```
 1: // $Id: ubigint.h,v 1.3 2022-01-11 17:47:47-08 - - $
 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: #include "relops.h"
14:
15: class ubigint {
16:    friend ostream& operator<< (ostream&, const ubigint&);
17:    private:
18:       using ubigvalue_t = unsigned long;
19:       ubigvalue_t uvalue {};
20:    public:
21:       void multiply_by_2();
22:       void divide_by_2();
23:
24:       ubigint() = default; // Need default ctor as well.
25:       ubigint (unsigned long);
26:       ubigint (const string&);
27:
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:       ubigint operator% (const ubigint&) const;
33:
34:       bool operator== (const ubigint&) const;
35:       bool operator<  (const ubigint&) const;
36:
37:       void print() const;
38: };
39:
40: #endif
41:
```

```
 1: // $Id: ubigint.cpp,v 1.3 2022-01-11 17:47:47-08 - - $
 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 "ubigint.h"
12:
13: ubigint::ubigint (unsigned long that): uvalue (that) {
14:    DEBUGF ('~', this << " -> " << uvalue)
15: }
16:
17: ubigint::ubigint (const string& that): uvalue(0) {
18:    DEBUGF ('~', "that = \"" << that << "\"");
19:    for (char digit: that) {
20:       if (not isdigit (digit)) {
21:          throw invalid_argument ("ubigint::ubigint(" + that + ")");
22:       }
23:       uvalue = uvalue * 10 + digit - '0';
24:    }
25: }
26:
27: ubigint ubigint::operator+ (const ubigint& that) const {
28:    DEBUGF ('u', *this << "+" << that);
29:    ubigint result (uvalue + that.uvalue);
30:    DEBUGF ('u', result);
31:    return result;
32: }
33:
34: ubigint ubigint::operator- (const ubigint& that) const {
35:    if (*this < that) throw domain_error ("ubigint::operator-(a<b)");
36:    return ubigint (uvalue - that.uvalue);
37: }
38:
39: ubigint ubigint::operator* (const ubigint& that) const {
40:    return ubigint (uvalue * that.uvalue);
41: }
42:
43: void ubigint::multiply_by_2() {
44:    uvalue *= 2;
45: }
46:
47: void ubigint::divide_by_2() {
48:    uvalue /= 2;
49: }
50:
```

```
 51:
 52: struct quo_rem { ubigint quotient; ubigint remainder; };
 53: quo_rem udivide (const ubigint& dividend, const ubigint& divisor_) {
 54:    // NOTE: udivide is a non-member function.
 55:    ubigint divisor {divisor_};
 56:    ubigint zero {0};
 57:    if (divisor == zero) throw domain_error ("udivide by zero");
 58:    ubigint power_of_2 {1};
 59:    ubigint quotient {0};
 60:    ubigint remainder {dividend}; // left operand, dividend
 61:    while (divisor < remainder) {
 62:       divisor.multiply_by_2();
 63:       power_of_2.multiply_by_2();
 64:    }
 65:    while (power_of_2 > zero) {
 66:       if (divisor <= remainder) {
 67:          remainder = remainder - divisor;
 68:          quotient = quotient + power_of_2;
 69:       }
 70:       divisor.divide_by_2();
 71:       power_of_2.divide_by_2();
 72:    }
 73:    DEBUGF ('/', "quotient = " << quotient);
 74:    DEBUGF ('/', "remainder = " << remainder);
 75:    return {.quotient = quotient, .remainder = remainder};
 76: }
 77:
 78: ubigint ubigint::operator/ (const ubigint& that) const {
 79:    return udivide (*this, that).quotient;
 80: }
 81:
 82: ubigint ubigint::operator% (const ubigint& that) const {
 83:    return udivide (*this, that).remainder;
 84: }
 85:
 86: bool ubigint::operator== (const ubigint& that) const {
 87:    return uvalue == that.uvalue;
 88: }
 89:
 90: bool ubigint::operator< (const ubigint& that) const {
 91:    return uvalue < that.uvalue;
 92: }
 93:
 94: void ubigint::print() const {
 95:    DEBUGF ('p', this << " -> " << *this);
 96:    cout << uvalue;
 97: }
 98:
 99: ostream& operator<< (ostream& out, const ubigint& that) {
100:    return out << "ubigint(" << that.uvalue << ")";
101: }
102:
```

```
 1: // $Id: bigint.h,v 1.3 2022-01-07 17:55:54-08 - - $
 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:       void print() const;
41: };
42:
43: #endif
44:
```

```
 1: // $Id: bigint.cpp,v 1.2 2021-12-28 14:11:26-08 - - $
 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: void bigint::print() const {
69:    DEBUGF ('p', this << " -> " << *this);
70:    if (is_negative) cout << "-";
71:    uvalue.print();
72: }
73:
74: ostream& operator<< (ostream& out, const bigint& that) {
75:    return out << "bigint(" << (that.is_negative ? "-" : "+")
76:               << "," << that.uvalue << ")";
77: }
78:
```

```
 1: // $Id: libfns.h,v 1.1 2021-12-28 13:54:01-08 - - $
 2:
 3: // Library functions not members of any class.
