```
#include "SymbolTable.h"

/**
 * @brief Construct a new SymbolTable object
 */
SymbolTable::SymbolTable() : memaddress(5000) {}

/**
 * @brief Destroy the SymbolTable object
 */
SymbolTable::~SymbolTable()
{
    instr_address = 1;
}

/**
 * @brief Increment the current memory address by one
 */
void SymbolTable::incrementMem()
{
    ++this->memaddress;
}

/**
 * @brief Insert a Symbol into the symbol table
 *
 * @param t The token object
 * @return true if successful
 * @return false if unsuccessful
 */
bool SymbolTable::insert(Lexer::Token t, std::string type)
{
    bool success = false;

    if (!lookup(t))
    {
        Symbol *s = new Symbol(t, this->memaddress, type);
        this->table.push_back(*s);
        incrementMem();
        success = true;
    }

    return success;
}

/**
 * @brief Check to see if an identifier already exists
 * in the symbol table.
 *
 * @param id The identifier (lexeme)
 * @return address if existing, 0 if not
 */
int SymbolTable::lookup(Lexer::Token t)
{
    std::vector<Symbol>::iterator it = this->table.begin();
    bool found = false;

    while (!found && it != this->table.end())
    {
        if (it->token.lexeme == t.lexeme && it->token.token == t.token)
        {
            found = true;
        }
        else
        {
            ++it;
        }
    }

    return found ? it->address : 0;
}

/**
 * @brief Remove a Symbol from the symbol table.
 *
 * @param id The identifier (lexeme)
 * @return true if successful
 * @return false if unsuccessful
 */
bool SymbolTable::remove(Lexer::Token t)
{
    bool success = false;
    int pos = 0;

    if (lookup(t))
    {
        std::vector<Symbol>::const_iterator it = this->table.begin();
        while (!success && it != this->table.end())
        {
            if (it->token.lexeme == t.lexeme)
            {
                this->table.erase(this->table.begin() + pos);
                success = true;
            }
            else
            {
```

```
 97:                 ++pos;
 98:                 ++it;
 99:             }
100:         }
101:     }
102:
103:     return success;
104: }
105:
106: /**
107:  * @brief Return a string representation of the
108:  * current symbol table.
109:  *
110:  * @return std::string
111:  */
112: std::string SymbolTable::list()
113: {
114:     std::ostringstream os;
115:     const int COL_WIDTH = 15;
116:
117:     os << std::left << std::setw(COL_WIDTH) << "Identifier" << std::setw(COL_WIDTH) << "Type"
118:        << "Memory Address" << std::endl;
119:         os << std::setfill('-') << std::setw(COL_WIDTH * 2 + 14) << '-' << std::setfill(' ') << std::endl;
120:
121:     for (std::vector<Symbol>::const_iterator it = this->table.begin(); it != this->table.end(); ++it)
122:     {
123:         os << std::setw(COL_WIDTH) << it->token.lexeme << std::setw(COL_WIDTH) << it->type << it->address << std::endl;
124:     }
125:
126:     return os.str();
127: }
128:
129: /**
130:  * @brief List all of the instructions
131:  * in the instruction table
132:  *
133:  * @return std::string A formatted list of instructions
134:  */
135: std::string SymbolTable::list_instr()
136: {
137:     std::ostringstream os;
138:     const int COL_WIDTH = 15;
139:
140:     os << std::left << std::setw(COL_WIDTH) << "Address" << std::setw(COL_WIDTH) << "OpCode"
141:        << "Operand" << std::endl;
142:         os << std::setfill('-') << std::setw(COL_WIDTH * 2 + 7) << '-' << std::setfill(' ') << std::endl;
143:
144:     for (std::vector<Instr>::const_iterator it = this->instructions.begin(); it != this->instructions.end(); ++it)
145:     {
146:         os << std::setw(COL_WIDTH) << it->address << std::setw(COL_WIDTH) << it->op;
147:
148:         if (it->operand != NIL)
149:         {
150:             os << it->operand;
151:         }
152:
153:         os << std::endl;
154:     }
155:
156:     return os.str();
157: }
158:
159: /**
160:  * @brief Generate a new instruction and
161:  * add it to the instruction table.
