ENGN2219/COMP6719
Computer Systems & Organization

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian National University

Word Count (wordcount.c)

Problem statement: Count the number of words input by the user via keyboard

A word contains a sequence of characters (a-z, A-Z) without a blank space or a new line

- ✓ ENGN2219
- ✓ 2219ENGN
- ✓ ENGN
- ✗ 2219

Input/Ouput

Input: *Read from keyboard character-wise until end of file is encountered. Blank space and new-line starts a new word.*

Output: *# of words*

Flow of our solution

1. Read characters until the end of file is encountered
   EOF is a symbolic constant in C (equivalent to pressing Ctrl-d)

2. For each character, check if it is in the range: *a to z or A to Z*

3. Keep count of the number of words seen so far

Reading characters

```
char c;
while ((c = getchar()) != EOF) {
```

Checking for characters in range

In C, the *char* type or the character variable holds an ASCII value between 0 and 127

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

A note on characters in C

```
if (((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z'))) {
```

Counting words

First, we need a variable to store the *state* we are in

Inside a word (**IN**)

Outside a word (**OUT**)

If the initial state is **OUT**

State transitions to **IN** if the user types a valid character

State transitions to **OUT** if the user inputs blank space or \n

OUT → IN

We have a new word (words++)

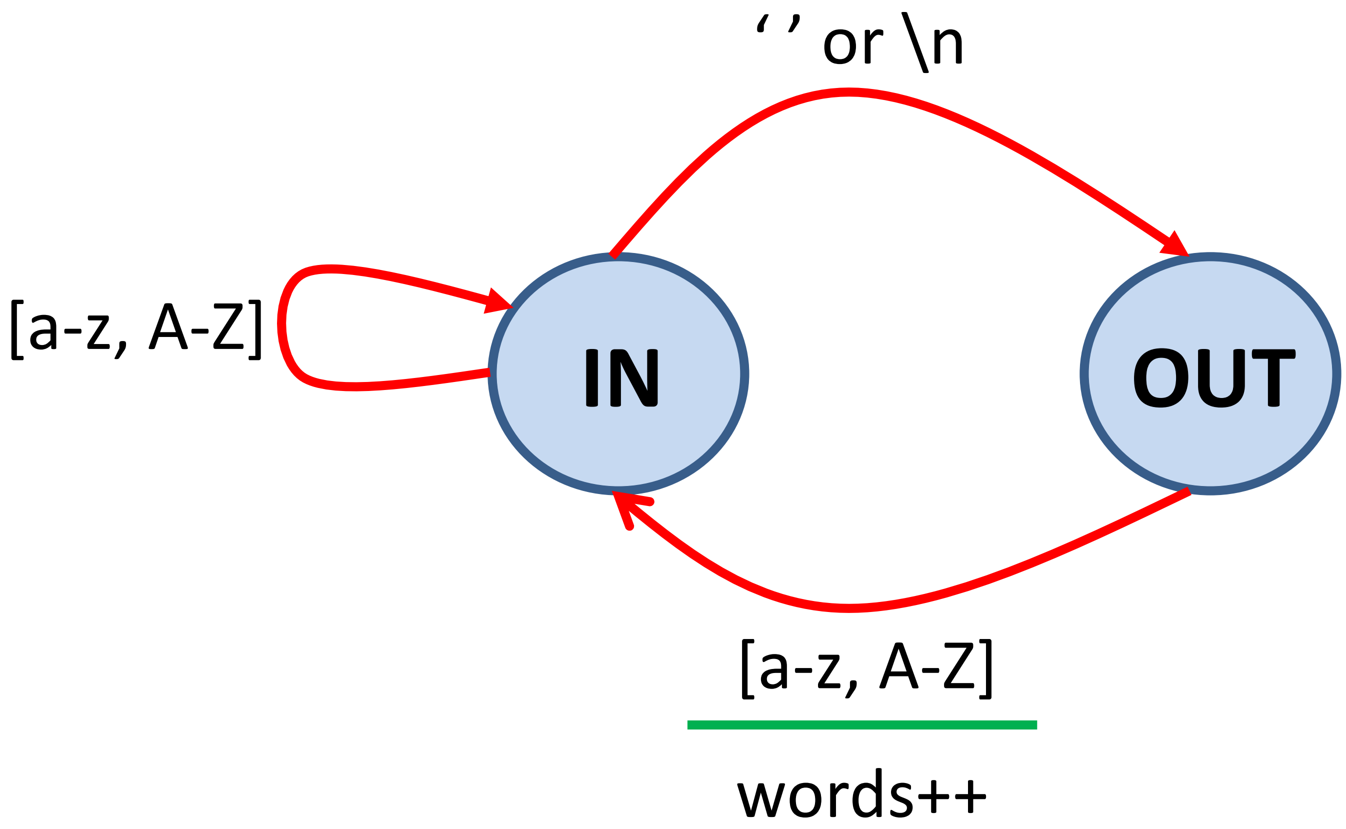
Question?

What happens if we are in state **IN** and the user inputs a valid character?

