Linker and Loader

By

Ujjwal Biswas

Introduction

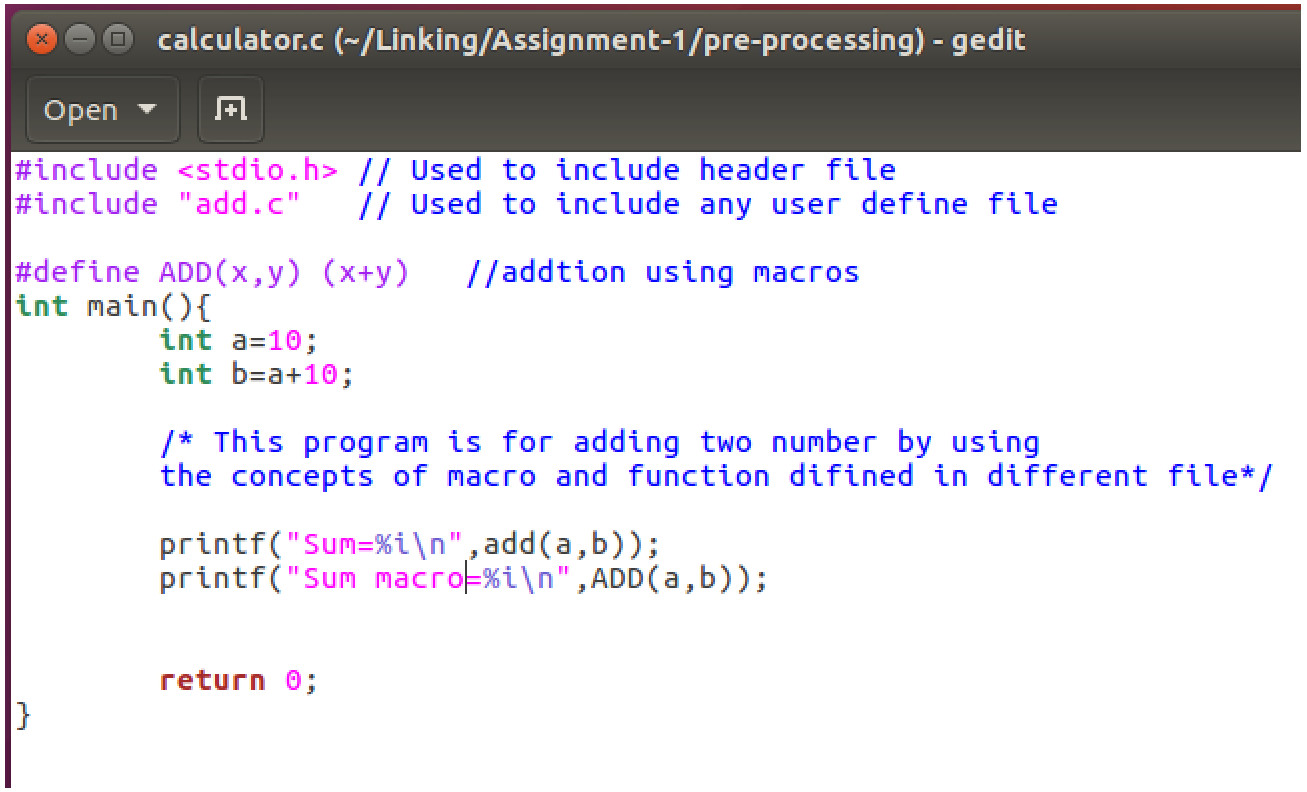
- Program must be brought into memory and placed within a process memory space for it to be executed
- Linkers and loaders prepare program to execution
- Linkers and loaders enable to binds programmer’s abstract names to concrete numeric values – addresses

Note: Need preprocessing, Compilation, assembly, linking, and loading

Pre-processing

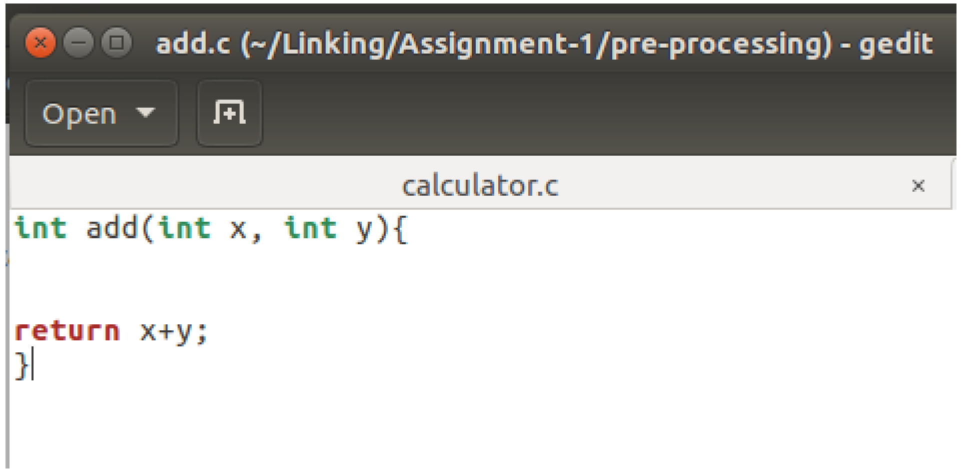
- This is the first phase of compilation process. This phase include:
  - Removal of Comments
  - Expansion of Macros
  - Expansion of the included files
- The preprocessed output is stored in the **filename.i**.
  - Translates the C source file main.c into an intermediate file main.i

