Karlsruhe Institute of Technology

Operating Systems

Introduction to the C Programming Language II

Dr.-Ing. Marc Rittinghaus | WT 2020/2021


```c
#include "malloc.h"

#include <stdio.h>
#include <assert.h>

typedef struct _Block {
    /*
     * Pointer to the header of the next free block.
     * Only valid if this block is also free.
     * This is null for the last Block of the free list.
     */
    struct _Block *next;

    /*
     * Our header should always have a size of 16 Bytes.
     * This is just for 32 bit systems.
     */
    uint8_t padding[8 - sizeof(void*)];

    /*
     * The size of this block, including the header
     * Always a multiple of 16 bytes.
```

```c
} Block;

#define HEADER_SIZE sizeof(Block)
#define INV_HEADER_SIZE_MASK ~((uint64_t)HEADER_SIZE - 1)

/*
 * This is the heap you should use.
 * 16 MiB heap space per default. The heap does not grow.
 */
#define HEAP_SIZE (HEADER_SIZE * 1024 * 1024)
uint8_t __attribute__ ((aligned(HEADER_SIZE))) _heapData[HEAP_SIZE];

/*
 * This should point to the first free block in memory.
 */
Block *_firstFreeBlock;

/*
 * Initializes the memory block. You don't need to change this.
 */
void initAllocator()
{
```

C Preprocessor

- C preprocessor modifies *source* code *before* compilation
  - Based on preprocessor directives and macros
  - Insert/replace source code
  - Conditional compilation
  - Macro expansion (e.g., `__LINE__`, `__FILE__`, `__func__`)

- **#include**: (literally) copies contents of file into current file

```c
#include <stdio.h>  // Preprocessor directive
int main()
{
    printf("Hello World from line %d!\n", __LINE__);
    return 0;
}
```

Macro

```
Hello World from line 4!
```

C Preprocessor

```
#include <stdio.h>    // #include <file>
```

- System include; search in system include directories such as `/usr/local/include`
  (try: `gcc -v -x c -E /dev/null`)
- Can add own paths with `-I<dir>`

```
#include "myheader.h" // #include "file"
```

- Local include; search in directory containing the current file
- Then in the paths specified by `-I<dir>`
- Then in system include paths

Declaration vs. Definition

- Compiler only knows functions previously declared
  - Definition implicitly declares functions

main.c

```c
int sum(int a, int b, int c)
{
  return a+b+c;
}

int main()
{
  return sum(1,2); // ERROR
}
```

Does not compile.

```
$ gcc main.c -o sumprog
error: two few arguments to
function sum
```

main.c

```c
int main()
{
  return sum(1,2); // ??
}

int sum(int a, int b, int c)
{
  return a+b+c;
}
```

Wrong implicit declaration!
→ Value of c in call undefined

```
$ gcc main.c -o sumprog
$ ./sumprog → 50
$ ./sumprog → 114
```

Declaration vs. Definition

- Example: Declaration of function in another file

math.c

```c
int sum(int a, int b)
{
  return a+b;
}
```

main.c

```c
int sum(int a, int b);
int main()
{

  return sum(2, 8); // 10
}
```

```
$ gcc math.c main.c -o sumprog
$ ./sumprog → 10
$ ./sumprog → 10
```

Declaration vs. Definition

- Example: Declaration of function in another file

Changed without fixing `main.c`!

math.c

```c
int sum(int a, int b, int c)
{
  return a+b+c;
}
```

main.c

```c
int sum(int a, int b);
int main()
{

  return sum(2, 8); // 10
}
```

⚠ Compiles but returns undefined results! ⚠

```
$ gcc math.c main.c -o sumprog
$ ./sumprog → 194
$ ./sumprog → 66
```

Declaration vs. Definition

- Better: Use header file to import declaration

+ Ensures consistent signature of `sum()`

math.h

```c
int sum(int a, int b);
```

math.c

```c
#include "math.h"


int sum(int a, int b)
{
  return a+b;
}
```

main.c

```c
#include "math.h"

int main()
{

  return sum(2, 8); // 10
}
```

```
$ gcc math.c main.c -o sumprog
$ ./sumprog → 10
$ ./sumprog → 10
```

Extern

- Use **extern** to declare global variables defined elsewhere

math.h

```
extern int myvar;
int sum(int a, int b);
```

math.c

```
#include "math.h"

int myvar;
int sum(int a, int b)
{
  return a+b+myvar;
}
```

main.c

```
#include "math.h"

int main()
{
  myvar = 3;
  return sum(2, 8); // 13
}
```

```
$ gcc math.c main.c -o sumprog
$ ./sumprog → 13
$ ./sumprog → 13
```

C Preprocessor

- **#define**: Defines macros for string replacement

```
#define MY_CONDITION
#define TRUE 1
#define MAX(a,b) (((a) > (b)) ? (a):(b))
```

```
int func(int ia, int ib) {
  return MAX(ia, ib);
}
```

```
int func(int ia, int ib) {
  return (((ia) > (ib)) ? (ia):(ib));
}
```

- Helps making code portable and adjustable
  - Quickly switch on/off features based on architecture or config

```
#ifdef __unix__
#include <unistd.h>
#elif defined _WIN32
#include <windows.h>
#endif
```

```
#ifdef DEBUG
#define TRACE(x) printf(x)
#else
#define TRACE(x)
#endif
```

```
$ gcc -D DEBUG main.c
```

C Preprocessor

- **#define**: Defines macros for string replacement

```
#define MY_CONDITION
#define TRUE 1
#define MAX(a,b) (((a) > (b)) ? (a):(b))
```

```
int func(int ia, int ib) {
  return MAX(ia, ib);
}          f(ia)
```

```
int func(int ia, int ib) {
  return (((ia) > (ib)) ? (ia):(ib));
}          f(ia)              f(ia)
```

- Helps making code portable and adjustable
  - Quickly switch on/off features based on architecture or config

```
#ifdef __unix__
#include <unistd.h>
#elif defined _WIN32
#include <windows.h>
#endif
```

```
#ifdef DEBUG
#define TRACE(x) printf(x)
#else
#define TRACE(x)
#endif
```

```
$ gcc -D DEBUG main.c
```

