

```
1: // $Id: ubigint.h,v 1.5 2020-10-11 12:25:22-07 - - $
2:
3: #ifndef __UBIGINT_H__
4: #define __UBIGINT_H__
5:
6: #include <exception>
7: #include <iostream>
8: #include <limits>
9: #include <utility>
10: using namespace std;
11:
12: #include "debug.h"
13:
14: class ubigint {
15:     friend ostream& operator<< (ostream&, const ubigint&);
16:     private:
17:         using unumber = unsigned long;
18:         unumber uvalue {};
19:     public:
20:         void multiply_by_2();
21:         void divide_by_2();
22:
23:         ubigint() = default; // Need default ctor as well.
24:         ubigint (unsigned long);
25:         ubigint (const string&);
26:
27:         ubigint operator+ (const ubigint&) const;
28:         ubigint operator- (const ubigint&) const;
29:         ubigint operator* (const ubigint&) const;
30:         ubigint operator/ (const ubigint&) const;
31:         ubigint operator% (const ubigint&) const;
32:
33:         bool operator== (const ubigint&) const;
34:         bool operator< (const ubigint&) const;
35: };
36:
37: #endif
38:
```

```
1: // $Id: ubigint.cpp,v 1.12 2020-10-19 13:14:59-07 - - $
2:
3: #include <cctype>
4: #include <cstdlib>
5: #include <exception>
6: #include <stack>
7: #include <stdexcept>
8: using namespace std;
9:
10: #include "debug.h"
11: #include "relops.h"
12: #include "ubigint.h"
13:
14: ubigint::ubigint (unsigned long that): uvalue (that) {
15:     DEBUGF ('~', this << " -> " << uvalue)
16: }
17:
18: ubigint::ubigint (const string& that): uvalue(0) {
19:     DEBUGF ('~', "that = \"" << that << "\"");
20:     for (char digit: that) {
21:         if (not isdigit (digit)) {
22:             throw invalid_argument ("ubigint::ubigint(" + that + ")");
23:         }
24:         uvalue = uvalue * 10 + digit - '0';
25:     }
26: }
27:
28: ubigint ubigint::operator+ (const ubigint& that) const {
29:     DEBUGF ('u', *this << "+" << that);
30:     ubigint result (uvalue + that.uvalue);
31:     DEBUGF ('u', result);
32:     return result;
33: }
34:
35: ubigint ubigint::operator- (const ubigint& that) const {
36:     if (*this < that) throw domain_error ("ubigint::operator-(a<b)");
37:     return ubigint (uvalue - that.uvalue);
38: }
39:
40: ubigint ubigint::operator* (const ubigint& that) const {
41:     return ubigint (uvalue * that.uvalue);
42: }
43:
44: void ubigint::multiply_by_2() {
45:     uvalue *= 2;
46: }
47:
48: void ubigint::divide_by_2() {
49:     uvalue /= 2;
50: }
51:
```

```
52:
53: struct quo_rem { ubigint quotient; ubigint remainder; };
54: quo_rem udivide (const ubigint& dividend, const ubigint& divisor_) {
55:     // NOTE: udivide is a non-member function.
56:     ubigint divisor {divisor_};
57:     ubigint zero {0};
58:     if (divisor == zero) throw domain_error ("udivide by zero");
59:     ubigint power_of_2 {1};
60:     ubigint quotient {0};
61:     ubigint remainder {dividend}; // left operand, dividend
62:     while (divisor < remainder) {
63:         divisor.multiply_by_2();
64:         power_of_2.multiply_by_2();
65:     }
66:     while (power_of_2 > zero) {
67:         if (divisor <= remainder) {
68:             remainder = remainder - divisor;
69:             quotient = quotient + power_of_2;
70:         }
71:         divisor.divide_by_2();
72:         power_of_2.divide_by_2();
73:     }
74:     DEBUGF ('/', "quotient = " << quotient);
75:     DEBUGF ('/', "remainder = " << remainder);
76:     return {.quotient = quotient, .remainder = remainder};
77: }
78:
79: ubigint ubigint::operator/ (const ubigint& that) const {
80:     return udivide (*this, that).quotient;
81: }
82:
83: ubigint ubigint::operator% (const ubigint& that) const {
84:     return udivide (*this, that).remainder;
85: }
86:
87: bool ubigint::operator== (const ubigint& that) const {
88:     return uvalue == that.uvalue;
89: }
90:
91: bool ubigint::operator< (const ubigint& that) const {
92:     return uvalue < that.uvalue;
93: }
94:
95: ostream& operator<< (ostream& out, const ubigint& that) {
96:     return out << "ubigint(" << that.uvalue << ")";
97: }
98:
```

```
1: // $Id: bigint.h,v 1.2 2020-01-06 13:39:55-08 - - $
2:
3: #ifndef __BIGINT_H__
4: #define __BIGINT_H__
5:
6: #include <exception>
7: #include <iostream>
8: #include <limits>
9: #include <utility>
10: using namespace std;
11:
12: #include "debug.h"
13: #include "relops.h"
14: #include "ubigint.h"
15:
16: class bigint {
17:     friend ostream& operator<< (ostream&, const bigint&);
18:     private:
19:         ubigint uvalue;
20:         bool is_negative {false};
21:     public:
22:
23:         bigint() = default; // Needed or will be suppressed.