 4:
 5: #ifndef LIBFNS_H
 6: #define LIBFNS_H
 7:
 8: #include "bigint.h"
 9:
10: bigint pow (const bigint& base, const bigint& exponent);
11:
12: #endif
13:
```

```
 1: // $Id: libfns.cpp,v 1.1 2021-12-28 13:54:01-08 - - $
 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 2021-12-28 13:54:01-08 - - $
 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() const { 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 1.1 2021-12-28 13:54:01-08 - - $
 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 2021-12-28 13:54:01-08 - - $
 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 trace code.  First argument is a
34: //    trace 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 1.1 2021-12-28 13:54:01-08 - - $
 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 << "DEBUG(" << flag << ") "
32:         << file << "[" << line << "] " << endl
33:         << "... " << pretty_function << endl;
34: }
35:
```

```
 1: // $Id: util.h,v 1.1 2021-12-28 13:54:01-08 - - $
 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: const string octal (long number);
39:
```

```
40:
41: //
42: // main -
43: //    Keep track of execname and exit status.  Must be initialized
44: //    as the first thing done inside main.  Main should call:
45: //       main::execname (argv[0]);
46: //    before anything else.
47: //
48:
49: class exec {
50:    private:
51:       static string execname_;
52:       static int status_;
53:       static void execname (const string& argv0);
54:       friend int main (int, char**);
55:    public:
56:       static void status (int status);
57:       static const string& execname() {return execname_; }
58:       static int status() {return status_; }
59: };
60:
61: //
62: // complain -
63: //    Used for starting error messages.  Sets the exit status to
64: //    EXIT_FAILURE, writes the program name to cerr, and then
65: //    returns the cerr ostream.  Example:
66: //       complain() << filename << ": some problem" << endl;
67: //
68:
69: ostream& note();
70: ostream& error();
71:
72: #endif
73:
```

```
 1: // $Id: util.cpp,v 1.1 2021-12-28 13:54:01-08 - - $
 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: const string octal (long number) {
24:    ostringstream stream;
25:    stream << showbase << oct << number;
26:    return stream.str();
27: }
28:
29: ostream& note() {
30:    return cerr << exec::execname() << ": ";
31: }
32:
33: ostream& error() {
34:    exec::status (EXIT_FAILURE);
35:    return note();
36: }
37:
```

```
 1: // $Id: iterstack.h,v 1.1 2021-12-28 13:54:01-08 - - $
 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 1.4 2022-01-11 22:20:02-08 - - $
 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 RELOPS_H
12: #define RELOPS_H
13:
14: template <typename value>
15: inline bool operator>  (const value& left, const value& right) {
16:    return right < left;
17: }
18:
19: template <typename value>
20: inline bool operator<= (const value& left, const value& right) {
21:    return not (right < left);
22: }
23:
24: template <typename value>
25: inline bool operator>= (const value& left, const value& right) {
26:    return not (left < right);
27: }
28:
29: #endif
30:
```

```
 1: // $Id: main.cpp,v 1.2 2021-12-28 14:11:26-08 - - $
 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) {
 59:       elem.print();
 60:       cout << endl;
 61:    }
 62: }
 63:
 64: void do_print (bigint_stack& stack, const char) {
 65:    if (stack.size() < 1) throw ydc_error ("stack empty");
 66:    stack.top().print();
 67:    cout << endl;
 68: }
 69:
 70: void do_debug (bigint_stack&, const char) {
 71:    cout << "Y not implemented" << endl;
 72: }
 73:
 74: class ydc_quit: public exception {};
 75: void do_quit (bigint_stack&, const char) {
 76:    throw ydc_quit();
 77: }
 78:
 79: string unimplemented (char oper) {
 80:    if (isgraph (oper)) {
 81:       return "'"s + oper + "' ("s + octal (oper) + ") unimplemented";
 82:    }else {
 83:       return octal (oper) + " unimplemented"s;
 84:    }
 85: }
 86:
 87: void do_function (bigint_stack& stack, const char oper) {
 88:    switch (oper) {
 89:       case '+': do_arith    (stack, oper); break;
 90:       case '-': do_arith    (stack, oper); break;
 91:       case '*': do_arith    (stack, oper); break;
 92:       case '/': do_arith    (stack, oper); break;
 93:       case '%': do_arith    (stack, oper); break;
 94:       case '^': do_arith    (stack, oper); break;
 95:       case 'Y': do_debug    (stack, oper); break;
 96:       case 'c': do_clear    (stack, oper); break;
 97:       case 'd': do_dup      (stack, oper); break;
 98:       case 'f': do_printall (stack, oper); break;
 99:       case 'p': do_print    (stack, oper); break;
100:       case 'q': do_quit     (stack, oper); break;
101:       default : throw ydc_error (unimplemented (oper));
102:    }
103: }
104:
```

```
105:
106: //
107: // scan_options
108: //    Options analysis:  The only option is -Dflags.