Printing the #words

```
printf("%d\n", words);
```

State machine diagram



wordcount.c

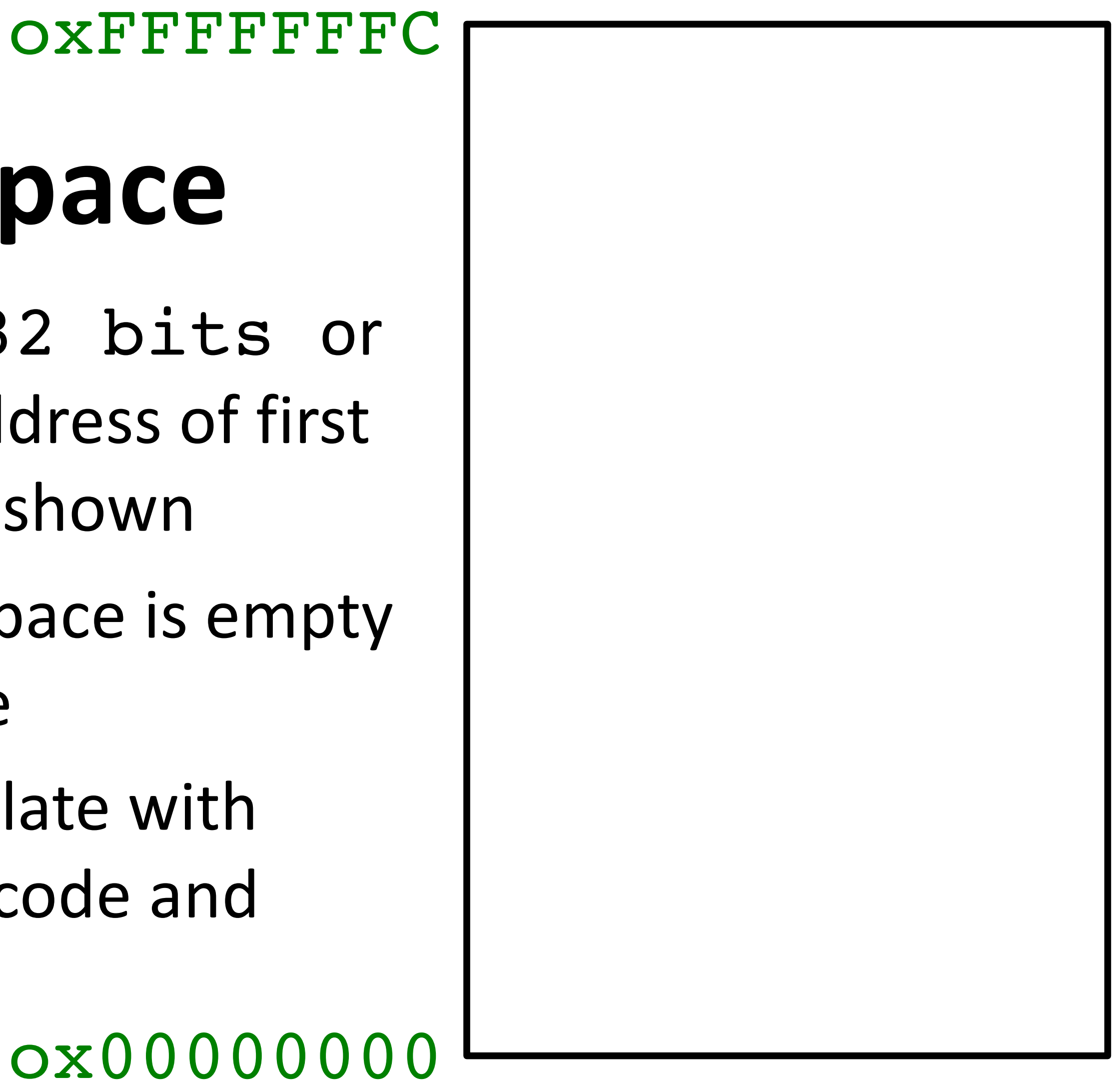
```c
#include <stdio.h>

#define IN  1  // inside a word
#define OUT 0  // outside a word


int main(void) {


    char c;
    int  inorout = OUT;
    int  words   = 0;


    while ((c = getchar()) != EOF) {
        if (((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z'))) {
            if (inorout == OUT) {
                inorout = IN;
                words++;
            }
        } else if ((c == ' ') || (c == '\n')) {
            inorout = OUT;
        } else {
          // ignore
        }
    }


    printf("%d\n", words);


    return 0;
}
```

Address Space

- **Address range**
  - A `32-bit` (ARM) CPU generates addresses in the range `0` to `0xFFFFFFFC` (`4294967292`)
  - With a 4×10^9 address range, the CPU can access `4 billion` individual bytes
- **Address space**
  - The address space of a `32-bit` CPU is 2^{32} bytes which equals `4 Gigabytes (GB)`

Address Space

- Each word is `32 bits` or `4 bytes`. Address of first & last word is shown
- The address space is empty as shown here
  - Let's populate with stack and code and data