24:         bigint (long);
25:         bigint (const ubigint&, bool is_negative = false);
26:         explicit bigint (const string&);
27:
28:         bigint operator+() const;
29:         bigint operator-() const;
30:
31:         bigint operator+ (const bigint&) const;
32:         bigint operator- (const bigint&) const;
33:         bigint operator* (const bigint&) const;
34:         bigint operator/ (const bigint&) const;
35:         bigint operator% (const bigint&) const;
36:
37:         bool operator== (const bigint&) const;
38:         bool operator< (const bigint&) const;
39: };
40:
41: #endif
42:
```

```
1: // $Id: bigint.cpp,v 1.4 2021-09-26 10:35:44-07 - - $
2:
3: #include <cstdlib>
4: #include <exception>
5: #include <stack>
6: #include <stdexcept>
7: using namespace std;
8:
9: #include "bigint.h"
10:
11: bigint::bigint (long that): uvalue (that), is_negative (that < 0) {
12:     DEBUGF ('~', this << " -> " << uvalue)
13: }
14:
15: bigint::bigint (const ubigint& uvalue_, bool is_negative_):
16:     uvalue(uvalue_), is_negative(is_negative_) {
17: }
18:
19: bigint::bigint (const string& that) {
20:     is_negative = that.size() > 0 and that[0] == '_';
21:     uvalue = ubigint (that.substr (is_negative ? 1 : 0));
22: }
23:
24: bigint bigint::operator+ () const {
25:     return *this;
26: }
27:
28: bigint bigint::operator- () const {
29:     return {uvalue, not is_negative};
30: }
31:
32: bigint bigint::operator+ (const bigint& that) const {
33:     ubigint result {uvalue + that.uvalue};
34:     return result;
35: }
36:
37: bigint bigint::operator- (const bigint& that) const {
38:     ubigint result {uvalue - that.uvalue};
39:     return result;
40: }
41:
```

```
42:
43: bigint bigint::operator* (const bigint& that) const {
44:     bigint result {uvalue * that.uvalue};
45:     return result;
46: }
47:
48: bigint bigint::operator/ (const bigint& that) const {
49:     bigint result {uvalue / that.uvalue};
50:     return result;
51: }
52:
53: bigint bigint::operator% (const bigint& that) const {
54:     bigint result {uvalue % that.uvalue};
55:     return result;
56: }
57:
58: bool bigint::operator== (const bigint& that) const {
59:     return is_negative == that.is_negative and uvalue == that.uvalue;
60: }
61:
62: bool bigint::operator< (const bigint& that) const {
63:     if (is_negative != that.is_negative) return is_negative;
64:     return is_negative ? uvalue > that.uvalue
65:         : uvalue < that.uvalue;
66: }
67:
68: ostream& operator<< (ostream& out, const bigint& that) {
69:     return out << "bigint(" << (that.is_negative ? "-" : "+")
70:         << "," << that.uvalue << ")";
71: }
72:
```

```
1: // $Id: libfns.h,v 1.1 2019-12-12 18:19:23-08 - - $
2:
3: // Library functions not members of any class.
4:
5: #include "bigint.h"
6:
7: bigint pow (const bigint& base, const bigint& exponent);
8:
```

```
1: // $Id: libfns.cpp,v 1.1 2019-12-12 18:19:23-08 - - $
2:
3: #include "libfns.h"
4:
5: //
6: // This algorithm would be more efficient with operators
7: // *=, /=2, and is_odd. But we leave it here.
