Bachelor Project in Compiler Construction

Symbol Table

May 2019

Report from group 9:

Anton Nørgaard (antno16), Bjørn Glue Hansen (bhans09) & Thor Skjold Haagensen (thhaa16)

1 Introduction

1.1 Implementation Status

The current implementation is based on not knowing the language that is going to be compiled. Some assumptions regarding scope rules has been made and can be found in section 2.1. The program functions as intended and passes all tests. Most parameters are not currently tested for correct input, e.g. that pointers are not NULL. The symbol table doesn't currently have a method for deallocating memory, it will be added later.

2 The Symbol Table

2.1 Scope Rules

The current implementation is made without knowledge of the language to be compiled. It does have some assumptions about about the scope rules.

Rule	Description
Recursive scope expansion	Only symbols in the current scope, main scope or in the path from the current scope to the main scope, can be used.
Most local first	If there are multiple symbols with the same identifier, the most local symbol is selected.
Cannot init symbol if another symbol has the same identifier	A symbol can not be initialized in a scope that already has a symbol with the same identifier.

2.2 Data Structure

A single scope is represented as a hash table. This allows for O(1) look up and insertion time in a single scope, assuming the symbols have been distributed evenly. The scopes are connected in a tree structure, but it is only possible to traverse the tree in one direction. A node can access its parent, but a parent can not access its child nodes. A leaf is the most local scope, while the root is the global scope.

When looking up a symbol, it is first searched for in the current scope, if it is not found there it is looked up in the parent of the scope. This continues until either a symbol has been found or after checking the global scope. If the symbol could not be found in any of the scopes, then it is assumed to be an error in the source code.

2.3 implementation specifics

Currently a symbol can only have int as a value. This is mainly because the symbol table is currently implemented without taking into account the language that it should compile. In the future the value will likely be a union of different types that can be expected and an enum will be added to keep track of the current type.

The hash table starts at a small size of 11 and will resize itself when this condition is met $T/A \geq log_{10}(A)$, with A being the size of the internal array and T the number of symbols in the table. The reason for making the upper bound dynamic, is to ensure that the hash table resizes quickly at a small size, but allow for a higher number of collisions and less wasted space in the array at a larger size.

The size of the array is always a prime to avoid common divisors between the hashed string and the modulus when calculating the hashed index. The prime is found by first multiplying the original array size by some constant $n_{i+1} = cA$ and then find the lowest prime larger than the result $p_{i+1} \geq n_{i+1}$. A number is checked if prime by using the naive algorithm with some optimizations, resulting in a run time of $O(\sqrt{n})$. Compared to rehashing all of the symbols for the new array, finding the prime will not be the dominant aspect of resizing and doesn't need further optimization.

2.4 Tests, their motivations and results

We test the tables' methods using the test_symbol_table.c file. Here, we create a root table and two child tables, using the scope_symbol_table() method. We test if any of the child tables has lost the reference to the parent. It is reasonable to expect that this behavior will be similar for larger symbol tables as well.

In this test, we also ensure that in the event that two tables have a variable with the same name, get_symbol() finds the first table that has the variable. Specifically, in the event that we look for a symbol named x, starting from child table 1, it should be found in child table 1 and since child 2 in this test does not have a symbol named x, it should be found in the root table. We again use the same principle as before i.e it will work for larger tables.

On the next page is a list of some of the tests in test_symbol_table.c and their results.

Test	Expected Result	Actual Result
put symbol in table	dump of table has the symbol	dump of table has the symbol
get previous symbol	the symbol is printed	the symbol is printed
put symbol with same name in child table	dump has a symbol in both tables	dump has a symbol in both tables
init multiple scopes from same table	both scopes point to the same table	both scopes point to the same table
put symbol in child that has same name as symbol in parent	dump shows that both symbols are in the expected scopes	dump shows that both symbols are in the expected scopes
get symbol that has the same identifier as a symbol in the parent scope	get method returns the correct symbol	get method returns the correct symbol
get symbol that only exists in parent scope	get the symbol from the parent	got the symbol from the parent.
try to get symbol that is in a different child of the parent scope	Can not find child and instead get an error message	Can not find child and instead get an error message

3 Conclusion

The symbol table passes all tests and works as currently intended. It will need to be modified in the future to support the environment of the chosen language and the usage by other modules. More error checking could be added to help understand potential problems when the symbol table is used by other modules. A method for deallocation should also be added to avoid memory leaks.

