

```
1: #include "SymbolTable.h"
2:
3: /**
4:  * @brief Construct a new SymbolTable object
5:  */
6: SymbolTable::SymbolTable() : memaddress(5000) {}
7:
8: /**
9:  * @brief Destroy the SymbolTable object
10:  */
11: SymbolTable::~SymbolTable()
12: {
13:     instr_address = 1;
14: }
15:
16: /**
17:  * @brief Increment the current memory address by one
18:  */
19: void SymbolTable::incrementMem()
20: {
21:     ++this->memaddress;
22: }
23:
24: /**
25:  * @brief Insert a Symbol into the symbol table
26:  *
27:  * @param t The token object
28:  * @return true if successful
29:  * @return false if unsuccessful
30:  */
31: bool SymbolTable::insert(Lexer::Token t, std::string type)
32: {
33:     bool success = false;
34:
35:     if (!lookup(t))
36:     {
37:         Symbol *s = new Symbol(t, this->memaddress, type);
38:         this->table.push_back(*s);
39:         incrementMem();
40:         success = true;
41:     }
42:
43:     return success;
44: }
45:
46: /**
47:  * @brief Check to see if an identifier already exists
48:  * in the symbol table.
49:  *
50:  * @param id The identifier (lexeme)
51:  * @return address if existing, 0 if not
52:  */
53: int SymbolTable::lookup(Lexer::Token t)
54: {
55:     std::vector<Symbol>::iterator it = this->table.begin();
56:     bool found = false;
57:
58:     while (!found && it != this->table.end())
59:     {
60:         if (it->token.lexeme == t.lexeme && it->token.token == t.token)
61:         {
62:             found = true;
63:         }
64:         else
65:         {
66:             ++it;
67:         }
68:     }
69:
70:     return found ? it->address : 0;
71: }
72:
73: /**
74:  * @brief Remove a Symbol from the symbol table.
75:  *
76:  * @param id The identifier (lexeme)
77:  * @return true if successful
78:  * @return false if unsuccessful
79:  */
80: bool SymbolTable::remove(Lexer::Token t)
81: {
82:     bool success = false;
83:     int pos = 0;
84:
85:     if (lookup(t))
86:     {
87:         std::vector<Symbol>::const_iterator it = this->table.begin();
88:         while (!success && it != this->table.end())
89:         {
90:             if (it->token.lexeme == t.lexeme)
91:             {
92:                 this->table.erase(this->table.begin() + pos);
93:                 success = true;
94:             }
95:             else
96:             {
```

```
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:     return type;
295: }
296:
297: /**
298:  * @brief Check if typestack is empty
299:  *
300:  * @return true if empty
301:  * @return false if not empty
302:  */
303: bool SymbolTable::typestack_empty() const
304: {
305:     return this->typestack.empty();
306: }
```