8: //
9:
10: bigint pow (const bigint& base_arg, const bigint& exponent_arg) {
11:     bigint base (base_arg);
12:     bigint exponent (exponent_arg);
13:     static const bigint ZERO (0);
14:     static const bigint ONE (1);
15:     static const bigint TWO (2);
16:     DEBUGF ('^', "base = " << base << ", exponent = " << exponent);
17:     if (base == ZERO) return ZERO;
18:     bigint result = ONE;
19:     if (exponent < ZERO) {
20:         base = ONE / base;
21:         exponent = - exponent;
22:     }
23:     while (exponent > ZERO) {
24:         if (exponent % TWO == ONE) {
25:             result = result * base;
26:             exponent = exponent - 1;
27:         } else {
28:             base = base * base;
29:             exponent = exponent / 2;
30:         }
31:     }
32:     DEBUGF ('^', "result = " << result);
33:     return result;
34: }
35:
```



```
1: // $Id: scanner.h,v 1.1 2019-12-12 18:19:23-08 - - $
2:
3: #ifndef __SCANNER_H__
4: #define __SCANNER_H__
5:
6: #include <iostream>
7: #include <utility>
8: using namespace std;
9:
10: #include "debug.h"
11:
12: enum class tsymbol {SCANEOF, NUMBER, OPERATOR};
13:
14: struct token {
15:     tsymbol symbol;
16:     string lexinfo;
17:     token (tsymbol sym, const string& lex = string()):
18:         symbol(sym), lexinfo(lex){
19:     }
20: };
21:
22: class scanner {
23:     private:
24:         istream& instream;
25:         int nextchar {instream.get()};
26:         bool good() { return nextchar != EOF; }
27:         char get();
28:     public:
29:         scanner (istream& instream_ = cin): instream(instream_) {}
30:         token scan();
31: };
32:
33: ostream& operator<< (ostream&, tsymbol);
34: ostream& operator<< (ostream&, const token&);
35:
36: #endif
37:
```

```
1: // $Id: scanner.cpp,v 1.2 2021-09-26 10:35:44-07 - - $
2:
3: #include <cassert>
4: #include <iostream>
5: #include <locale>
6: #include <stdexcept>
7: #include <type_traits>
8: #include <unordered_map>
9: using namespace std;
10:
11: #include "scanner.h"
12: #include "debug.h"
13:
14: char scanner::get() {
15:     if (not good()) throw runtime_error ("scanner::get() past EOF");
16:     int currchar {nextchar};
17:     nextchar = instream.get();
18:     return char (currchar);
19: }
20:
21: token scanner::scan() {
22:     while (good() and isspace (nextchar)) get();
23:     if (not good()) return {tsymbol::SCANEOF};
24:     if (nextchar == '_' or isdigit (nextchar)) {
25:         token result {tsymbol::NUMBER, {get()}};
26:         while (good() and isdigit (nextchar)) result.lexinfo += get();
27:         return result;
28:     }
29:     return {tsymbol::OPERATOR, {get()}};
30: }
31:
32: ostream& operator<< (ostream& out, tsymbol symbol) {
33:     const char* sym_name {" "};
34:     switch (symbol) {
35:         case tsymbol::NUMBER : sym_name = "NUMBER" ; break;
36:         case tsymbol::OPERATOR: sym_name = "OPERATOR"; break;
37:         case tsymbol::SCANEOF : sym_name = "SCANEOF" ; break;
38:         default : assert (false) ; break;
39:     }
40:     return out << sym_name;
41: }
42:
43: ostream& operator<< (ostream& out, const token& token) {
44:     out << "{" << token.symbol << ", \"" << token.lexinfo << "\"}";
45:     return out;
46: }
47:
```

```
1: // $Id: debug.h,v 1.1 2019-12-12 18:19:23-08 - - $
2:
3: #ifndef __DEBUG_H__
4: #define __DEBUG_H__
5:
6: #include <bitset>
7: #include <climits>
8: #include <string>
9: using namespace std;
10:
11: // debug -
12: //     static class for maintaining global debug flags.
13: // setflags -
14: //     Takes a string argument, and sets a flag for each char in the
15: //     string. As a special case, '@', sets all flags.