109: //
110: void scan_options (int argc, char** argv) {
111:    opterr = 0;
112:    for (;;) {
113:       int option = getopt (argc, argv, "@:");
114:       if (option == EOF) break;
115:       switch (option) {
116:          case '@':
117:             debugflags::setflags (optarg);
118:             break;
119:          default:
120:             error() << "-" << static_cast<char> (optopt)
121:                     << ": invalid option" << endl;
122:             break;
123:       }
124:    }
125:    if (optind < argc) {
126:       error() << "operand not permitted" << endl;
127:    }
128: }
129:
```

```
130:
131: //
132: // Main function.
133: //
134: int main (int argc, char** argv) {
135:    exec::execname (argv[0]);
136:    scan_options (argc, argv);
137:    bigint_stack operand_stack;
138:    scanner input;
139:    try {
140:       for (;;) {
141:          try {
142:             token lexeme = input.scan();
143:             switch (lexeme.symbol) {
144:                case tsymbol::SCANEOF:
145:                   throw ydc_quit();
146:                   break;
147:                case tsymbol::NUMBER:
148:                   operand_stack.push (bigint (lexeme.lexinfo));
149:                   break;
150:                case tsymbol::OPERATOR: {
151:                   char oper = lexeme.lexinfo[0];
152:                   do_function (operand_stack, oper);
153:                   break;
154:                   }
155:                default:
156:                   assert (false);
157:             }
158:          }catch (ydc_error& error) {
159:             cout << exec::execname() << ": " << error.what() << endl;
160:          }
161:       }
162:    }catch (ydc_quit&) {
163:       // Intentionally left empty.
164:    }
165:    return exec::status();
166: }
167:
```

```
# $Id: Makefile,v 1.2 2022-01-05 02:54:32-08 - - $

MKFILE      = Makefile
DEPSFILE    = ${MKFILE}.deps
NOINCL      = ci clean spotless
NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
GMAKE       = ${MAKE} --no-print-directory
GPPOPTS     = -std=gnu++2a -fdiagnostics-color=never
GPPWARN     = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
GPP         = g++ ${GPPOPTS} ${GPPWARN}
COMPILECPP  = ${GPP} -g -O0 ${GPPOPTS}
MAKEDEPSCPP = ${GPP} -MM ${GPPOPTS}

MODULES     = ubigint bigint libfns scanner debug util
CPPHEADER   = ${MODULES:=.h} iterstack.h relops.h
CPPSOURCE   = ${MODULES:=.cpp} main.cpp
EXECBIN     = ydc
OBJECTS     = ${CPPSOURCE:.cpp=.o}
MODULESRC   = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.cpp}
OTHERSRC    = ${filter-out ${MODULESRC}, ${CPPHEADER} ${CPPSOURCE}}
ALLSOURCES  = ${MODULESRC} ${OTHERSRC} ${MKFILE}
LISTING     = Listing.ps

export PATH := ${PATH}:/afs/cats.ucsc.edu/courses/cse110a-wm/bin

all : ${EXECBIN}

${EXECBIN} : ${OBJECTS} ${MKFILE}
        ${COMPILECPP} -o $@ ${OBJECTS}

%.o : %.cpp
        - checksource $<
        - cpplint.py.perl $<
        ${COMPILECPP} -c $<

ci : check
        cid -is ${ALLSOURCES}

check : ${ALLSOURCES}
        - checksource ${ALLSOURCES}
        - cpplint.py.perl ${CPPSOURCE}

lis : ${ALLSOURCES}
        mkpspdf ${LISTING} ${ALLSOURCES} ${DEPSFILE}

clean :
        - rm ${OBJECTS} ${DEPSFILE} core ${EXECBIN}.errs

spotless : clean
        - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
```

```
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 Tue Jan 11 18:57:25 PST 2022
    2: ubigint.o: ubigint.cpp debug.h ubigint.h relops.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
```