Notes on Heap Manager in C

Jeff Chase

Duke University

Basic hints on using Unix

- Find a properly installed Unix system: linux.cs.duke.edu, or MacOS with Xcode will do nicely.
- Learn a little about the Unix shell command language. On MacOS open the standard **Terminal** utility.
- Learn some basic commands: **cd, ls, cat, grep, more/less, pwd, rm, cp, mv**, **diff**, and an editor of some kind (**vi, emacs**, …). Spend one hour.
- Learn basics of **make**. Look at the **makefile**. Run "**make –i**" to get it to tell you what it is doing. Understand what it is doing.
- Wikipedia is a good source for basics. Use the **man** command to learn about commands (1), syscalls (2), or C libraries (3). E.g.: type "man man".
- Know how to run your programs under a debugger: **gdb**. If it crashes you can find out where. It's easy to set breakpoints, print variables, etc.
- If your program doesn't compile, deal with errors from the top down. Try "make >out 2>out". It puts all output in the file "out" to examine at leisure.
- Keep source in gitlab.cs.duke.edu and **Do. Not. Share. It.**

The “shell”: Unix CLI

```
chase:scratch> curl http:…/lab1.tgz >lab1.tgz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100212k  100  212k    0     0   148k      0  0:00:01  0:00:01 --:--:--  163k
chase:scratch> ls
lab1.tgz
chase:scratch> tar zxvf lab1.tgz
…
chase:scratch> ls
lab1.tgz
chase:scratch> cd lab1
chase:lab1> ls
Makefile dmm.c    lab1.pdf  test_coalesce.c    test_stress2.c
README            dmm.h    test_basic.c   test_stress1.c      test_stress3.c
chase:lab1>
```

CLI == Command Line Interface

E.g., the Terminal application on MacOS

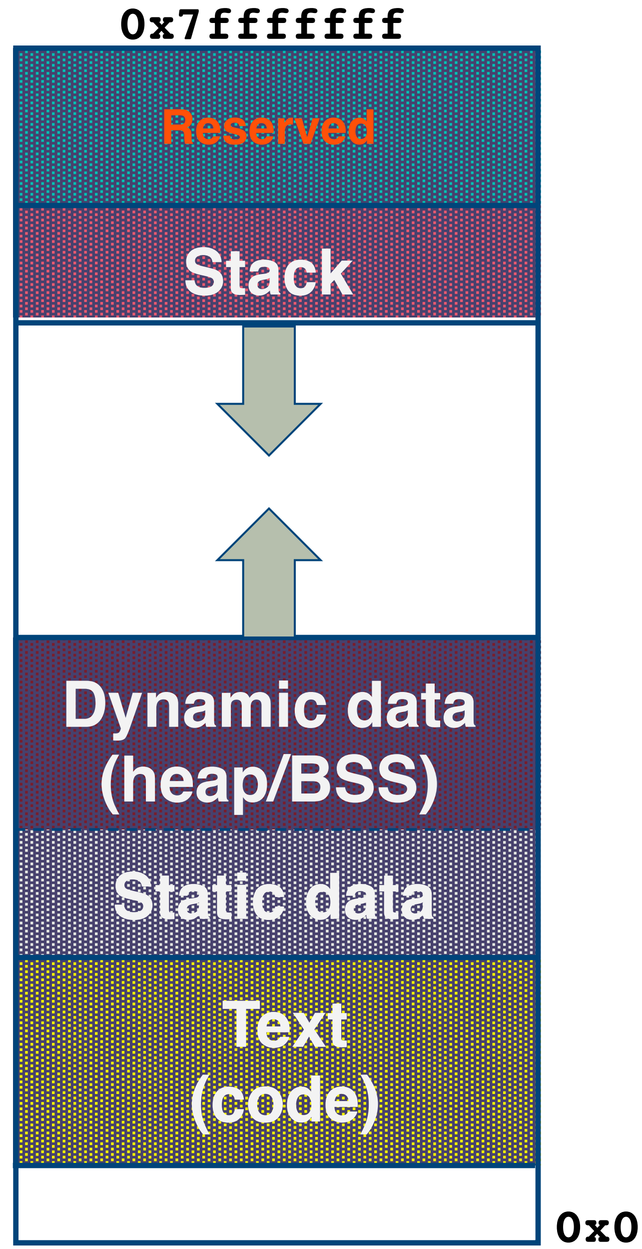
A simple C program

```
int
main()
{

}
```


78024
78034
78044
78015
78025
78035
78045
78016
78026
78036
78046
78017

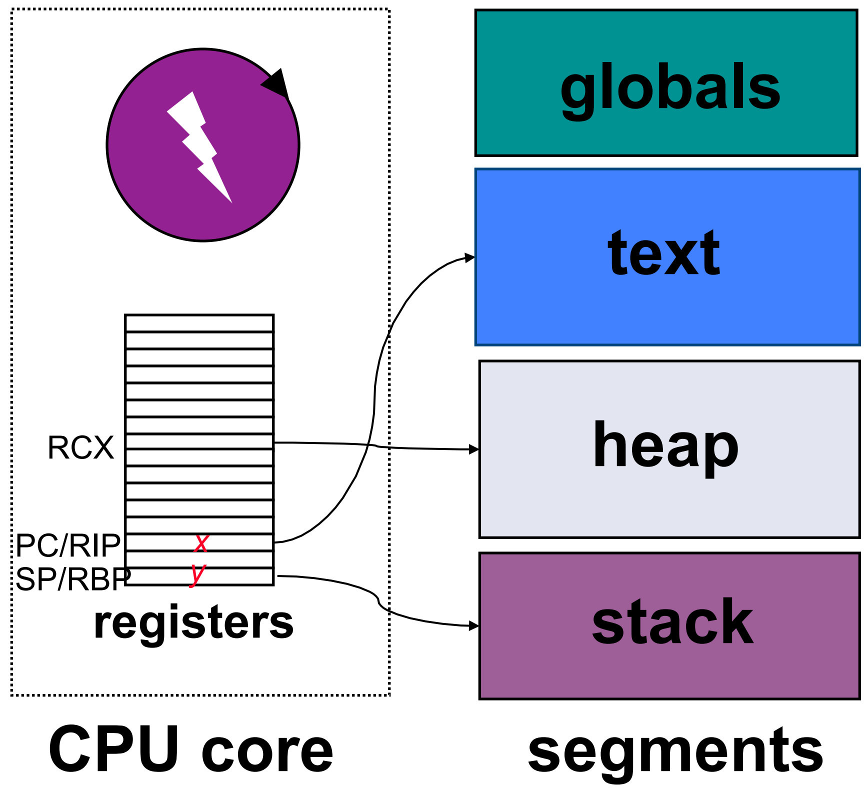
VAS example (32-bit)

- The program uses virtual memory through its process' Virtual Address Space:
- An addressable array of bytes…
- Containing every instruction the process thread can execute…
- And every piece of data those instructions can read/write…
  - i.e., read/write == **load/store** on memory
- Partitioned into logical **segments** (**regions**) with distinct purpose and use.
- Every memory reference by a thread is interpreted in the context of its VAS.
  - Resolves to a location in machine memory



