Computer Architecture & Real-Time Operating System

5. Building and Loading Programs

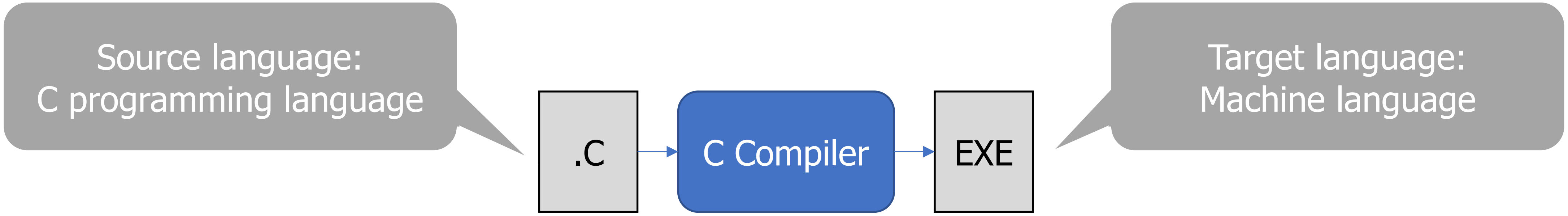
Prof. Jong-Chan Kim

Dept. Automobile and IT Convergence

국민대학교
KOOKMIN UNIVERSITY

Compiler

- Computer program that translates computer code written in one programming language into another language – Wikipedia
  - e.g.) Python to C compiler
- C compiler translates C code into machine code



- Most famous C compilers
  - GNU C Compiler (Now, GNU Compiler Collection)
  - Clang (based on LLVM (Low Level Virtual Machine))

GCC: A Brief Manual

- Single source file

```
$ gcc file.c
$ ./a.out
```

`a.out` is the default program name

```
$ gcc -o prog file.c
$ ./prog
```

`-o` option to specify the program name

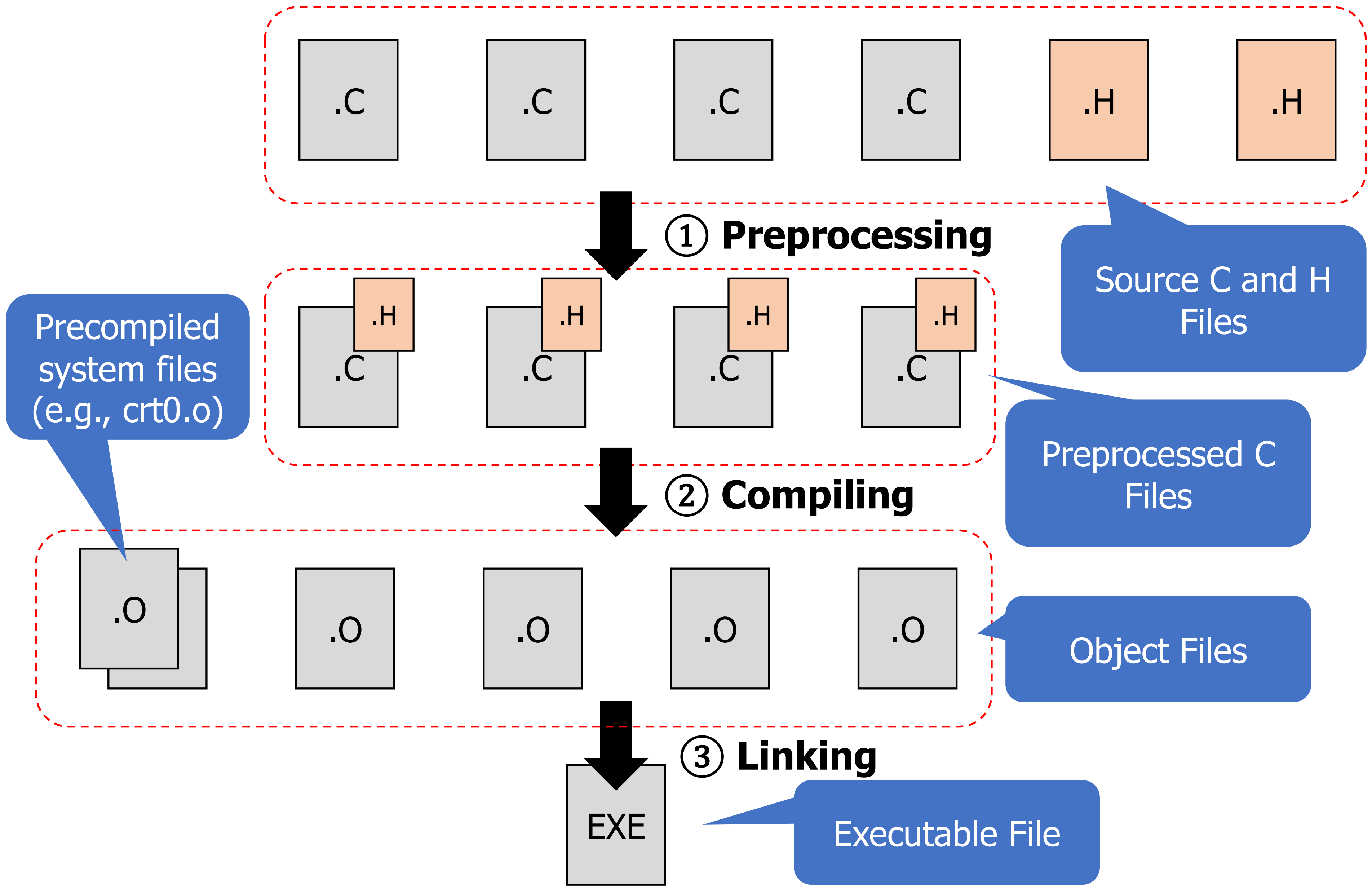
- Multiple source files

```
$ gcc -c file1.c
$ gcc -c file2.c
$ gcc -c file3.c
$ gcc -o prog file1.o file2.o file3.o
$ ./prog
```

`-c`: Do not link. Just compile C files into object files

Link user object files and system files altogether, producing an executable file

Three Steps of Build Process



Compiling and Linking: A Quick Tutorial

- Two C files (`main.c` and `func.c`) and one header file (`func.h`)

```
$ ls
func.c  func.h  main.c
$ gcc -c main.c
$ gcc -c func.c
$ gcc -o prog main.o func.o
$ ./prog
Hello World
$ ls
func.c  func.h  func.o  main.c  main.o  prog
```