Include Guards

- Problem: Multiple includes of same header

math.h

```
typedef struct coord {
  int x; int y;
} coord;
```

mathex.h

```
#include "math.h"
int sum(coord a, coord b);
```

main.c

```
#include "math.h"
#include "mathex.h"
```

mathex.h
- math.h
  - math.h

```
typedef struct coord {
  int x; int y;
} coord;
typedef struct coord {
  int x; int y;
} coord;
int sum(coord a, coord b);
```

✖ Compile error: Redefinition of type `struct coord`

Include Guards

- Prevent contents of `math.h` being included multiple times

math.h

```
#ifndef MATH_H
#define MATH_H

typedef struct coord {
  int x; int y;
} coord;

#endif
```

mathex.h

```
#include "math.h"
int sum(coord a, coord b);
```

main.c

```
#include "math.h"
#include "mathex.h"
```

← Second include is empty. MATH_H already defined!

Application Binary Interface

- Defines binary interface between programs/modules/OS
  - Specifies executable/object file formats, calling convention, dynamic linking semantics, alignment rules, …
  - Example: System V AMD64 ABI used in Linux, BSD, and macOS

- **Calling conventions** standardize how parameters and return values are passed between a calling function (caller) and the called function (callee)
  - cdecl (32-bit)
  - System V AMD64 (64-bit)
  - Microsoft x64 (64-bit)
  - System call ABIs

Calling Conventions

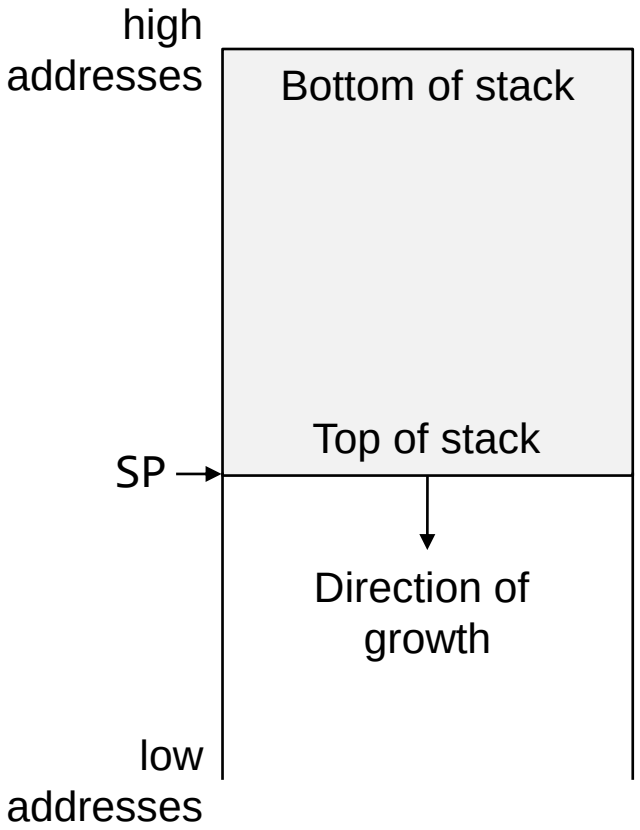
- When a function is called, the caller…
  1. Saves the state of the local scope (e.g., still used registers)
  2. Sets up parameters where the subroutine can find them
  3. Jumps to function

- The called function then…
  1. Sets up a new local scope
  2. Performs its duty
  3. Puts the return value where the caller can find it
  4. Jumps back to calling function

x86(-64) Stack

32-bit: ESP
64-bit: RSP

- Stack pointer (SP)
  - Points at last allocated word on stack (*pre-decrement stack*)
  - Stack grows downward
- Push
  - Decrement SP (allocate space)
  - Place value at SP
- Pop
  - Retrieve value at SP
  - Increment SP (free space)



cdecl

```
EAX ← func(int a, int b, int c, int d, int e, int f, int g);
           s[6]   s[5]   s[4]   s[3]   s[2]   s[1]   s[0]
```

- Used on 32-bit platforms
- Arguments passed on stack in reverse order
  - Caller frees parameter stack space
  - Stack frames aligned to 4 (msvc) or 16 bytes (gcc)
    (16-byte alignment needed if 128-bit XMM registers used - SIMD)
- Result returned via EAX
  - 64-bit in EAX:EDX

32-bit general-purpose registers

Callee

```
int sum(int a, int b) {
  int val = a + b;

  return val;
}
```

Caller

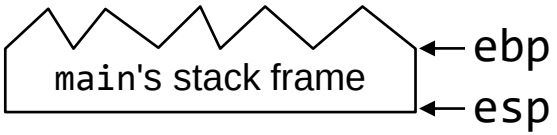
```
int main() {
  return sum(2, 8); // 10
}
```

cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```



ebp: base pointer
(a.k.a. frame pointer)
allows referencing contents on stack with fixed offsets