162:  *
163:  * @param op The op (ADD, SUB, EQU, etc...)
164:  * @param operand The operand (an integer, memory address, etc...)
165:  */
166: void SymbolTable::gen_instr(std::string op, int operand)
167: {
168:     Instr *instr = new Instr(op, operand);
169:
170:     this->instructions.push_back(*instr);
171: }
172:
173: /**
174:  * @brief Push an address onto the jumpstack
175:  *
176:  * @param address the address
177:  */
178: void SymbolTable::push_jumpstack(int address)
179: {
180:     this->jumpstack.push_back(address);
181: }
182:
183: /**
184:  * @brief Used to close off JUMP instructions.
185:  * Find the previous JUMP instruction and fill out its
186:  * operand with the current instruction address.
187:  *
188:  * @param jump_addr The address of the previous JUMP instruction
189:  */
190: void SymbolTable::back_patch(int jump_addr)
191: {
192:     const int addr = jumpstack.back();
```

```
193:     jumpstack.pop_back();
194:
195:     if (this->instructions.size() >= addr)
196:     {
197:         this->instructions.at(addr - 1).operand = jump_addr;
198:     }
199:     else
200:     {
201:         // TODO: ERROR
202:     }
203: }
204:
205: /**
206:  * @brief Get a particular token’s memory address (e.g: 500, 5001...)
207:  *
208:  * @param token the token
209:  * @return int the token’s address. 0 if unsuccessful.
210:  */
211: int SymbolTable::get_address(Lexer::Token token)
212: {
213:     return lookup(token);
214: }
215:
216: /**
217:  * @brief Get the current memory address (e.g: 5000, 5001...)
218:  *
219:  * @return int the memory address
220:  */
221: int SymbolTable::get_mem()
222: {
223:     return this->memaddress;
224: }
225:
226: /**
227:  * @brief Get the current instruction address (i.e: 1, 2, 3...)
228:  *
229:  * @return int The instruction address
230:  */
231: int SymbolTable::get_instr_address() const
232: {
233:     return instr_address;
234: }
235:
236: /**
237:  * @brief Push a value onto the typestack
238:  *
239:  * @param type the type
240:  */
241: void SymbolTable::push_typestack(std::string type)
242: {
243:     this->typestack.push(type);
244: }
245:
246: /**
247:  * @brief Pop a value from the typestack
248:  *
249:  * @return true if successful
250:  * @return false if stack is empty
251:  */
252: bool SymbolTable::pop_typestack()
253: {
254:     bool success = false;
255:
256:     if(!this->typestack.empty())
257:     {
258:         success = true;
259:         this->typestack.pop();
260:     }
261:
262:     return success;
263: }
264:
265: /**
266:  * @brief Retrieve the top element on the stack
267:  *
268:  * @return std::string the type
269:  */
270: std::string SymbolTable::top_typestack() const
271: {
272:     return this->typestack.top();
273: }
274:
275: /**
276:  * @brief Get the type of the given Token
277:  *
278:  * @param token
279:  * @return std::string
280:  */
281: std::string SymbolTable::get_type(Lexer::Token token) const
282: {
283:     std::string type = "";
284:
285:     for(Symbol s : this->table)
286:     {
287:         if(s.token.lexeme == token.lexeme)
288:         {
```

```
289:             type = s.type;
290:             break;
291:         }
292:     }
293:
294:         if (token.token == "Integer")
295:         {
296:                 type = "int";
297:         }
298:
299:     return type;
300: }
301:
302: /**
303:  * @brief Check if typestack is empty
304:  *
305:  * @return true if empty
306:  * @return false if not empty
307:  */
308: bool SymbolTable::typestack_empty() const
309: {
310:     return this->typestack.empty();
311: }
```