BHARATIYA VIDYA BHAVAN'S

SARDAR PATEL INSTITUTE OF TECHNOLOGY

(Empowered Autonomous Institute Affiliated to University of Mumbai)

[Knowledge is Nectar]

Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300108
Name	Hatim Sawai
Class and Batch	TE Computer Engineering - Batch C
Date	1-04-2024
Lab #	10
Aim	To Design a Linker/Loader
Objective	Implement file parsing, symbol table creation, and pretty table representation to analyze C code structure efficiently
Theory	**Linkers and Loaders:** In computer science, linkers and loaders are essential components of the software development process. They are responsible for converting source code into executable programs that can be run by a computer. Linkers and loaders perform tasks such as linking together multiple object files, resolving symbolic references, and loading the executable program into memory for execution. **Linking:** Linking is the process of combining multiple object files generated by a compiler into a single executable program. During linking, the linker resolves symbolic references between different object files, ensuring that all references to external functions and variables are correctly resolved. **Types of Linkers:** There are two main types of linkers: **Static Linker:** A static linker combines all the object files and libraries needed to create an executable program into a single executable file. It resolves all symbolic references at link time. The resulting executable file contains all the necessary code and data, making it self-contained and independent of external libraries. Static linking produces larger executable files but ensures that the program will run on any system without the need for external dependencies.

Dynamic Linker:

A dynamic linker links the program to external libraries at runtime rather than at link time.
It resolves symbolic references when the program is loaded into memory for execution.
Dynamic linking allows multiple programs to share a single copy of a library in memory, reducing memory usage and improving system performance.
Dynamic linking produces smaller executable files but requires the presence of external libraries on the system where the program will run.

Features of Linkers:

Symbol Resolution:
Linkers resolve symbolic references between different object files and libraries, ensuring that all references to external functions and variables are correctly resolved.
Reallocation:
Linkers perform reallocation of memory addresses to resolve conflicts between different object files and libraries that may have overlapping memory addresses.
Optimization:
Linkers perform optimization techniques such as dead code elimination and code compression to reduce the size of the executable file and improve program performance.
Relocation:
Linkers perform relocation of code and data to ensure that they are correctly positioned in memory when the program is loaded for execution.

Loading:

Loading is the process of transferring an executable program from disk into memory for execution. Loaders are responsible for loading the executable program into memory, resolving memory addresses, and initializing program variables before transferring control to the program's entry point.

Types of Loaders:

There are two main types of loaders:

Compile-time Loader:

A compile-time loader loads the entire program into memory before execution begins.
It is used in systems where memory space is not an issue and programs are small enough to fit entirely in memory.

Run-time Loader:

A run-time loader loads the program into memory on demand, loading only those parts of the program that are needed for execution.
It is used in systems where memory space is limited, and programs are too large to fit entirely in memory.
Run-time loaders use techniques such as demand paging and virtual memory to manage memory efficiently.

Reallocation:

Reallocation is the process of adjusting memory addresses to resolve conflicts between different object files and libraries that may have overlapping memory addresses.
It involves relocating code and data to ensure that they are correctly positioned in memory when the program is loaded for execution.

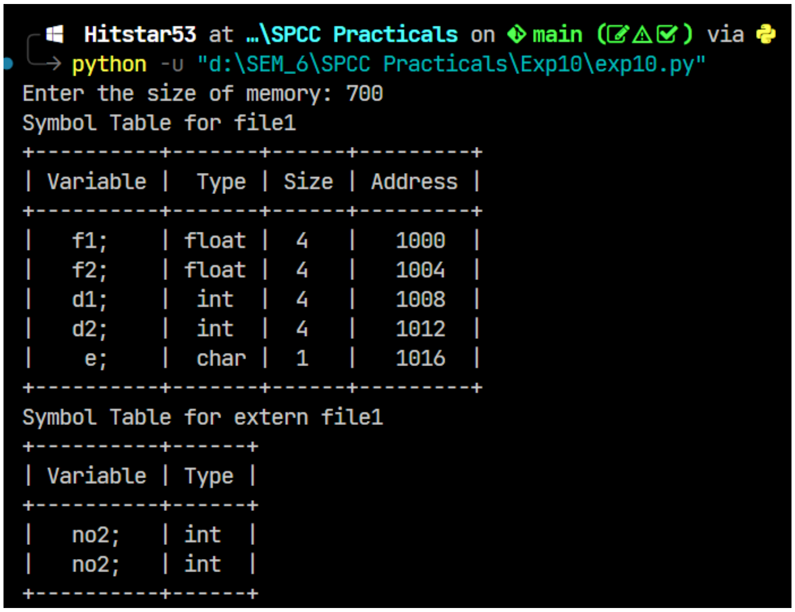
Loading:

Loading is the process of transferring an executable program from disk into memory for execution. Loaders are responsible for loading the executable program into memory, resolving memory addresses, and initializing program variables before transferring control to the program's entry point.