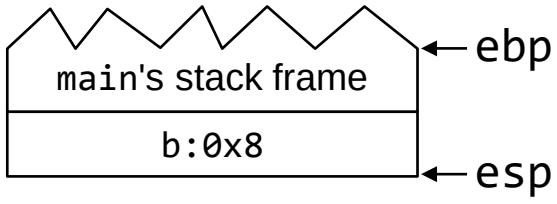
ebp+X: arguments
ebp-X: local vars

cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```

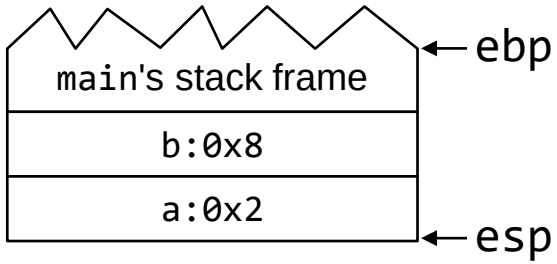


cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```

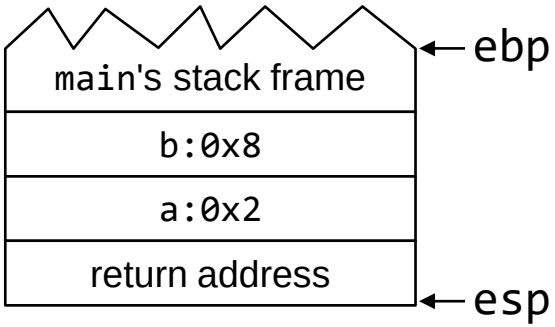


cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```

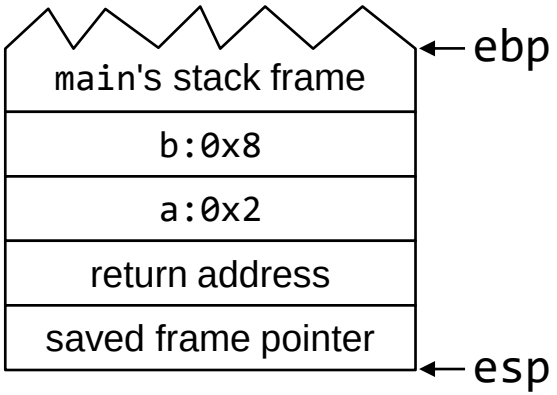


cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```

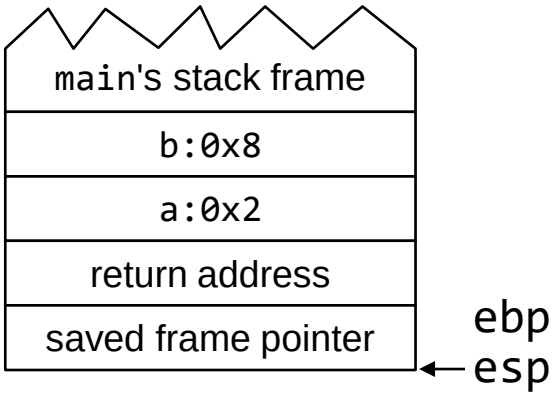


cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```

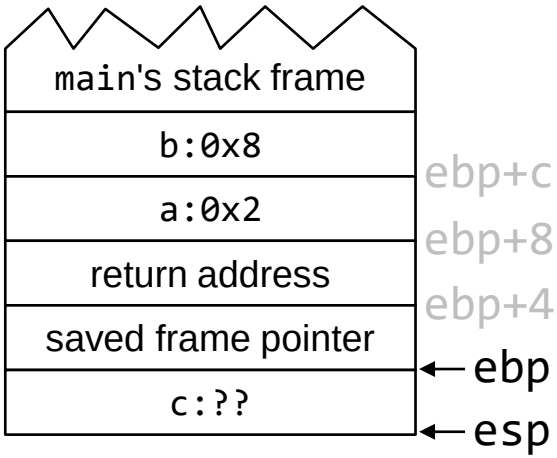


cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8                ; push 8 onto stack
 push 0x2                ; push 2 onto stack
 call <sum>              ; save eip/jump to callee
 add  esp, 0x8           ; release stack space
```

```
 push ebp                ; save frame pointer
 mov  ebp, esp           ; start new frame
int c;
 sub  esp, 0x04          ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]     ; load a into edx
 mov  eax, [ebp+0xc]     ; load b into eax
 add  eax, edx           ; perform addition
 mov  [ebp-0x4], eax     ; save result in c
return c;
 mov  eax, [ebp-0x4]     ; load result into eax
 leave                   ; mov esp, ebp; pop ebp;
 ret                     ; return to caller
```

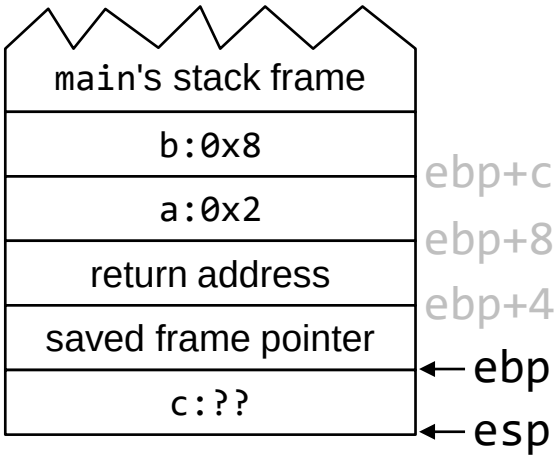


cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8                ; push 8 onto stack
 push 0x2                ; push 2 onto stack
 call <sum>              ; save eip/jump to callee
 add  esp, 0x8           ; release stack space
```

```
 push ebp                ; save frame pointer
 mov  ebp, esp           ; start new frame
int c;
 sub  esp, 0x04          ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]     ; load a into edx
 mov  eax, [ebp+0xc]     ; load b into eax
 add  eax, edx           ; perform addition
 mov  [ebp-0x4], eax     ; save result in c
return c;
 mov  eax, [ebp-0x4]     ; load result into eax
 leave                   ; mov esp, ebp; pop ebp;
 ret                     ; return to caller
```



edx: 0x2
eax: 0x8

eax: 0xa

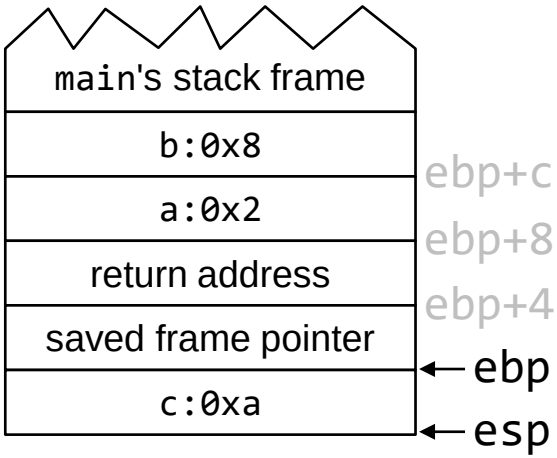
cdecl

Assembly not optimized!

