can change it in one place

So why did we use “magic constants”?

- To demonstrate what is really bad style
  - Now you see how ugly that first code was
- Because we forget (get busy, etc.) and write ugly code
  - “Cleaning up code” is a real and important activity
    - Not only for students
    - Re-test the program whenever you have made a change
  - So, periodically stop adding functionality, “go back” and review your code
    - Finally it will save your time!

Recover from errors

- Any user's error terminates the program
  - That's not ideal, so design your code to be able to recover after errors:

```
int main()
try {
    // … do "everything" …
} catch (exception& e) {   // catch errors we know 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 (step 1)

```
int main()
try {
    calculate();    // all code is inside
    return 0;
} catch (exception& e) {    // errors we understand something about
    cerr << e.what() << endl;
    return 1;
} catch (...) {                    // other errors
    cerr << "exception"<<endl;
    return 2;
}
```

Recover from errors

- "packing" of the read and evaluate loop into function **calculate()** allows us to simplify recovery from errors.

```
void calculate()
{
    while (cin) {  // loop over input expressions
        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;
    }   // end of loop over input expressions
}
```

Recover from errors

- Move code that handles exceptions from which we can recover from **main()** to **calculate()** (step 2)

```
int main()
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 {
         cout << prompt;
         Token t = ts.Get();
          while (t.kind == print) t=ts.Get();    // skip all "prints"
          if (t.kind == quit) return;            // quit
          ts.PutBack(t);
          cout << result << expression() << endl;
      } catch (exception& e) {          // catch any std exception
          cerr << e.what() << endl;     // what() gives exception type
          clean_up_mess();              // <<< The tricky part we have to write!
      }
    }
}
```

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 well. Why? Consider the input **1@$z; 1+3;** When you try to **clean_up_mess()** from the bad token **@**, you get a **"Bad token"** exception (thrown by **ts.Get()**) trying to get rid of **$**
  - We always have to 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 jump down to the level of characters
  - The solution must be a modification of Token_stream:

```
class Token_stream {
    bool full;          // is there a Token in the buffer?
    Token buffer;       // here is where we keep a Token
                        // which was put back using PutBack()
public:
    Token Get();                // get a Token
    void PutBack(Token t);      // put back a Token t
    Token_stream();             // constructor of Token_stream
    void ignore(char c);        // discard tokens up to and including c
};
```

Recover from errors

```
// skip characters until we find c,
// (also discard this c)

void Token_stream::ignore(char c)
{
    // first look in buffer if c is already there:
    if (full && (buffer.kind == c)) {    // && means and
        full = false;
        return;
    }
    full = false;       // discard the contents of buffer (tag as 'empty')
                        // now search c in input:
    char ch = 0;
    while (cin>>ch) { if (ch==c) return; }
}
```

Recover from errors

- clean_up_mess() now is trivial

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

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

Remark

- Why this example of “simple” calculator is interesting?
  - It is an illustration of the program which parse text and do some actions according to set of “grammatical” rules (i.e. “language”)
  - all modern programming language compilers are built on similar principles.

Next talk

- functions
- scope
- references