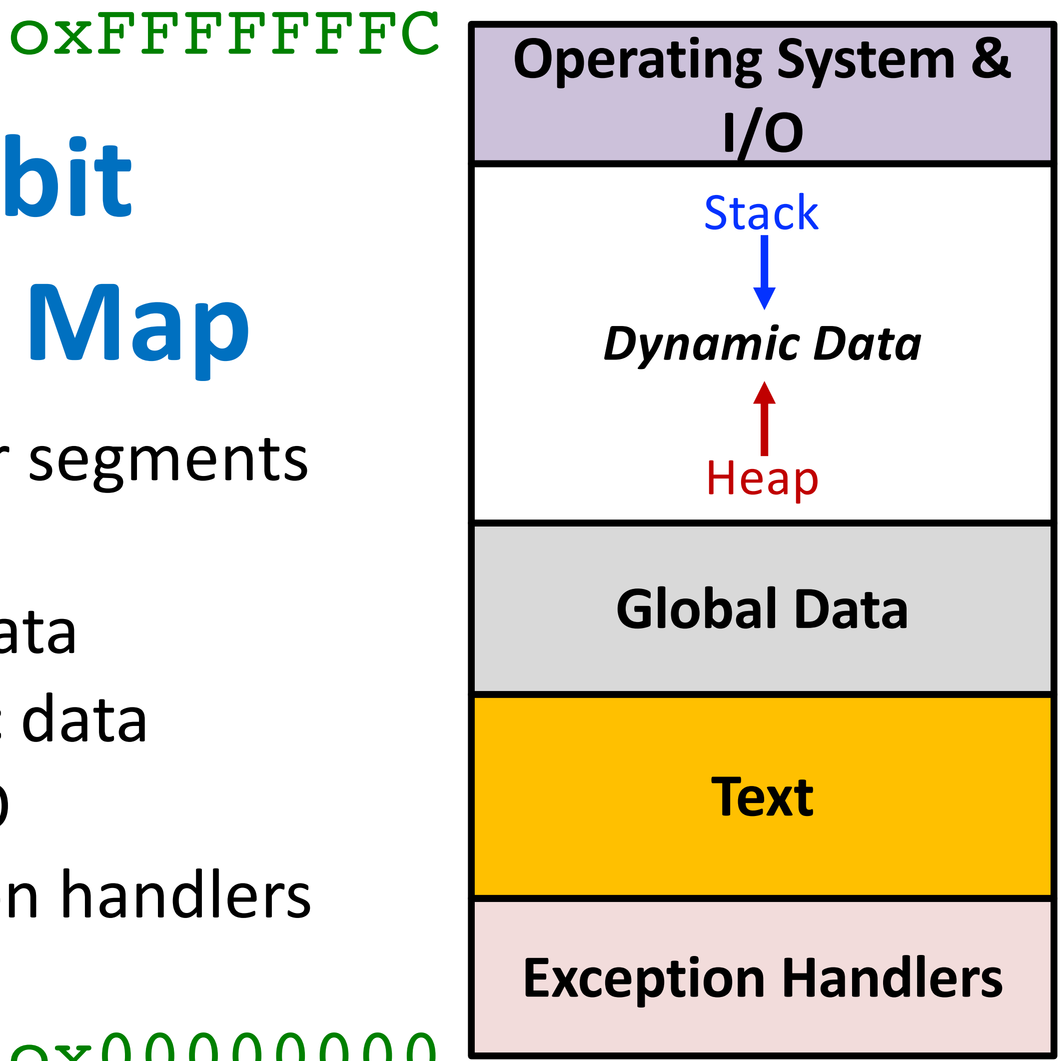


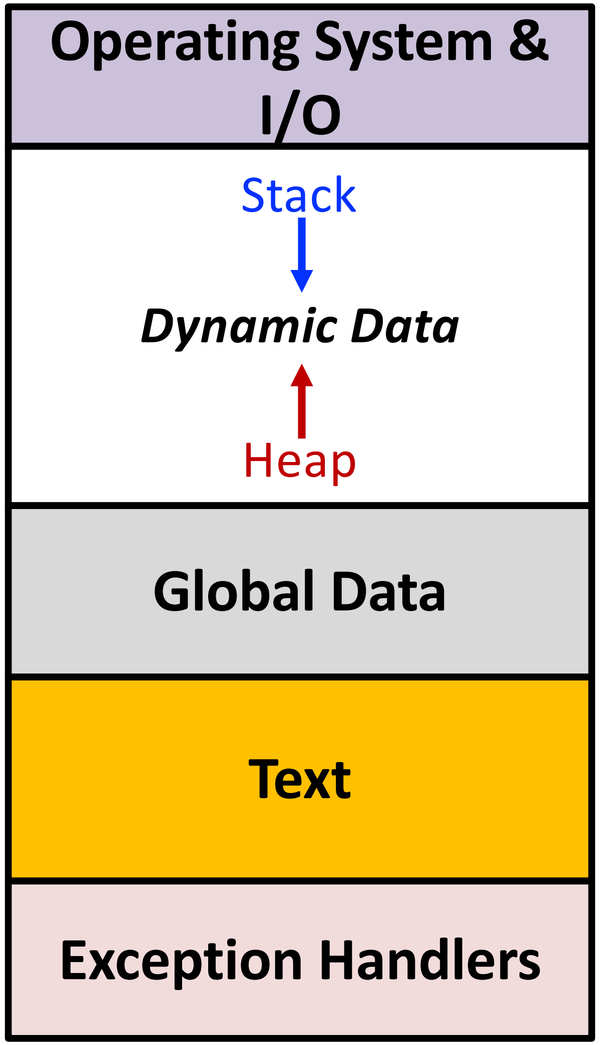
Questions

- Where is the code, data, and the stack in the address space?
- **Memory map**
  - Defines where code, data, and stack memory are in the program address space
  - Differs from architecture to architecture
  - The subsequent discussion pertains to ARM

ARM 32-bit Memory Map

- Five parts or segments
  - text
  - global data
  - dynamic data
  - OS & I/O
  - Exception handlers



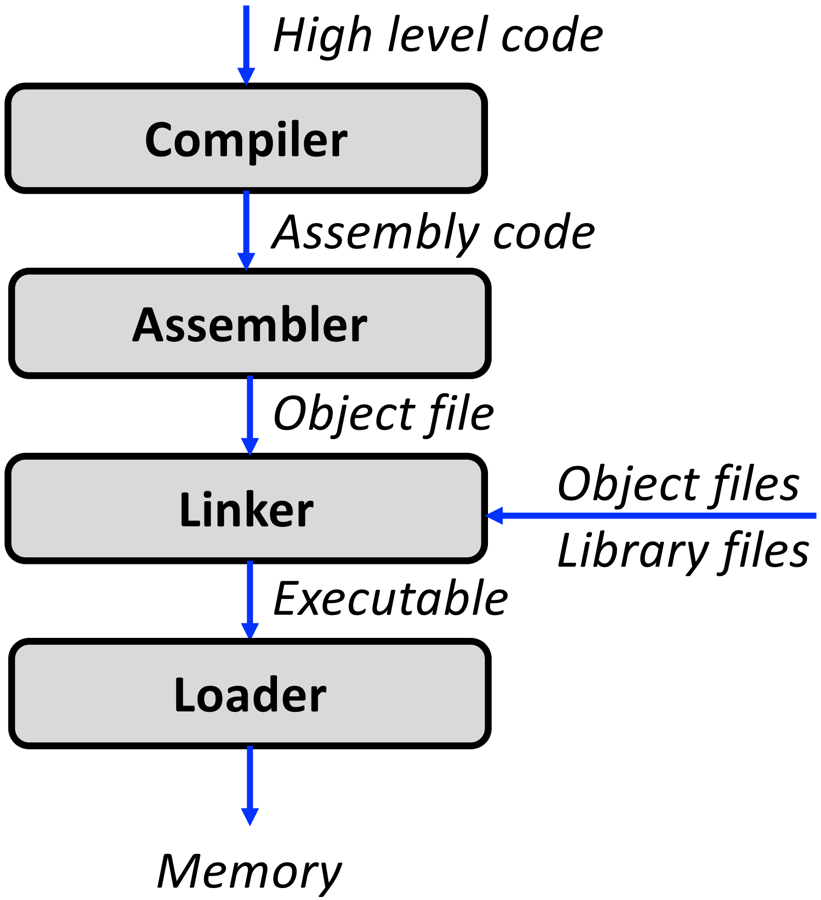


- *Data in this segment is dynamically allocated and deallocated during program execution*
- *Heap data is allocated by the program at runtime*
  - `malloc()` and `new`
- *Heap grows upward, stack grows downward*

- *Global variables visible to all functions (contrasted with local variables that are only visible to a function)*

- *Machine language program*
- *Also called read-only (**RO**) segment*
- *Literals (constants) such as "Hello"*

