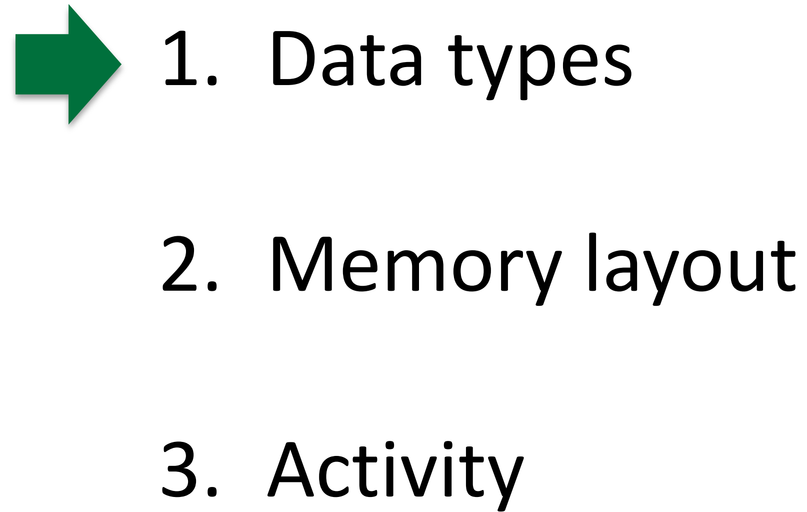
CS 50: Software Design and Implementation

C basics

Agenda



C allows more precision when allocating variables than other languages

Data type	Print	Description
void		the void type
bool	%d	the Boolean type, representing true/false

C allows more precision when allocating variables than other languages

Data type	Print	Description
void		the void type
bool	%d	the Boolean type, representing true/false
Integers		
char	%c	the character type
short	%h	the short integer type (sometimes shorter than int)
int	%d	the standard integer type
long	%ld	the longer integer type (sometimes longer than int)

C allows more precision when allocating variables than other languages

Data type	Print	Description
void		the void type
bool	%d	the Boolean type, representing true/false
Integers		
char	%c	the character type
short	%h	the short integer type (sometimes shorter than int)
int	%d	the standard integer type
long	%ld	the longer integer type (sometimes longer than int)
Floating point		
float	%f	the standard floating-point (real) type
double	%f	the extra precision floating-point type
long double	%LF	the super precision floating-point type

Integer types have different sizes and two versions; define range of possible values

Integer types

Typical size, not guaranteed to this width on every system!

Use sizeof() to check size

Data type	Typical Size (bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
int	4	-2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4,294,967,295
long int	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long int	8	0 to 18,446,744,073,709,551,615

Signed version (default), most significant bit is sign (0=positive, 1=negative)
Unsigned version, most significant bit part of value (so 2 times larger possible)

Adapted from https://www.tutorialspoint.com/cprogramming/c_data_types.htm

Integer data types have a signed and an unsigned version that affects their range

Integer types

Data type	**Typical Size (bytes)**	**Range**
char	1	-128 to 127
unsigned char	1	0 to 255
int	4	-2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4,294,967,295
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
long int	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long int	8	0 to 18,446,744,073,709,551,615
bool	**1 bit**	**0 or 1**

Include <stdbool.h> to get Boolean data type
Often see `int` used for Boolean where 0 = false, everything else is true

Adapted from https://www.tutorialspoint.com/cprogramming/c_data_types.htm

Floating point data types can hold numeric values with decimal components

Floating point types

Data type	Typical size (bytes)	Range	Decimal places
float	4	1.2E-38 to 3.4E+38	6 decimal places
double	8	2.3E-308 to 1.7E+308	15 decimal places
long double	10	3.4E-4932 to 1.1E+4932	19 decimal places

Adapted from https://www.tutorialspoint.com/cprogramming/c_data_types.htm

Use `sizeof` to check how many bytes each type takes

range.c

```
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <limits.h>
15 #include <float.h>
16 #include <stdbool.h>
17
18 int main(int argc, char** argv) {
19     //note: sizeof returns an unsigned long
20    printf("Integers\tBytes \tMin \t\tMax\n");
21     printf("boolean\t\t%ld\t%d\t\t%d\n",sizeof(bool),0,1);
22    printf("char\t\t%ld\t%d\t\t%d\n",sizeof(bool),CHAR_MIN,CHAR_MAX);
23     printf("short\t\t%ld\t%d\t\t%d\n",sizeof(short),SHRT_MIN,SHRT_MAX);
24     printf("int\t\t%ld\t%d\t%d\n",sizeof(int),INT_MIN,INT_MAX);
25     //casting long min and max as double for formatting in scientific notation
26     printf("long\t\t%ld\t%e\t%e\n",sizeof(long),(double)LONG_MIN,(double)LONG_MAX);
27
28     printf("\nFloating points\n");
29     printf("float\t\t%ld\t%e\t%e\n",sizeof(float),FLT_MIN,FLT_MAX);
30     printf("double\t\t%ld\t%e\t%e\n",sizeof(double),DBL_MIN,DBL_MAX);
31
32     return 0;
33 }
```

Use sizeof to check how many bytes each type takes

range.c

```
$ mygcc range.c
$ ./a.out
Integers Bytes Min           Max
boolean  1     0             1
char     1     -128          127
short    2     -32768        32767
int      4     -2147483648   2147483647
long     8     -9.223372e+18 9.223372e+18

Floating points
float    4     1.175494e-38  3.402823e+38
double   8     2.225074e-308 1.797693e+308
```

```
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <limits.h>
15 #include <float.h>
16 #include <stdbool.h>
17
18 int main(int argc, char** argv) {
19     //note: sizeof returns an unsigned long
20    printf("Integers\tBytes \tMin \t\tMax\n");
21     printf("boolean\t\t%ld\t%d\t\t%d\n",sizeof(bool),0,1);
22    printf("char\t\t%ld\t%d\t\t%d\n",sizeof(bool),CHAR_MIN,CHAR_MAX);
23     printf("short\t\t%ld\t%d\t\t%d\n",sizeof(short),SHRT_MIN,SHRT_MAX);
24     printf("int\t\t%ld\t%d\t%d\n",sizeof(int),INT_MIN,INT_MAX);
25     //casting long min and max as double for formatting in scientific notation
26     printf("long\t\t%ld\t%e\t%e\n",sizeof(long),(double)LONG_MIN,(double)LONG_MAX);
27
28     printf("\nFloating points\n");
29     printf("float\t\t%ld\t%e\t%e\n",sizeof(float),FLT_MIN,FLT_MAX);
30     printf("double\t\t%ld\t%e\t%e\n",sizeof(double),DBL_MIN,DBL_MAX);
31
32     return 0;
33 }
```