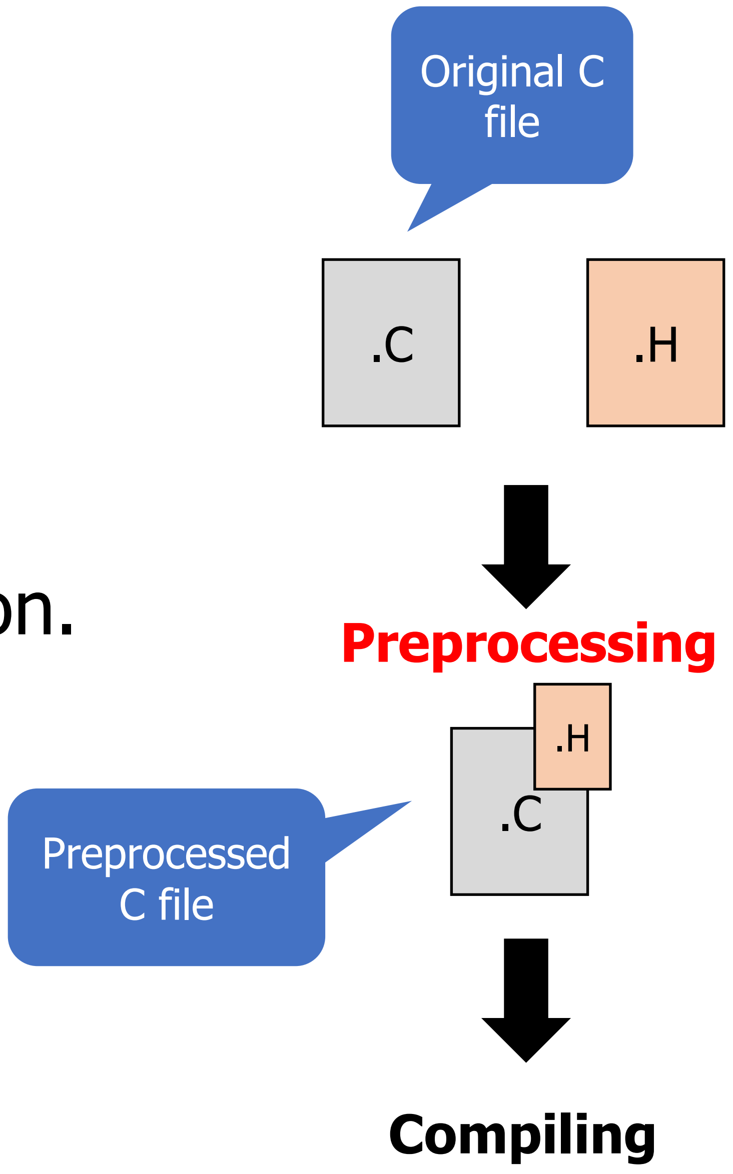
Preprocessing

- Removing comments
- Including header files
- Expanding macros
- Conditional compilation
- Line control

Preprocessed C files are gone after compilation.
However, you can check it by -E option

```
$ gcc -E file.c
```

-E : stop after preprocessing



Before and After Preprocessing

```
/*
 * This is the file with the main function
 *
 * Created by Jong-Chan Kim
 */

#include <stdio.h>
#include "func.h"

#define LEN_STR (30 + 1)
short g_sss = 10;
int g_global = 0;

int main(void)
{
#if 1
    char str[LEN_STR] = "Hello World\n";
#else
    char str[LEN_STR] = "Goodbye World\n";
#endif

    func(str);

    return 0;
}
```

main.c

```
$ gcc -E main.c > after.c
$ vi after.c
```

```
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c”
```

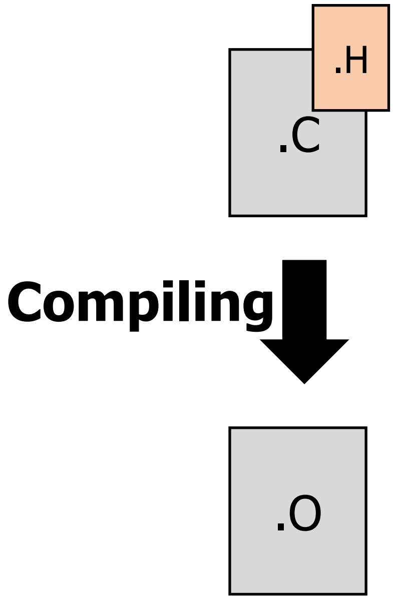
```
# 9 "main.c" 2


short g_sss = 10;
int g_global = 0;

int main(void)
{

    char str[(30 + 1)] = "Hello World\n";




    func(str);

    return 0;
}
```

Compiling

- Translating preprocessed C files into object files
- Object file
  - Written in CPU-dependent machine languages
  - Not portable across different CPU architectures
  - Contains both instructions and data

Can you tell instructions and data?

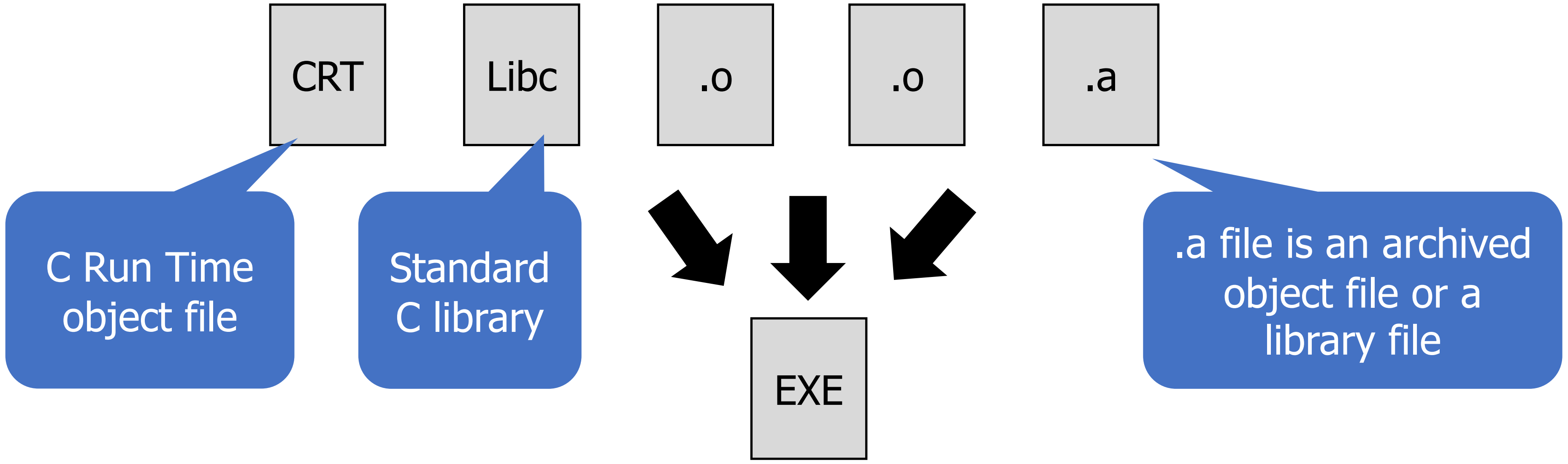
```
for (i = 1; i < 10; i++)
{
    sum += i;
}
```



- Compiler optimization options
  - Fast code vs fast compilation vs small object file vs …
  - -O option is used to instruct a specific code generation policy
  - For more information, `$ man gcc`