16: // getflag -
17: //     Used by the DEBUGF macro to check to see if a flag has been set.
18: //     Not to be called by user code.
19:
20: class debugflags {
21:     private:
22:         using flagset = bitset<UCHAR_MAX + 1>;
23:         static flagset flags;
24:     public:
25:         static void setflags (const string& optflags);
26:         static bool getflag (char flag);
27:         static void where (char flag, const char* file, int line,
28:                           const char* pretty_function);
29: };
30:
```

```
31:
32: // DEBUGF -
33: //     Macro which expands into debug code.  First argument is a
34: //     debug flag char, second argument is output code that can
35: //     be sandwiched between <<.  Beware of operator precedence.
36: //     Example:
37: //         DEBUGF ('u', "foo = " << foo);
38: //     will print two words and a newline if flag 'u' is on.
39: //     Traces are preceded by filename, line number, and function.
40:
41: #ifdef NDEBUG
42: #define DEBUGF(FLAG, CODE) ;
43: #define DEBUGS(FLAG, STMT) ;
44: #else
45: #define DEBUGF(FLAG, CODE) { \
46:     if (debugflags::getflag (FLAG)) { \
47:         debugflags::where (FLAG, __FILE__, __LINE__, \
48:             __PRETTY_FUNCTION__); \
49:         cerr << CODE << endl; \
50:     } \
51: }
52: #define DEBUGS(FLAG, STMT) { \
53:     if (debugflags::getflag (FLAG)) { \
54:         debugflags::where (FLAG, __FILE__, __LINE__, \
55:             __PRETTY_FUNCTION__); \
56:         STMT; \
57:     } \
58: }
59: #endif
60:
61: #endif
62:
```

```
1: // $Id: debug.cpp,v 1.2 2021-03-31 19:46:57-07 - - $
2:
3: #include <climits>
4: #include <iostream>
5: #include <vector>
6:
7: using namespace std;
8:
9: #include "debug.h"
10: #include "util.h"
11:
12: debugflags::flagset debugflags::flags {};
13:
14: void debugflags::setflags (const string& initflags) {
15:     for (const unsigned char flag: initflags) {
16:         if (flag == '@') flags.set();
17:         else flags.set (flag, true);
18:     }
19: }
20:
21: // getflag -
22: //     Check to see if a certain flag is on.
23:
24: bool debugflags::getflag (char flag) {
25:     // WARNING: Don't TRACE this function or the stack will blow up.
26:     return flags.test (static_cast<unsigned char> (flag));
27: }
28:
29: void debugflags::where (char flag, const char* file, int line,
30:                        const char* pretty_function) {
31:     cerr << exec::execname() << ": DEBUG(" << flag << ") "
32:          << file << "[" << line << "]" " << endl
33:          << "    " << pretty_function << endl;
34: }
35:
```

```
1: // $Id: util.h,v 1.2 2019-12-12 19:22:40-08 - - $
2:
3: //
4: // util -
5: //     A utility class to provide various services
6: //     not conveniently included in other modules.
7: //
8:
9: #ifndef __UTIL_H__
10: #define __UTIL_H__
11:
12: #include <iomanip>
13: #include <iostream>
14: #include <sstream>
15: #include <stdexcept>
16: #include <vector>
17: using namespace std;
18:
19: #include "debug.h"
20:
21: //
22: // ydc_error -
23: //     Indicate a problem where processing should be abandoned and
24: //     the main function should take control.
25: //
26:
27: class ydc_error: public runtime_error {
28:     public:
29:         explicit ydc_error (const string& what): runtime_error (what) {
30:             }
31: };
32:
33: //
34: // octal -
35: //     Convert integer to octal string.
36: //
37:
38: template <typename numeric>
39: const string octal (numeric number) {
40:     ostringstream stream;
41:     stream << showbase << oct << (number + 0);
42:     return stream.str();
43: }
44:
```

```
45:
46: //
47: // main -
48: //     Keep track of execname and exit status.  Must be initialized
49: //     as the first thing done inside main.  Main should call:
50: //         main::execname (argv[0]);
51: //     before anything else.