Use sizeof to check how many bytes each type takes

range.c

```
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <limits.h>
15 #include <float.h>
16 #include <stdbool.h>
17
18 int main(int argc, char** argv) {
19     //note: sizeof returns an unsigned long
20    printf("Integers\tBytes \tMin \t\tMax\n");
21     printf("boolean\t\t%ld\t%d\t\t%d\n",sizeof(bool),0,1);
22    printf("char\t\t%ld\t%d\t\t%d\n",sizeof(bool),CHAR_MIN,CHAR_MAX);
23     printf("short\t\t%ld\t%d\t\t%d\n",sizeof(short),SHRT_MIN,SHRT_MAX);
24     printf("int\t\t%ld\t%d\t%d\n",sizeof(int),INT_MIN,INT_MAX);
25     //casting long min and max as double for formatting in scientific notation
26     printf("long\t\t%ld\t%e\t%e\n",sizeof(long),(double)LONG_MIN,(double)LONG_MAX);
27
28     printf("\nFloating points\n");
29     printf("float\t\t%ld\t%e\t%e\n",sizeof(float),FLT_MIN,FLT_MAX);
30     printf("double\t\t%ld\t%e\t%e\n",sizeof(double),DBL_MIN,DBL_MAX);
31
32     return 0;
33 }
```

```
$ mygcc range.c
$ ./a.out
Integers Bytes Min           Max
boolean  1     0             1
char     1     -128          127
short    2     -32768        32767
int      4     -2147483648   2147483647
long     8     -9.223372e+18 9.223372e+18

Floating points
float    4     1.175494e-38  3.402823e+38
double   8     2.225074e-308 1.797693e+308
```

Chose the size that fits your needs

If you know you will not need large integer numbers, can choose short (2 bytes) instead of int (4 bytes)

Use sizeof to check how many bytes each type takes

range.c

```
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <limits.h>
15 #include <float.h>
16 #include <stdbool.h>
17
18 int main(int argc, char** argv) {
19     //note: sizeof returns an unsigned long
20    printf("Integers\tBytes \tMin \t\tMax\n");
21     printf("boolean\t\t%ld\t%d\t\t%d\n",sizeof(bool),0,1);
22    printf("char\t\t%ld\t%d\t\t%d\n",sizeof(bool),CHAR_MIN,CHAR_MAX);
23     printf("short\t\t%ld\t%d\t\t%d\n",sizeof(short),SHRT_MIN,SHRT_MAX);
24     printf("int\t\t%ld\t%d\t%d\n",sizeof(int),INT_MIN,INT_MAX);
25     //casting long min and max as double for formatting in scientific notation
26     printf("long\t\t%ld\t%e\t%e\n",sizeof(long),(double)LONG_MIN,(double)LONG_MAX);
27
28     printf("\nFloating points\n");
29     printf("float\t\t%ld\t%e\t%e\n",sizeof(float),FLT_MIN,FLT_MAX);
30     printf("double\t\t%ld\t%e\t%e\n",sizeof(double),DBL_MIN,DBL_MAX);
31
32     return 0;
33 }
```

```
$ mygcc range.c
$ ./a.out
Integers Bytes Min           Max
boolean  1     0             1
char     1     -128          127
short    2     -32768        32767
int      4     -2147483648   2147483647
long     8     -9.223372e+18 9.223372e+18

Floating points
float    4     1.175494e-38  3.402823e+38
double   8     2.225074e-308 1.797693e+308
```

Chose the size that fits your needs

Same for float (4 bytes) vs double (8 bytes)

Other data types include arrays, strings, structs, and pointers

Array	Contiguous block of memory that holds multiple items of the same type Zero-indexed	int myNumbers[] = {25, 50, 75, 100}; printf("%d", myNumbers[0]); //25

Other data types include arrays, strings, structs, and pointers

Array	Contiguous block of memory that holds multiple items of the same type Zero-indexed	int myNumbers[] = {25, 50, 75, 100}; printf("%d", myNumbers[0]); //25
String	An array of characters, terminated by a \0 character	char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0’}; printf(“%s\n”, greeting);

Other data types include arrays, strings, structs, and pointers

Array	Contiguous block of memory that holds multiple items of the same type Zero-indexed	`int myNumbers[] = {25, 50, 75, 100};` `printf("%d", myNumbers[0]); //25`
String	An array of characters, terminated by a \0 character	`char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0’};` `printf(“%s\n”, greeting);`

Struct	A little like an object in Java, but without code (data only)	`struct Books {`
		`  char title[50];`
		`  char author[50]`
		`};`

Other data types include arrays, strings, structs, and pointers

Array	Contiguous block of memory that holds multiple items of the same type Zero-indexed	`int myNumbers[] = {25, 50, 75, 100};` `printf("%d", myNumbers[0]); //25`
String	An array of characters, terminated by a \0 character	`char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0’};` `printf(“%s\n”, greeting);`

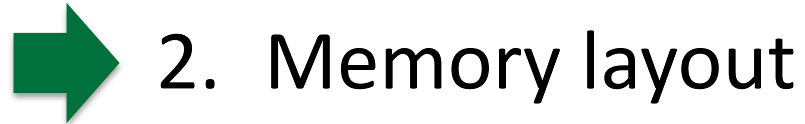
Struct	A little like an object in Java, but without code (data only)	`struct Books {`
		`  char title[50];`
		`  char author[50]`
		` };`

Pointer	A variable that stores the address of another variable	`char *p;`

Will cover these soon!