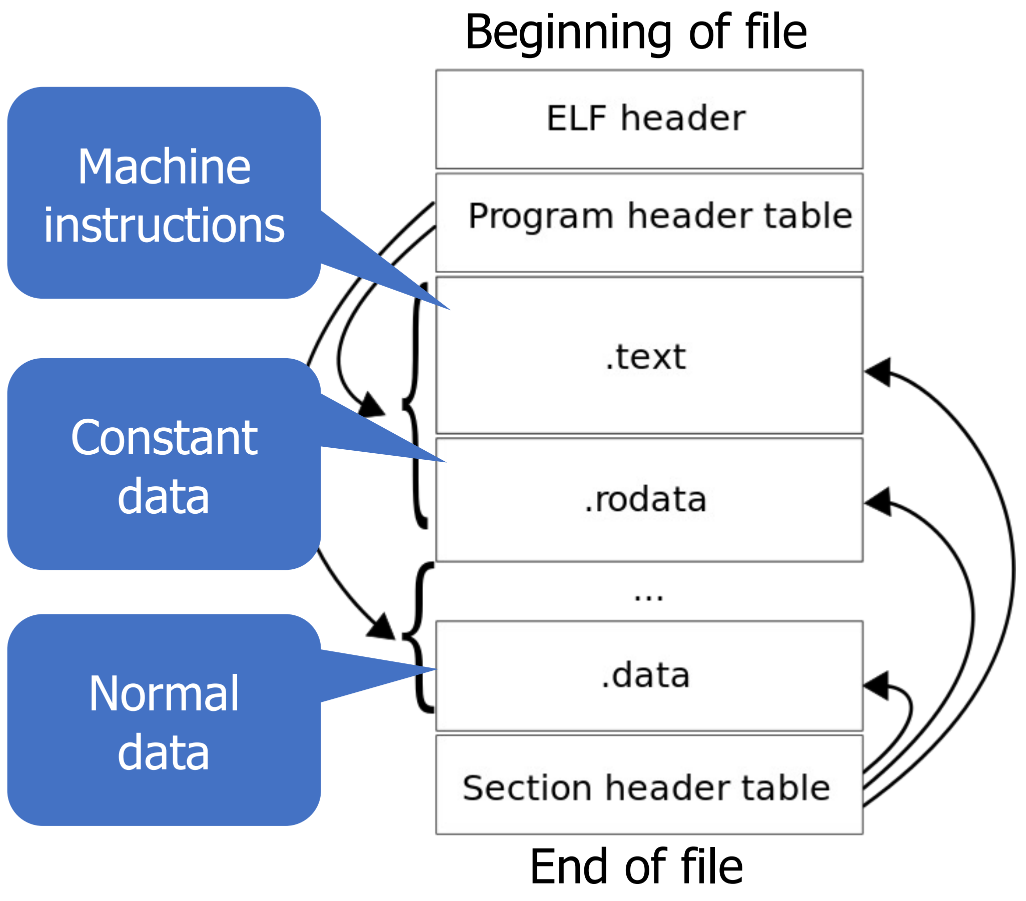
Linking

- Combines object files (including CRT and libc) into an executable file
  - Each object file has calls and accesses to other object files
  - The links between them should be made
- CRT has the initialization code calling the main function
- Standard C library (libc) is a set of object files for printf, scanf, …
- GCC calls the `ld` command internally



Executable File

- Instructions and data linked from multiple object files
- When executed, its entry function in CRT is called by OS
- Different file formats for different OSes
  - ELF* for Linux
  - PE COFF** for MS Windows



https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

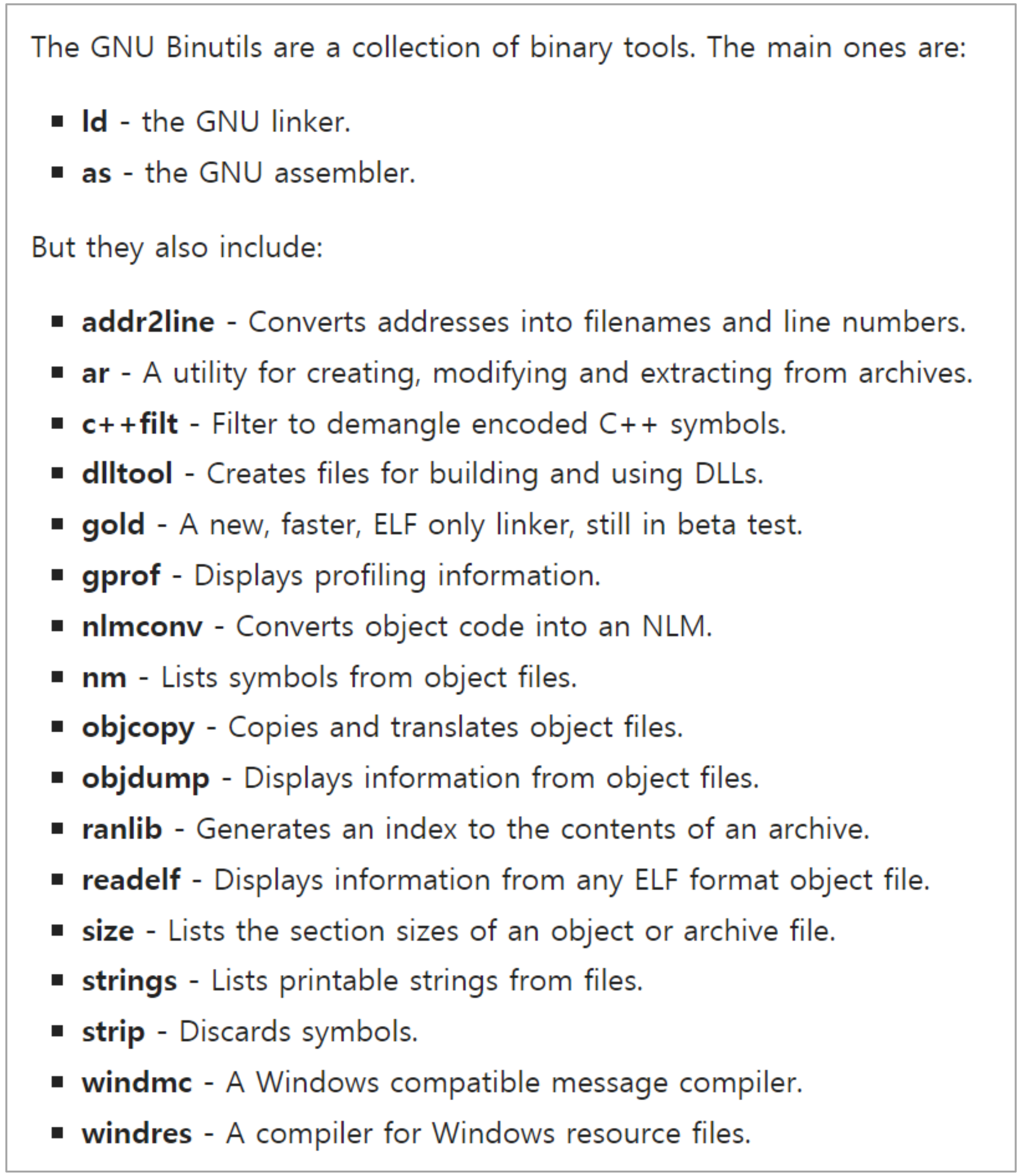
* Executable and Linkable Format
** Portable Executable Common Object File Format

GNU Binutils

- Utilities for handling binary files
  - Object files and executable files

```
$ objdump -s file.o
```
Shows the full content

```
$ objdump -D file.o
```
Shows disassembled assembly language



The GNU Binutils are a collection of binary tools. The main ones are:

- **ld** - the GNU linker.
- **as** - the GNU assembler.