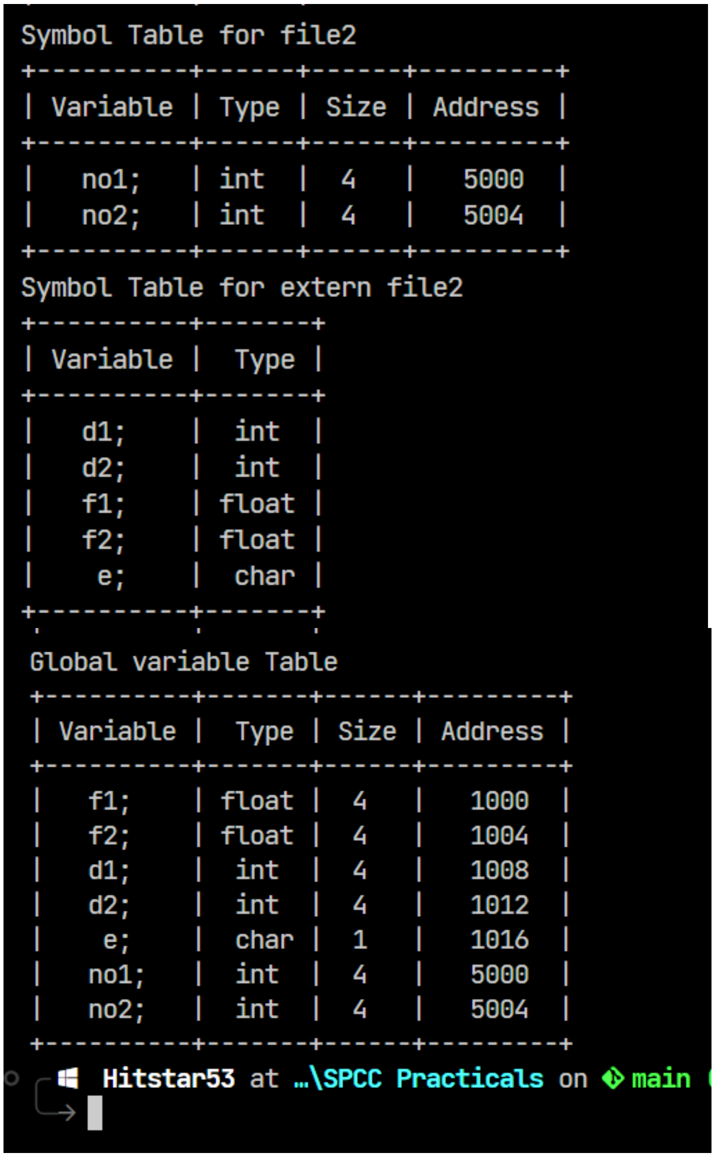
Implementation / Code

```python
import os
from prettytable import PrettyTable

class Variable:
    def __init__(self, name, type, size, address):
        self.name = name
        self.type = type
        self.size = size
        self.address = address

class SymbolTable:
    def __init__(self):
        self.variables = []

def read_file_size_and_content(filename):
    with open(filename, 'rb') as file:
        content = file.read()
        size = os.path.getsize(filename)
    return size, content.decode('utf-8')

def parse_variables(content, symbol_table, start_address):
    lines = content.split('\n')
    for line in lines:
        parts = line.split()
        if len(parts) >= 2:
            type, name = parts[:2]
            size = None
            if type == "int":
                size = 4
            elif type == "char":
                size = 1
            elif type == "float":
                size = 4
            elif type == "double":
                size = 8
            if size is not None:
                symbol_table.variables.append(Variable(name, type,
size, start_address))
                start_address += size

def parse_ext_variables(content, symbol_table):
    lines = content.split('\n')
    for line in lines:
        parts = line.split()
        if len(parts) >= 3 and parts[0] == "extern":
            symbol_table.variables.append(Variable(parts[2], parts[1],
0, -1))

def print_symbol_table(symbol_table):
    table = PrettyTable(["Variable", "Type", "Size", "Address"])
    for variable in symbol_table.variables:
        table.add_row([variable.name, variable.type, variable.size,
variable.address])
    print(table)

def print_symbol_tablet(symbol_table):
    table = PrettyTable(["Variable", "Type"])
    for variable in symbol_table.variables:
        table.add_row([variable.name, variable.type])
    print(table)

def print_symbol_tables(symbol_table1, symbol_table2):
    table = PrettyTable(["Variable", "Type", "Size", "Address"])
    for variable in symbol_table1.variables:
        table.add_row([variable.name, variable.type, variable.size,
variable.address])
    for variable in symbol_table2.variables:
        table.add_row([variable.name, variable.type, variable.size,
variable.address])
    print(table)

def main():
    memory_size = int(input("Enter the size of memory: "))

    size_a, content_a = read_file_size_and_content("Experiment
10\\file1.c")
    size_b, content_b = read_file_size_and_content("Experiment
10\\file2.c")

    total_size = size_a + size_b

    if total_size > memory_size:
        print("Insufficient memory.")
        return

    symbol_table_a = SymbolTable()
    symbol_table_b = SymbolTable()
    symbol_table_c = SymbolTable()
    symbol_table_d = SymbolTable()

    parse_variables(content_a, symbol_table_a, 1000)
    parse_variables(content_b, symbol_table_b, 5000)

    parse_ext_variables(content_a, symbol_table_c)
    parse_ext_variables(content_b, symbol_table_d)

    print("Symbol Table for file1")
    print_symbol_table(symbol_table_a)
    print("Symbol Table for extern file1")
    print_symbol_tablet(symbol_table_c)

    print("Symbol Table for file2")
    print_symbol_table(symbol_table_b)

    print("Symbol Table for extern file2")
    print_symbol_tablet(symbol_table_d)

    print("Global variable Table")
    print_symbol_tables(symbol_table_a, symbol_table_b)

if __name__ == "__main__":
    main()
```