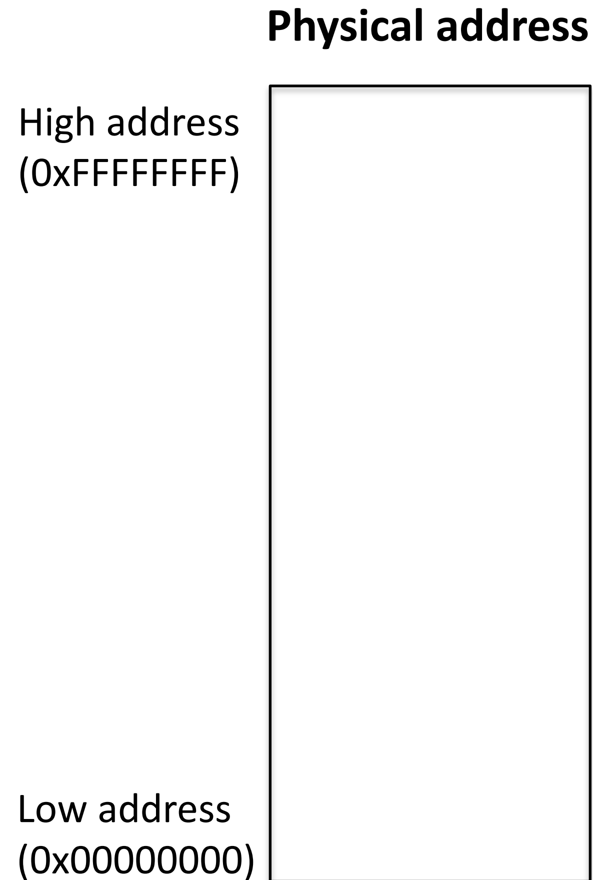
Agenda

1. Data types



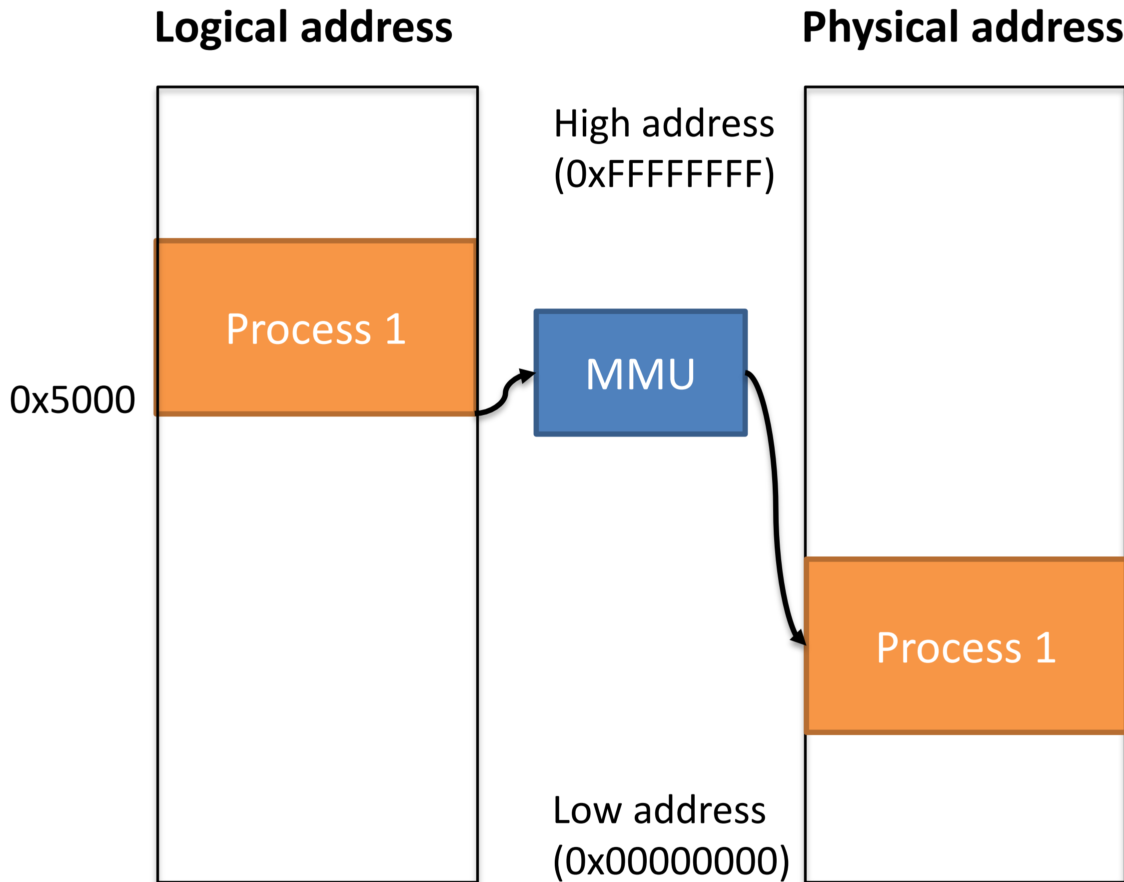
2. Memory layout
3. Activity

Physical memory is addressed from low to high



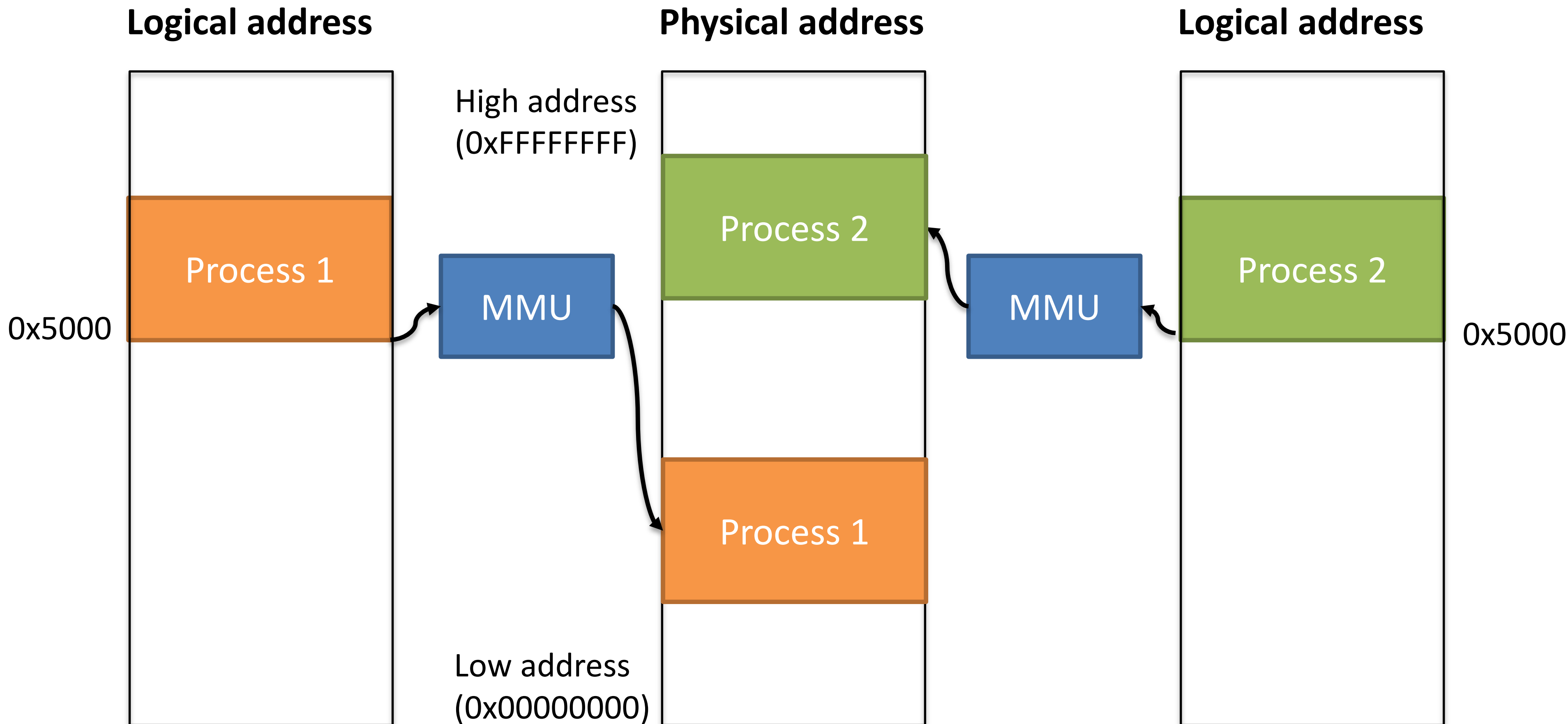
Adapted from *Computer and Internet Security* by Du