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8                ; push 8 onto stack
 push 0x2                ; push 2 onto stack
 call <sum>              ; save eip/jump to callee
 add  esp, 0x8           ; release stack space
```

```
 push ebp                ; save frame pointer
 mov  ebp, esp           ; start new frame
int c;
 sub  esp, 0x04          ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]     ; load a into edx
 mov  eax, [ebp+0xc]     ; load b into eax
 add  eax, edx           ; perform addition
 mov  [ebp-0x4], eax     ; save result in c
return c;
 mov  eax, [ebp-0x4]     ; load result into eax
 leave                   ; mov esp, ebp; pop ebp;
 ret                     ; return to caller
```



eax: 0xa

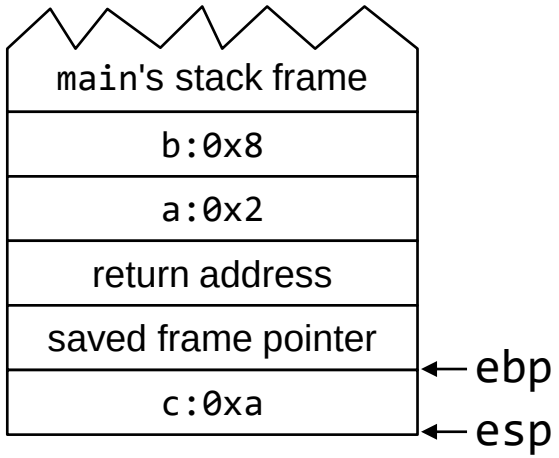
eax: 0xa

cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8                ; push 8 onto stack
 push 0x2                ; push 2 onto stack
 call <sum>              ; save eip/jump to callee
 add  esp, 0x8           ; release stack space
```

```
 push ebp                ; save frame pointer
 mov  ebp, esp           ; start new frame
int c;
 sub  esp, 0x04          ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]     ; load a into edx
 mov  eax, [ebp+0xc]     ; load b into eax
 add  eax, edx           ; perform addition
 mov  [ebp-0x4], eax     ; save result in c
return c;
 mov  eax, [ebp-0x4]     ; load result into eax
 leave                   ; mov esp, ebp; pop ebp;
 ret                     ; return to caller
```



release stack frame

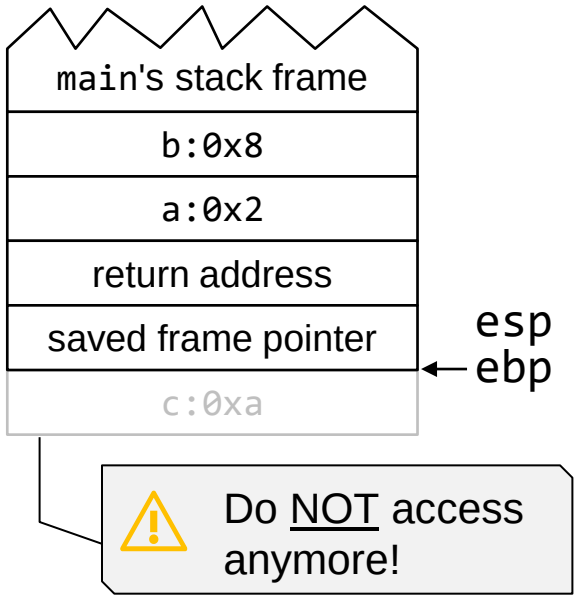
```
mov esp, ebp
pop ebp
```

cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8                ; push 8 onto stack
 push 0x2                ; push 2 onto stack
 call <sum>              ; save eip/jump to callee
 add  esp, 0x8           ; release stack space
```

```
 push ebp                ; save frame pointer
 mov  ebp, esp           ; start new frame
int c;
 sub  esp, 0x04          ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]     ; load a into edx
 mov  eax, [ebp+0xc]     ; load b into eax
 add  eax, edx           ; perform addition
 mov  [ebp-0x4], eax     ; save result in c
return c;
 mov  eax, [ebp-0x4]     ; load result into eax
 leave                   ; mov esp, ebp; pop ebp;
 ret                     ; return to caller
```



restore old frame

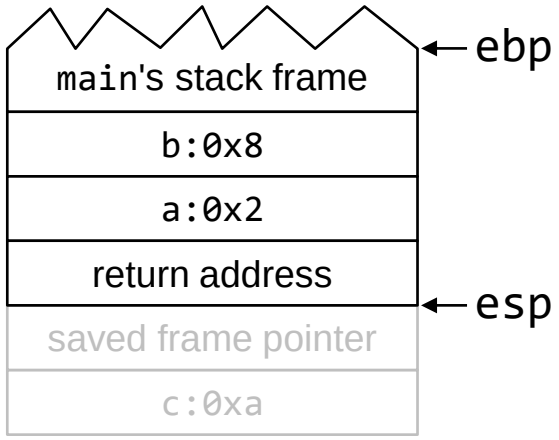
```
mov esp, ebp
pop ebp
```

cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```



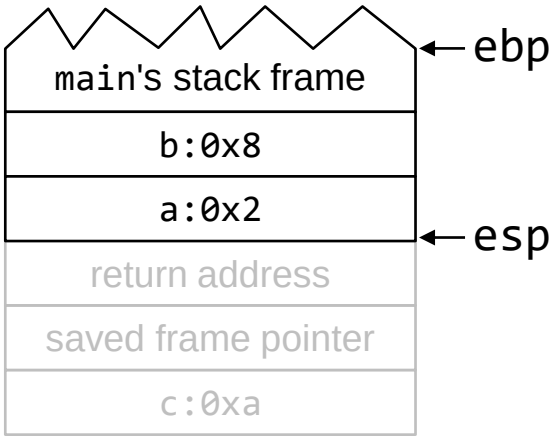