A Source Code

Makefile

```
GCC = gcc
SRC = symbol.c array_list.c memory.c prime_generator.c
IDIR = include
OBJS := $(patsubst %.c, %.o, $(SRC))
OBJ_TEST1 = test_symbol_table.o
OBJ_TEST2 = test_symbol_resize.o
CFLAGS = -I$(IDIR) -std=c11 -Wall -Wextra -pedantic

.PHONY: compiler test2 clean

all: compiler

%.o: %.c
  $(GCC) $(CFLAGS) -c -o $@ $<

compiler: $(OBJ_TEST1) $(OBJS)
  $(GCC) $(OBJ_TEST1) $(OBJS) $(CFLAGS) -o compiler -lm

test2: $(OBJ_TEST2) $(OBJS)
  $(GCC) $(OBJ_TEST2) $(OBJS) $(CFLAGS) -o compiler -lm

clean:
  rm -f $(OBJ_TEST1) $(OBJ_TEST2) $(OBJS) compiler
```

memory.h

```
#ifndef __MEMORY_H__
#define __MEMORY_H__

void *Malloc(unsigned n);

#define NEW(type) (type *)Malloc(sizeof(type))

#endif
```

memory.c

```

#ifdef __APPLE__
#include <malloc/malloc.h>
#else
#include <malloc.h>
#endif

#include <stdio.h>
#include <stdlib.h>

void *Malloc(unsigned n)
{
  void *p;
  if(!(p = malloc(n)))
  {
    fprintf(stderr,"Malloc(%d) failed.\n",n);
```

```
    fflush(stderr);
    abort();
  }
  return p;
}
```

array_list.h

```
#ifndef __H_ARRAY_LIST__
#define __H_ARRAY_LIST__

#define AL_GET(T, V, L, I) T* result = al_get(L, I); V = *result // computes the result and dereferences the pointer, saving the job of having to do this explicitly

typedef struct al_list_ array_list;
//typedef enum { NUM, REal, CHAR, VOID} al_kind;

struct al_list_
{
    int size;
    int max_size;
    int element_size;
    void *array;
    int (*next_size)(int size);
};

/* Methods */
array_list *al_init_list(int initial_max_size, int element_size);
void al_add(array_list *list, void *value);
void al_set_max(array_list *list);
void al_set(array_list *list, int index, void *value);
void *al_get(array_list *list, int index);
int _al_next_size(int size);
void _al_resize(array_list *list);
void al_clean(array_list *list, void (*clean_value)(void *));

#endif
```

array_list.c

```
#include "array_list.h"
#include "memory.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

array_list *al_init_list(int initial_max_size, int element_size)
{
    array_list *list = NEW(array_list);
    void *array = Malloc(element_size * initial_max_size);
    memset(array, 0, element_size * initial_max_size);

    list->size = 0;
    list->max_size = initial_max_size;
    list->element_size = element_size;
    list->array = array;
    list->next_size = &_al_next_size;
```

```
18
19      return list;
20  }
21
22  void al_add(array_list *list, void *value)
23  {
24      list->size += 1;
25      if (list->size > list->max_size)
26      {
27          _al_resize(list);
28      }
29
30      al_set(list, list->size - 1, value);
31  }
32
33  void al_set_max(array_list *list)
34  {
35      list->size = list->max_size;
36  }
37
38  void al_set(array_list *list, int index, void *value)
39  {
40      if (index >= list->size)
41      {
42          fprintf(stderr, "array_list index to large %d >= %d.\n", index, list->
        size);
43          fflush(stderr);
44          abort();
45      }
46
47      char *to;
48
49      to = list->array;
50      to += index * list->element_size;
51
52      memcpy(to, value, list->element_size);
53  }
54
55  void *al_get(array_list *list, int index)
56  {
57      if (index >= list->size)
58      {
59          fprintf(stderr, "array_list index to large %d >= %d.\n", index, list->
        size);
60          fflush(stderr);
61          abort();
62      }
63
64      char *ptr = list->array;
65
66      ptr += index * list->element_size;
67      return ptr;
68  }
69
70  int _al_next_size(int size)
71  {
72      return size * 2;
```

```
}

void _al_resize(array_list *list)
{
    int new_size = list->next_size(list->max_size);
    void *new_array = Malloc(new_size * list->element_size);

    memcpy(new_array, list->array, list->max_size * list->element_size);
    free(list->array);

    list->max_size = new_size;
    list->array = new_array;
}

void al_clean(array_list *list, void (*clean_value)(void *))
{
    if (list->element_size == sizeof(void *) && clean_value != NULL)
    {
        void **element = list->array;
        for (int index = 0; index < list->size; index++)
        {
            clean_value(element);
            element++;
        }
    }

    free(list->array);
    free(list);
}
```