When a process allocates memory, MMU maps from the logical address to physical



Adapted from *Computer and Internet Security* by Du

Another process can allocate same logical address, but will map to different physical

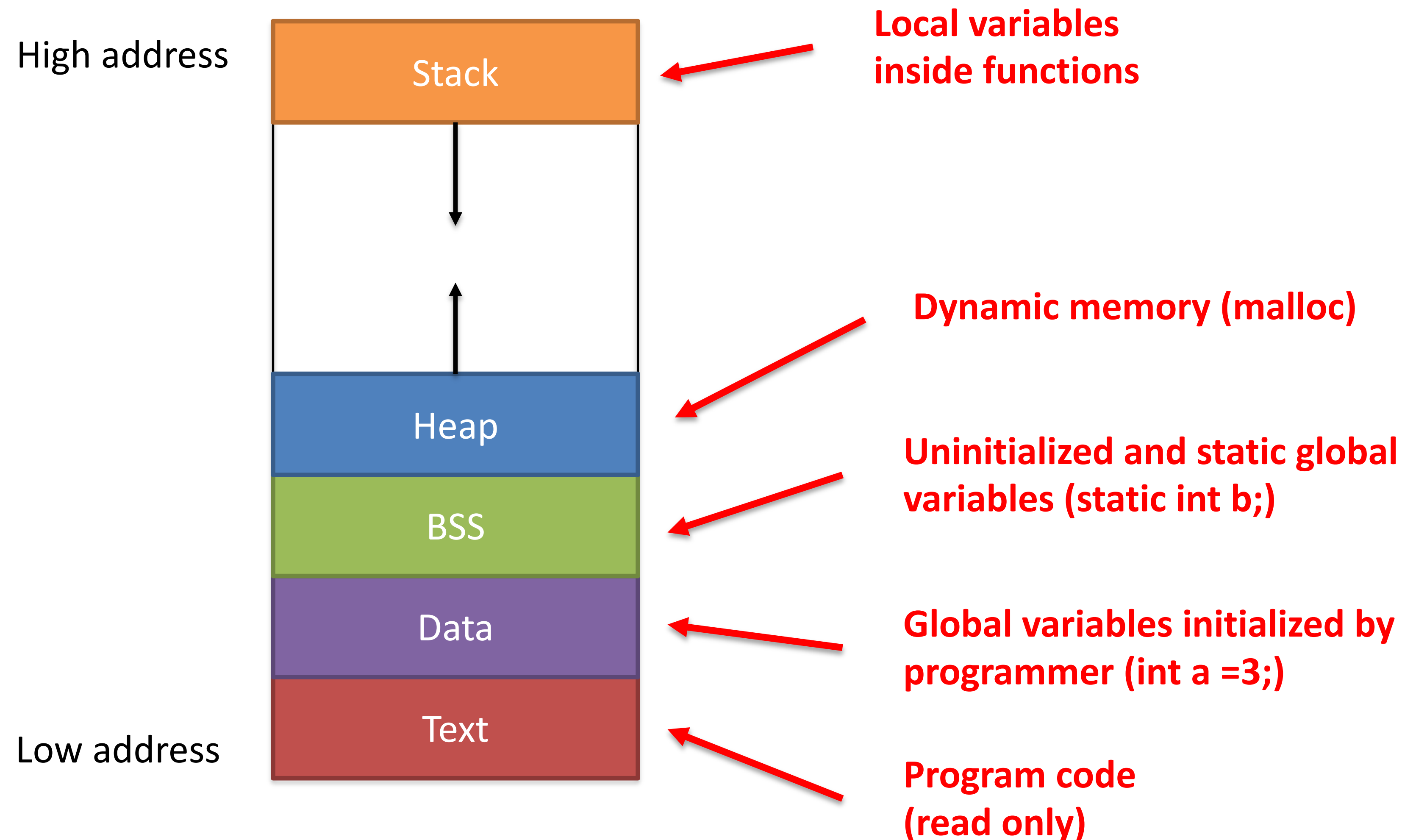


- Processes do not know exactly where they are in physical memory
- Process reference virtual address space as if it was all available to them
- MMU converts logical address to physical address in RAM