52: //
53:
54: class exec {
55:     private:
56:         static string execname_;
57:         static int status_;
58:         static void execname (const string& argv0);
59:         friend int main (int, char**);
60:     public:
61:         static void status (int status);
62:         static const string& execname() {return execname_; }
63:         static int status() {return status_; }
64: };
65:
66: //
67: // complain -
68: //     Used for starting error messages.  Sets the exit status to
69: //     EXIT_FAILURE, writes the program name to cerr, and then
70: //     returns the cerr ostream.  Example:
71: //         complain() << filename << ": some problem" << endl;
72: //
73:
74: ostream& note();
75: ostream& error();
76:
77: #endif
78:
```

```
1: // $Id: util.cpp,v 1.2 2019-12-12 19:22:40-08 - - $
2:
3: #include <cstring>
4: using namespace std;
5:
6: #include "util.h"
7:
8: string exec::execname_; // Must be initialized from main().
9: int exec::status_ = EXIT_SUCCESS;
10:
11: void exec::execname (const string& argv0) {
12:     execname_ = basename (argv0.c_str());
13:     cout << boolalpha;
14:     cerr << boolalpha;
15:     DEBUGF ('Y', "execname = " << execname_);
16: }
17:
18: void exec::status (int new_status) {
19:     new_status &= 0xFF;
20:     if (status_ < new_status) status_ = new_status;
21: }
22:
23: ostream& note() {
24:     return cerr << exec::execname() << ": ";
25: }
26:
27: ostream& error() {
28:     exec::status (EXIT_FAILURE);
29:     return note();
30: }
31:
```



```
1: // $Id: iterstack.h,v 1.1 2019-12-12 18:19:23-08 - - $
2:
3: //
4: // The class std::stack does not provide an iterator, which is
5: // needed for this class. So, like std::stack, class iterstack
6: // is implemented on top of a container.
7: //
8: // We use private inheritance because we want to restrict
9: // operations only to those few that are approved. All functions
10: // are merely inherited from the container, with only ones needed
11: // being exported as public.
12: //
13: // No implementation file is needed because all functions are
14: // inherited, and the convenience functions that are added are
15: // trivial, and so can be inline.
16: //
17: // Any underlying container which supports the necessary operations
18: // could be used, such as vector, list, or deque.
19: //
20:
21: #ifndef __ITERSTACK_H__
22: #define __ITERSTACK_H__
23:
24: #include <vector>
25: using namespace std;
26:
27: template <typename value_t, typename container = vector<value_t>>
28: class iterstack {
29:     public:
30:         using value_type = value_t;
31:         using const_iterator = typename container::const_reverse_iterator;
32:         using size_type = typename container::size_type;
33:     private:
34:         container stack;
35:     public:
36:         void clear() { stack.clear(); }
37:         bool empty() const { return stack.empty(); }
38:         size_type size() const { return stack.size(); }
39:         const_iterator begin() { return stack.crbegin(); }
40:         const_iterator end() { return stack.crend(); }
41:         void push (const value_type& value) { stack.push_back (value); }
42:         void pop() { stack.pop_back(); }
43:         const value_type& top() const { return stack.back(); }
44: };
45:
46: #endif
47:
```

```
1: // $Id: relops.h,v 1.1 2019-12-12 18:19:23-08 - - $
2:
3: //
4: // Assuming that for any given type T, there are operators
5: // bool operator< (const T&, const T&);
6: // bool operator== (const T&, const T&);
7: // as fundamental comparisons for type T, define the other
8: // six operators in terms of the basic ones.