Memory segments: a view from C

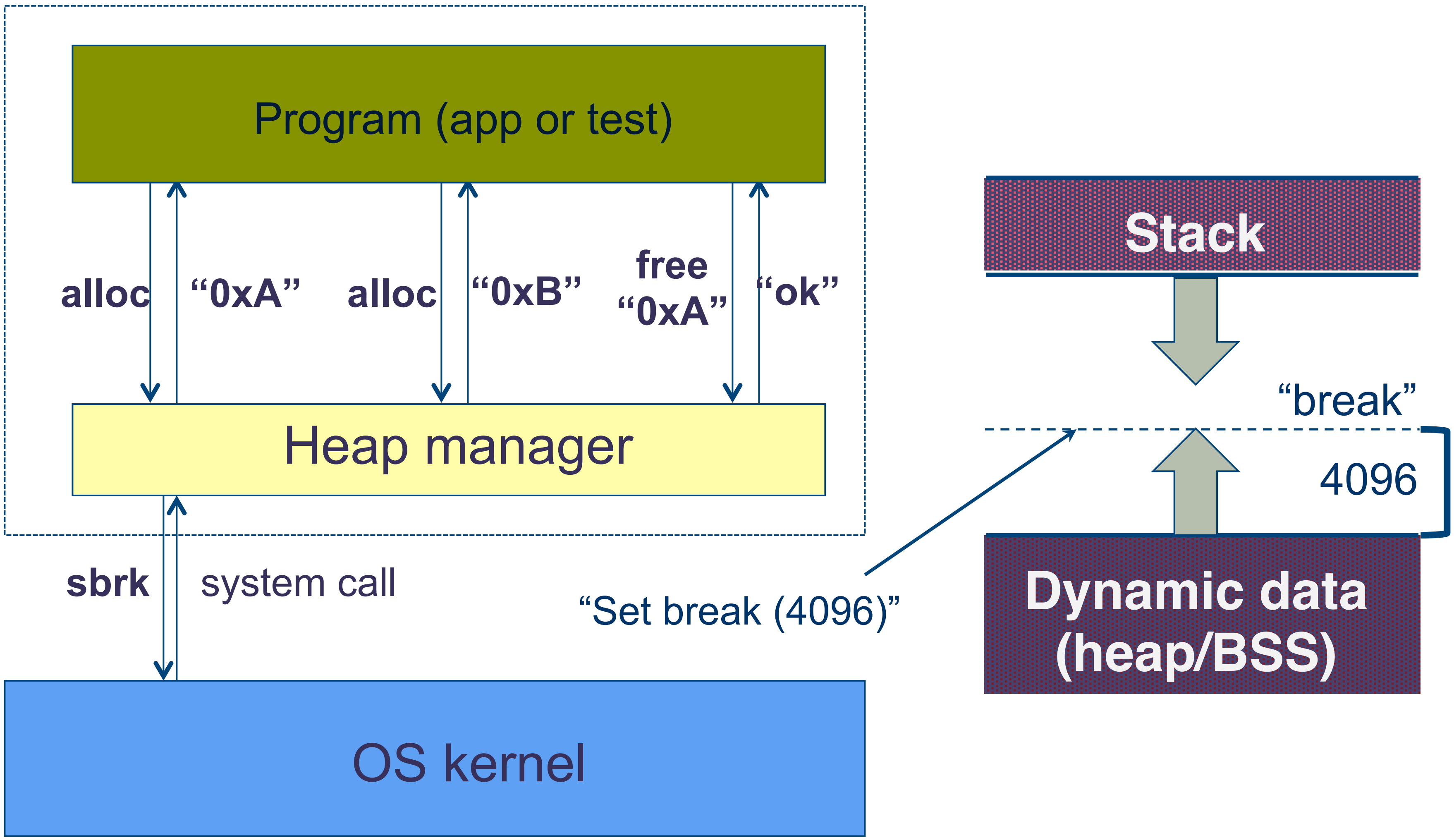
- **Globals:**
  - Fixed-size segment
  - Writable by user program
  - May have initial values
- **Text (instructions)**
  - Fixed-size segment
  - Executable
  - Not writable
- **Heap and stack**
  - Variable-size segments
  - Writable
  - Zero-filled on demand



Heap abstraction, simplified

1. User program calls heap manager to **allocate** a block of any desired size to store some dynamic data.
2. Heap manager returns a pointer to a block. The program uses that block for its purpose. The block's memory is reserved exclusively for that use.
3. Program calls heap manager to free (**deallocate**) the block when the program is done with it.
4. Once the program frees the block, the heap manager may reuse the memory in the block for another purpose.
5. User program is responsible for initializing the block, and deciding what to store in it. Initial contents could be old. Program must not try to use the block after freeing it.

Heap manager

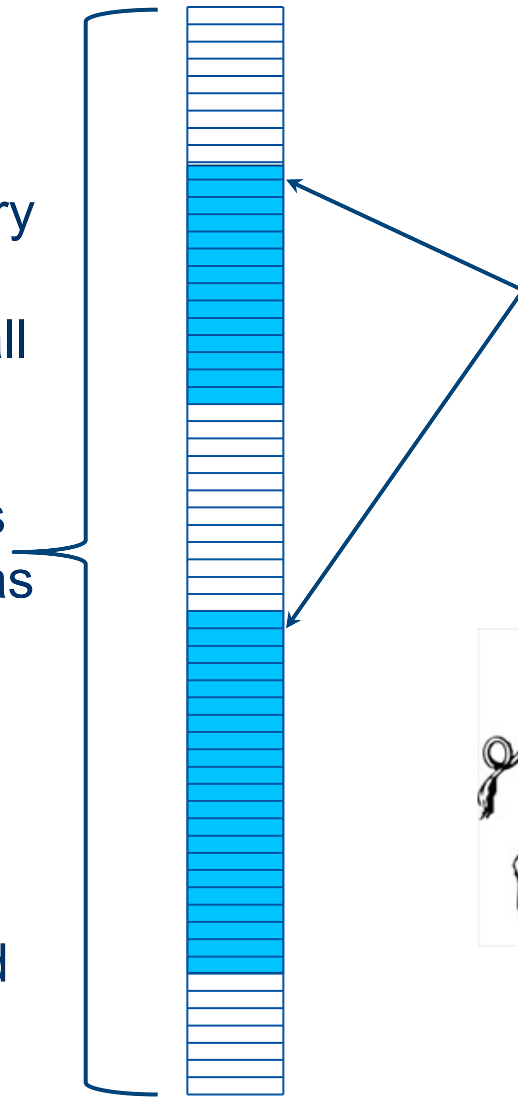


Heap: dynamic memory

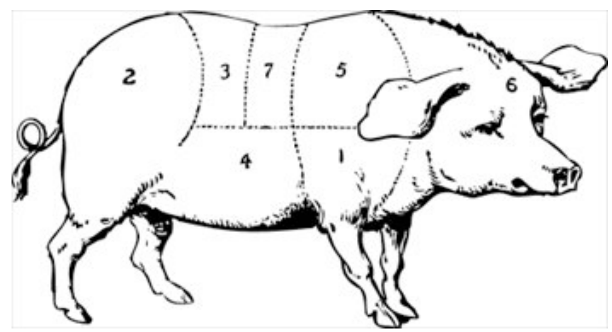
Heap segment. A contiguous chunk of memory obtained from OS kernel. E.g., with Unix *sbrk*() syscall

A **runtime library** obtains the block and manages it as a “heap” for use by the programming language environment, to store dynamic objects.

E.g., with Unix *malloc* and *free* library calls.



Allocated heap blocks for structs or objects. Align!

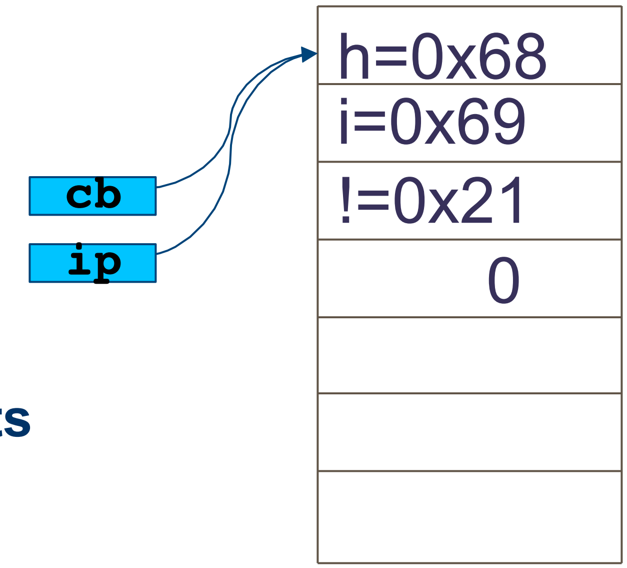


Using the heap (1)

```
#include <stdlib.h>
#include <stdio.h>

int
main()
{
  char* cb = (char*) malloc(14);
  cb[0]='h';
  cb[1]='i';
  cb[2]='!';
  cb[3]='\0';
  printf("%s\n", cb);
  free(cb);
}
```

```
chase$ cc -o heap heap.c
chase$ ./heap
hi!
chase$
```

Using the heap (2)




```c
#include <stdlib.h>
#include <stdio.h>

int
main()
{
  char* cb = (char*) malloc(14);
  cb[0]='h';
  cb[1]='i';
  cb[2]='!';
  cb[3]='\0';
  printf("%s\n", cb);
  int *ip = (int*)cb;
  printf("0x%x\n", *ip);
  free(cb);
}
```