Adapted from *Computer and Internet Security* by Du

Virtual memory is laid out so that the heap and stack grow toward each other

Linux virtual memory layout

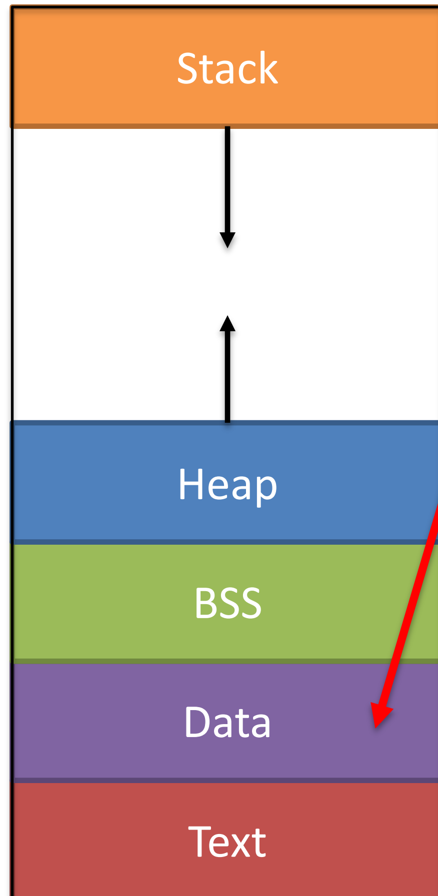


Adapted from *Computer and Internet Security* by Du

Virtual memory is laid out so that the heap and stack grow toward each other

Linux virtual memory layout

High address



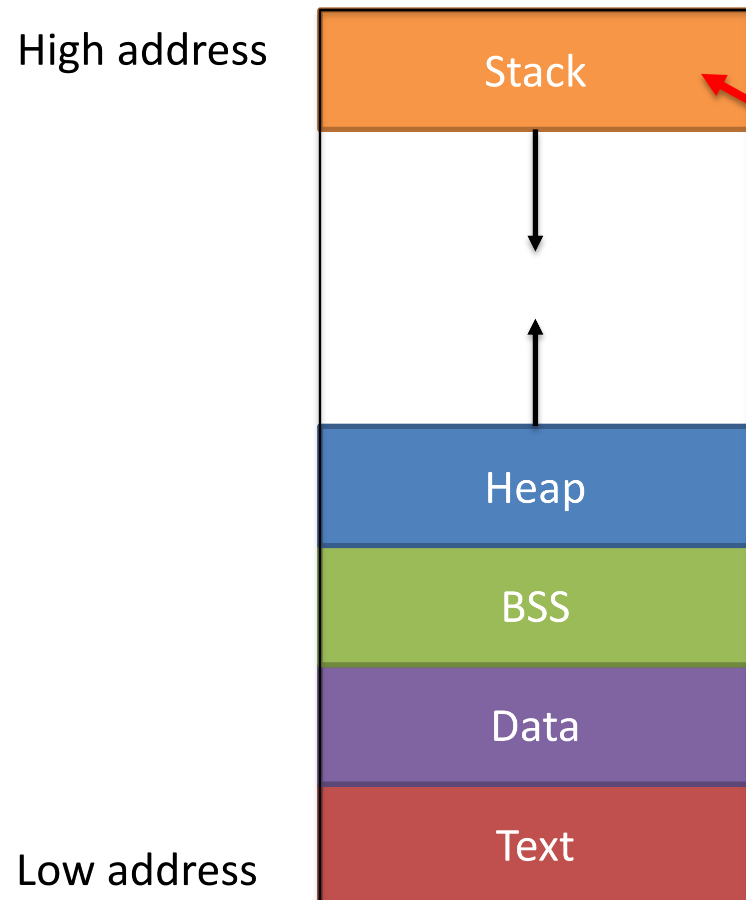
Low address

```c
int x = 100; //allocated in data segment

void main() {
	//allocated on stack
	int a=2;
	float b=2.5;

	//allocated on heap
	int *ptr = (int *)malloc(2*sizeof(int));

	//values 5 and 6 stored on heap
	ptr[0]=5;
	ptr[1]=6;

	//deallocate memory on heap
	free(ptr);
}
```


Adapted from *Computer and Internet Security* by Du

Virtual memory is laid out so that the heap and stack grow toward each other

Linux virtual memory layout

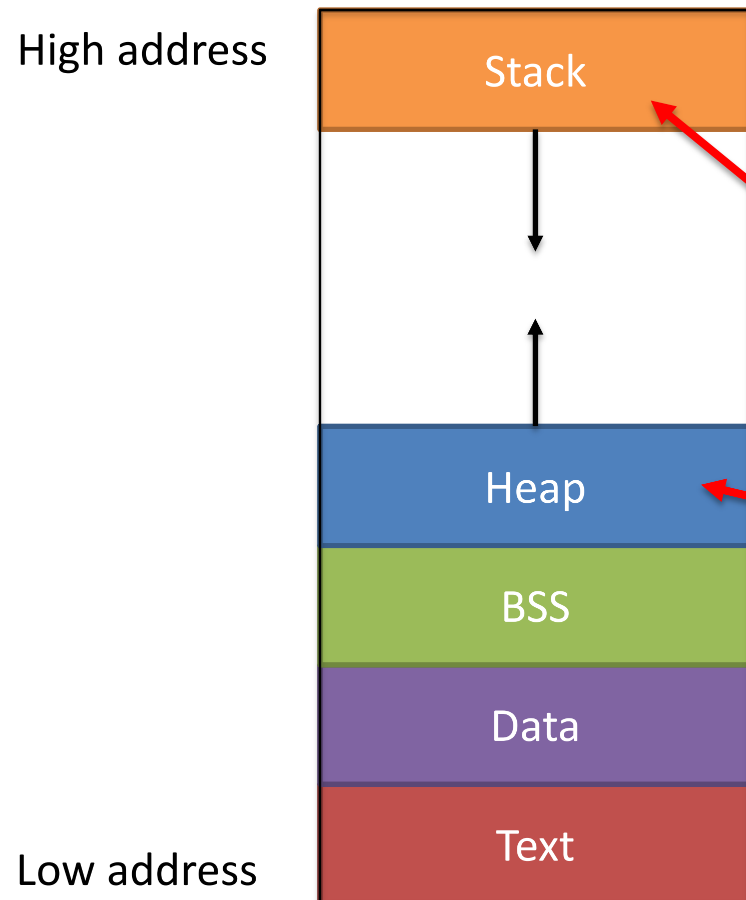



```
int x = 100; //allocated in data segment

void main() {
        //allocated on stack
        int a=2;
        float b=2.5;

        //allocated on heap
        int *ptr = (int *)malloc(2*sizeof(int));

        //values 5 and 6 stored on heap
        ptr[0]=5;
        ptr[1]=6;

        //deallocate memory on heap
        free(ptr);
}
```


Adapted from *Computer and Internet Security* by Du

Virtual memory is laid out so that the heap and stack grow toward each other

Linux virtual memory layout




```
int x = 100; //allocated in data segment

void main() {
        //allocated on stack
        int a=2;
        float b=2.5;

        //allocated on heap
        int *ptr = (int *)malloc(2*sizeof(int));

        //values 5 and 6 stored on heap
        ptr[0]=5;
        ptr[1]=6;

        //deallocate memory on heap
        free(ptr);
}
```