But they also include:

- **addr2line** - Converts addresses into filenames and line numbers.
- **ar** - A utility for creating, modifying and extracting from archives.
- **c++filt** - Filter to demangle encoded C++ symbols.
- **dlltool** - Creates files for building and using DLLs.
- **gold** - A new, faster, ELF only linker, still in beta test.
- **gprof** - Displays profiling information.
- **nlmconv** - Converts object code into an NLM.
- **nm** - Lists symbols from object files.
- **objcopy** - Copies and translates object files.
- **objdump** - Displays information from object files.
- **ranlib** - Generates an index to the contents of an archive.
- **readelf** - Displays information from any ELF format object file.
- **size** - Lists the section sizes of an object or archive file.
- **strings** - Lists printable strings from files.
- **strip** - Discards symbols.
- **windmc** - A Windows compatible message compiler.
- **windres** - A compiler for Windows resource files.

Source: https://www.gnu.org/software/binutils/

objdump

```c
int a = 0xaa;
int b = 0xbb;

char *s = "You are a girl";
char p[] = "I am a boy";

int main(void)
{
    printf("%02x\n", 0x88);
    printf("%02x\n", a);
    printf("%02x\n", b);

    printf("%s", s);
    printf("%s", p);

    return 0;
}
```

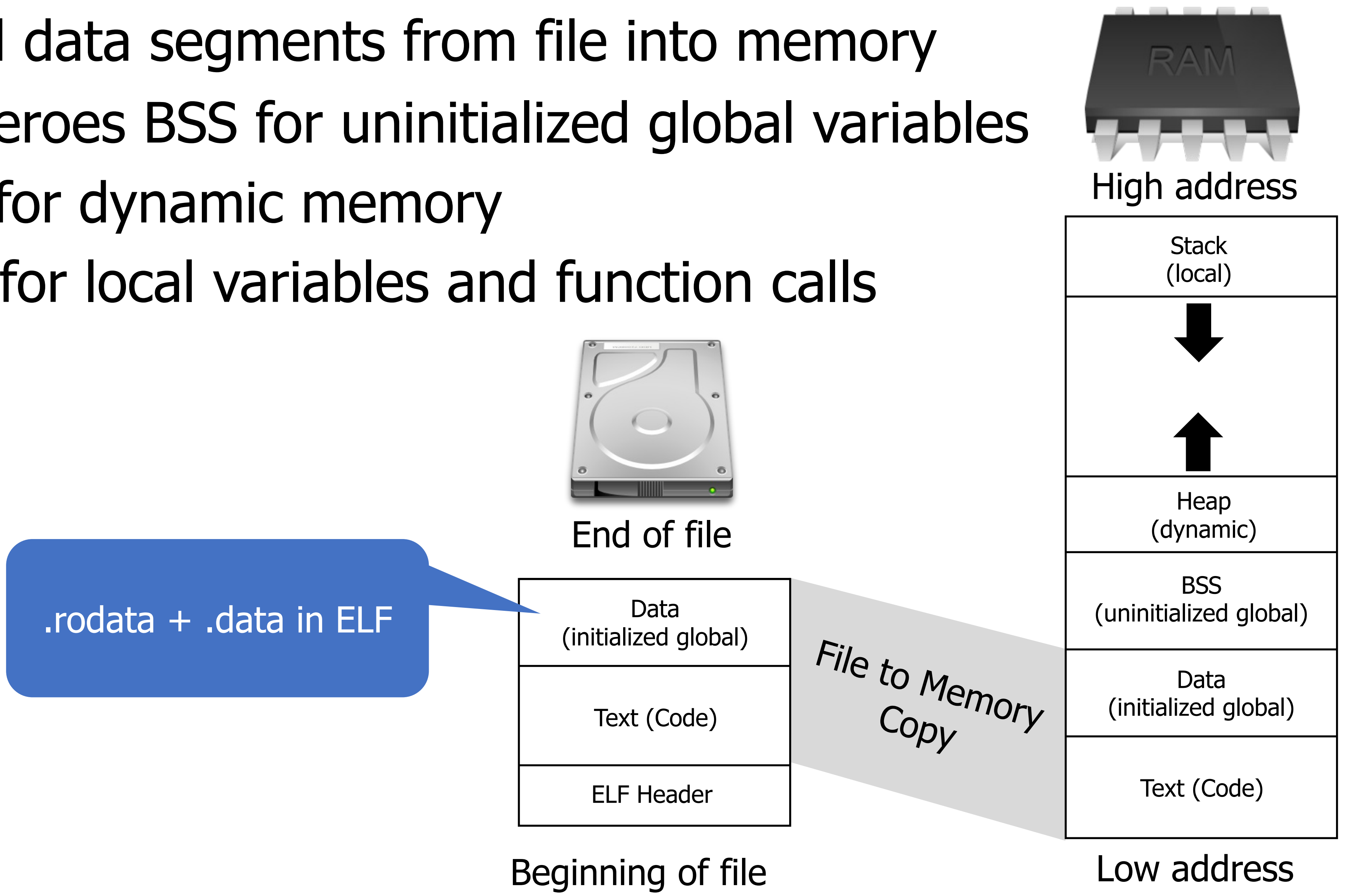
test.c

```
$ gcc -c test.c
$ objdump -s test.o
```

```
Contents of section .text:
 0000 f30f1efa 554889e5 be880000 00488d05  ....UH.......H..
 0010 00000000 4889c7b8 00000000 e8000000  ....H...........
 0020 008b0500 00000089 c6488d05 00000000  .........H......
 0030 4889c7b8 00000000 e8000000 008b0500 H...............
 0040 00000089 c6488d05 00000000 4889c7b8  .....H......H...
 0050 00000000 e8000000 00488b05 00000000  .........H......
 0060 4889c648 8d050000 00004889 c7b80000 H..H......H.....
 0070 0000e800 00000048 8d050000 00004889  .......H......H.
 0080 c6488d05 00000000 4889c7b8 00000000  .H......H.......
 0090 e8000000 00b80000 00005dc3           ..........].
Contents of section .data:
 0000 aa000000 bb000000 4920616d 20612062  ........I am a b
 0010 6f7900                               oy.
Contents of section .rodata:
 0000 596f7520 61726520 61206769 726c0025 You are a girl.%
 0010 3032780a 00257300                   02x..%s.
Contents of section .data.rel.local:
 0000 00000000 00000000                    ........
Contents of section .comment:
 0000 00474343 3a202855 62756e74 75203131  .GCC: (Ubuntu 11
 0010 2e332e30 2d317562 756e7475 317e3232  .3.0-1ubuntu1~22
 0020 2e303429 2031312e 332e3000           .04) 11.3.0.
Contents of section .note.gnu.property:
 0000 04000000 10000000 05000000 474e5500  ............GNU.
 0010 020000c0 04000000 03000000 00000000  ................
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 9c000000 00450e10 8602430d  .........E....C.
 0030 0602930c 07080000                    ........
```