```
mov esp, ebp
pop ebp
```

cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```

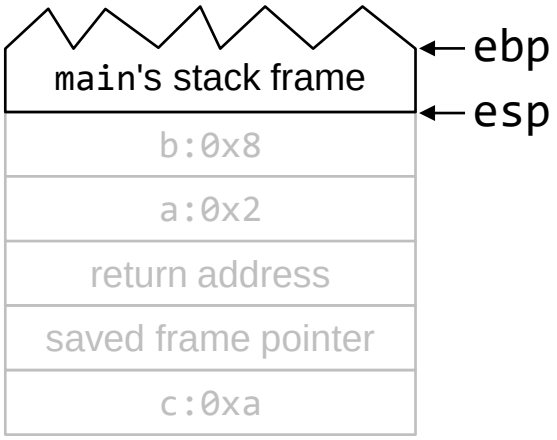


cdecl

```
$ gcc -O0 -g -fno-stack-protector -fcf-protection=none -fno-pic –mpreferred-stack-boundary=2 -m32 –o prog main.c
$ objdump -sd -M intel ./prog
```

```
return sum(2, 8); // 10
 push 0x8              ; push 8 onto stack
 push 0x2              ; push 2 onto stack
 call <sum>            ; save eip/jump to callee
 add  esp, 0x8         ; release stack space
```

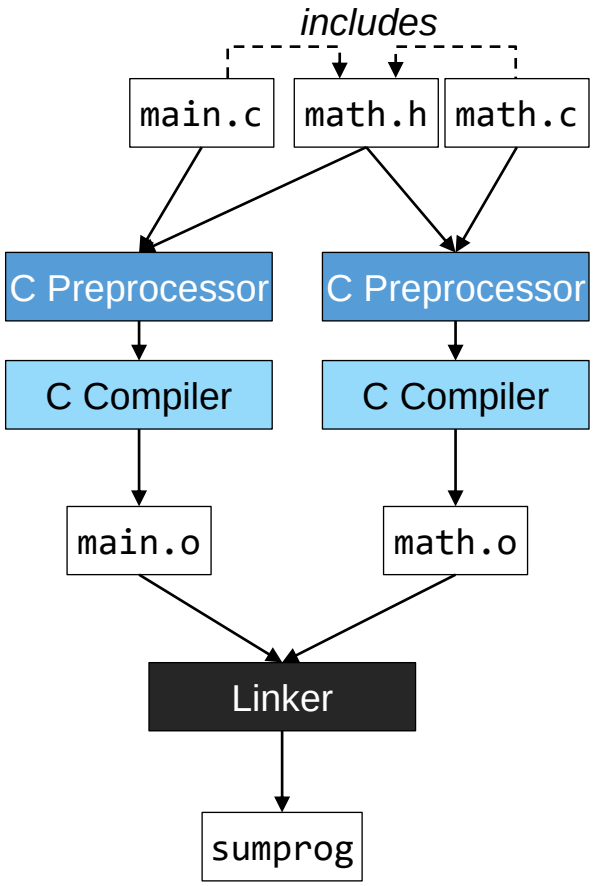
```
 push ebp              ; save frame pointer
 mov  ebp, esp         ; start new frame
int c;
 sub  esp, 0x04        ; reserve space for c
c = a + b;
 mov  edx, [ebp+0x8]   ; load a into edx
 mov  eax, [ebp+0xc]   ; load b into eax
 add  eax, edx         ; perform addition
 mov  [ebp-0x4], eax   ; save result in c
return c;
 mov  eax, [ebp-0x4]   ; load result into eax
 leave                 ; mov esp, ebp; pop ebp;
 ret                   ; return to caller
```



Linking

- Multiple source files
  - Built independently to object files (*.o)
  - ➢ Compiler cannot resolve external symbols
- Linker (Linux: `ld`)
  - Combines all object files into a single executable binary
  - Patches unresolved references

One file's definition of a global symbol satisfies another file's undefined reference to the same global symbol.



Static Linking: Example

math.h

```
#ifndef MATH_H
#define MATH_H
extern int myvar;
int sum(int a, int b);
#endif
```

includes →

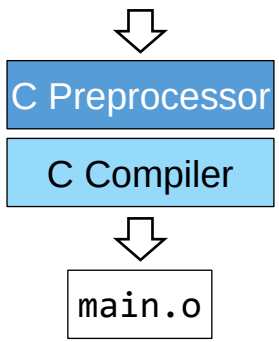
← *includes*

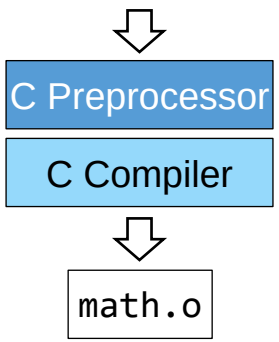
main.c

```
#include "math.h"

int main()
{
  myvar = 3;
  return sum(2, 8); // 13
}
```

math.c

```
#include "math.h"
int myvar;

int sum(int a, int b)
{
  return a+b+myvar;
}
```





Static Linking: Example

References to unresolved symbols are filled with zero bytes

main.o

```
#include "math.h"                          .text
int main() {
 000:  55                 push ebp
 001:  89 e5              mov  ebp, esp
myvar = 3;
 003:  C7 05 00 00 00     mov  [0], 3
       00 03 00 00 00
return sum(2, 8); // 13
 00d:  6A 08              push 0x8
 00f:  6A 02              push 0x2
 011:  E8 00 00 00 00     call 0
 016:  83 C4 08           add  esp, 0x8
}
 019:  C9                 leave
 01a:  C3                 ret
```

math.o

```
#include "math.h"                          .text
int myvar;

int sum(int a, int b) {
 000:  55                 push ebp
 001:  89 e5              mov  ebp, esp
return a+b+myvar;
 003:  8b 55 08           mov  edx, [ebp+0x8]
 006:  8b 45 0C           mov  eax, [ebp+0xc]
 009:  01 C2              add  edx, eax
 00b:  A1 00 00 00 00     mov  eax, [0]
 010:  01 D0              add  eax, edx
}
 012:  5D                 pop  ebp
 013:  C3                 ret
```

Static Linking: Before

main.o

.rel.text

Offset	Info	Type	
00000005	00000e01	R_386_32	myvar
00000012	00000f02	R_386_PC32	sum

.symtab

Num	Size	Type	Bind	
13	27	FUNC	GLOBAL	main
14	0	NOTYPE	GLOBAL	myvar
15	0	NOTYPE	GLOBAL	sum

.text