Translating/Starting Programs

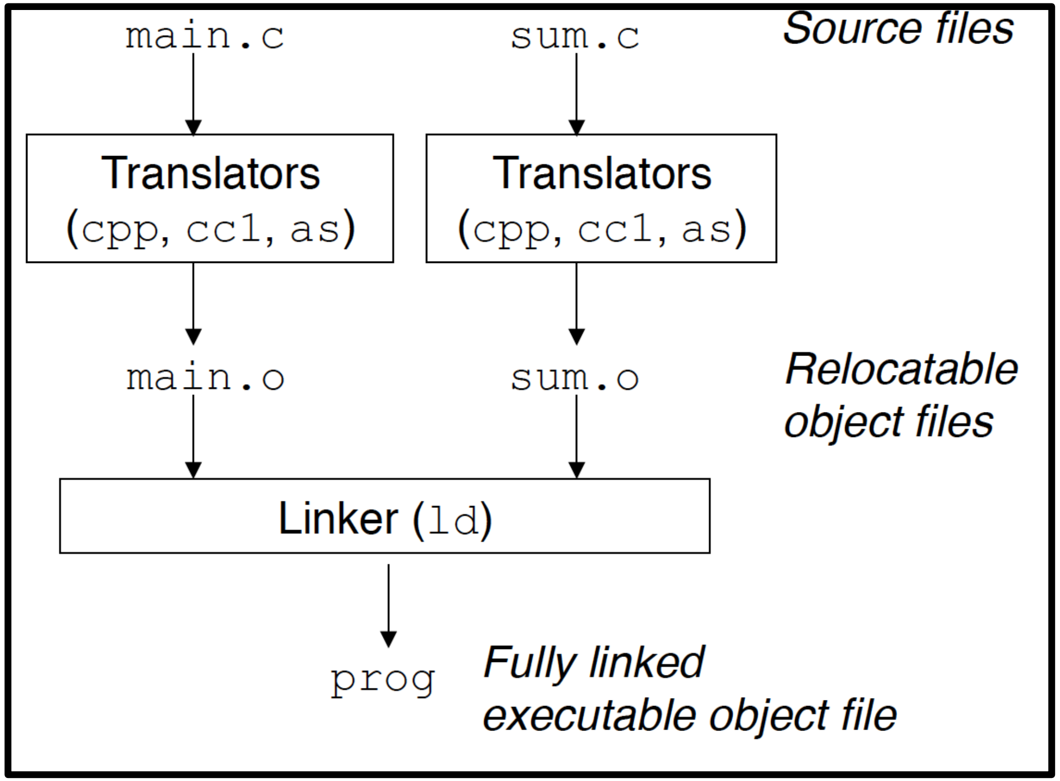


Translating/Starting Programs

- GNU compilation system & Linux specific
  - `gcc –o prog main.c sum.c`
- Invokes the GCC driver
  - GCC performs # steps

Translating/Starting Programs

- GNU compilation system & Linux specific
  - `gcc –o prog main.c sum.c`
- Invokes the GCC driver
  - GCC performs # steps



Example

sum.h:

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

sum.c:

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

main.c:

```
#include <stdio.h>
#include "sum.h"
int main() {
   int c = sum(a,b);
   printf("c = %i \n", c);
   return 0;
}
```

`cpp, cc1, and as`

- `cpp` is the C preprocessor: handles header file inclusion and macro expansion and generates an intermediate (`.i`) file
  - `#include <stdio.h>` copies the contents of `stdio.h`
  - `#define LEN 100` replaces `LEN` with 100 everywhere in the code
- `cc1` is the C compiler that generates an assembly (`.s`) file from the intermediate format (internal command)
- `as` is the assembler, which translates the assembly file into a binary relocatable (`.o`) object file

Relocatable Object File

- Contains binary code and data that can be combined with other relocatable object files at compile time to create an executable object file
- Symbols are not resolved
  - Variables with the same name declared in multiple sources
  - Symbol resolution: removes ambiguity by creating a single *linked executable file*
- Functions and variables are not bound to any specific address
- Addresses are still symbols (i.e., starting from 0 in each object file), and not properly assigned to an address in the memory map

Useful `gcc` commands

- Can break down the steps to generate the final executable file
  - `gcc –c main.c` (generates `main.o`)
  - `gcc –c sum.c` (generates `sum.o`)
  - `gcc –o prog main.o sum.o`
- Can generate the assembly (`.s`) file to view the assembly
  - `gcc –S main.c` (generates `main.s`)

Program vs. Process

- Program
  - A compiled executable binary lies dormant on a storage medium such as disk
- A process is a running program
  - The loader loads the executable file (image) in memory and fills up the memory map with code and data
  - Process has an execution environment (stack and heap)

Scope

- C identifiers (variables, functions, macros) have scope that delimits the regions where they can be accessed
- Four type of scope
  - File
  - Block
  - Function prototype
  - Function
- Scope is determined by where a variable is declared in the program

Example: Scope

```
int j;  // file scope of j begins

void f(int i) { // block scope of i begins
   int j = 1;  // block scope of j begins; hides file-scope j
   i++;        // i refers to function parameter
   for (int i = 0; i < 2; i++) { // block scope of loop-local i begins
      int j = 2; // block scope of inner j begins, hides outer j
      printf("%d\n", j); // inner j is in scope, prints ?
   }
   printf("%d\n", j); // outer j is in scope, prints ?
} // the block scope of i and j ends

void g(int j); // j has function prototype scope; hides file-scope j
```

Storage Duration

- Variables (objects) have a storage duration that determines their lifetime
  - There is a physical memory location reserved for the variable
- **Example:** Variables declared inside a code block or function definition are alive during the execution of the block or function
- **Example:** The loop index (declared inside the `for` statement) dies when loop terminates
- Four durations available in C
  - `automatic`
  - `static`
  - `allocated`
  - `thread` (related to concurrency, won't cover)

Storage Duration: `automatic`

- Consider the function definition below
  - Variable `life_and_death` has `automatic` storage duration
    - Implicit, no need to specify explicitly
  - It is alive only during the execution of function in which it is declared

```
void function(int i) {
   int life_and_death = 1;
   printf("%i\n", life_and_death);
}

int main() {
   int life_and_death = 2;
   printf("%i\n", life_and_death);
   return 0;
}
```

Storage Duration: `automatic`

- More generally, variables declared in a code block demarcated by `{...}` have automatic storage duration

```
int main() {
    for (int i = 0; i < 100; i++)
        printf("%i\n", i);
    return 0;
}
```

`i` has automatic storage duration

- The compiler (or assembly programmer) can reclaim the register in which `i` is stored after the `for` loop terminates

Storage Duration: static

- Objects declared in file scope have static storage duration
- Array big_array has static storage duration
  - Lifetime → Entire execution of program
- Static storage duration is implicit for variables declared in file scope

```
int big_array[1L<<24]; //static
int huge_array[1L<<31]; // static

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

Storage Duration: `static`

- We can explicitly use `static` storage duration for variables inside functions
- These variables persist after the function exits

```
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
   // ctr is only initialized once
   static unsigned int ctr = 0;
   ctr++;
   printf("ctr = %d\n", ctr);
}

int main() {
   for (int i=0; i<5; i++)
      increment();
   return 0;
}
```

Storage Duration: `static`

- We can explicitly use `static` storage duration for variables inside functions
- These variables persist after the function exits
- Output
  - `ctr = 1`
  - `ctr = 2`
  - `ctr = 3`
  - `ctr = 4`
  - `ctr = 5`