Program Loading

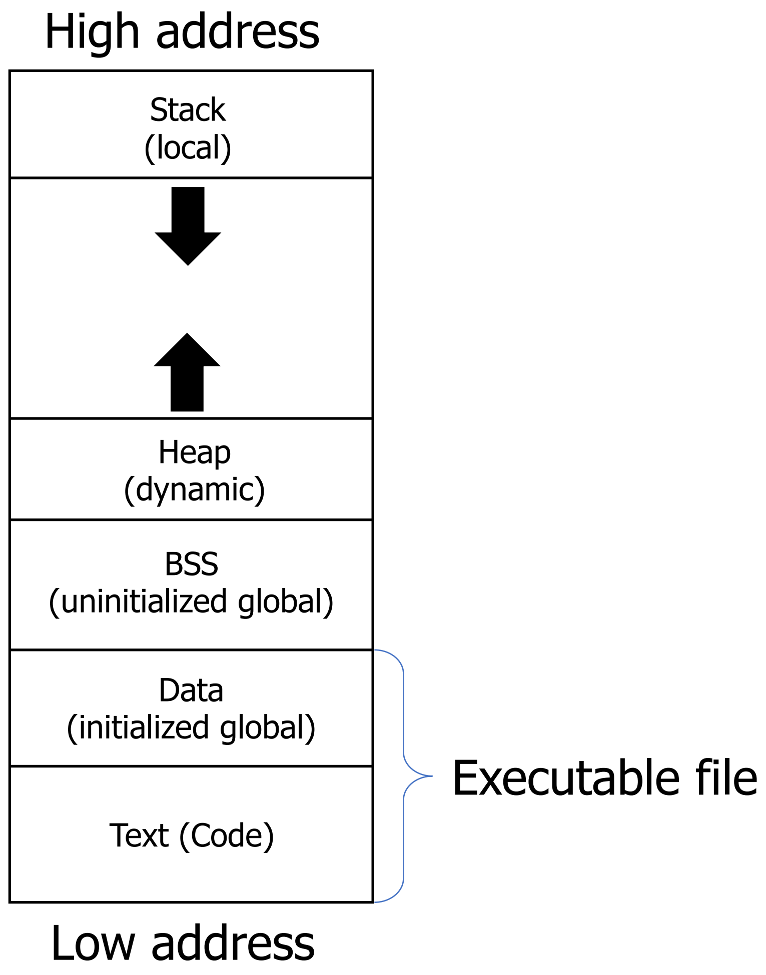
- Copies text and data segments from file into memory
- Allocates and zeroes BSS for uninitialized global variables
- Prepares heap for dynamic memory
- Prepares stack for local variables and function calls



Program in Memory

- Variables and functions have their own locations in the address space

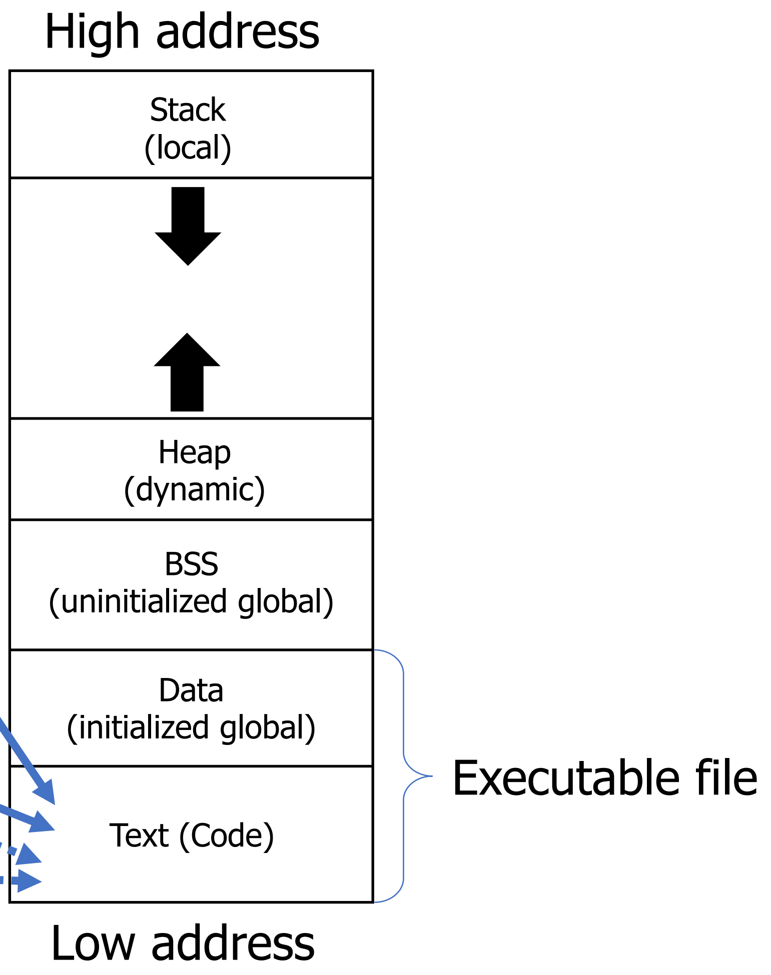
```
#include <stdio.h>
int g = 1;      /* initialized global */
int b;          /* uninitialized global*/
void func(void)        /* function */
{
    int l2;            /* local */
    char *p;           /* local */
    static int s;      /* static local */
    p = malloc(10);    /* dynamic */
    return;
}
int main(void)         /* function */
{
    int l1;            /* local */
    return 0;
}
```