```
#include "math.h"
int main() {
 000:  55                push ebp
 001:  89 e5             mov  ebp, esp
myvar = 3;
 003:  C7 05 00 00 00    mov  [0], 3
       00 03 00 00 00
return sum(2, 8); // 13
 00d:  6A 08             push 0x8
 00f:  6A 02             push 0x2
 011:  E8 00 00 00 00    call 0
 016:  83 C4 08          add  esp, 0x8
}
 019:  C9                leave
 01a:  C3                ret
```

math.o

.rel.text

Offset	Info	Type	
0000000c	00000d01	R_386_32	myvar

.symtab

Num	Size	Type	Bind	
13	4	OBJECT	GLOBAL	myvar
14	20	FUNC	GLOBAL	sum

.text

```
#include "math.h"
int myvar;

int sum(int a, int b) {
 000:  55                push ebp
 001:  89 e5             mov  ebp, esp
return a+b+myvar;
 003:  8b 55 08          mov  edx, [ebp+0x8]
 006:  8b 45 0C          mov  eax, [ebp+0xc]
 009:  01 C2             add  edx, eax
 00b:  A1 00 00 00 00    mov  eax, [0]
 010:  01 D0             add  eax, edx
}
 012:  5D                pop  ebp
 013:  C3                ret
```

Static Linking: After

sumprog

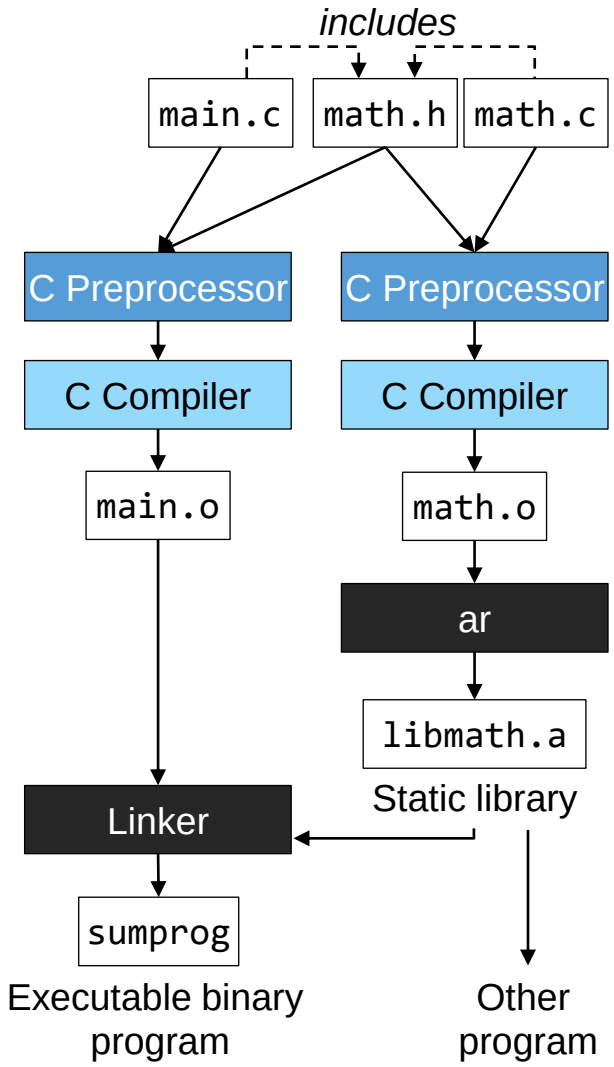
Num	Size	Type	Bind	.symtab
71	27	FUNC	GLOBAL	main ← Entry point
59	4	OBJECT	GLOBAL	myvar
63	20	FUNC	GLOBAL	sum