Example Code for Analysis Pre-processing



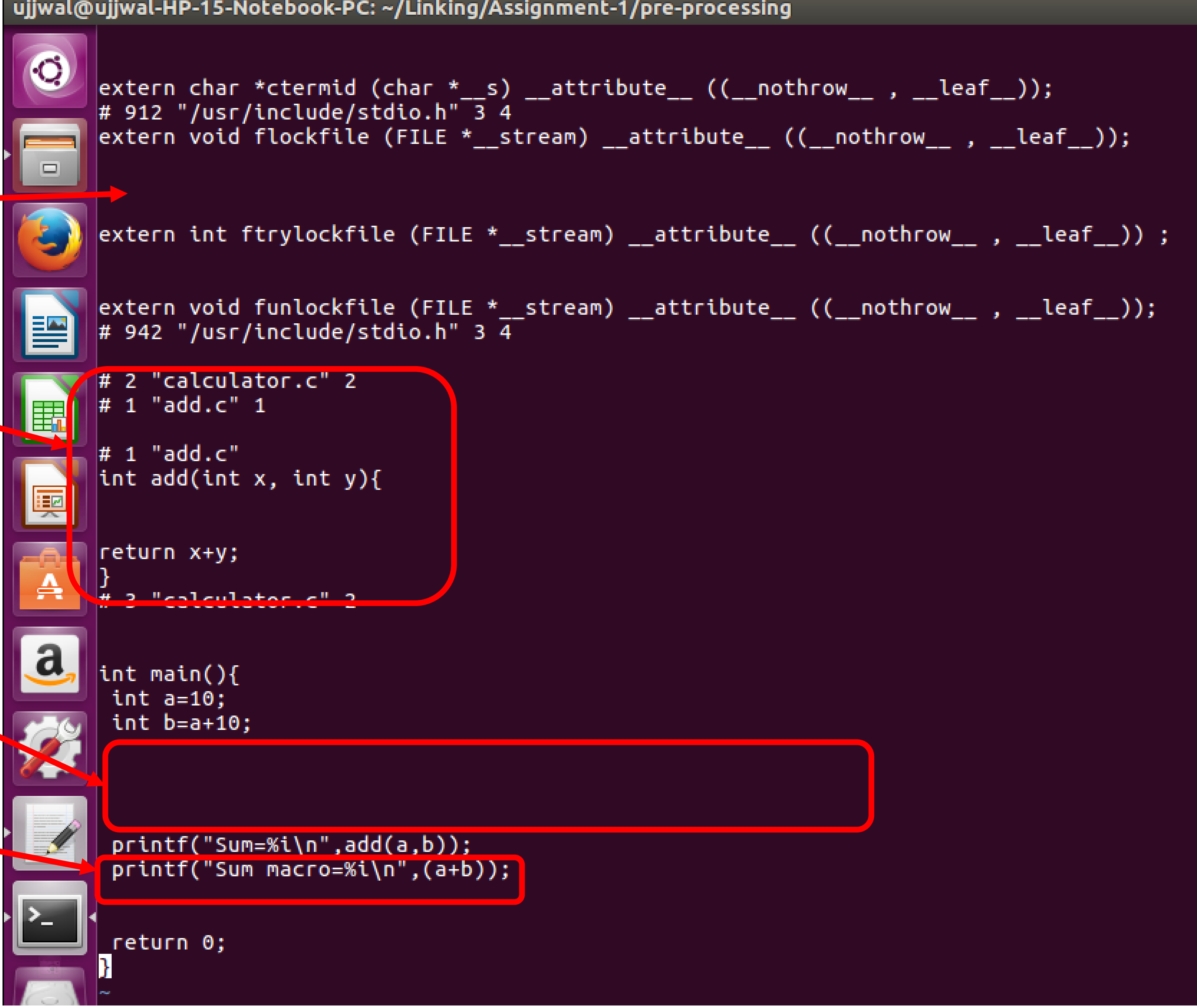
calculator.c (~/Linking/Assignment-1/pre-processing) - gedit
Open

```c
#include <stdio.h> // Used to include header file
#include "add.c"   // Used to include any user define file

#define ADD(x,y) (x+y)   //addtion using macros
int main(){
        int a=10;
        int b=a+10;

        /* This program is for adding two number by using
        the concepts of macro and function difined in different file*/

        printf("Sum=%i\n",add(a,b));
        printf("Sum macro=%i\n",ADD(a,b));


        return 0;
}
```



```c
int add(int x, int y){


return x+y;
}
```