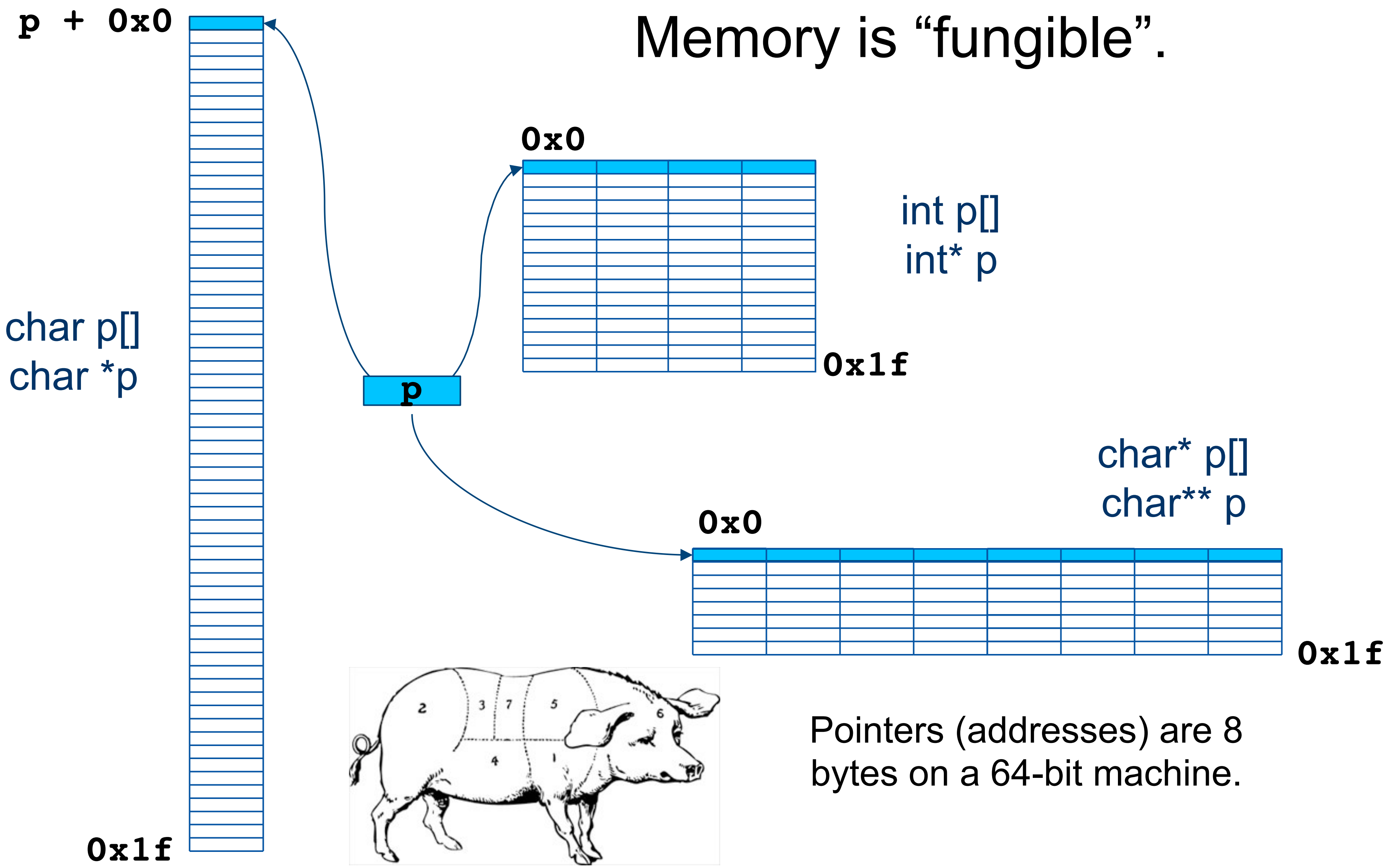
Type casts

```
chase$ cc -o heap heap.c
chase$ ./heap
hi!
0x216968
chase$
```

Try:
http://wikipedia.org/wiki/ASCII
http://wikipedia.org/wiki/Endianness

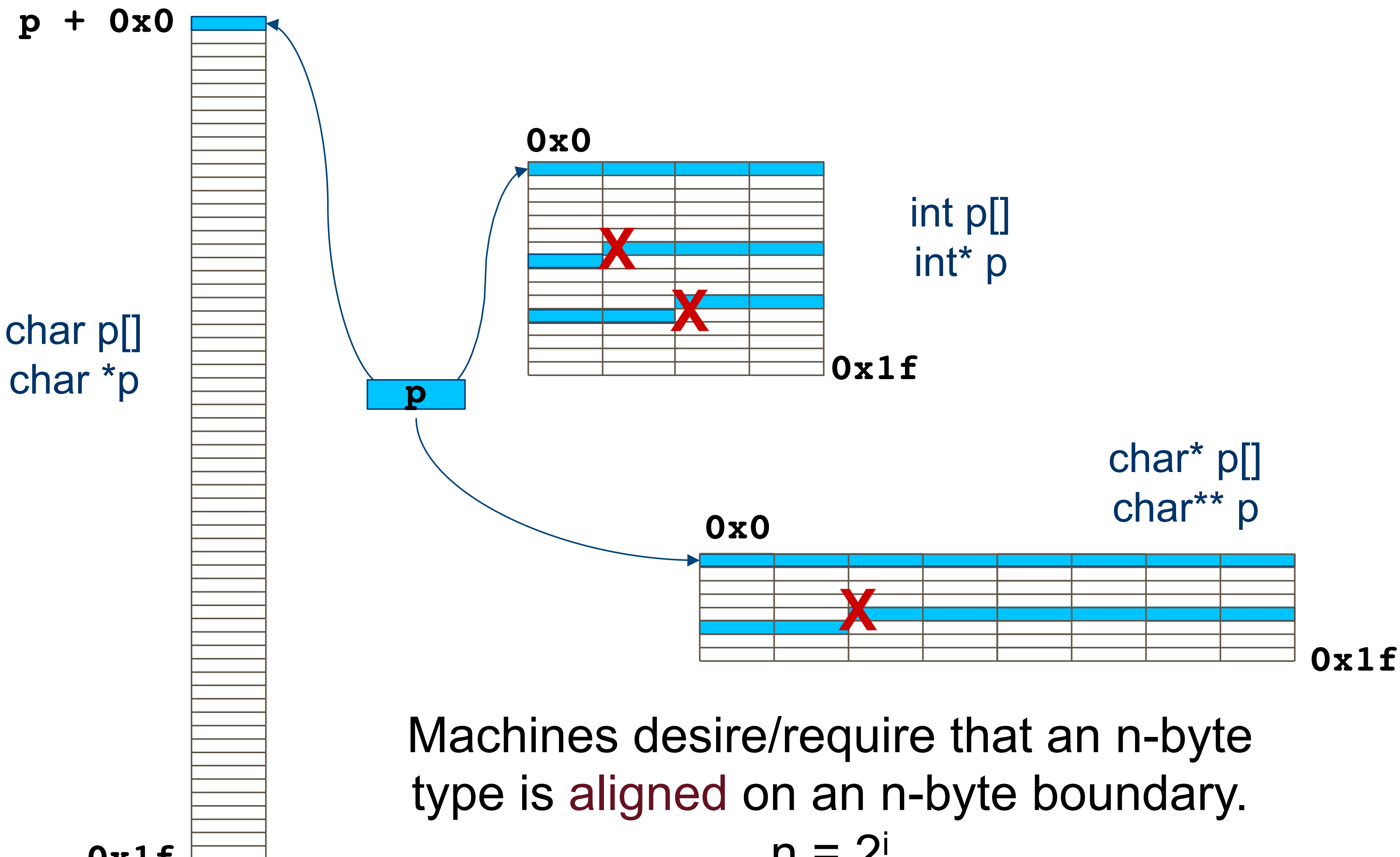
64 bytes: 3 ways

Memory is “fungible”.



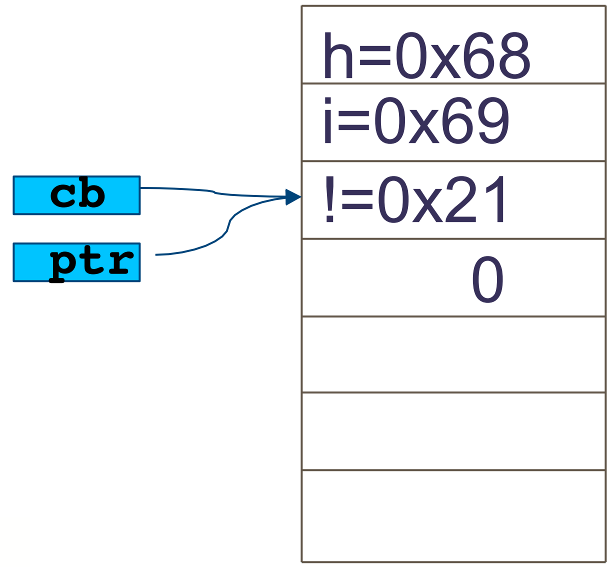
Pointers (addresses) are 8 bytes on a 64-bit machine.