prime_generator.h

```
#ifndef __H_PRIME_GENERATOR__
#define __H_PRIME_GENERATOR__


int prime_next(int lower_bound);
int prime_check(int num);

#endif
```

prime_generator.c

```
#include "prime_generator.h"
#include <math.h>

int prime_next(int lower_bound)
{
    if (lower_bound % 2 == 0)
    {
        lower_bound++;
    }

    while (!prime_check(lower_bound))
    {
        lower_bound++;
    }

```

```
    return lower_bound;
}

int prime_check(int num)
{
    if (num == 2)
    {
        return 1;
    }
    else if (num % 2 == 0)
    {
        return 0;
    }
    else
    {
        int sq = sqrt(num);
        for (int divisor = 3; divisor <= sq; divisor += 2)
        {
            if (num % divisor == 0)
            {
                return 0;
            }
        }
    }

    return 1;
}
```

symbol.h

```

#ifndef __H_SYMBOL_TABLE__
#define __H_SYMBOL_TABLE__

#include "array_list.h"

// elements in table compared to size of array before resizing
#define SYM_RESIZE_RATIO 1.3
// how much larger the prime of the next size of the array should at least be
#define SYM_RESIZE_SCALE 2
// max number of charachters used to calculate hash
#define MAX_VARIABLE_SIGNIFICANCE (32)

typedef struct _sym_symbol symbol;
typedef struct _sym_table symbol_table;

struct _sym_symbol {
  char *name;
  int value;
  symbol* next;
};

struct _sym_table {
  int size;
  array_list *hash_table;
  symbol_table *parent;
};

```

```
// Get a numerical representation of the string str
long long int Hash(char *str);

// Get the hashed index of str for hash_table
int hash_index(array_list *hash_table, char *str);

// Create a new symbol table
symbol_table *init_symbol_table();

// Create a new scope with table as parant
symbol_table *scope_symbol_table(symbol_table *table);

// Add identifier name to the scope of table
symbol *put_symbol(symbol_table *table, char *name, int value);

// Change identifier name in scope or from parent scope
void sym_assign(symbol_table *table, char *name, int value);

// Create a new symbol with identifier name
symbol *_sym_init_symbol(char *name, int value);

// Increase the size of the symbol table and resize if necesary
void _sym_increment_size(symbol_table *table);

// Get the first symbol that has idenfier name in the table or its parents
symbol *get_symbol(symbol_table *table, char *name);

// Resize the hash table to a new prime
void _sym_resize(symbol_table *table);

// Print a string representation of the symbol table and its parents
void dump_symbol_table(symbol_table *table);

// TODO: method for freeing memory

#endif
```