```
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
   // ctr is only initialized once
   static unsigned int ctr = 0;
   ctr++;
   printf("ctr = %d\n", ctr);
}

int main() {
   for (int i=0; i<5; i++)
    increment();
   return 0;
}
```

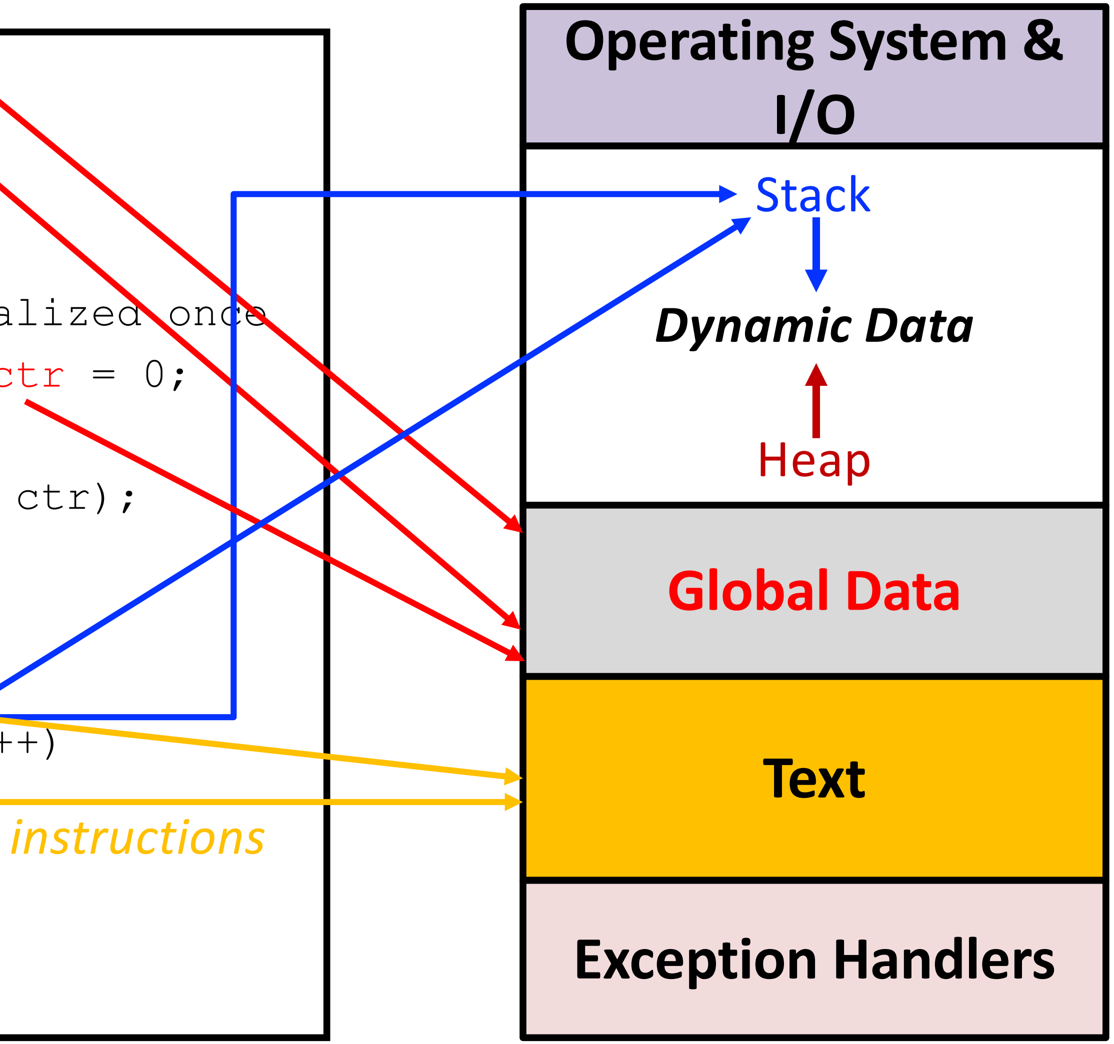
Storage Duration: `static`

- We can declare `ctr` outside the function `increment()`
- Good software engineering practice to limit the scope of a variable whenever possible
- Static variables cannot be initialized to the symbol/name of another variable
- Initialized only to constants such as `0, 1, “Hello”`

```
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
    // ctr is only initialized once
    static unsigned int ctr = 0;
    ctr++;
    printf(“ctr = %d\n”, ctr);
}

int main() {
    for (int i=0; i<5; i++)
     increment();
    return 0;
}
```

Where is everything mapped?

```c
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
   // ctr is only initialized once
   static unsigned int ctr = 0;
   ctr++;
   printf("ctr = %d\n", ctr);
}

int main() {
   for (int i=0; i<5; i++)
    increment();
   int *ptr;
   return 0;
}
```



Statically Allocated Memory

```
int big_array[1L<<24];
int huge_array[1L<<31];
```

- The above arrays are statically allocated
  - Size must be known at compile time (*limitation of static*)
- We need to specify the size of the statically-allocated arrays

```
int N;
int big_array[N];✗

int main() {
    scanf("&i",&N);✓
    int array[N];
}
```

This array is allocated on the stack

Limitations of Stack Allocation

- Memory is allocated and deallocated (freed) in a specific order
  - Last In First Out (LIFO)
- Memory is retained on the stack even if it is not needed
  - Memory is a precious resource
- No way for programmer to inform the compiler/OS to free unused memory
  - Deallocated only when function returns
  - And in a specific order

Example: Mem Waste

```
int caller_useless_func() {
    int sum = 0;
    int array[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++)
        sum += array[i];
    int x = useless_func(sum);
    return x * sum;
}

int useless_func(int var) {
    int multiplied = var * 5;
    return multiplied;
}
```

`array` is no longer needed after this statement, but it is still retained on the stack

Another Problem: Stack Allocation

- Sharing data across functions