```
#include "math.h"                          #include "math.h"                    .text
int main() {                               int myvar;
119d:  55              push ebp
119e:  89 e5           mov  ebp, esp       int sum(int a, int b) {
myvar = 3;                                 11b8:  55              push ebp
11a0:  C7 05 0c 40 00  mov  [400c], 5      11b9:  89 e5           mov  ebp, esp
       00 03 00 00 00                      return a+b+myvar;
return sum(2, 8); // 13                    11bb:  8b 55 08        mov  edx, [ebp+0x8]
11aa:  6A 08           push 0x8            11be:  8b 45 0C        mov  eax, [ebp+0xc]
11ac:  6A 02           push 0x2            11c1:  01 C2           add  edx, eax
11ae:  E8 05 00 00 00  call 11b8 <sum>     11c3:  A1 0C 40 00 00  mov  eax, [400c]
11b3:  83 C4 08        add  esp, 0x8       11c8:  01 D0           add  eax, edx
}                                          }
11b6:  C9              leave               11ca:  5D              pop  ebp
11b7:  C3              ret                 11cb:  C3              ret
```

Static Libraries (*.a)

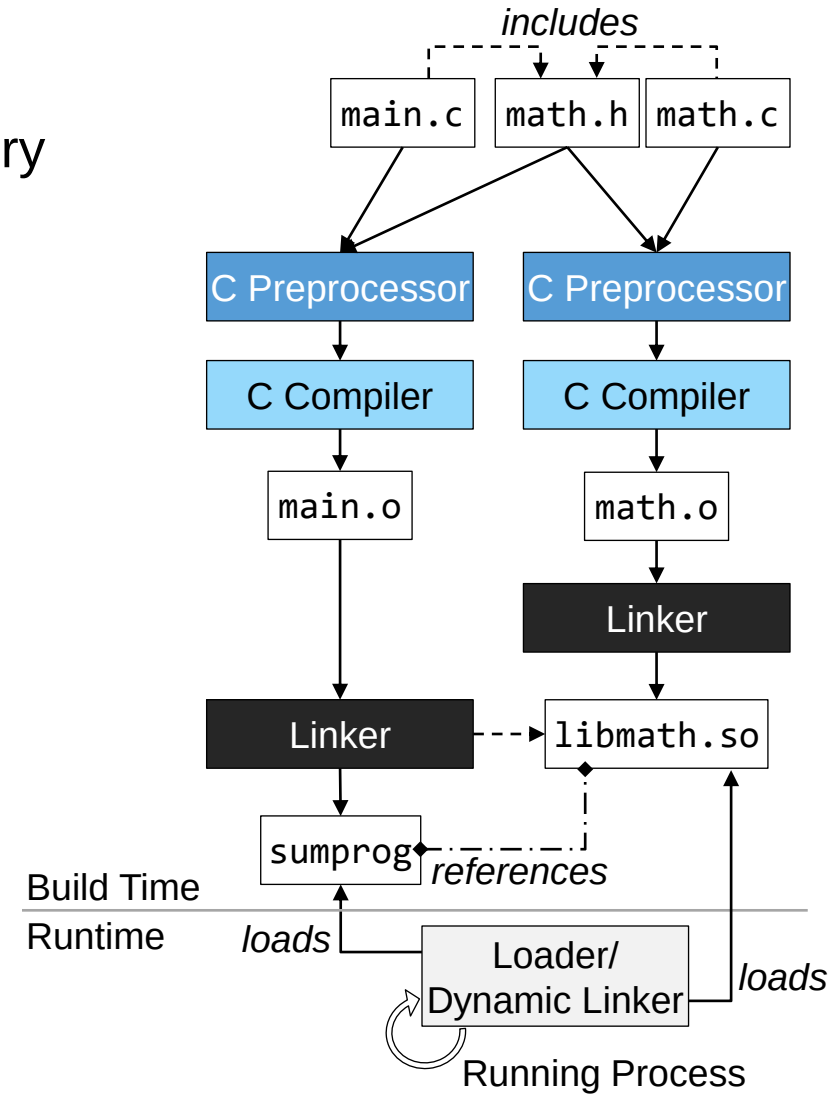
- Just archive of object files
  - Example: Collection of different utility functions (math, strings, …)
  - `ar rcs libX.a file1.o file2.o`

- Can be linked into executables just like normal object files
  - \+ Library calls as fast as local calls
  - \+ Only referenced symbols included
  - \- No sharing between processes
  - \- Need to recompile whole program on library update



Dynamic Shared Libraries (*.so)

- Loaded and linked at runtime
  - Only single copy in physical memory
  - Shared between all processes that reference it
- But: Address in virtual memory may vary between processes
  - Address range already occupied
  - Address Space Layout Randomization (ASLR)
- Relocation 👎
  - Time-consuming (→load time)
  - Prevents sharing



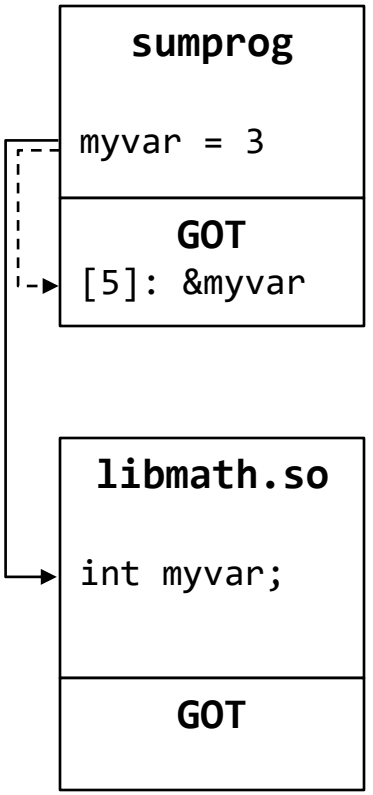
Global Offset Table (GOT)

- Idea: Position-Independent Code (PIC)
  - Can be placed at arbitrary address
  - Uses relative addressing only (no patching!)

5 bytes

```
11ae:  E8 05 00 00 00  call 11b8 <sum>
11b3:  …
int sum(int a, int b) {
11b8:  55              push ebp
```

  - Global symbols accessed via GOT



- Global Offset Table
  - Holds addresses of all external symbols
  - Created by static linker as part of binary
  - Initialized by dynamic linker at runtime