Note: *ptr* is allocated on the stack, memory it points to is on the heap

Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called

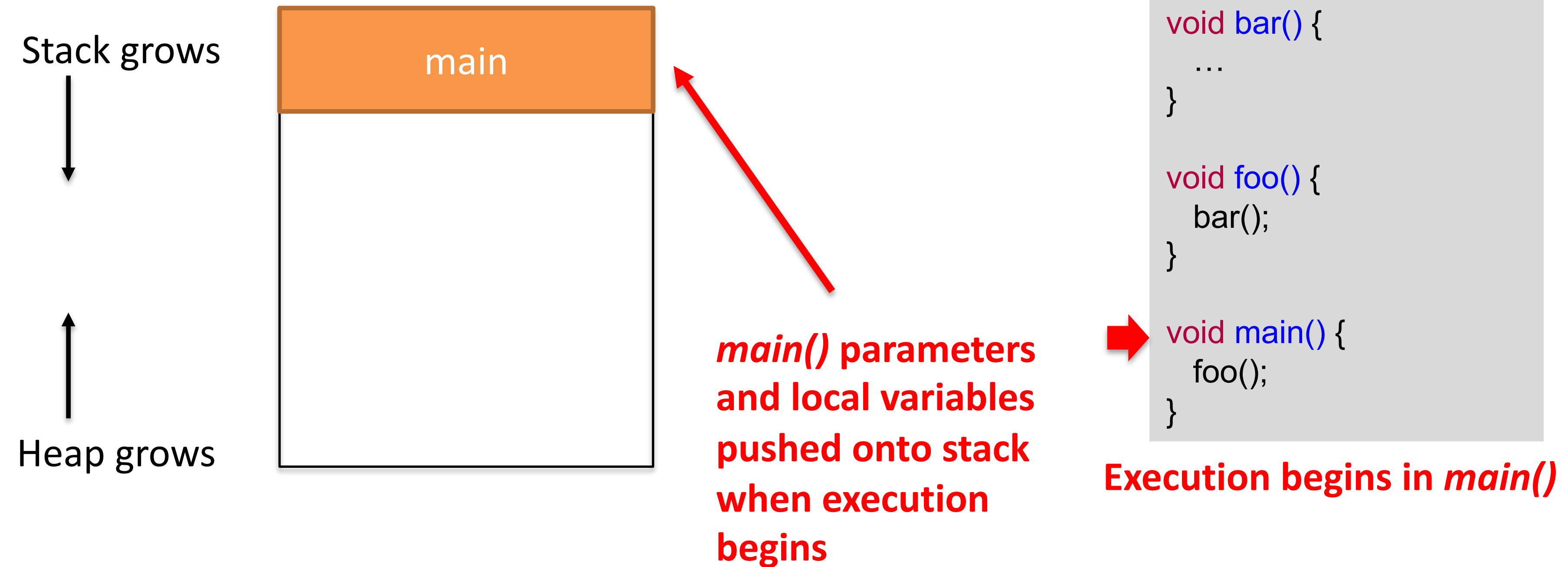
Stack

```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Execution begins in *main()*

Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called



Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called

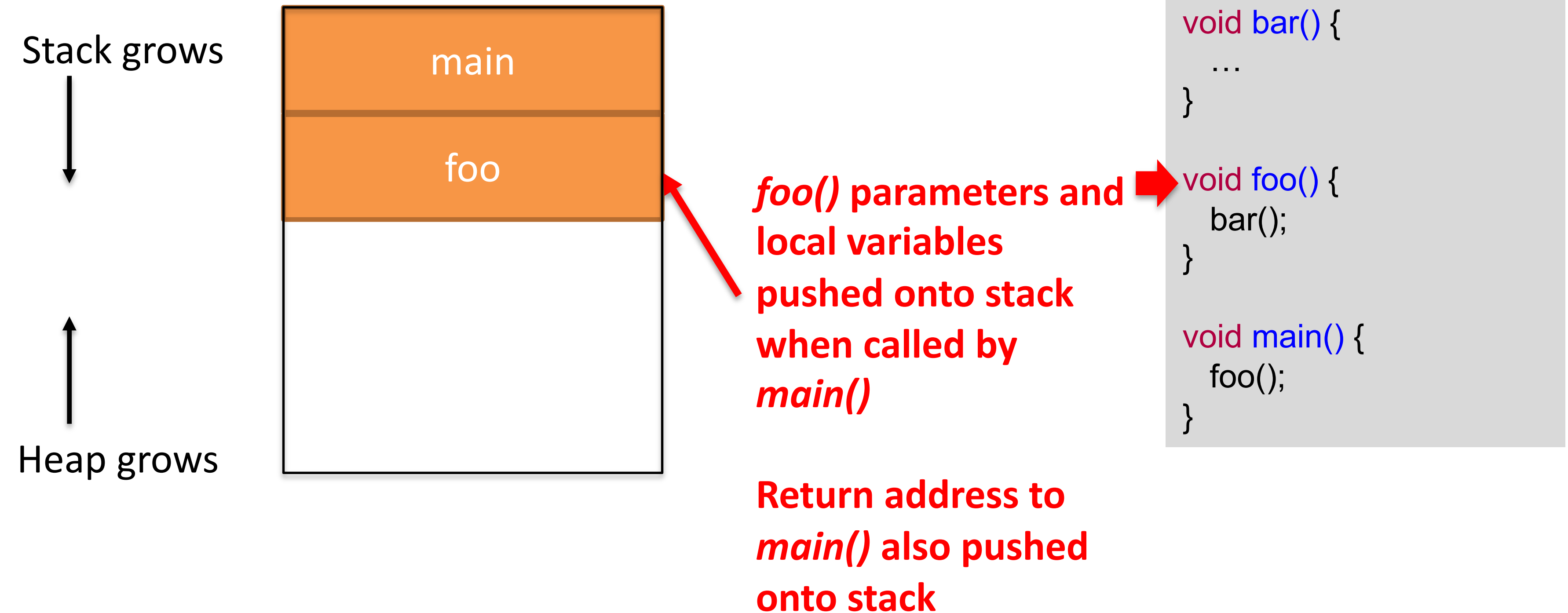
Stack

main() calls foo()

