Arrays & Pointers
Characters & strings

Jinyang Li

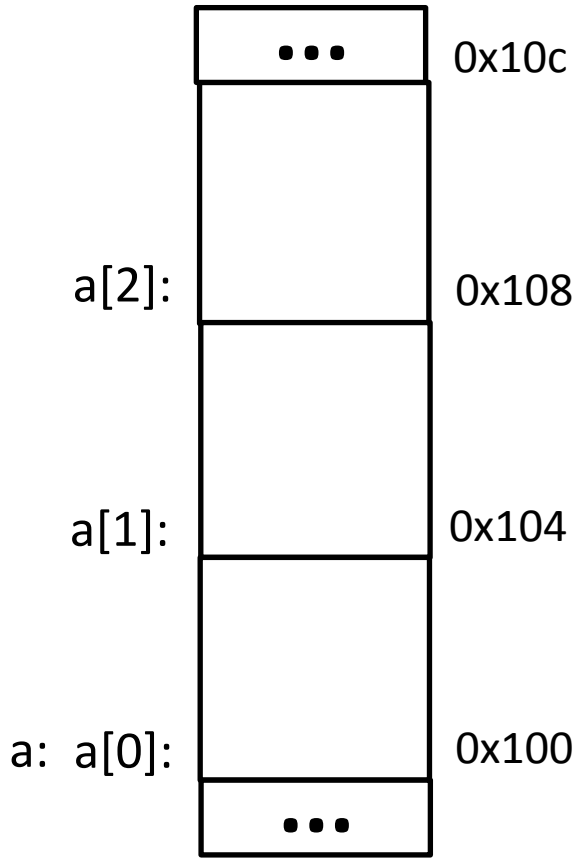
What we've learnt and today's plan

- Bitwise operations
- Pointers
  - Pointers are addresses
  - With pointers arguments, a callee can modify local variables in the caller.
- Today's lesson:
  - Array and its relationship with pointer
  - Pointer casting
  - Characters & strings