- **#include<stdio.h>** is missing instead we see lots of code. So header files has been expanded and included in our source file
- **#include "add.c"** is missing instead we see add.c files has been expanded and included in our source file (add.c may be replace by add.h)
- Comments are stripped off
- printf contains now a + b rather than ADD(a, b) that's because macros have expanded



ujjwal@ujjwal-HP-15-Notebook-PC: ~/Linking/Assignment-1/pre-processing

```
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 912 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 942 "/usr/include/stdio.h" 3 4

# 2 "calculator.c" 2
# 1 "add.c" 1

# 1 "add.c"
int add(int x, int y){


return x+y;
}
# 3 "calculator.c" 2



int main(){
 int a=10;
 int b=a+10;





 printf("Sum=%i\n",add(a,b));
 printf("Sum macro=%i\n",(a+b));


 return 0;
}
~
```

Two Pass Linking

- Two Passes Logic
  - Pass 1:
    - assign addresses to all external symbols
  - Pass 2:
    - relocates the object code
    - updates symbol references
    - adjust memory addresses in code and data segment
    - writes to the output file

Pass 1: Program Logic

- Scan the input files to find the sizes of the segments
    - Input files (command line arguments, linker control files, object files, normal/shared libraries)
- Collect the definitions and references of all the symbols
    - exported- names of routines within the file that can be called from
    - imported- names of routines called from but not present in the file

Pass 1: Program Logic (con..)

- Creates segment table and symbol table
- Assigns numeric locations to symbols
- Determines the sizes and location of the segments (contiguous chunks of code or data) in the output address space

Pass 2: Program Logic

- Reads and relocates the object code
- Substitutes numeric addresses for symbol references
- Adjust memory addresses in code and data to reflect relocated segment addresses
- Writes the output file with:
  - header information
  - relocated segments
  - symbol table information
- Note: The above steps are for static linking if program uses dynamic linking then it is slightly different

Pass 2: Program Logic (con..)

Dynamic linking approach :

The runtime linker

- binds the dynamic symbols, stored in the symbol table
- generates small amounts of code in the output file which need to be called at program startup time for:
    - calling the routines in dynamic link libraries
    - initializing routines using array of pointers

Other Variants of Two Pass Linker

- Appears to work in one pass
- Uses buffer:
  - To store the content of the input file in memory or disk during the linking process
  - To read the buffered material later

Note: Since this is an implementation trick that doesn’t fundamentally affect the two-pass nature of linking

Type of Loaders

- Bootstrap loader
- Compile and Go loader
- Absolute loader
- Dynamic linking and loading

Bootstrap Loader

- Loads the first program to be run by the computer—usually an operating system
- Small program which is to be fitted in the ROM
- Task is to load the necessary portion of the operating system in the main memory
- Initial address at which the bootstrap loader is to be loaded is generally the lowest/highest location

Compile and Go’ Loader

- Also called as “assemble and go”
- Instruction is read line by line
- Machine code is obtained and it is directly put in the main memory at some known address
- Compiler which uses such “load and go” scheme (i.e. FORTRAN)

Compile and Go’ Loader (contd..)

- Advantage
  - simple, developing environment
- Disadvantage
  - whenever the assembly program is to be executed, it has to be assembled again
  - combination of assembler and loader activities, this combination program occupies large block of memory
  - programs have to be coded in the same language

Absolute Loader

- Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory
- Logic:
    - Header record is checked to verify that the correct program has been presented for loading
    - As each Text record is read, the object code it contains is moved to the indicated address in memory
    - When the End record is encountered, the loader jumps to the specified address to begin execution

Algorithm for an absolute loader

```
Begin
read Header record
verify program name and length
read first Text record
while record type is not ‘E’ do
   begin
   {if object code is in character form, convert into internal representation}
   move object code to specified location in memory
   read next object program record
   end
jump to address specified in End record
end
```

Advantages and Disadvantages

- Advantages
    - Simple to implement
    - Allows multiple programs or the source programs written in different languages
    - Task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory
- Disadvantages
    - The programmer’s duty to adjust all the inter segment addresses and manually do the linking activity
    - It is necessary for a programmer to know the memory management

Dynamic Linking and Loader

- Most common type of relocatable loader
- Allowing the programmer multiple procedure segments and multiple data segments and giving programmer complete freedom in referencing data or instruction contained in other segments
- Dynamic linking permits a program to load and unload routines at runtime, a facility that can otherwise be very difficult to provide
- To place the object code in the memory there are two situations:
  - Either the address of the object code could be absolute which then can be directly placed at the specified location or the address can be relative
  - If at all the address is relative then it is the assembler who informs the loader about the relative addresses

Reference

- ➢ Book: J R Levine, Linkers & Loaders.
- ➢ Book: J J Donovan, Systems Programming.
- ➢Book: L L Beck & D Manjula, Systems Software.