9: //
10:
11: #ifndef __REL_OPS_H__
12: #define __REL_OPS_H__
13:
14: template <typename value>
15: inline bool operator!= (const value& left, const value& right) {
16:     return not (left == right);
17: }
18:
19: template <typename value>
20: inline bool operator> (const value& left, const value& right) {
21:     return right < left;
22: }
23:
24: template <typename value>
25: inline bool operator<= (const value& left, const value& right) {
26:     return not (right < left);
27: }
28:
29: template <typename value>
30: inline bool operator>= (const value& left, const value& right) {
31:     return not (left < right);
32: }
33:
34: #endif
35:
```

```
1: // $Id: main.cpp,v 1.3 2021-03-26 15:29:03-07 - - $
2:
3: #include <cassert>
4: #include <deque>
5: #include <iostream>
6: #include <stdexcept>
7: #include <unordered_map>
8: #include <utility>
9: using namespace std;
10:
11: #include <unistd.h>
12:
13: #include "bigint.h"
14: #include "debug.h"
15: #include "iterstack.h"
16: #include "libfns.h"
17: #include "scanner.h"
18: #include "util.h"
19:
20: using bigint_stack = iterstack<bigint>;
21:
22: void do_arith (bigint_stack& stack, const char oper) {
23:     if (stack.size() < 2) throw ydc_error ("stack empty");
24:     bigint right = stack.top();
25:     stack.pop();
26:     DEBUGF ('d', "right = " << right);
27:     bigint left = stack.top();
28:     stack.pop();
29:     DEBUGF ('d', "left = " << left);
30:     bigint result;
31:     switch (oper) {
32:         case '+': result = left + right; break;
33:         case '-': result = left - right; break;
34:         case '*': result = left * right; break;
35:         case '/': result = left / right; break;
36:         case '%': result = left % right; break;
37:         case '^': result = pow (left, right); break;
38:         default: throw invalid_argument ("do_arith operator "s + oper);
39:     }
40:     DEBUGF ('d', "result = " << result);
41:     stack.push (result);
42: }
43:
44: void do_clear (bigint_stack& stack, const char) {
45:     DEBUGF ('d', "");
46:     stack.clear();
47: }
48:
```

```
49:
50: void do_dup (bigint_stack& stack, const char) {
51:     if (stack.size() < 1) throw ydc_error ("stack empty");
52:     bigint top = stack.top();
53:     DEBUGF ('d', top);
54:     stack.push (top);
55: }
56:
57: void do_printall (bigint_stack& stack, const char) {
58:     for (const auto& elem: stack) cout << elem << endl;
59: }
60:
61: void do_print (bigint_stack& stack, const char) {
62:     if (stack.size() < 1) throw ydc_error ("stack empty");
63:     cout << stack.top() << endl;
64: }
65:
66: void do_debug (bigint_stack&, const char) {
67:     cout << "Y not implemented" << endl;
68: }
69:
70: class ydc_quit: public exception {};
71: void do_quit (bigint_stack&, const char) {
72:     throw ydc_quit();
73: }
74:
75: string unimplemented (char oper) {
76:     if (isgraph (oper)) {
77:         return "'" + oper + "' (" + octal (oper) + ") unimplemented";
78:     } else {
79:         return octal (oper) + " unimplemented"s;
80:     }
81: }
82:
83: void do_function (bigint_stack& stack, const char oper) {
84:     switch (oper) {
85:         case '+': do_arith      (stack, oper); break;
86:         case '-': do_arith      (stack, oper); break;
87:         case '*': do_arith      (stack, oper); break;
88:         case '/': do_arith      (stack, oper); break;
89:         case '%': do_arith      (stack, oper); break;
90:         case '^': do_arith      (stack, oper); break;
91:         case 'Y': do_debug      (stack, oper); break;
92:         case 'C': do_clear      (stack, oper); break;
93:         case 'd': do_dup        (stack, oper); break;
94:         case 'f': do_printall   (stack, oper); break;
95:         case 'p': do_print      (stack, oper); break;
96:         case 'q': do_quit       (stack, oper); break;
97:         default : throw ydc_error (unimplemented (oper));
98:     }
99: }
100:
```

```
101:
102: //
103: // scan_options
104: // Options analysis: The only option is -Dflags.
105: //
106: void scan_options (int argc, char** argv) {
107:     opterr = 0;
108:     for (;;) {
109:         int option = getopt (argc, argv, "@:");
110:         if (option == EOF) break;
111:         switch (option) {
112:             case '@':
113:                 debugflags::setflags (optarg);
114:                 break;
115:             default:
116:                 error() << "-" << static_cast<char> (optopt)
117:                     << ": invalid option" << endl;
118:                 break;
119:         }
120:     }
121:     if (optind < argc) {
122:         error() << "operand not permitted" << endl;
123:     }
124: }
125:
```

```
126:
127: //
128: // Main function.