Alignment



Machines desire/require that an n-byte type is aligned on an n-byte boundary.
$n = 2^i$

Pointer arithmetic

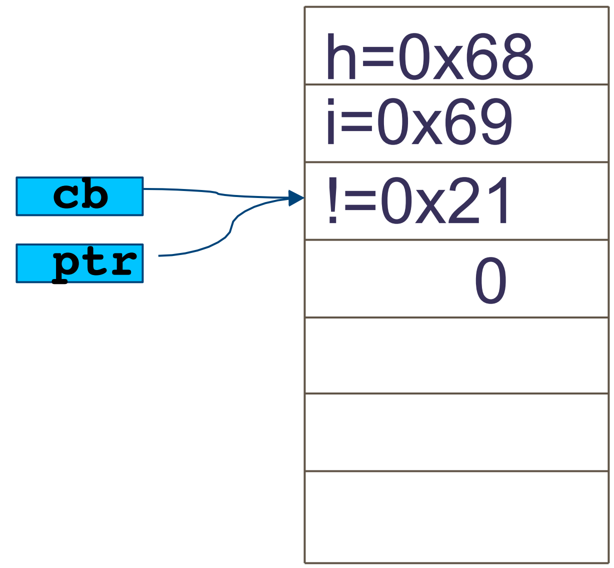



```
char* cb = (char*) malloc(14);
strcpy(cb, "hi!");

void* ptr = (void*)cb;
ptr = ptr + 2;
cb = (char*)ptr;
printf("%s\n", cb);

free(cb);
```


```
chase$ cc -o heap3 heap3.c
chase$ ./heap3
???
chase$
```

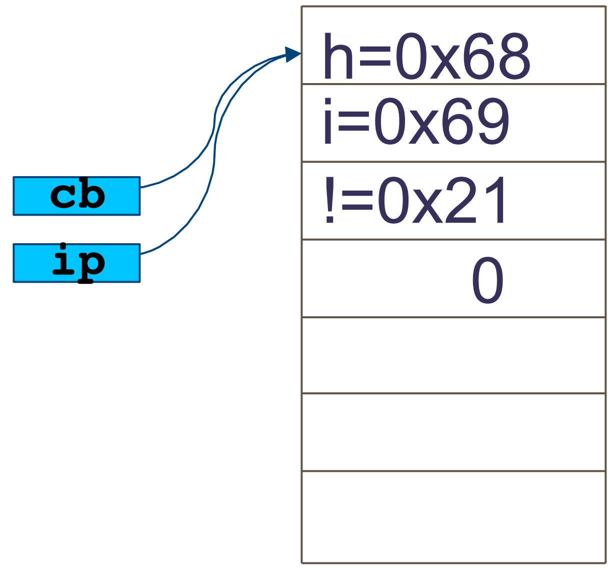
Pointer arithmetic

```
char* cb = (char*) malloc(14);
strcpy(cb, "hi!");

void* ptr = (void*)cb;
ptr = ptr + 2;
cb = (char*)ptr;
printf("%s\n", cb);

free(cb);
```




```
chase$ cc -o heap3 heap3.c
chase$ ./heap3
!
heap3(5478) malloc: *** error for
object 0x7f92a9c000e2: pointer being
freed was not allocated
Abort trap: 6
chase$
```

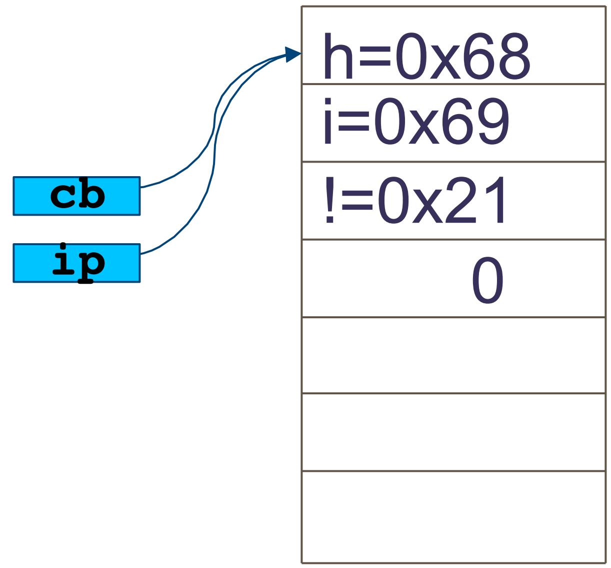
Using the heap (3)




```
char* cb = (char*)malloc(14);
strcpy(cb, "hi!");
free(cb);
/*
 * Dangling reference!
 */
printf("%s\n", cb);
int *ip = (int*)cb;
printf("0x%x\n", *ip);
/*
 * Uninitialized heap block!
 */
char* cb2 = (char*)malloc(14);
printf("%s\n", cb2);
```

```
chase$ cc -o heap2 heap2.c
chase$ ./heap2
???
chase$
```

Using the heap (4)




```
char* cb = (char*)malloc(14);
strcpy(cb, "hi!");
free(cb);
/*
 * Dangling reference!
 */
printf("%s\n", cb);
int *ip = (int*)cb;
printf("0x%x\n", *ip);
/*
 * Uninitialized heap block!
 */
char* cb2 = (char*)malloc(14);
printf("%s\n", cb2);
```

```
chase$ cc -o heap2 heap2.c
chase$ ./heap2
hi!
0x216968
hi!
chase$
```

WARNING

- These behaviors are undefined.
- Any program whose behavior relies on the meaning of a dangling reference is incorrect.
- For example, a change in the allocation policy of the heap manager could result in different behavior.
- Can a program stay safe from dangling references by just never calling free?

Memory leaks in the heap

What happens if some memory is heap allocated, but never deallocated? A program which forgets to deallocate a block is said to have a "memory leak" which may or may not be a serious problem. The result will be that the heap gradually fill up as there continue to be allocation requests, but no deallocation requests to return blocks for re-use.

For a program which runs, computes something, and exits immediately, memory leaks are not usually a concern. Such a "one shot" program could omit all of its deallocation requests and still mostly work. Memory leaks are more of a problem for a program which runs for an indeterminate amount of time. In that case, the memory leaks can gradually fill the heap until allocation requests cannot be satisfied, and the program stops working or crashes. Many commercial programs have memory leaks, so that when run for long enough, or with large data-sets, they fill their heaps and crash. Often the error detection and avoidance code for the heap-full error condition is not well tested, precisely because the case is rarely encountered with short runs of the program — that's why filling the heap often results in a real crash instead of a polite error message.

From **Essential C: Pointers and Memory**: http://cslibrary.stanford.edu/101]

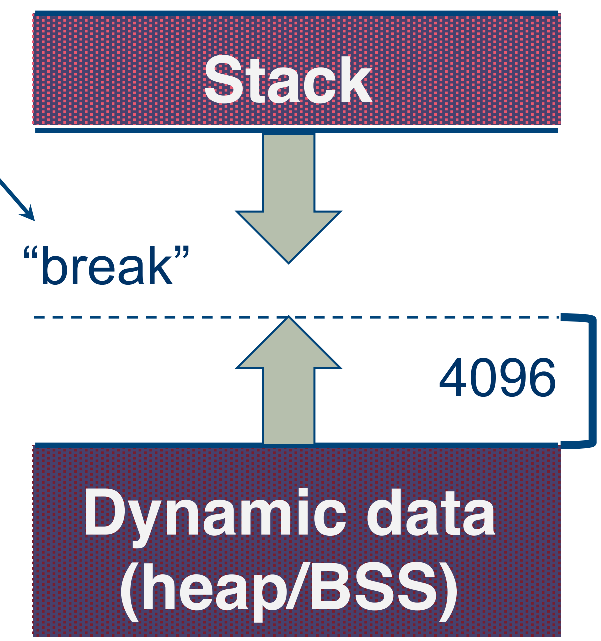
Heap: parking lot analogy

- Vehicles take up varying amounts of space, but they don’t grow or shrink, and they don’t have “holes”.
- They come and go at arbitrary times.
- Space allocation is **exclusive**.
- Vehicles are parked in contiguous space: must find a block that is available (free) and big enough.

Heap manager: "the easy way"

```
static void* freespace = NULL;

void* dmalloc(size_t n) {

  if (freespace == NULL)
    freespace = (void*)sbrk(4096);

  void* bp = freespace;
  freespace = freespace + ALIGN(n);

  return bp;

}
```

"Set break"
Choose your size.



Why is this a bad way to implement a heap manager?

Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes.

- **First-fit**: Allocate the *first* hole that is big enough.
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit [generally] better than worst-fit in terms of speed and storage utilization.

50-percent rule: Given N allocated blocks another 1/2N will be lost due to fragmentation $\Rightarrow$ 1/3 of memory lost.

Which Pointer Errors Do Students Make?

Bruce Adcock[1], Paolo Bucci[1], Wayne D. Heym[1], Joseph E. Hollingsworth[2], Timothy Long[1], and Bruce W. Weide[1]

[1] Department of Computer Science and Engineering
The Ohio State University

[2] Department of Computer Science
Indiana University Southeast

5. DATA SUMMARY

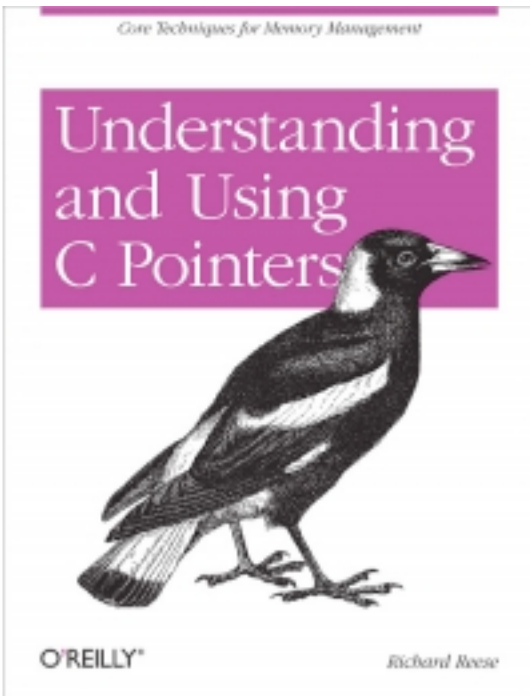
We logged all student pointer errors at OSU over a one-year period, then filtered the data as explained in Section 4, and also limited the focus to students doing assignments for CS2. Three programming assignments were involved: a closed lab (done in pairs) to implement a stack class, given a queue class as a model; and two open labs (done individually) to implement a singly linked list class and a doubly linked list class. We logged 2765 pointer errors made by 139 distinct students.