Output



```
 Hitstar53 at …\SPCC Practicals on  main (✎⚠☑) via 🐍
 → python -u "d:\SEM_6\SPCC Practicals\Exp10\exp10.py"
Enter the size of memory: 700
Symbol Table for file1
+----------+-------+------+---------+
| Variable |  Type | Size | Address |
+----------+-------+------+---------+
|   f1;    | float |  4   |   1000  |
|   f2;    | float |  4   |   1004  |
|   d1;    |  int  |  4   |   1008  |
|   d2;    |  int  |  4   |   1012  |
|    e;    |  char |  1   |   1016  |
+----------+-------+------+---------+
Symbol Table for extern file1
+----------+------+
| Variable | Type |
+----------+------+
|   no2;   | int  |
|   no2;   | int  |
+----------+------+
```



```
Symbol Table for file2
+----------+------+------+---------+
| Variable | Type | Size | Address |
+----------+------+------+---------+
|   no1;   | int  |  4   |   5000  |
|   no2;   | int  |  4   |   5004  |
+----------+------+------+---------+
Symbol Table for extern file2
+----------+-------+
| Variable |  Type |
+----------+-------+
|   d1;    |  int  |
|   d2;    |  int  |
|   f1;    | float |
|   f2;    | float |
|    e;    |  char |
+----------+-------+
 Global variable Table
 +----------+-------+------+---------+
 | Variable |  Type | Size | Address |
 +----------+-------+------+---------+
 |   f1;    | float |  4   |   1000  |
 |   f2;    | float |  4   |   1004  |
 |   d1;    |  int  |  4   |   1008  |
 |   d2;    |  int  |  4   |   1012  |
 |    e;    |  char |  1   |   1016  |
 |   no1;   |  int  |  4   |   5000  |
 |   no2;   |  int  |  4   |   5004  |
 +----------+-------+------+---------+
Hitstar53 at …\SPCC Practicals on  main
```

Conclusion	In this experiment, I learned how to implement linker/loaders in python using c program files as input.
References	[1] Chatgpt, https://chat.openai.com/share/2f45bddf-4a87-49ef-b4c0-c0d2fa2e4fb7