```
int caller_useless_func() {
    int sum = 0;
    int array[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++)
        sum += array[i];
    int *y =  useless_func(sum);
    printf("Now printing .... \n");
    printf("%d\n", *y * sum);
}

int *useless_func(int var) {
    int multiplied = var * 5;
    return &multiplied;
}
```

`printf's` stack frame likely corrupts the `useless_func's` stack frame

`multiplied` is dead after the function returns

Problem: Stack Allocation

- We can solve the problem by allocating `multiplied` as a global variable
- What if we want to share an array b/w functions?
  - Statically allocate the array
  - Ok, but what if we only find out the exact length of the array at run-time? What if we want to resize the array?
  - E.g., number of records in a database not known in advance
  - Student numbers grow dynamically during the semester
- We use the heap for such “dynamically allocated” memory

Pointers: Revision

- A pointer is a variable that contains the address of another variable

```
int A = 19;
int B = 10;
int C = 8;
int D = 17;
....
int *P = &B;
//unary operator & gives
   the address of a
   variable
```

Address	Data	Variable
⋮	⋮	⋮
00000010	00000004	P
0000000C	17	D
00000008	8	C
00000004	10	B
00000000	19	A

4 Bytes

Pointers: Revision

- Can use the pointer to access the value stored in a memory location

```
int A = 19;
int B = 10;
int C = 8;
int D = 17;
....
int *P = &B;
*P = 1;
//dereferencing or
// indirection operator
//that accesses the value
// stored at address in P
```

Address	Data	Variable
⋮	⋮	⋮
00000010	00000004	P
0000000C	17	D
00000008	8	C
00000004	1	B
00000000	19	A

4 Bytes

Pointers: Example

- A pointer is 4-bytes on a 32-bit system and 8-bytes on a 64-bits system & it can be stored on the stack or data segment like ordinary variables