Figure 4 lists each possible error message seen by these students, along with two pieces of information for each: the percentage of students making at least one error overall (i.e., 139) who made that particular error at least once, and the percentage of all errors (i.e., 2765) accounted for by that particular error.

Error	Students Making Error	Percent-age of All Errors
Creating memory leak by pointer leaving scope	**74%**	21%
Creating memory leak by using = (i.e., assignment)	**61%**	18%
Creating memory leak by using = NULL (i.e., assignment)	4%	1%
Creating memory leak by using New	1%	0%
Deleting dead pointer	19%	2%
Dereferencing dead pointer by using * or ->	**70%**	**33%**
Dereferencing null pointer by using * or ->	**57%**	**16%**
Using dead pointer with != (i.e., inequality checking)	**30%**	5%
Using dead pointer with != NULL (i.e., inequality checking)	10%	1%
Using dead pointer with == (i.e., equality checking)	13%	2%
Using dead pointer with == NULL (i.e., equality checking)	9%	1%

Figure 4: Results of Off-line Data Analysis

There is a book on this topic



“The easy way” isn’t good enough

- “The easy way” approach can’t free the blocks!
  - It doesn’t track the borders between blocks, or their sizes.
  - It allocates only from the “top” of the heap, never from the middle, so it can’t reuse freed blocks anyway.
  - It’s a **stack**! It is **fast and easy** for local variables, but it’s not good enough for a heap:

It can only free space at the front!

Admittedly this ferry picture illustrates FIFO (first-in-first-out: queue) rather than LIFO (last-in-first-out: stack), but you get the idea. It’s restricted, and I want to come and go as I please, i.e., allocate space when I want and free it when I want.

C structs, global, stack, heap

```
#include <stdio.h>
#include <stdlib.h>

struct stuff {
  int i;
  long j;
  char c[2];
};

struct stuff gstuff;
```

Data structure type definition

A global structure

```
int main() {
  struct stuff sstuff;
  struct stuff *hstuffp =
      (struct stuff *) malloc(sizeof(struct stuff));

  gstuff.i = 13;
  gstuff.j = 14;
  gstuff.c[0] = 'z';
  gstuff.c[1] = '\0';

  printf("%s\n", gstuff.c);

  sstuff.i = 13;
  sstuff.j = 14;…

  hstuffp->i = 13;
  hstuffp->j = 14;…
}
```

Local variables

Heap allocation

Accessing a global structure

Local data of a procedure (on the stack)

Accessing a heap-allocated struct through a pointer.

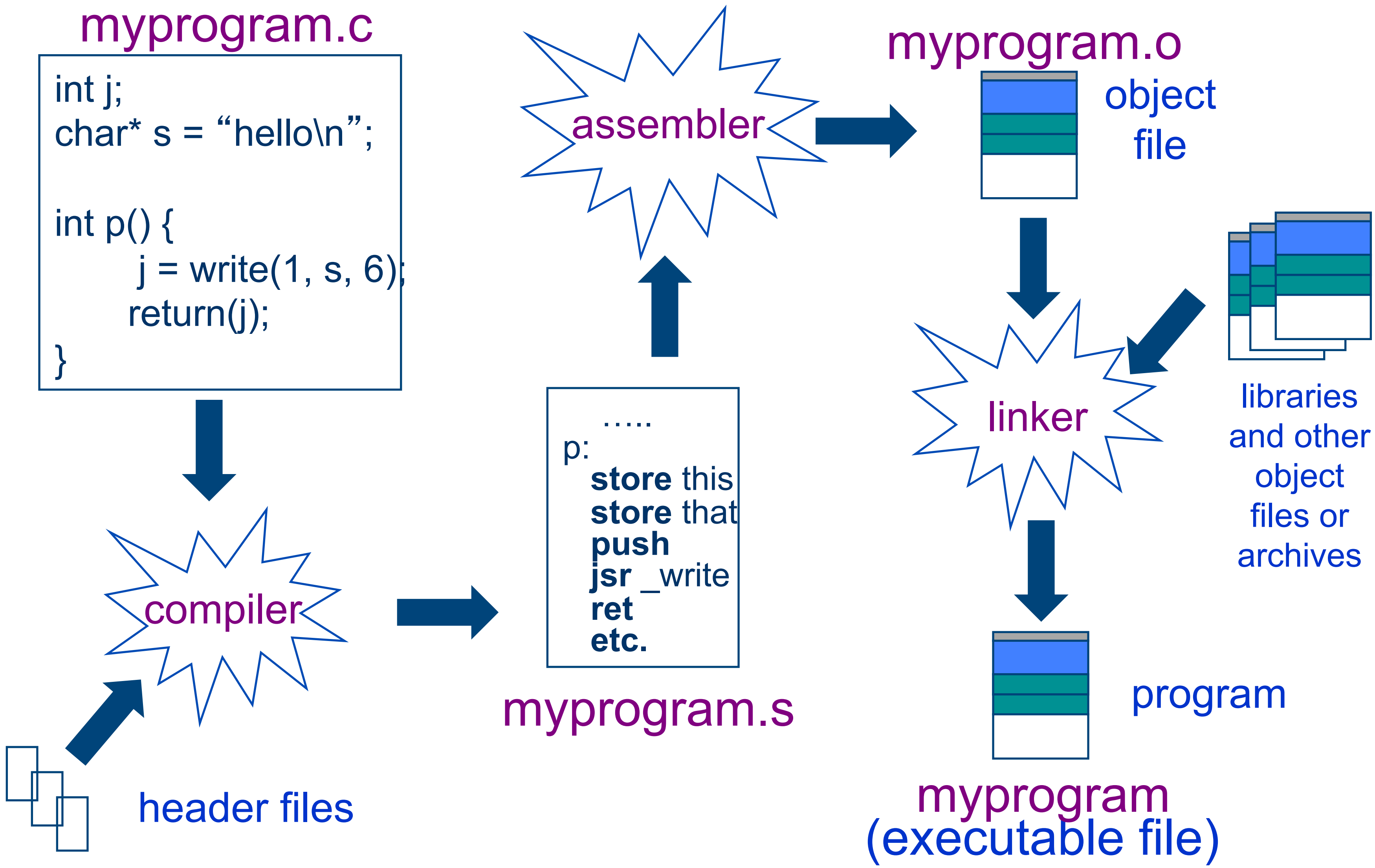
Spelling it out

- That C program is in a C source file called structs.c in the **C samples** directory on the course web.
- On any properly installed Unix/Linux or MacOSX system, you can compile it into an executable program by typing:
  - cc –o structs structs.c
  - That says: “run the C compiler (cc or gcc) on the input structs.c, and put the output in a new file called structs”.
  - This is a **shell command line**. Much more on that later (Lab #2). You will need to learn some shell command language.
- This command secretly runs other system utilities to finish the job.
  - The **compiler** outputs a file with compiled assembly language code.
  - **Assembler** reads that and generates a partial executable (**object file**).
  - A **linker** combines the object file with code from system libraries, and outputs the executable file structs.