Text Area

- Text area stores the instructions of a program

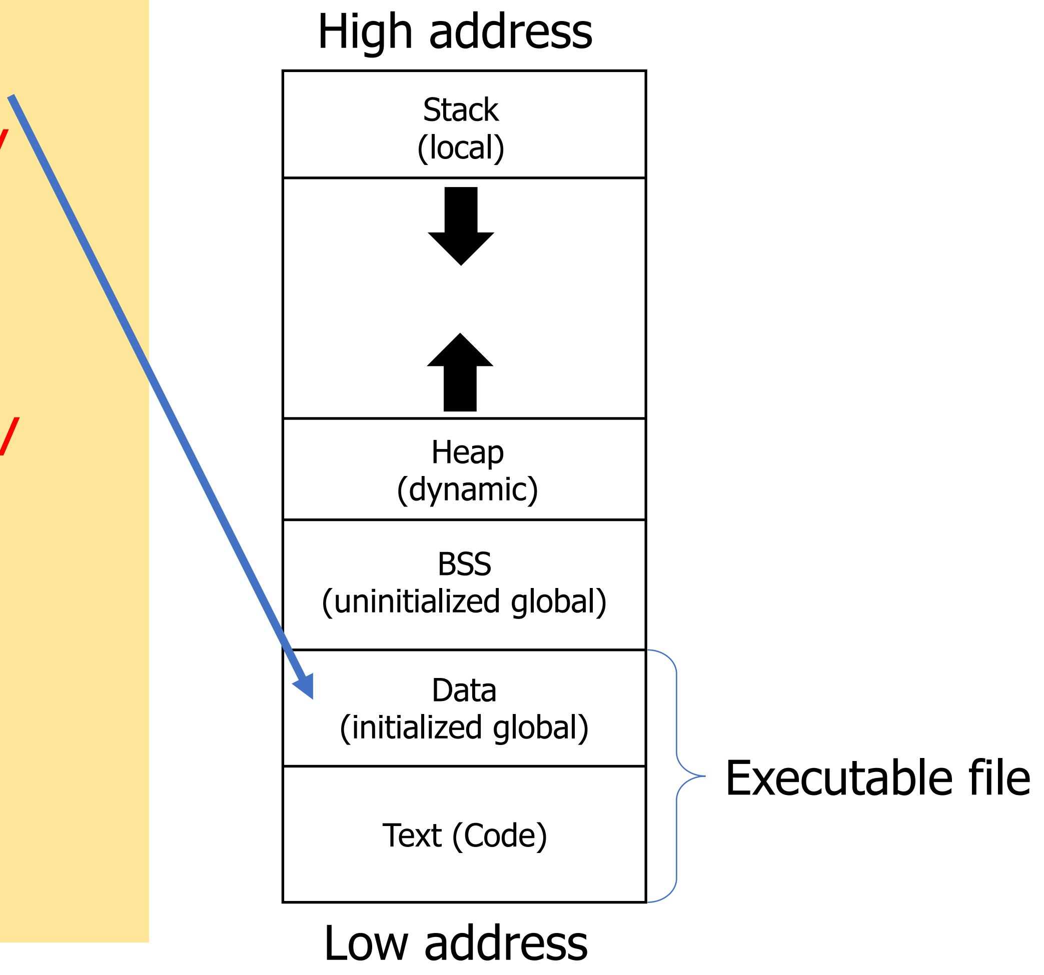
```
#include <stdio.h>
int g = 1;      /* initialized global */
int b;          /* uninitialized global*/
void func(void)        /* function */
{
    int l2;            /* local */
    char *p;           /* local */
    static int s;      /* static local */
    p = malloc(10);    /* dynamic */
    return;
}
int main(void)         /* function */
{
    int l1;            /* local */
    return 0;
}
```



Data Area

- Data area stores initialized global variables

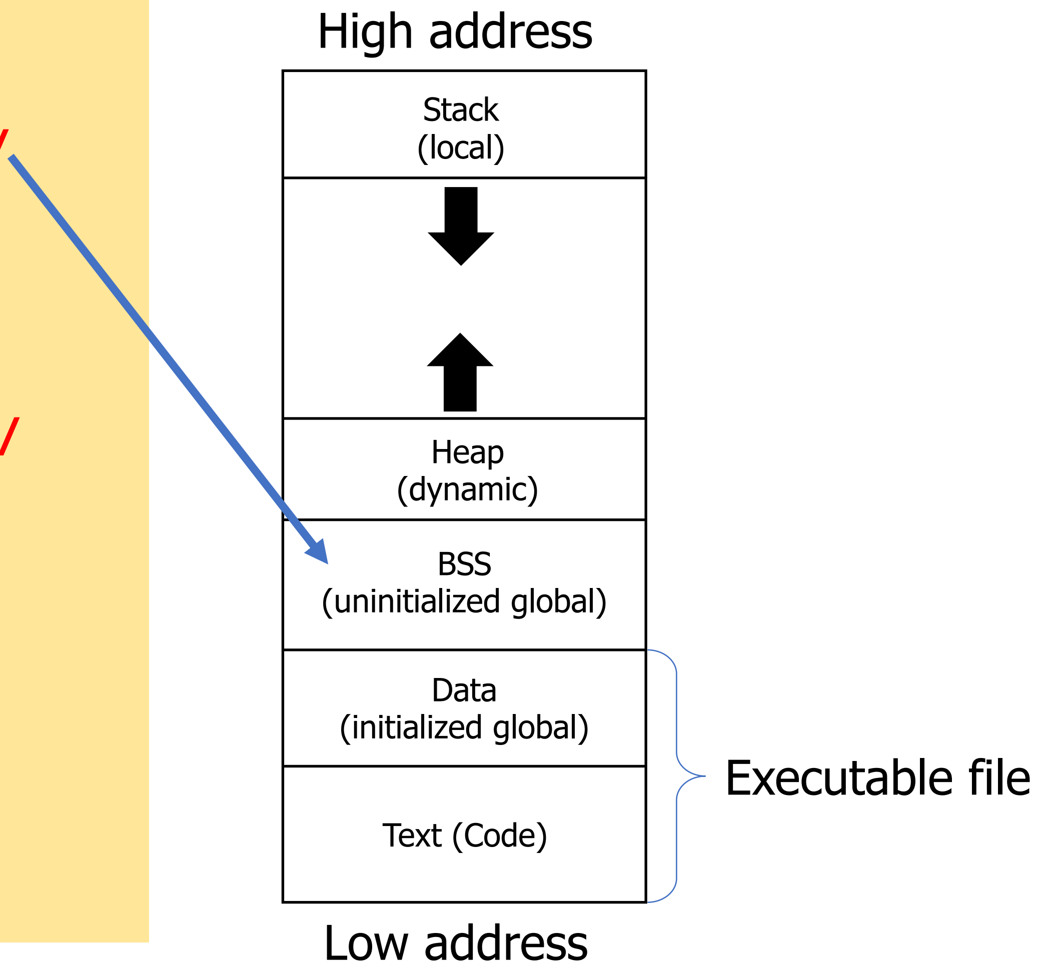
```
#include <stdio.h>
int g = 1;      /* initialized global */
int b;          /* uninitialized global*/
void func(void)        /* function */
{
    int l2;            /* local */
    char *p;           /* local */
    static int s;      /* static local */
    p = malloc(10);    /* dynamic */
    return;
}
int main(void)         /* function */
{
    int l1;            /* local */
    return 0;
}
```



BSS Area

- BSS* area stores uninitialized global variables
- Automatically initialized to zeros

```
#include <stdio.h>
int g = 1;      /* initialized global */
int b;          /* uninitialized global*/
void func(void)        /* function */
{
    int l2;            /* local */
    char *p;           /* local */
    static int s;      /* static local */
    p = malloc(10);    /* dynamic */
    return;
}
int main(void)         /* function */
{
    int l1;            /* local */
    return 0;
}
```