```
int A = 19;
int B = 1;
int C = 8;
int D = 17;
....
int *P = &B;
char *Q = &B;
// Both P and Q contain
   00000004
printf("%i\n",*P); ??
printf("%i\n",*Q); ??
```

Address	Data	Variable
:	:	:
00000010	00000004	P
0000000C	17	D
00000008	8	C
00000004	1	B
00000000	19	A

4 Bytes

Pointers: Bottomline

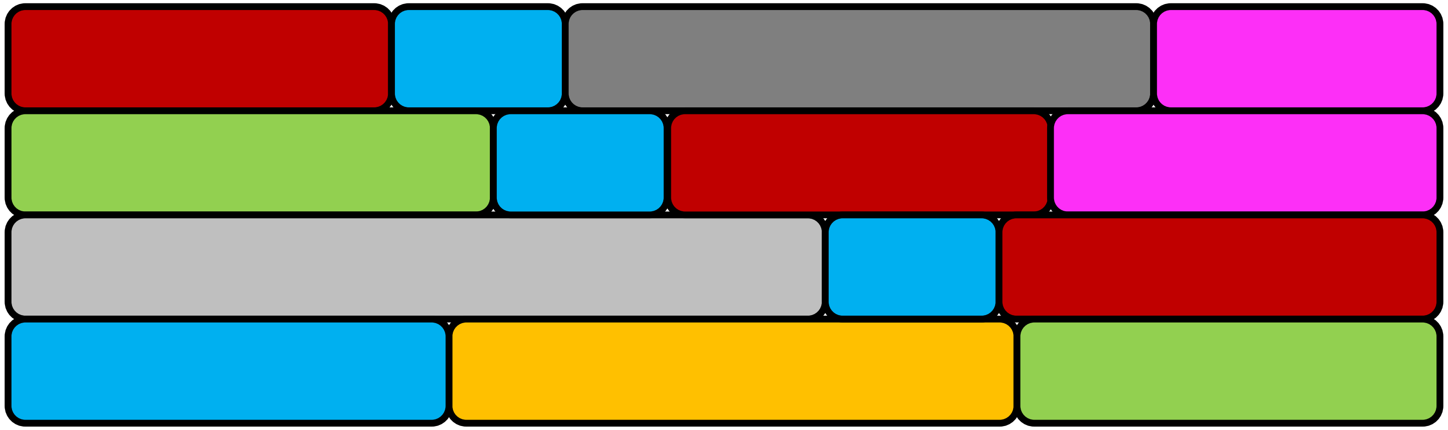
- A pointer points to a memory location
- Its content is a memory address
- It wears "datatype glasses"
  - Wherever it points, it sees through these glasses
- The variable stored at some memory address can be interpreted (via the dereferencing operator *) as character or integer or float, depending on the type of the pointer

Heap

- Heap is for dynamically allocated memory
  - Contrast with statically allocated memory
  - Large subdividable block of memory
  - Programmers explicitly allocate and deallocate memory on the heap
- Lifetime of heap variables/arrays extend from allocation until deallocation
- Memory managers are libraries that manage heap for the programmer (we will refine this view)

Heap Organization

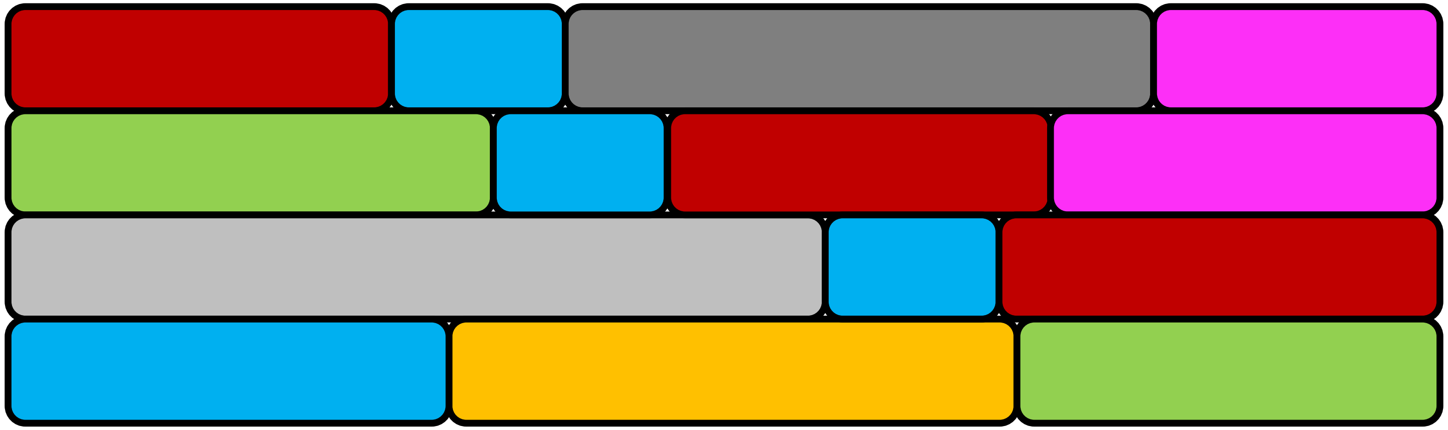
- Programs dynamically allocate objects (arrays, structures) of different types and sizes on the heap over time



- **Heap is now full!**

Heap Organization

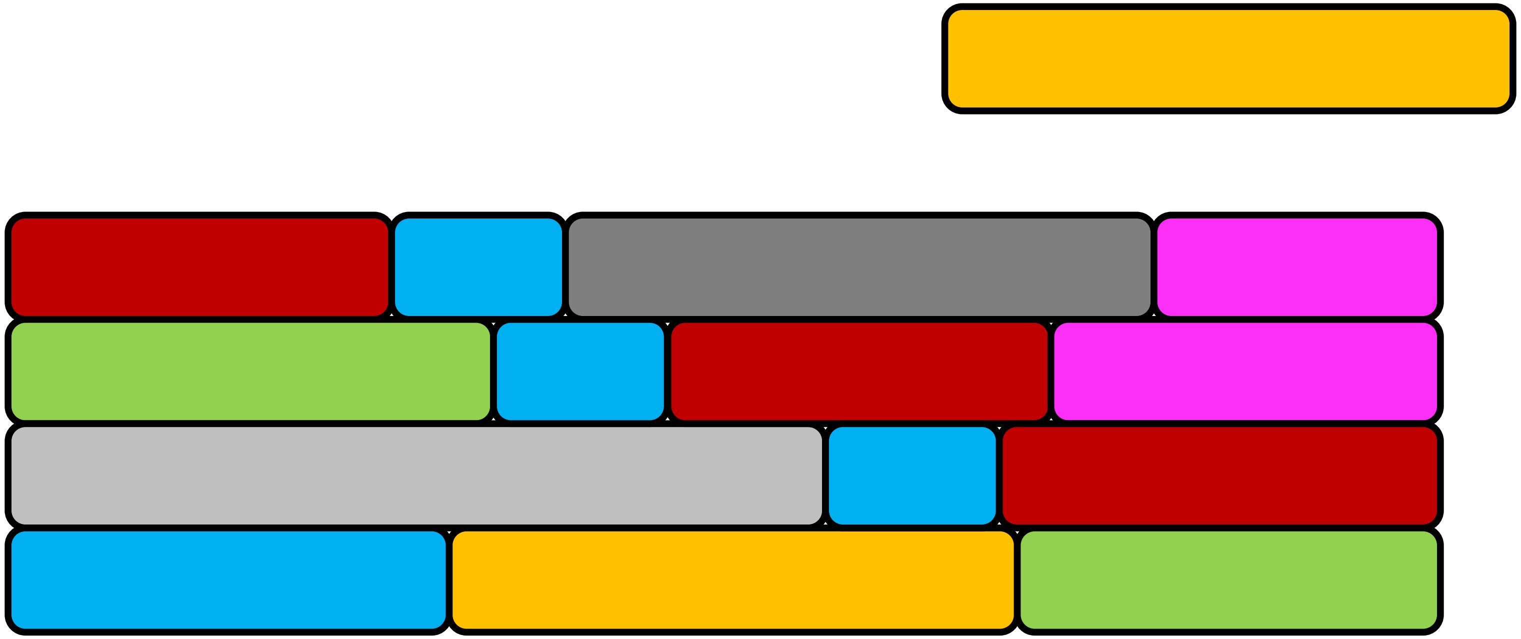
- C programmers need to track lifetime of heap variables
- Suppose we do not need anymore 



- **Heap is now full!**

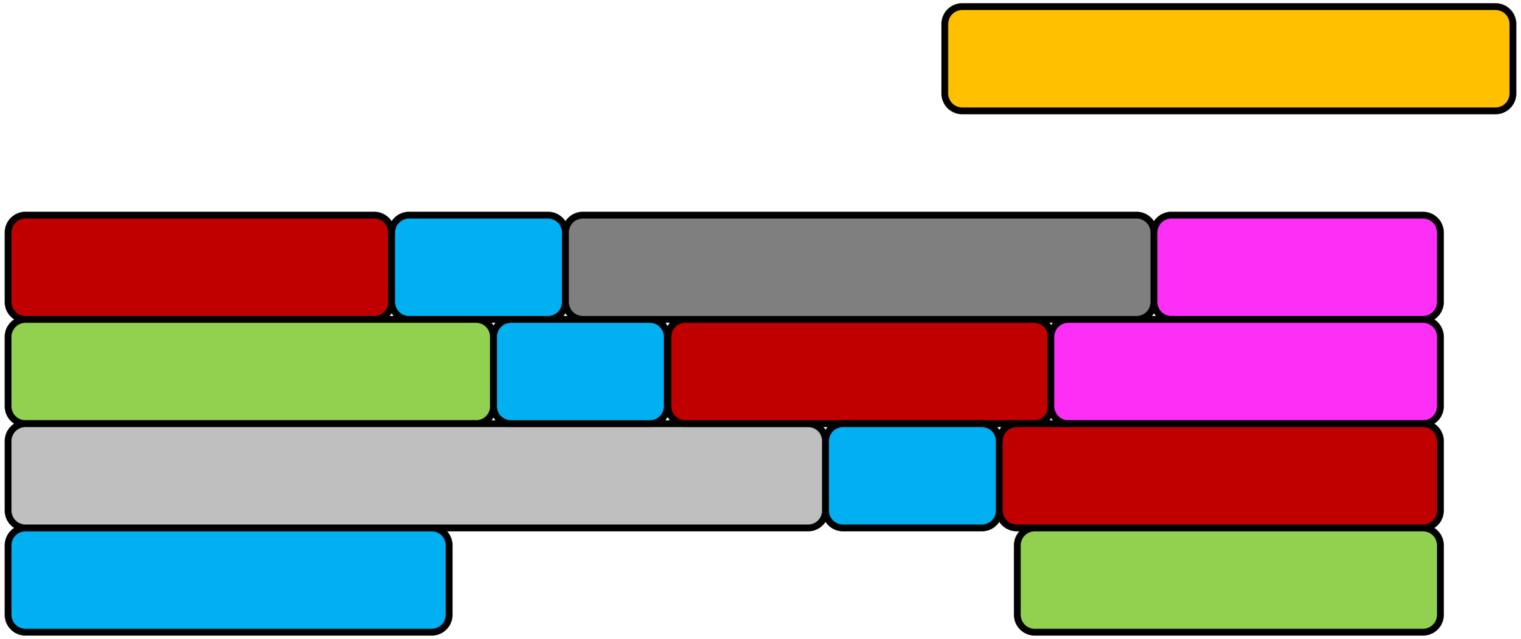
Heap Organization

- Let’s free some space on the heap



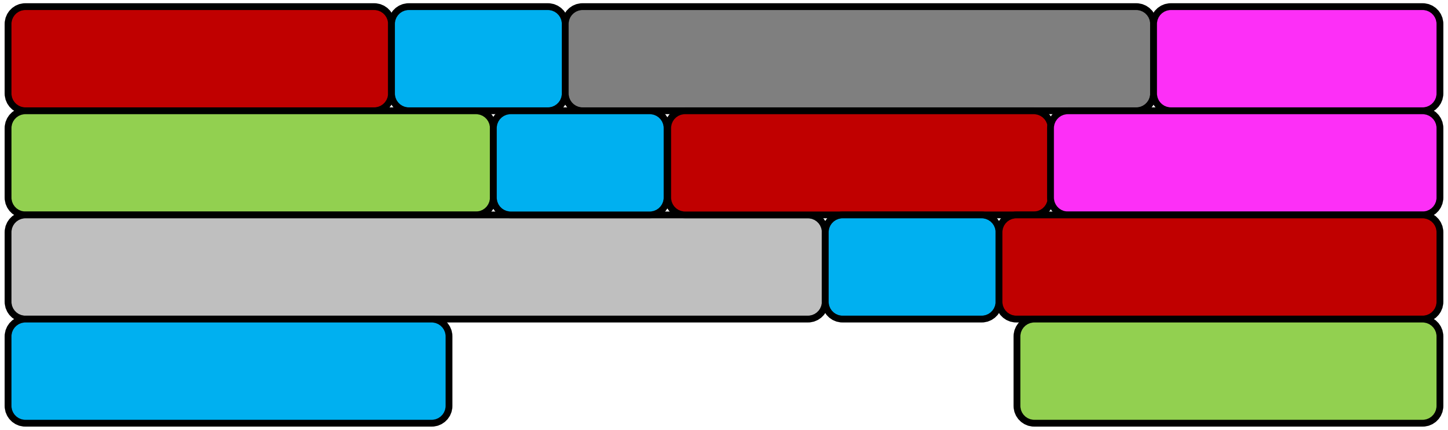
Heap Organization

- Let’s free some space on the heap



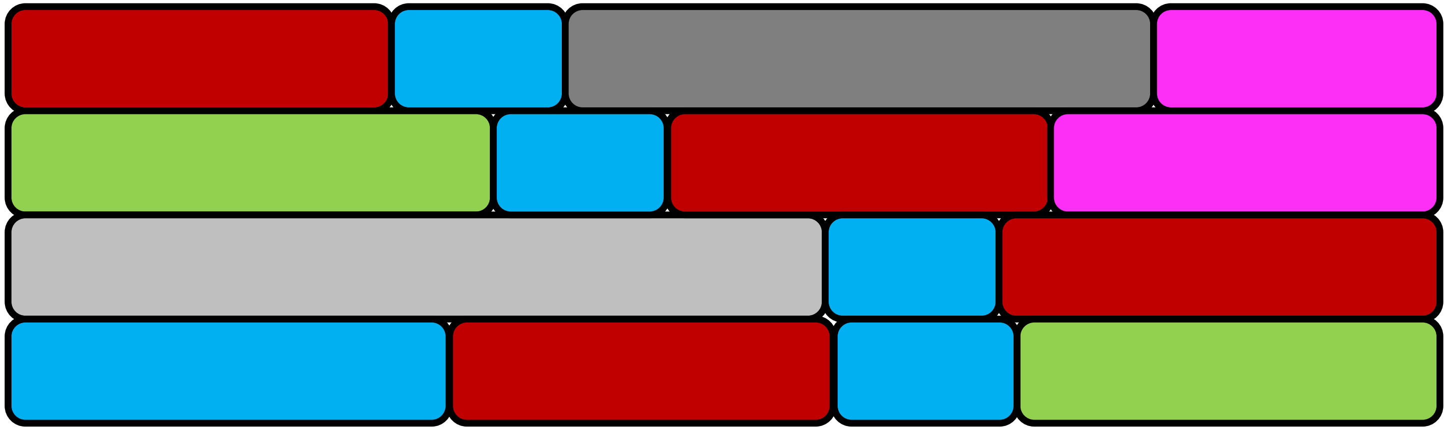
Heap Organization

- Now we can allocate new objects



Heap Organization

- Now we can allocate new objects



Heap Management

- We can deallocate (free) objects in any order (unlike the stack)
- Programmers need to deallocate objects that are no longer needed
  - Otherwise, heap will remain full even if we can have some free space
- Not returning unused heap back to the memory manager is called a *memory leak*

`malloc` and `free`

- C library provides `malloc` (short for memory allocate) and `free` to allocate and deallocate heap memory, respectively

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

void useless_func() {
    int *array = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
        array[i] = i * i;
    int sum = array[0] + array[9];
    free(array);
    printf("%i\n",sum);
    return;
}
```