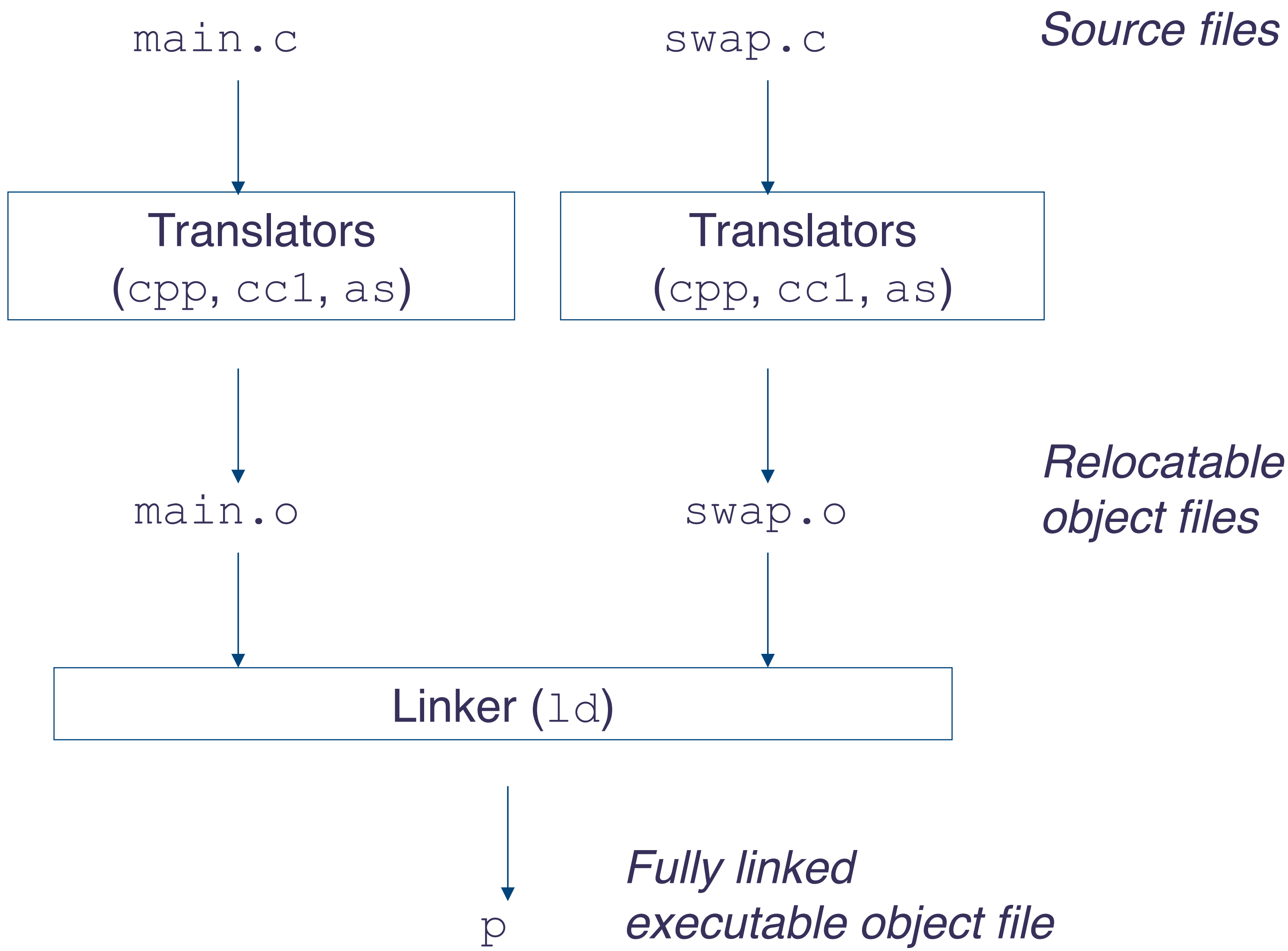
Building and running a program

```
chase:lab1> make
gcc -I. -Wall -lm -DNDEBUG -c dmm.c
…
gcc -I. -Wall -lm -DNDEBUG -o test_basic test_basic.c dmm.o
gcc -I. -Wall -lm -DNDEBUG -o test_coalesce test_coalesce.c dmm.o
gcc -I. -Wall -lm -DNDEBUG -o test_stress1 test_stress1.c dmm.o
gcc -I. -Wall -lm -DNDEBUG -o test_stress2 test_stress2.c dmm.o
chase:lab1> chase:lab1> ./test_basic
calling malloc(10)
call to dmalloc() failed
chase:lab1>
```

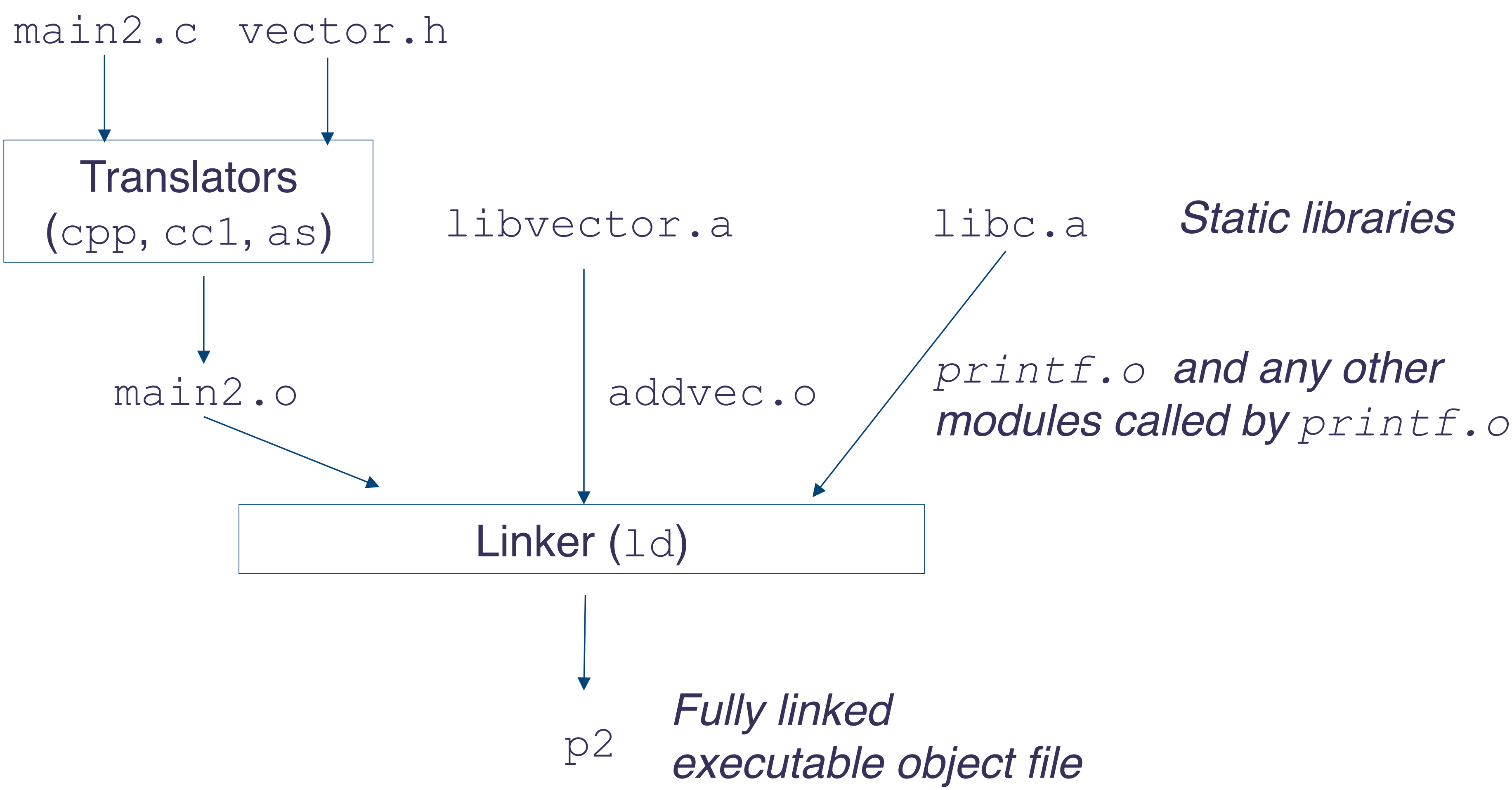
The Birth of a Program (C/Ux)



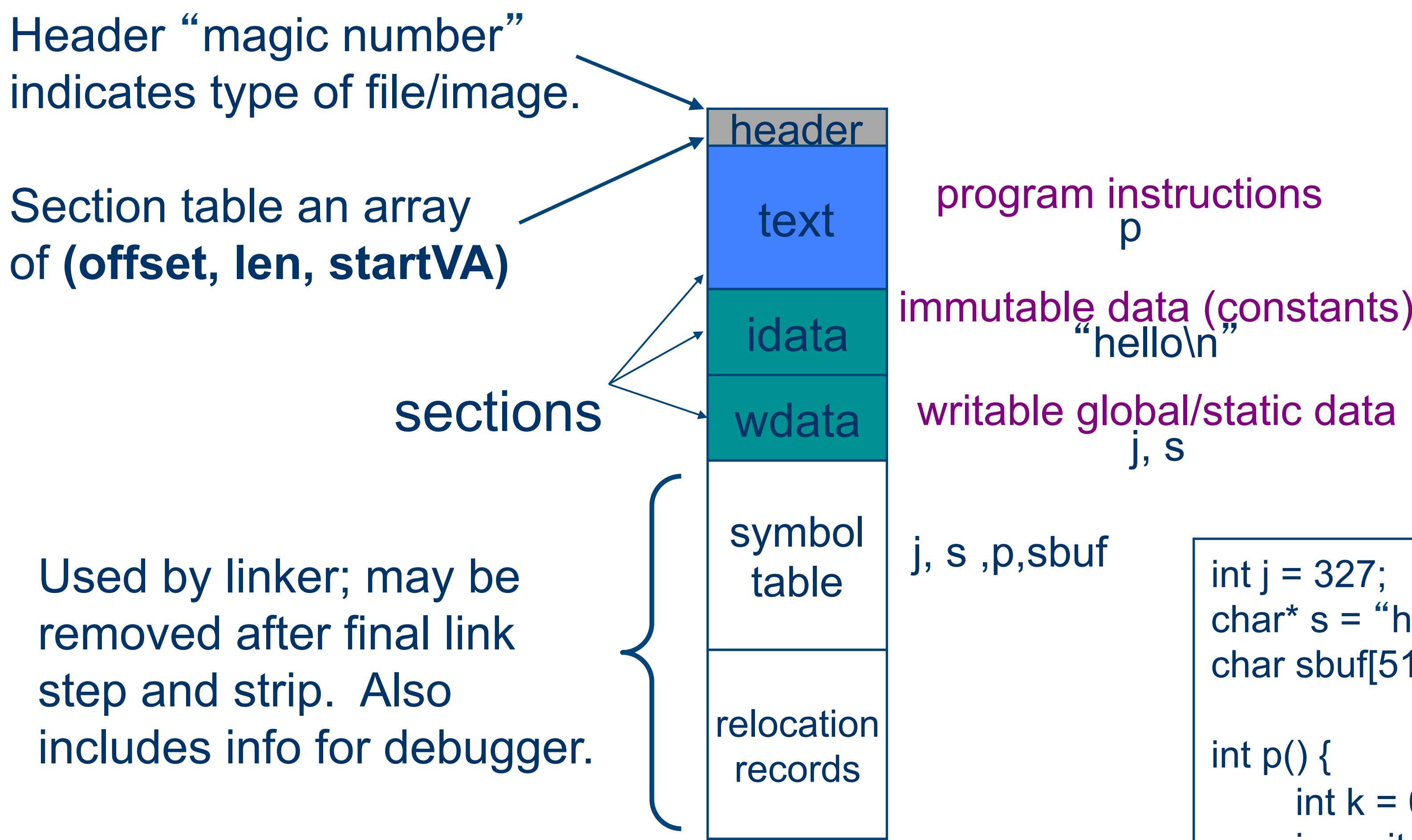
Linking (from cs:app)