```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```


Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called

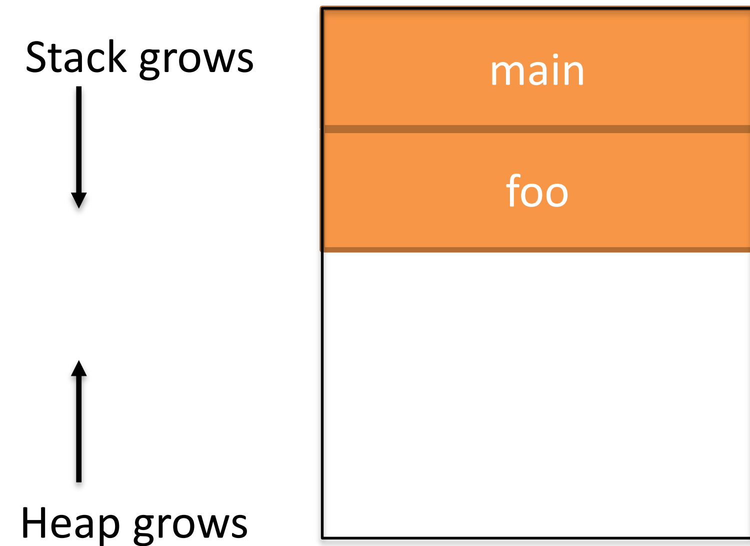


Return address to *main()* also pushed onto stack

Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called

Stack

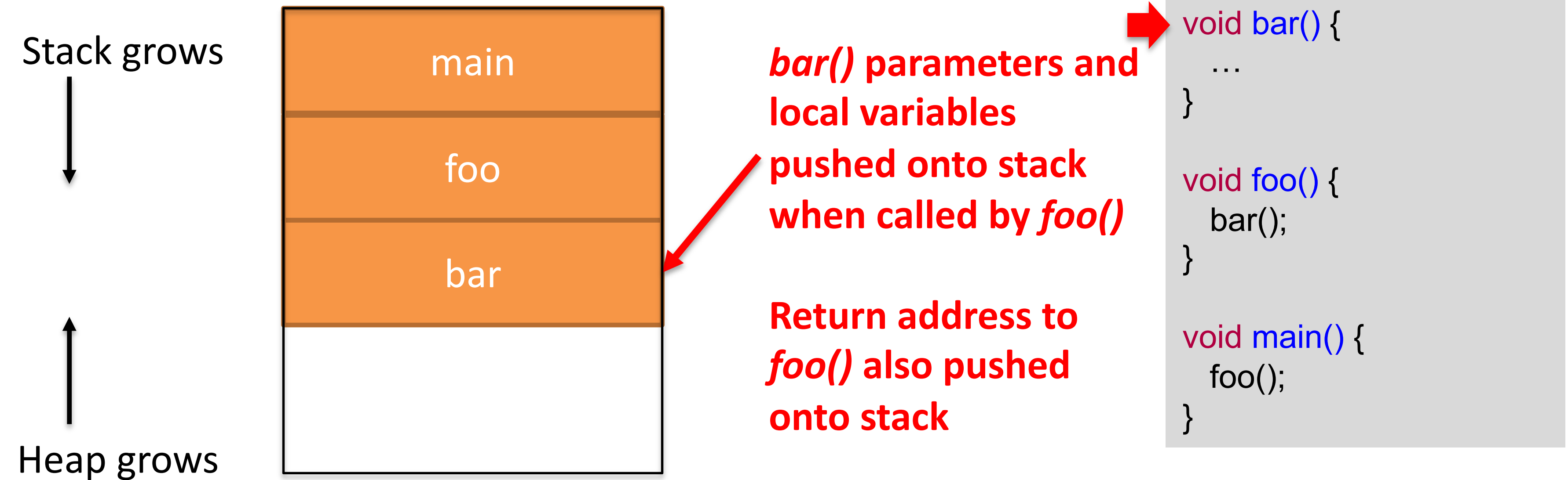


foo() calls bar()

```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```


Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called



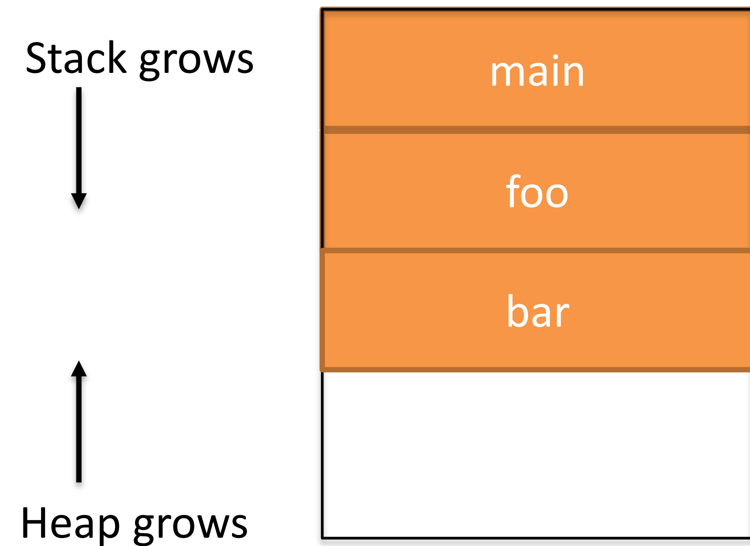
Functions popped from stack when they end

Recursion works by pushing new frames onto stack

Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called

Stack



***bar()* ends, popped from stack**

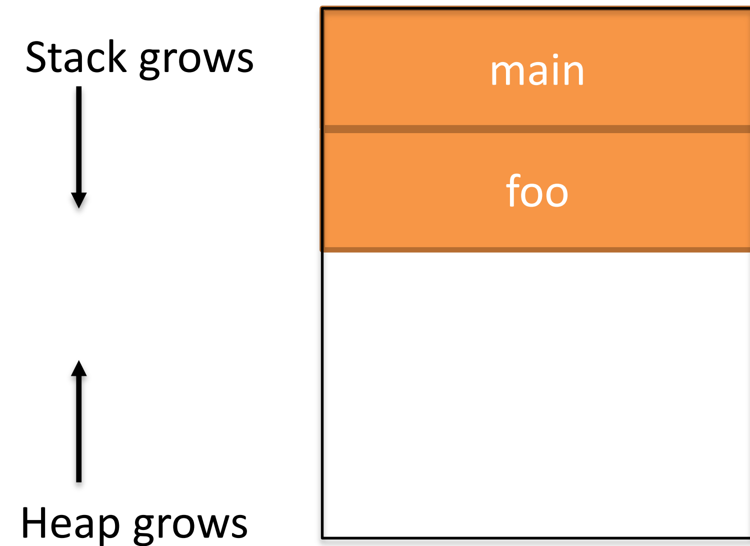
```
void bar() {
   ...
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Functions popped from stack when they end

Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called

Stack



bar() **ends, popped from stack**

Return address on stack allows execution to resume where ***bar()*** **was called**

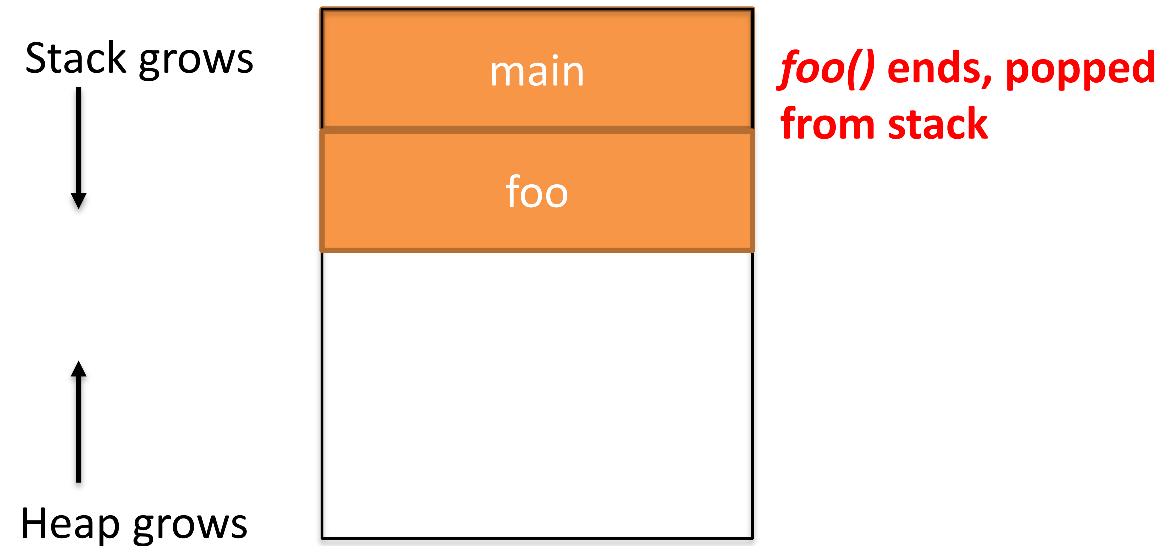
```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Functions popped from stack when they end

Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called

Stack



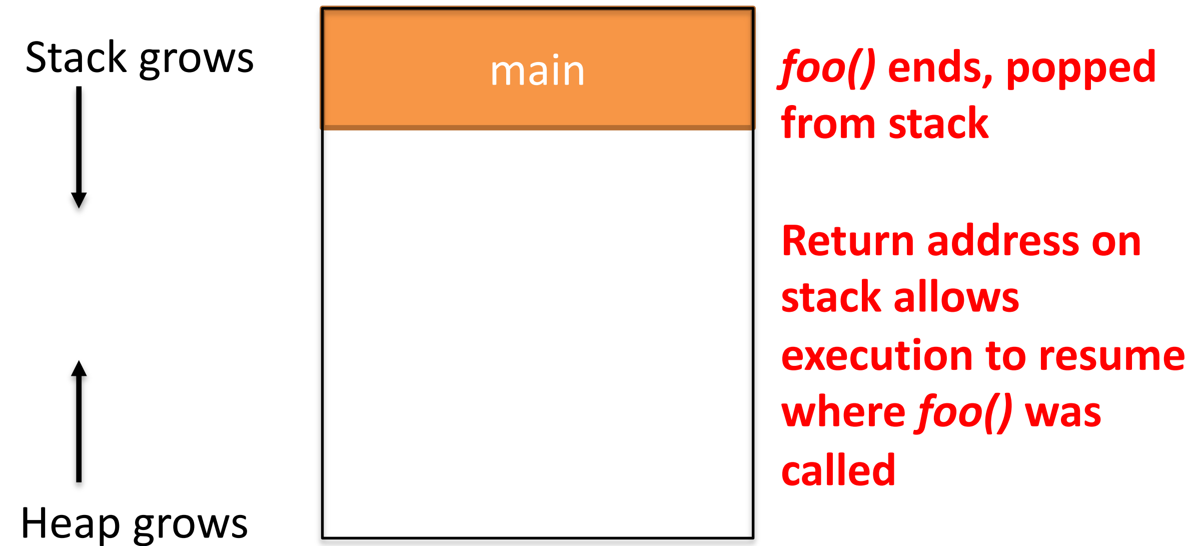