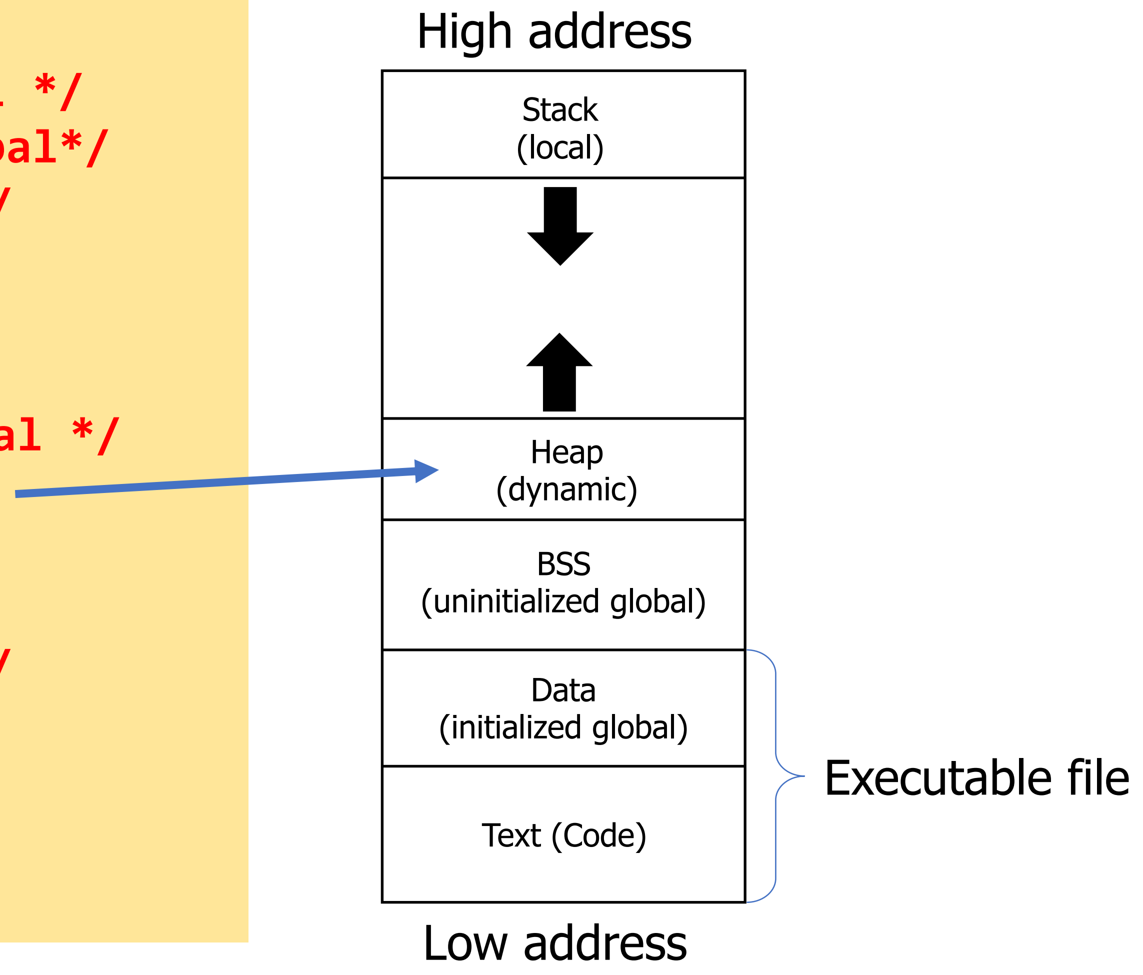


* Block Started by Symbol

Heap Area

- Heap area stores dynamic memory that grows dynamically
- Managed by the memory manager in the malloc() function

```
#include <stdio.h>
int g = 1;     /* initialized global */
int b;         /* uninitialized global*/
void func(void)       /* function */
{
    int l2;           /* local */
    char *p;          /* local */
    static int s;     /* static local */
    p = malloc(10);   /* dynamic */
    return;
}
int main(void)        /* function */
{
    int l1;           /* local */
    return 0;
}
```

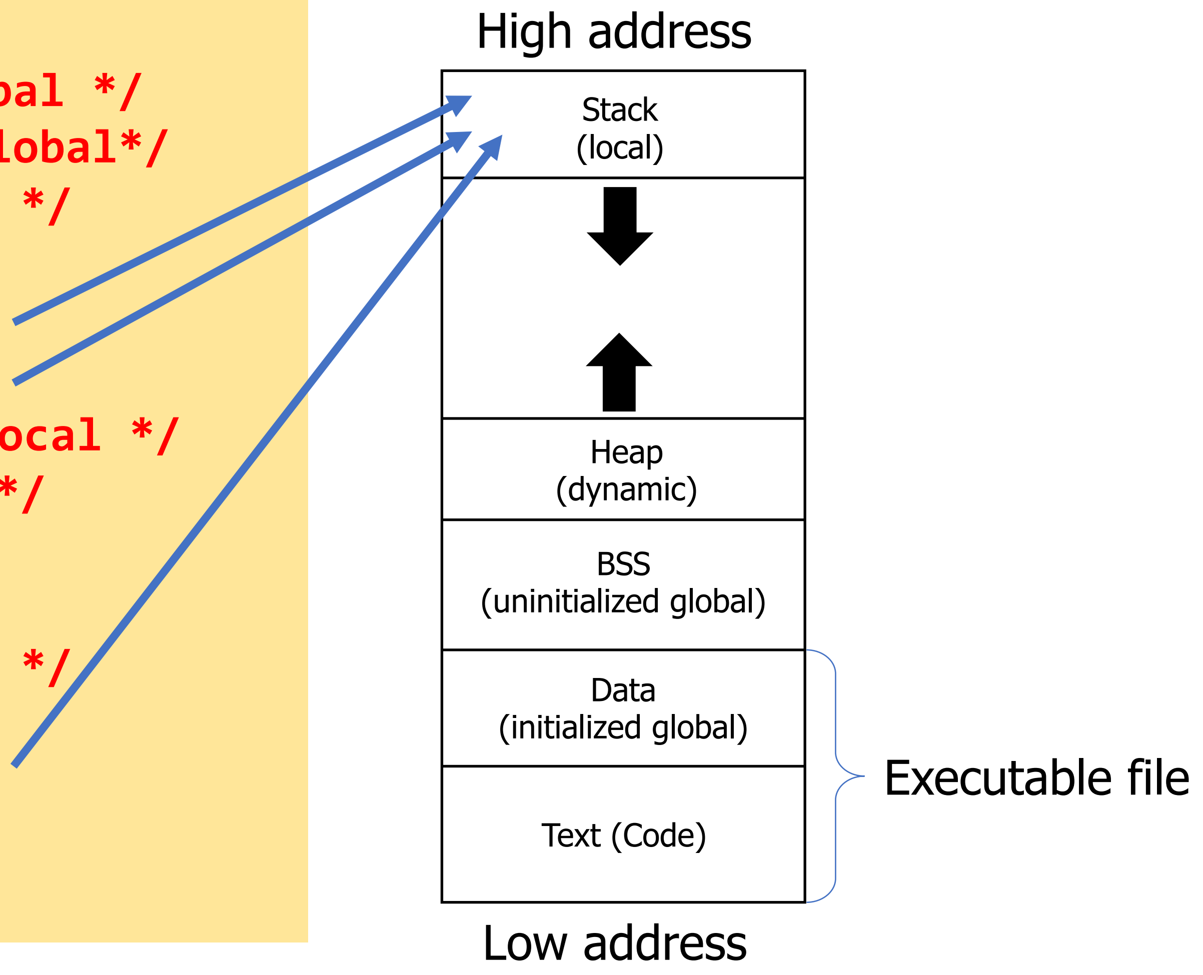


* Block Started by Symbol

Stack Area

- Stack area stores local variables
- Local variables exist only during the function is executing

```
#include <stdio.h>
int g = 1;      /* initialized global */
int b;          /* uninitialized global*/
void func(void)        /* function */
{
    int l2;            /* local */
    char *p;           /* local */
    static int s;      /* static local */
    p = malloc(10);    /* dynamic */
    return;
}
int main(void)         /* function */
{
    int l1;            /* local */
    return 0;
}
```