129: //
130: int main (int argc, char** argv) {
131:     exec::execname (argv[0]);
132:     scan_options (argc, argv);
133:     bigint_stack operand_stack;
134:     scanner input;
135:     try {
136:         for (;;) {
137:             try {
138:                 token lexeme = input.scan();
139:                 switch (lexeme.symbol) {
140:                     case tsymbol::SCANEOF:
141:                         throw ydc_quit();
142:                     break;
143:                     case tsymbol::NUMBER:
144:                         operand_stack.push (bigint (lexeme.lexinfo));
145:                     break;
146:                     case tsymbol::OPERATOR: {
147:                         char oper = lexeme.lexinfo[0];
148:                         do_function (operand_stack, oper);
149:                     } break;
150:                     }
151:                 default:
152:                     assert (false);
153:             }
154:         } catch (ydc_error& error) {
155:             cout << exec::execname() << ": " << error.what() << endl;
156:         }
157:     }
158:     } catch (ydc_quit&) {
159:         // Intentionally left empty.
160:     }
161:     return exec::status();
162: }
163:
```

```
1: # $Id: Makefile,v 1.10 2021-09-05 12:44:19-07 - - $
2:
3: MKFILE      = Makefile
4: DEPSFILE    = ${MKFILE}.deps
5: NOINCL      = ci clean spotless
6: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
7: GMAKE       = ${MAKE} --no-print-directory
8: GPPOPTS     = -std=gnu++2a -fdiagnostics-color=never
9: GPPWARN     = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
10: GPP         = g++ ${GPPOPTS} ${GPPWARN}
11: COMPILECPP  = ${GPP} -g -O0 ${GPPOPTS}
12: MAKEDEPSCPP = ${GPP} -MM ${GPPOPTS}
13:
14: MODULES     = ubigint bigint libfns scanner debug util
15: CPPHEADER   = ${MODULES:=.h} iterstack.h relops.h
16: CPPSOURCE   = ${MODULES:=.cpp} main.cpp
17: EXECBIN     = ydc
18: OBJECTS     = ${CPPSOURCE:.cpp=.o}
19: MODULESRC   = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.cpp}
20: OTHERSRC    = ${filter-out ${MODULESRC}, ${CPPHEADER} ${CPPSOURCE}}
21: ALLSOURCES  = ${MODULESRC} ${OTHERSRC} ${MKFILE}
22: LISTING     = Listing.ps
23:
24: export PATH := ${PATH}:/afs/cats.ucsc.edu/courses/cse110a-wm/bin
25:
26: all : ${EXECBIN}
27:
28: ${EXECBIN} : ${OBJECTS} ${MKFILE}
29:     ${COMPILECPP} -o $@ ${OBJECTS}
30:
31: %.o : %.cpp
32:     - checksource $<
33:     - cpplint.py.perl $<
34:     ${COMPILECPP} -c $<
35:
36: ci : check
37:     cid -is ${ALLSOURCES}
38:
39: check : ${ALLSOURCES}
40:     - checksource ${ALLSOURCES}
41:     - cpplint.py.perl ${CPPSOURCE}
42:
43: lis : ${ALLSOURCES}
44:     mkpspdf ${LISTING} ${ALLSOURCES} ${DEPSFILE}
45:
46: clean :
47:     - rm ${OBJECTS} ${DEPSFILE} core ${EXECBIN}.errs
48:
49: spotless : clean
50:     - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
51:
```

```
52:
53: deps : ${CPPSOURCE} ${CPPHEADER}
54:         @ echo "# ${DEPSFILE} created `LC_TIME=C date`" >${DEPSFILE}
55:         ${MAKEDEPSCPP} ${CPPSOURCE} >>${DEPSFILE}
56:
57: ${DEPSFILE} :
58:         @ touch ${DEPSFILE}
59:         ${GMAKE} deps
60:
61: again :
62:         ${GMAKE} spotless deps ci all lis
63:
64: ifeq (${NEEDINCL}, )
65: include ${DEPSFILE}
66: endif
67:
```



```
1: # Makefile.deps created Sun Sep 26 10:35:47 PDT 2021
2: ubigint.o: ubigint.cpp debug.h relops.h ubigint.h
3: bigint.o: bigint.cpp bigint.h debug.h relops.h ubigint.h
4: libfns.o: libfns.cpp libfns.h bigint.h debug.h relops.h ubigint.h
5: scanner.o: scanner.cpp scanner.h debug.h
6: debug.o: debug.cpp debug.h util.h
7: util.o: util.cpp util.h debug.h
8: main.o: main.cpp bigint.h debug.h relops.h ubigint.h iterstack.h libfns.
h \
9:  scanner.h util.h
```