symbol.c

```

#include "symbol.h"
#include "prime_generator.h"
#include <memory.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <string.h>

long long int Hash(char *str)
{
    long long int sum = 0;
    int current;
    int i = 0;
    sum = (int) str[i];
    i++;

    while(str[i] != '\0' && i < MAX_VARIABLE_SIGNIFICANCE)
    {
```

```
20          current = (int) str[i];
21          sum = sum << 1;
22          sum = sum + current;
23
24          i++;
25      }
26
27      return sum;
28  }
29
30  int hash_index(array_list *hash_table, char *str)
31  {
32      long long int hash = Hash(str);
33      int size = hash_table->size;
34
35      return hash % size;
36  }
37
38  symbol_table *init_symbol_table()
39  {
40      symbol_table *tbl = NEW(symbol_table);
41      tbl->hash_table = al_init_list(11, sizeof(void*)); // 11 is prime we choose
        this to avoid colisions -> good for the report
42      tbl->parent = NULL;
43      tbl->size = 0;
44
45      al_set_max(tbl->hash_table);
46      return tbl;
47  }
48
49  symbol_table *scope_symbol_table(symbol_table *table)
50  {
51      symbol_table *new = init_symbol_table();
52      new->parent = table;
53      return new;
54  }
55
56  symbol *put_symbol(symbol_table *table, char *name, int value)
57  {
58      int name_hash = hash_index(table->hash_table, name);
59      symbol *ptr;
60
61      AL_GET(symbol *, ptr, table->hash_table, name_hash);
62
63      if(ptr == NULL)
64      {
65          symbol* sprt = _sym_init_symbol(name, value);
66
67          al_set(table->hash_table, name_hash, &sprt);
68          _sym_increment_size(table);
69
70          return sprt;
71      }
72
73      while(ptr->next != NULL && strcmp(ptr->name, name) != 0)
74      {
75          ptr = ptr->next;
```

```
76      }
77
78      // ERROR: id already exists;
79      if(strcmp(ptr->name,name) == 0)
80      {
81          fprintf(stderr, "Identifier: %s, already exists.\n", ptr->name);
82          // abort(); TODO make compiler abort later;
83          return NULL;
84      }
85
86      symbol *sptr = _sym_init_symbol(name, value);
87
88      ptr->next = sptr;
89      _sym_increment_size(table);
90
91      return sptr;
92  }
93
94  void sym_assign(symbol_table *table, char *name, int value)
95  {
96      symbol *ptr = get_symbol(table, name);
97      // TODO: check values match
98
99      if (ptr != NULL)
100     {
101         ptr->value = value;
102     }
103 }
104
105 symbol *_sym_init_symbol(char *name, int value)
106 {
107     symbol* sprt = NEW(symbol);
108     sprt->name = name;
109     sprt->next = NULL;
110     sprt->value = value;
111
112     return sprt;
113 }
114
115 void _sym_increment_size(symbol_table *table)
116 {
117     table->size += 1;
118     double ratio = table->size / (double)table->hash_table->size;
119     double tolerence = log10(table->hash_table->size);
120
121     if (ratio >= tolerence)
122     {
123         _sym_resize(table);
124     }
125 }
126
127 symbol *get_symbol(symbol_table *table, char *name)
128 {
129     while(table != NULL)
130     {
131         int name_hash = hash_index(table->hash_table, name);
132         symbol *ptr;
```

```
133
134         AL_GET(symbol *, ptr, table->hash_table, name_hash);
135
136         if(ptr != NULL)
137         {
138             while(ptr->next != NULL && strcmp(ptr->name, name) != 0)
139             {
140                 ptr = ptr->next;
141             }
142
143             if(strcmp(ptr->name,name) == 0)
144             {
145                 return ptr;
146             }
147         }
148
149         table = table->parent;
150     }
151
152     fprintf(stderr, "Identifier: %s, doesn't exist.\n", name);
153     // abort(); TODO make compiler abort later;
154
155     return NULL;
156 }
157
158 void _sym_resize(symbol_table *table)
159 {
160     int new_size;
161     new_size = prime_next(table->hash_table->size * SYM_RESIZE_SCALE);
162
163     array_list *new_list, *old_list;
164     new_list = al_init_list(new_size, sizeof(void *));
165     al_set_max(new_list);
166
167     old_list = table->hash_table;
168     table->hash_table = new_list;
169     table->size = 0;
170
171     symbol *ptr;
172     for (int i = 0; i < old_list->size; i++)
173     {
174         AL_GET(symbol *, ptr, old_list, i);
175         while(ptr != NULL)
176         {
177             put_symbol(table, ptr->name, ptr->value);
178             ptr = ptr->next;
179         }
180     }
181
182     al_clean(old_list, NULL);
183 }
184
185 void dump_symbol_table(symbol_table *table)
186 {
187     int table_index = 1;
188
189     while(table != NULL){
```

```
        fprintf(stderr, "Table %d\n", table_index);

        for (int index = 0; index < table->hash_table->size; index++)
        {
            symbol *ptr;
            AL_GET(symbol *, ptr, table->hash_table, index);
            fprintf(stderr, "%d ", index);

            if (ptr == NULL)
            {
                fprintf(stderr, "NULL\n");
                continue;
            }

            fprintf(stderr, "(%s, %d)", ptr->name, ptr->value);
            while(ptr->next != NULL)
            {
                ptr = ptr->next;
                fprintf(stderr, " -> (%s, %d)", ptr->name, ptr->value);
            }

            fprintf(stderr, "\n");
        }

        fprintf(stderr, "----------------\n");
        fprintf(stderr, "\n");
        table = table->parent;
        table_index++;
    }

    // TODO: method for freeing memory
}
```