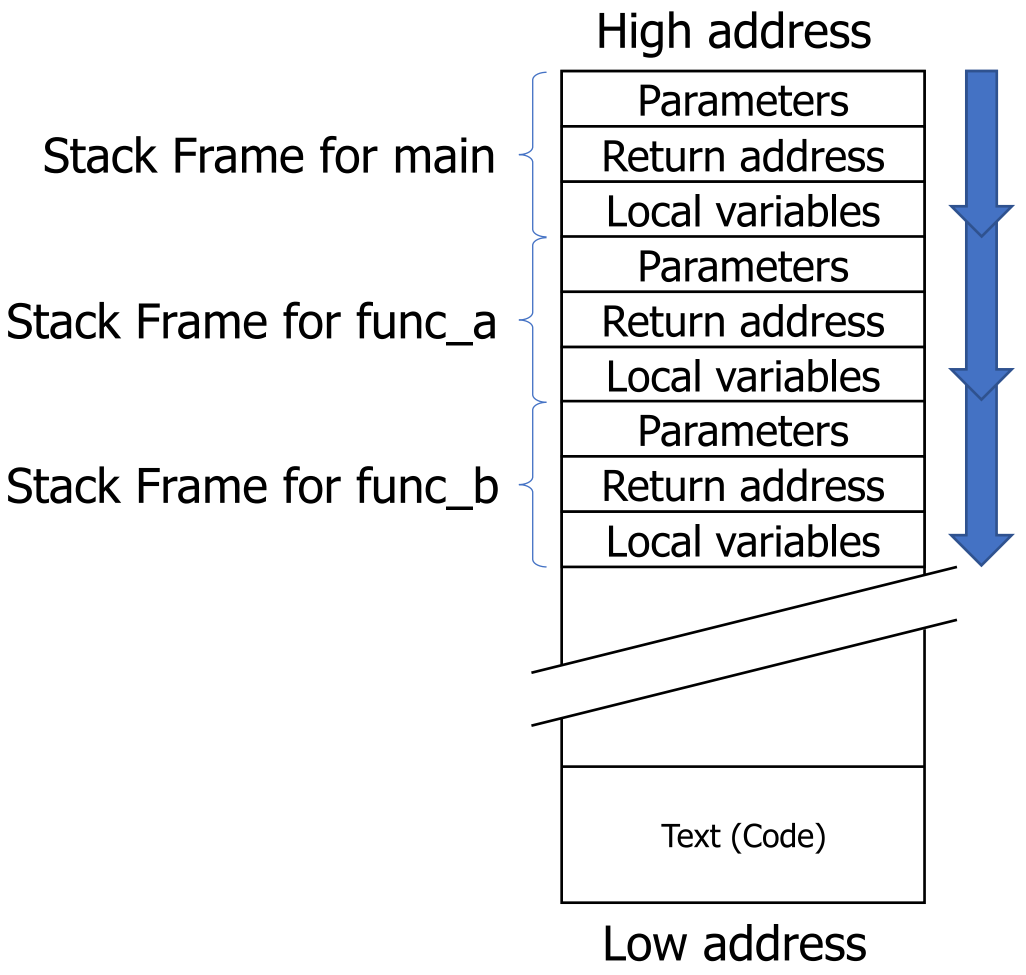


Stack Frame

- Stack grows and shrinks when a function is called and returned
- Stack is reused by many functions while not being cleaned

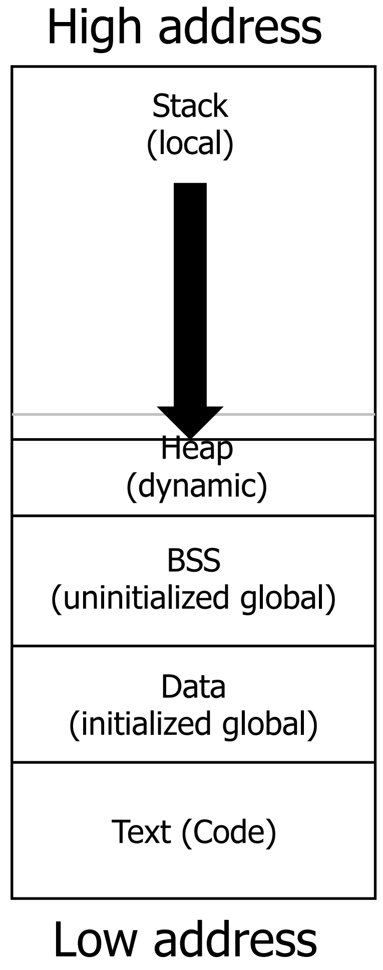
The reason why local variables have garbage values

```
int main(void) {
    int a;
    func_a(a);
    return 0;
}
void func_a(int p) {
    int v;
    func_b(p);
    return;
}
void func_b(int p) {
    int v;
    return;
}
```

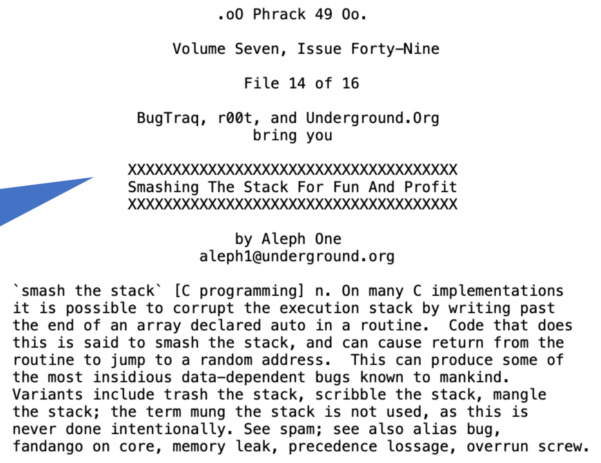


Risks on Stack

- Stack overflow
  - What if stack grows too much?
  - Dynamic memory area can be overwritten by the stack area
  - Avoid deep nested function calls (e.g., recursion)
  - Avoid using large local variables like arrays
- Stack smashing
  - Hackers use buffer overflow techniques to put malicious code on stack and overwrite a function's return address pointing to it



Notorious article about the stack smashing attack



```
                     .oO Phrack 49 Oo.

                 Volume Seven, Issue Forty-Nine

                         File 14 of 16

               BugTraq, r00t, and Underground.Org
                          bring you

            XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
            Smashing The Stack For Fun And Profit
            XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

                        by Aleph One
                    aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations
it is possible to corrupt the execution stack by writing past
the end of an array declared auto in a routine.  Code that does
this is said to smash the stack, and can cause return from the
routine to jump to a random address.  This can produce some of
the most insidious data-dependent bugs known to mankind.
Variants include trash the stack, scribble the stack, mangle
the stack; the term mung the stack is not used, as this is
never done intentionally. See spam; see also alias bug,
fandango on core, memory leak, precedence lossage, overrun screw.
```

http://phrack.org/issues/49/14.html

Summary

- Building programs
  - Preprocessing
  - Compiling
  - Linking
- Loading programs
  - Text
  - Data
  - BSS
  - Stack
  - Heap