```
myvar = 3;
11bb:  mov edx, [ebx+0x20]
11c1:  mov [edx], 0x3
```

Address of GOT (fixed offset from code)

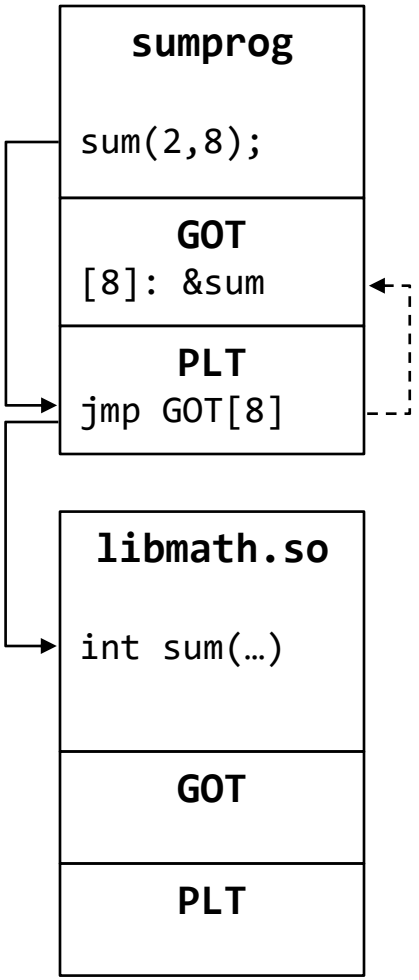
Offset of `myvar` in GOT

Procedure Linkage Table (PLT)

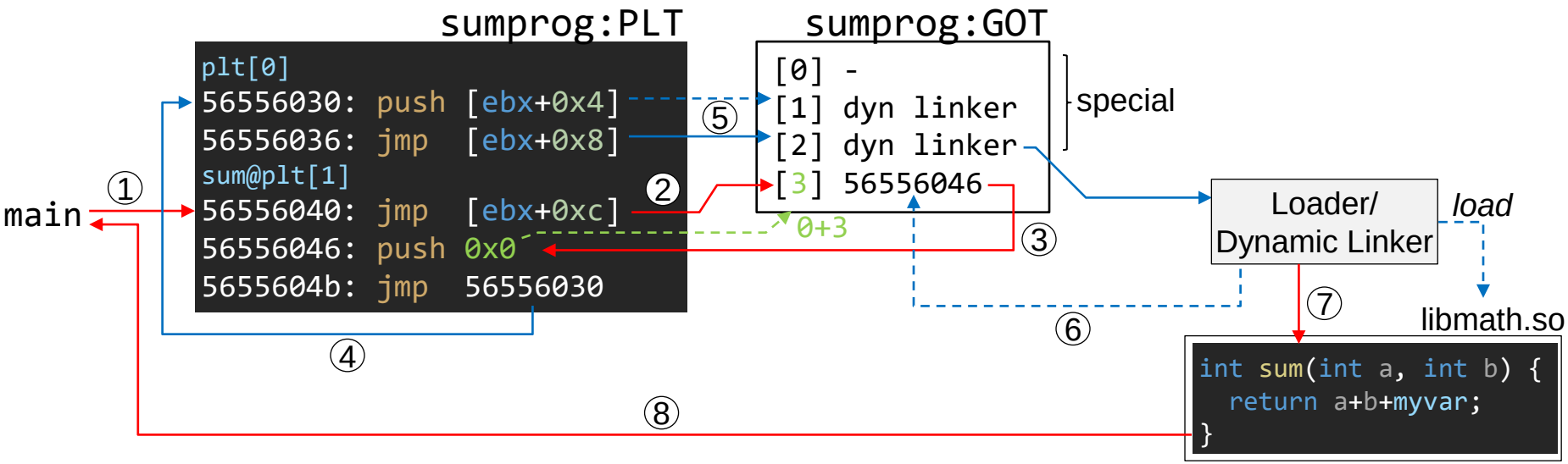
- Initialization of GOT costly
  - Not all program executions call all imported functions → wasted time
  - Want to load libraries only when needed (lazy loading)
- Procedure Linkage Table
  - Function calls into PLT entry

```
return sum(2, 8);
11c7:  push 0x8
11c9:  push 0x2
11cb:  call <sum@plt>
```

  - PLT entry redirects first call to dynamic linker to initialize GOT entry
  - Following calls into PLT entry jump to sum()
  - ➢ Lazy binding

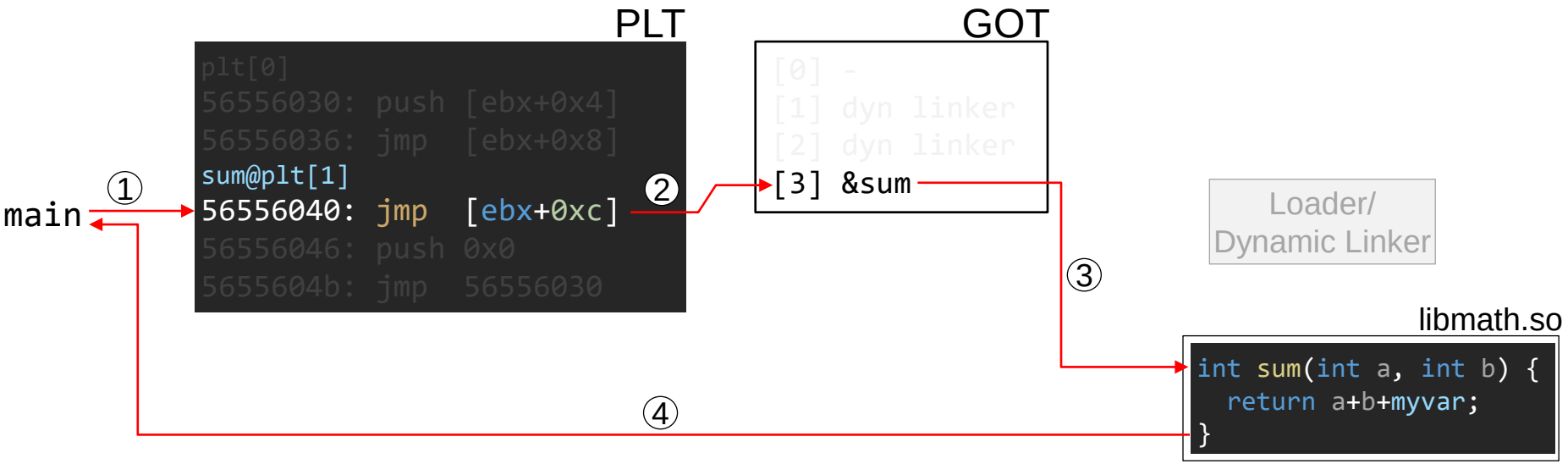


Lazy Binding – First Call



1. main() calls into sum@plt to jump to sum() in libmath.so
2. PLT entry takes address from sum()'s GOT entry
3. GOT entry still points to next instruction in sum@plt. Jump.
   CPU pushes index of sum()'s GOT entry onto stack
4. Jump to first PLT entry. This is reserved for dynamic linker
5. Jump to dynamic linker via GOT entry (initialized by linker at load)
6. Update of GOT entry to point to sum() in libmath.so (possibly loaded first if lazy loading)
7. Jump to sum()
8. Return to main()

Lazy Binding – Following Calls



1. main() calls into sum@plt to jump to sum() in libmath.so
2. PLT entry takes address from sum()'s GOT entry
3. Jump to sum()
4. Return to main()

Summary

- Put public declarations into header files
  - Use include guards
- Calling conventions: cdecl
  - Parameters passed via stack
  - Return value passed via EAX:EDX
  - Caller clears parameter stack space
- Static vs. dynamic linking
  - Global offset table (GOT)
  - Procedure linkage table (PLT)
  - Lazy binding + lazy loading

Further Reading

- “The C Programming Language” by Kernighan and Ritchie
- comp.lang.c Frequently Asked Questions (http://c-faq.com/)