test_array_list.c

```
#include "array_list.h"
#include <stdio.h>

int main()
{
    array_list *list = al_init_list(10, sizeof(int));
    al_set_max(list);
    int temp = 10;
    al_set(list, 2, &temp);

    int *result = al_get(list, 2);
    if (temp == *result)
        fprintf(stderr, "List working\n");
    else
        fprintf(stderr, "List not working: got %d\n", *result);


    return 0;
}
```

test_symbol_resize.c

```
#include "symbol.h"
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *random_name(int size)
{
    char *str = malloc(size + 1);
    char *ptr = str;
    while(size){
        memset(ptr, 'a' + (rand() % 26), 1);
        ptr++;
        size--;
    }

    memset(ptr, 0, 1);
    return str;
}

int main()
{
    symbol_table *tbl = init_symbol_table();

    // make sure no symbol got lost in resizing
    fprintf(stderr, "\nChecking all symbols added\n");
    for (char value = 'a'; value <= 'z'; value++)
    {
        char *str = malloc(sizeof(char) *3);
        memset(str, 0, sizeof(*str) * 3);
        memcpy(str, &value, sizeof(char));
        put_symbol(tbl, str, (int)value);
    }

    int count = 0;

    for (char value = 'a'; value <= 'z'; value++)
    {
        char *str = malloc(sizeof(char) *3);
        memset(str, 0, sizeof(*str) * 3);
        memcpy(str, &value, sizeof(char));
        symbol *ptr = get_symbol(tbl, str);

        if (ptr == NULL)
        {
            fprintf(stderr, "Name not found %s\n", str);
            count++;
        }
    }

    if (count == 0)
    {
        fprintf(stderr, "No input lost!\n\n");
    }

    // Check spread of random names
    fprintf(stderr, "Checking resize with large quantity of names.\n");
    symbol_table *tbl_rnd = init_symbol_table();

```

```
    for (int i = 0; i < 1000; i++)
    {
        char *str = random_name(10);
        put_symbol(tbl_rnd, str, rand() % 26);
    }

    dump_symbol_table(tbl_rnd);
    fprintf(stderr, "\n");

    // Check symbol name length
    fprintf(stderr, "Checking name lenght\n");
    symbol_table *tbl_large = init_symbol_table();

    for (int i = 0; i < 10; i++)
    {
        char *str = random_name(70);
        put_symbol(tbl_large, str, rand() % 26);
    }

    dump_symbol_table(tbl_large);
    fprintf(stderr, "\n");
}
```