Static linking with libraries



What's in an Object File or Executable?




```
int j = 327;
char* s = "hello\n";
char sbuf[512];

int p() {
        int k = 0;
        j = write(1, s, 6);
       return(j);
}
```

```
cc –o structs structs.c
otool –vt structs
```

The otool –vt command shows contents of the text section of an executable file.
Here are the instructions for the **structs** program.
When the program runs, the OS loads them into a contiguous block of virtual memory (the text segment) at the listed virtual addresses.

```
0000000100000ea0 pushq    %rbp
0000000100000ea1 movq     %rsp, %rbp
0000000100000ea4 subq     $48, %rsp
0000000100000ea8 movabsq  $24, %rdi
0000000100000eb2 callq    0x100000f42
0000000100000eb7 leaq     182(%rip), %rdi
0000000100000ebe leaq     347(%rip), %rcx
0000000100000ec5 movq     %rcx, %rdx
0000000100000ec8 addq     $16, %rdx
0000000100000ecf movq     %rax, -32(%rbp)
0000000100000ed3 movl     $13, (%rcx)
0000000100000ed9 movq     $14, 8(%rcx)
0000000100000ee1 movb     $122, 16(%rcx)
0000000100000ee5 movb     $0, 17(%rcx)
0000000100000ee9 movq     %rdx, %rsi
0000000100000eec movb     $0, %al
0000000100000eee callq    0x100000f48
0000000100000ef3 movl     $0, %r8d
0000000100000ef9 movl     $13, -24(%rbp)
0000000100000f00 movq     $14, -16(%rbp)
0000000100000f08 movb     $122, -8(%rbp)
0000000100000f0c movb     $0, -7(%rbp)
0000000100000f10 movq     -32(%rbp), %rcx
0000000100000f14 movl     $13, (%rcx)
0000000100000f1a movq     -32(%rbp), %rcx
0000000100000f1e movq     $14, 8(%rcx)
0000000100000f26 movq     -32(%rbp), %rcx
0000000100000f2a movb     $122, 16(%rcx)
0000000100000f2e movq     -32(%rbp), %rcx
0000000100000f32 movb     $0, 17(%rcx)
0000000100000f36 movl     %eax, -36(%rbp)
0000000100000f39 movl     %r8d, %eax
0000000100000f3c addq     $48, %rsp
0000000100000f40 popq     %rbp
0000000100000f41 ret
```

Code: instructions

- The **text** section contains executable code for the program: a sequence of machine instructions**.**
- Most instructions just move data in and out of named **registers**, or do arithmetic in registers.

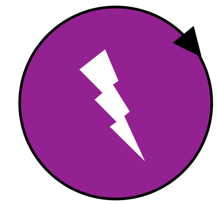
```
movq    %rsp, %rbp
subq    $48, %rsp
```

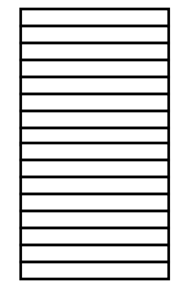
  - These x86 instructions move the contents of the %rsp register (stack pointer) into the %rbp register (base pointer), then subtract 48 from %rsp.
- Some instructions **load** from or **store** to memory at a virtual address in a register, plus some **offset** (displacement).

```
movq    -32(%rbp), %rcx
```

  - This x86 instruction loads a quadword (8-byte) value at the address in the %rbp register, minus 32, and puts that value in the %rcx register.

Registers





- The next few slides give some pictures of the register sets for various processors.
  - x86 (IA32 and x86-64): Intel and AMD chips, MIPS
  - The details aren't important, but there's always an SP (**stack pointer**) and PC (**program counter** or instruction pointer: the address of the current/next instruction to execute).
- The system's Application Binary Interface (ABI) defines conventions for use of the registers by executable code.
- Each processor core has at least one register set for use by a code stream running on that core.
  - Multi-threaded cores ("SMT") have multiple register sets and can run multiple streams of instructions simultaneously.

Simplified…

```
pushq    %rbp
movq     %rsp, %rbp
subq     $48, %rsp
```

Push a frame on the stack: this one is 48 bytes.
Who decided that?

int main() {
 struct stuff sstuff;

 struct stuff *hstuffp =
 (struct stuff *)
 malloc(24);

```
movabsq $24, %rdi
callq    0x100000f42
movq     %rax, -32(%rbp)
```

Call **malloc**(24): move return value into a local variable (at an offset from the stack base pointer).

 gstuff.i = 13;
 gstuff.j = 14;
 gstuff.c[0] = 'z';
 gstuff.c[1] = '\0';

```
leaq     347(%rip), %rcx
```

...address global data relative to (%rcx)

Address global data relative to the code address (%rip=PC).
position-independent

...address stack data relative to (%rbp)

 sstuff.i = 13;
 sstuff.j = 14;…

; move pointer to heap block into %rcx

```
movq     -32(%rbp), %rcx
```

...address heap block relative to (%rcx)

 hstuffp->i = 13;
 hstuffp->j = 14;…
}

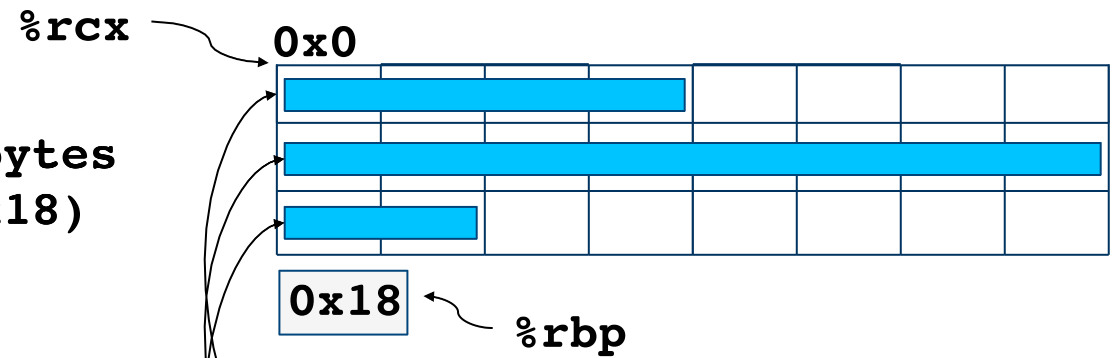
```
addq     $48, %rsp
popq     %rbp
ret
```

Pop procedure frame from the stack before returning from main().