`malloc` and `free`

- `malloc`
  - Declaration in C library: `void *malloc(size_t size)`
  - Takes input as size (# bytes)
  - Returns a void pointer that can be casted to any pointer type
- `free`
  - Declaration in C library: `void free(void *ptr)`
  - Memory manager knows how many bytes to free, all it needs is the starting address

Quiz: Bug?

- Is there a memory-related bug in the following program?

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

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    int *array2 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++) {
       array1[i] = i * i;
       array2[i] = i + i;
    }
    int sum = array1[0] + array2[9];
    free(array1);
    free(array2);
    printf("%i\n",sum + array1[8]);
    return;
}
```

Quiz: Bug?

- Is there a memory-related bug in the following program?

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

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    int *array2 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++) {
       array1[i] = i * i;
       array2[i] = i + i;
    }
    int sum = array1[0] + array2[9];
    free(array1);
    free(array2);
    printf("%i\n",sum + array1[8]);
    return;
}
```

use after free

Quiz: Bug?

- Is there a memory-related bug in the following program?

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

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
       array1[i] = i * i;
    array1 = malloc(5 * sizeof(int));
    for (int i = 0; i < 10; i++)
       array1[i] = i * i;
    free(array1);
    printf("Done ....\n");
    return;
}
```

Quiz: Bug?

- Is there a memory-related bug in the following program?

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

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
       array1[i] = i * i;
    array1 = malloc(5 * sizeof(int));
    for (int i = 0; i < 10; i++)
       array1[i] = i * i;
    free(array1);
    printf("Done ....\n");
    return;
}
```

memory leak
Original 10-element array still on heap (address is gone, not saved)

out of bounds