test_symbol_table.c

```
#include "symbol.h"
#include <stdio.h>

int scopeTest(){

    fprintf(stderr, "Testing scope_symbol\n");
    //creating root scope
    symbol_table *tbl = init_symbol_table();

    //creating leaf scopes and testing if successful
    symbol_table *child1 = scope_symbol_table(tbl);
    symbol_table *child2 = scope_symbol_table(tbl);

    if(child1->parent != tbl)
        fprintf(stderr, "scope_symbol_table failed creating child1:\n ");
    else
        fprintf(stderr, "child1 created \n");
    if(child2->parent != tbl)
        fprintf(stderr, "scope_symbol_table failed creating child2:\n ");
    else
        fprintf(stderr, "child2 created \n");

    fprintf(stderr, "\n");
    fprintf(stderr, "starting put/get test on multiple tables \n");

    // inserting values in root table
    symbol *x = put_symbol(tbl,"x",10);

    //symbol *q = put_symbol(tbl,"q",9);
    put_symbol(tbl,"q",9);

    // inserting values in child1
    symbol *x1 = put_symbol(child1,"x",8);
```

```
34
35      fprintf(stderr, "Show that both symbols are in the expected scopes \n");
36      dump_symbol_table(child1);
37
38      // inserting values in child2
39      put_symbol(child2,"y",7);
40      symbol *z = put_symbol(child2,"z",6);
41
42      // looking for symbol x from child1
43      fprintf(stderr, "Get identifier x from child1 \n");
44      symbol *x2 = get_symbol(child1,"x");
45      if(x2 == x1)
46          fprintf(stderr, "x found in child1: \n");
47      else
48          fprintf(stderr, "x NOT found in child 1: \n");
49
50      // looking for x from child2
51      fprintf(stderr, "Get identifier x in parent \n");
52      symbol *y2 = get_symbol(child2,"x");
53      if(y2 == x)
54          fprintf(stderr, "x found in root node from child2: \n");
55      else
56          fprintf(stderr, "x NOT found from child2: \n");
57
58      // looking for z from child1 in child2. Test should fail (out of scope)
59
60      fprintf(stderr, "Try to get identifier from a different child scope \n");
61      symbol *z1 = get_symbol(child1, "z");
62      if(z1 == z)
63          fprintf(stderr, "z found from child1: (test failed) \n");
64      else
65          fprintf(stderr, "z  NOT found from child1: (test success) \n");
66
67      fprintf(stderr, "\n");
68      fprintf(stderr, "-------------------\n");
69
70      return 0;
71  }
72
73  void dubplicateTest()
74  {
75      // test putting same identifier
76      fprintf(stderr, "Testing put existing identifier\n");
77      symbol_table *tbl = init_symbol_table();
78      put_symbol(tbl,"abc",10);
79      put_symbol(tbl,"abc",1);
80
81      // test changing value of identifier
82      fprintf(stderr, "Testing assign \n");
83      sym_assign(tbl,"abc",5);
84      dump_symbol_table(tbl);
85  }
86
87  void largeTest()
88  {
89      // Complex table
90      fprintf(stderr, "\n======================\n");
```

```
91      fprintf(stderr, "Testing complex table\n");
92      symbol_table *root = init_symbol_table();
93      symbol_table *depth1[3];
94      symbol_table *depth2[4];
95      symbol_table *depth3[2];
96
97      depth1[0] = scope_symbol_table(root);
98      depth1[1] = scope_symbol_table(root);
99      depth1[2] = scope_symbol_table(root);
100
101     depth2[0] = scope_symbol_table(depth1[0]);
102     depth2[1] = scope_symbol_table(depth1[1]);
103     depth2[2] = scope_symbol_table(depth1[1]);
104     depth2[3] = scope_symbol_table(depth1[1]);
105
106     depth3[0] = scope_symbol_table(depth2[2]);
107     depth3[1] = scope_symbol_table(depth2[2]);
108
109     // try to put a symbol with a negative value
110     fprintf(stderr, "put negative value\n");
111     symbol *a = put_symbol(root, "a", -2);
112     dump_symbol_table(root);
113
114     // put a symbol with same name start as another
115     put_symbol(depth2[2], "aa", 5);
116
117     symbol *bb = put_symbol(root, "bb", 100);
118     put_symbol(depth2[1], "b", 3);
119
120     symbol *ret;
121
122     fprintf(stderr, "traverse empty tables\n");
123     ret = get_symbol(depth3[0], "a");
124     if(ret == a)
125         fprintf(stderr, "a found in root node from depth3[0]: \n");
126     else
127         fprintf(stderr, "a NOT found from depth3[0]: \n");
128
129     fprintf(stderr, "get symbol with substring id of another symbol\n");
130     ret = get_symbol(depth2[2], "a");
131     if(ret == a)
132         fprintf(stderr, "a found in root node from depth2[2]: \n");
133     else
134         fprintf(stderr, "a NOT found from depth2[2]: \n");
135
136     fprintf(stderr, "get symbol with superstring id of another symbol \n");
137     ret = get_symbol(depth2[1], "bb");
138     if(ret == bb)
139         fprintf(stderr, "bb found in root node from depth2[1]: \n");
140     else
141         fprintf(stderr, "bb NOT found from depth2[1]: \n");
142 }
143
144 int main()
145 {
146     // Testing put_symbol() and get_symbol();
147     fprintf(stderr, "Testing put in single table\n");
```

```
148     symbol_table *tbl = init_symbol_table();
149     put_symbol(tbl, "abc", 10);
150 
151     dump_symbol_table(tbl);
152 
153     fprintf(stderr, "Testing get in single table\n");
154     symbol *ptr = get_symbol(tbl, "abc");
155 
156     if (ptr == NULL)
157         fprintf(stderr, "symbol: NULL\n");
158     else
159         fprintf(stderr, "symbol: (%s, %d)\n", ptr->name, ptr->value);
160 
161 
162     fprintf(stderr, "Testing put on child table\n");
163     symbol_table *tbl2 = scope_symbol_table(tbl);
164     put_symbol(tbl2, "abc", 2);
165 
166     dump_symbol_table(tbl2);
167 
168     scopeTest();
169     dubplicateTest();
170     largeTest();
171     return 0;
172 }
```