Addressing stuff

```
struct stuff {
  int i;
  long j;
  char c[2];
};
```

24 bytes
(=0x18)




```
sstuff.i = 13;
sstuff.j = 14;
sstuff.c[0] = 'z';
sstuff.c[1] = '\0';
```

```
movl    $13, -24(%rbp)
movq    $14, -16(%rbp)
movb    $122, -8(%rbp)
movb    $0, -7(%rbp)
```

```
hstuffp->i = 13;
hstuffp->j = 14;
hstuffp->c[0] = 'z';
hstuffp->c[1] = '\0';
```

```
movl    $13, (%rcx)
movq    $14, 8(%rcx)
movb    $122, 16(%rcx)
movb    $0, 17(%rcx)
```

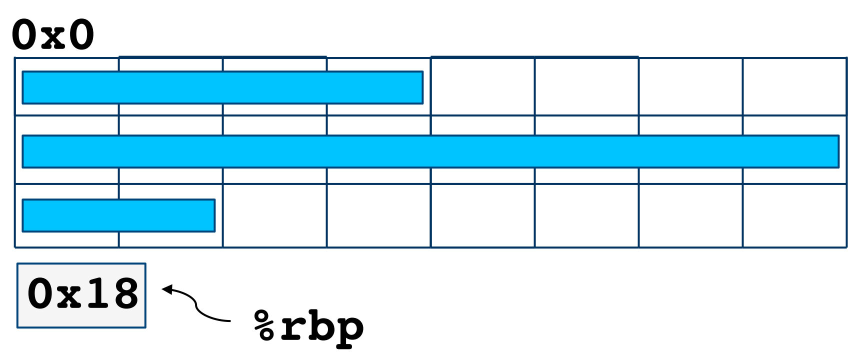
Basic C data types (64-bit)

64-bit data models

Data model	short (integer)	int	long (integer)	long long	pointers/ size_t	Sample operating systems
LLP64/ IL32P64	16	32	32	64	64	Microsoft Windows (x86-64 and IA-64)
LP64/ I32LP64	16	32	64	64	64	Most Unix and Unix-like systems, e.g. Solaris, Linux, BSD, and OS X; z/OS
ILP64	16	64	64	64	64	HAL Computer Systems port of Solaris to SPARC64
SILP64	64	64	64	64	64	"Classic" UNICOS[31] (as opposed to UNICOS/mp, etc.)

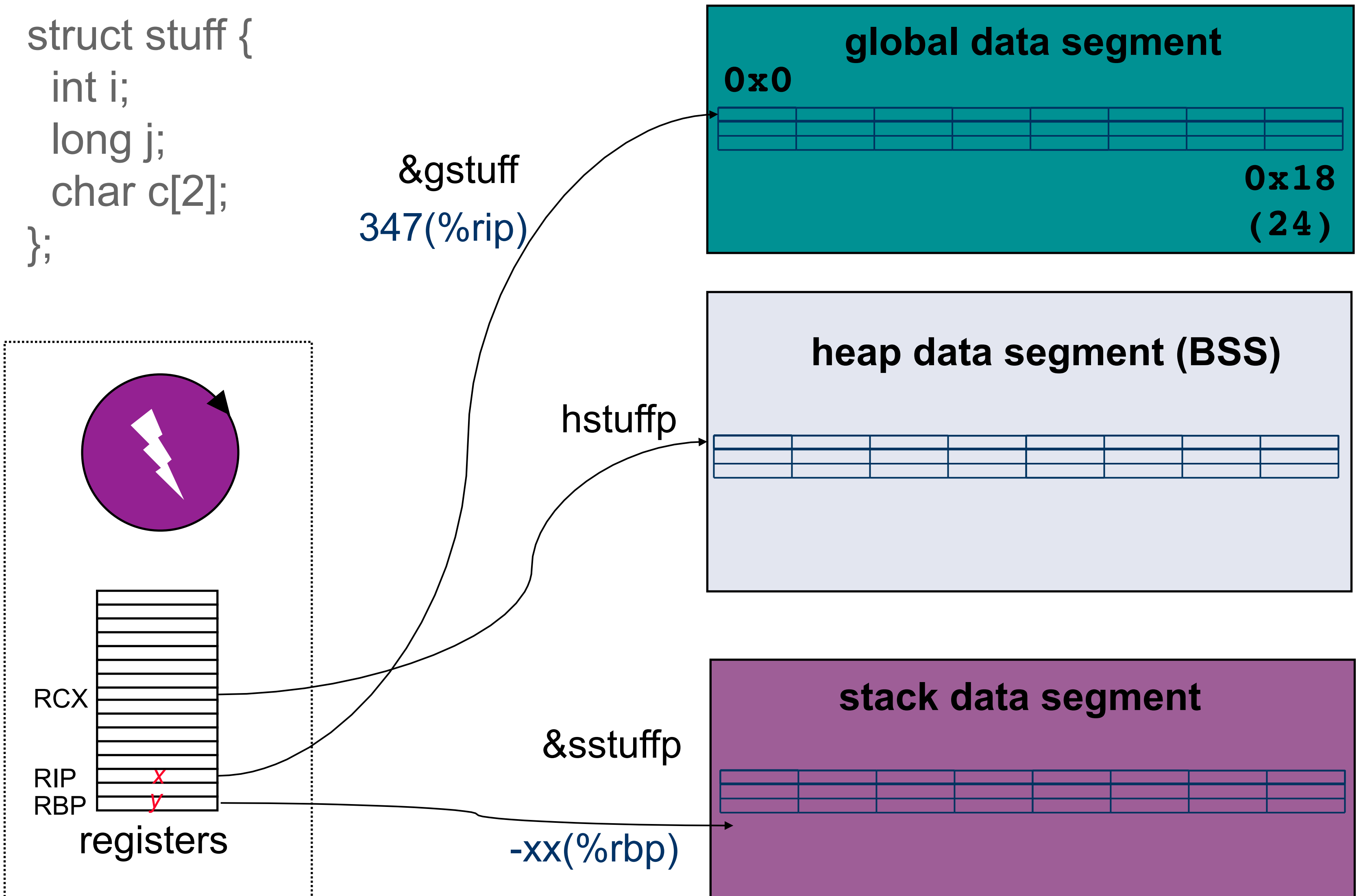
```
struct stuff {
  int i;
  long j;
  char c[2];
};
```

24 bytes (=0x18)



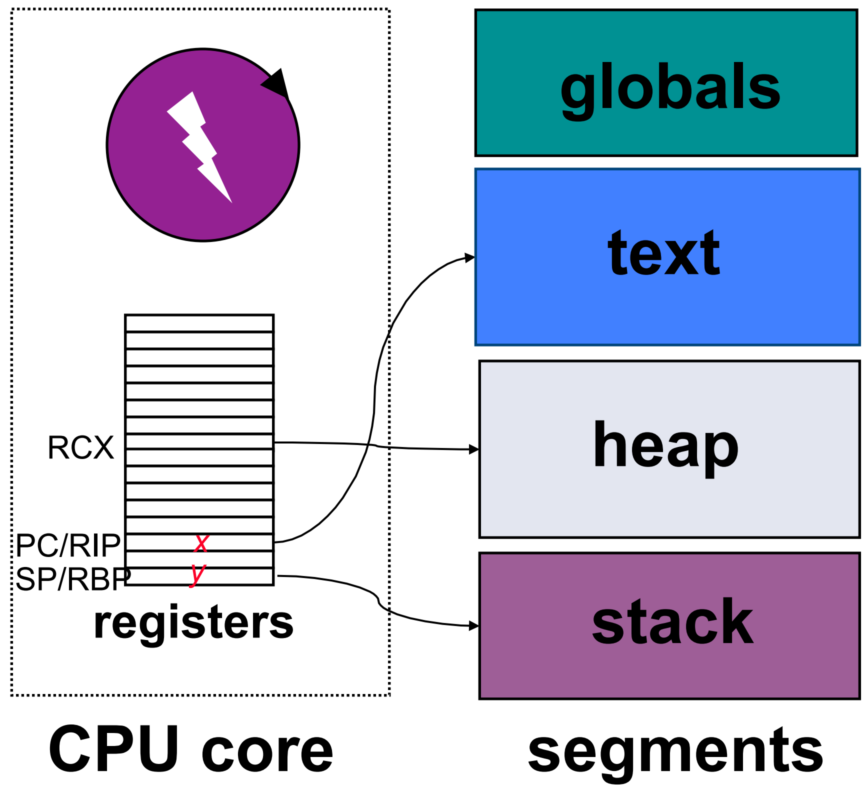
Addressing stuff

```
struct stuff {
  int i;
  long j;
  char c[2];
};
```



Memory model: the view from C

- **Globals:**
  - fixed size segment
  - Writable by user program
  - May have initial values
- **Text (instructions)**
  - fixed size segment
  - executable
  - not writable
- **Heap and stack**
  - variable size segments
  - writable
  - zero-filled on demand



Assembler directives: quick peek

From x86 Assembly Language Reference Manual

The .align directive causes the next data generated to be aligned modulo integer bytes.

The .ascii directive places the characters in *string* into the object module at the current location but does **not** terminate the string with a null byte (\0).

The .comm directive allocates storage in the data section. The storage is referenced by the identifier *name*. *Size* is measured in bytes and must be a positive integer.

The .globl directive declares each *symbol* in the list to be **global**. Each symbol is either defined externally or defined in the input file and accessible in other files.

The .long directive generates a long integer (32-bit, two's complement value) for each *expression* into the current section. Each *expression* must be a 32–bit value and must evaluate to an integer value.

.proto Type	Notes	C++ Type	Java Type	Python Type[2]
double		double	double	float
float		float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long[3]
uint32	Uses variable-length encoding.	uint32	int[1]	int/long[3]
uint64	Uses variable-length encoding.	uint64	long[1]	int/long[3]
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long[3]
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int[1]	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long[1]	int/long[3]
sfixed32	Always four bytes.	int32	int	int
sfixed64	Always eight bytes.	int64	long	int/long[3]
bool		bool	boolean	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode[4]
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str