```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Functions popped from stack when they end

Adapted from *Computer and Internet Security* by Du

Frames are pushed onto the stack as functions are called

Stack

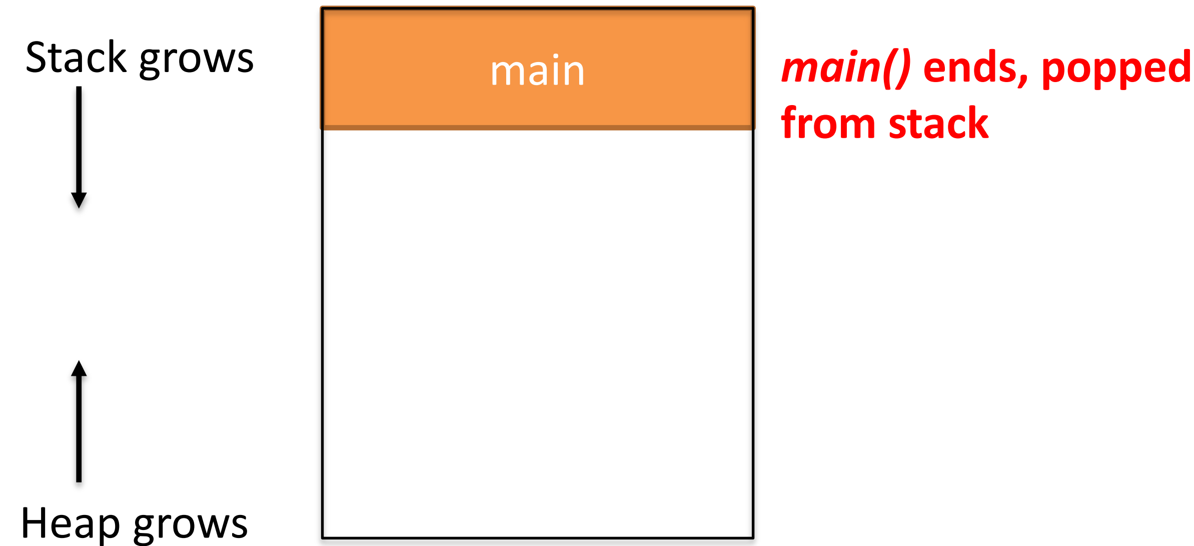



```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Functions popped from stack when they end

Frames are pushed onto the stack as functions are called

Stack




```
void bar() {
   …
}

void foo() {
   bar();
}

void main() {
   foo();
}
```

Functions popped from stack when they end

Static local variables are not stored on the stack and retain value between calls

static_test.c

```
#include <stdio.h>

int f() {
    static int i = 1;
    i++;
    return i;
}


int main() {
    printf("%d\n", f());
    printf("%d\n", f());
    return 0;
}
```

```
$ mygcc -o static_test static_test.c
$ ./static_test
2
3
```

Goes out of scope if not *static*
***Static* variable retains value between calls**
Somewhat like global variable, but only visible in this function
Stored in Data segment not on stack, so does not get popped from stack

Static global variables and static functions can only be called by functions in the same C program file in which the static global variable or function is defined

Use *extern* keyword to access functions and variables in another C file

We will soon get to multi-file programs!

If `i` is static, it is not allocated on the stack, let's confirm this is true

static_test.c

```
#include <stdio.h>

int f() {
    static int i = 1;
    i++;
    return i;
}


int main() {
    printf("%d\n", f());
    printf("%d\n", f());
    return 0;
}
```

-c flag says stop after compile (do not link)

objdump program dumps executable

-Sr interleaves source code

i is in the data segment

```
$ mygcc -c -o static_test static_test.c
$ objdump -Sr static_test

static_test:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <f>:
#include <stdio.h>
/* static_test.c - demonstrate static local variables
 *
 */
int f() {
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
    static int i = 1;
    i++;
   8: 8b 05 00 00 00 00     mov    0x0(%rip),%eax        # e <f+0xe>
a: R_X86_64_PC32 .data-0x4
   e: 83 c0 01              add    $0x1,%eax
  11: 89 05 00 00 00 00     mov    %eax,0x0(%rip)        # 17 <f+0x17>
13: R_X86_64_PC32 .data-0x4
    return i;
  17: 8b 05 00 00 00 00     mov    0x0(%rip),%eax        # 1d <f+0x1d>
19: R_X86_64_PC32 .data-0x4
}
  1d: 5d                    pop    %rbp
  1e: c3                    ret
```

If `i` is static, it is not allocated on the stack, let's confirm this is true

static_test.c

Not initializing puts i in the BSS segment

```
#include <stdio.h>

int f() {
    static int i;// = 1;
    i++;
    return i;
}


int main() {
    printf("%d\n", f());
    printf("%d\n", f());
    return 0;
}
```

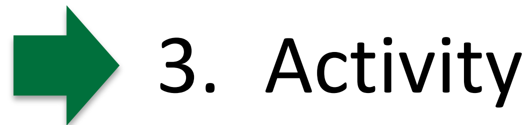
i is in the BSS segment

```
$ mygcc -c -o static_test static_test.c
$ objdump -Sr static_test

static_test:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <f>:
#include <stdio.h>
/* static_test.c - demonstrate static local variables
 *
 */
int f() {
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
    static int i = 1;
    i++;
   8: 8b 05 00 00 00 00       mov    0x0(%rip),%eax        # e <f+0xe>
a: R_X86_64_PC32 .bss-0x4
   e: 83 c0 01                add    $0x1,%eax
  11: 89 05 00 00 00 00       mov    %eax,0x0(%rip)        # 17 <f+0x17>
13: R_X86_64_PC32 .bss-0x4
    return i;
  17: 8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 1d <f+0x1d>
19: R_X86_64_PC32 .bss-0x4
}
  1d: 5d                      pop    %rbp
  1e: c3                      ret
```

Agenda

1. Data types

2. Memory layout



3. Activity