Array: a collection of contiguous objects with the same type

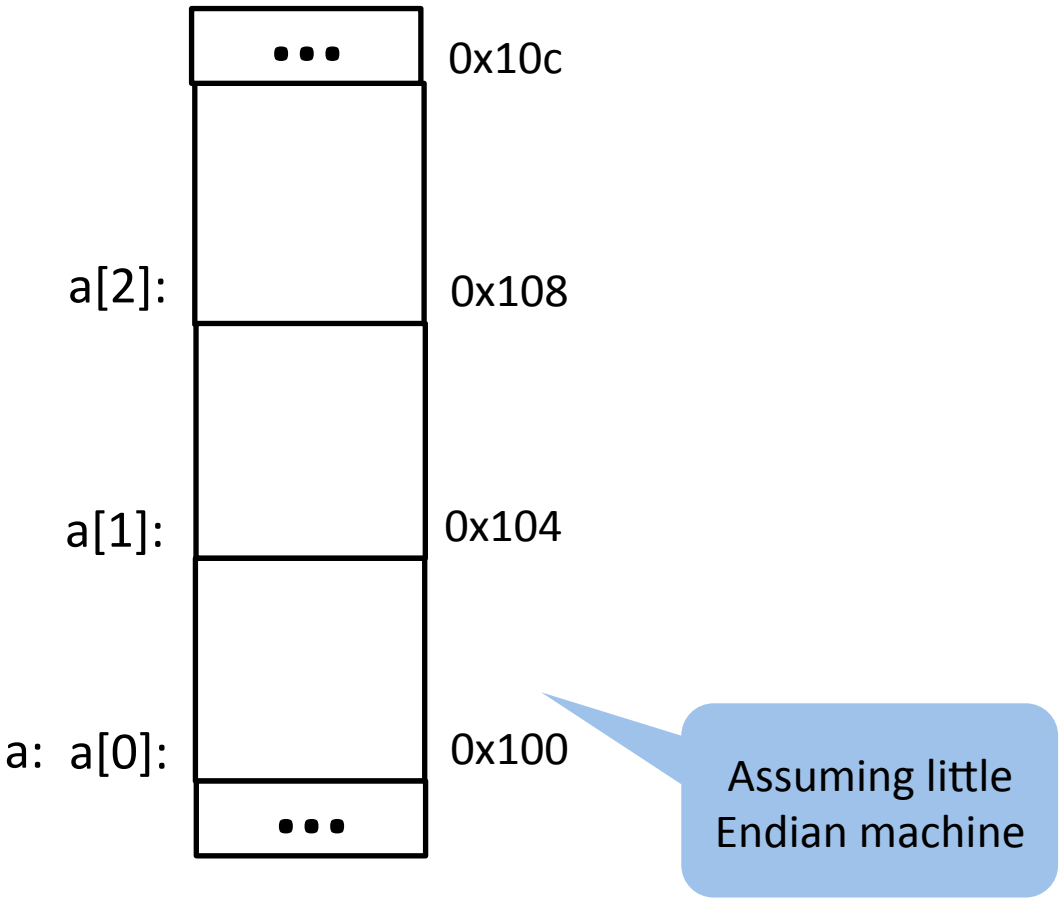
Array

```
int a[3];
```



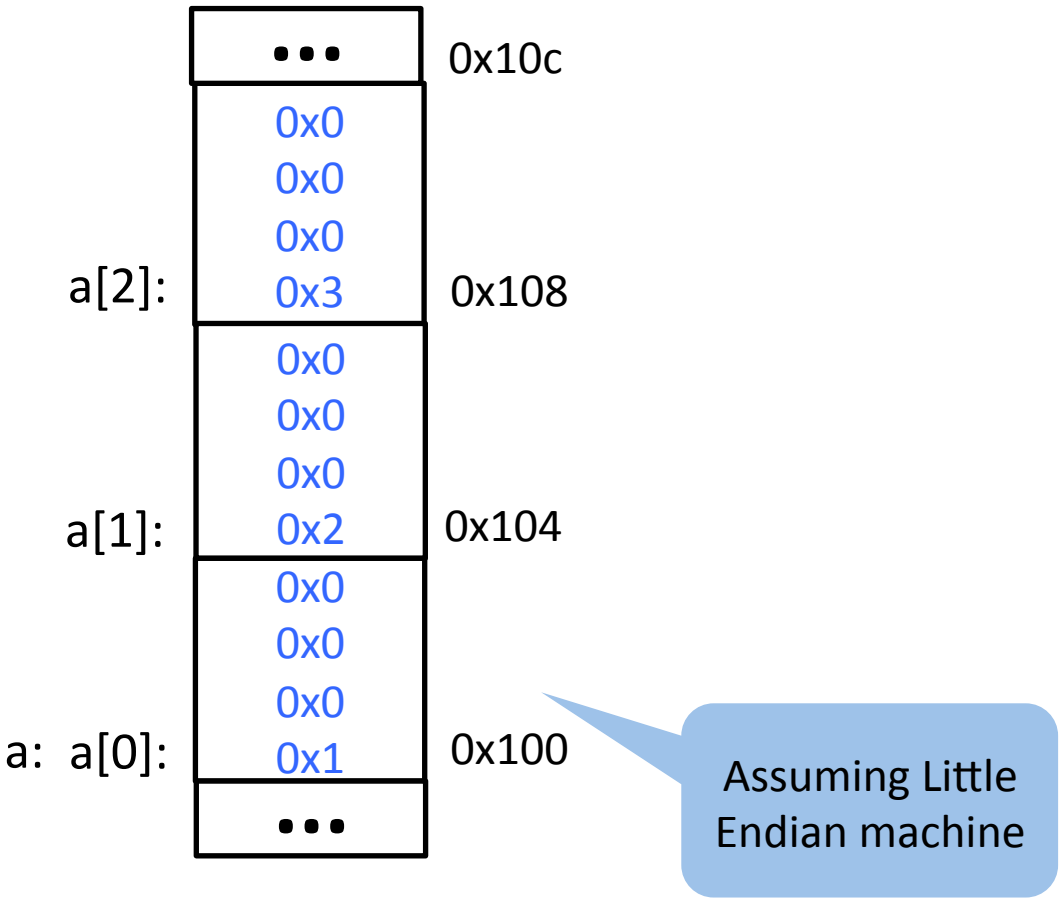
Array

```
int a[3] = {1, 2, 3};
```

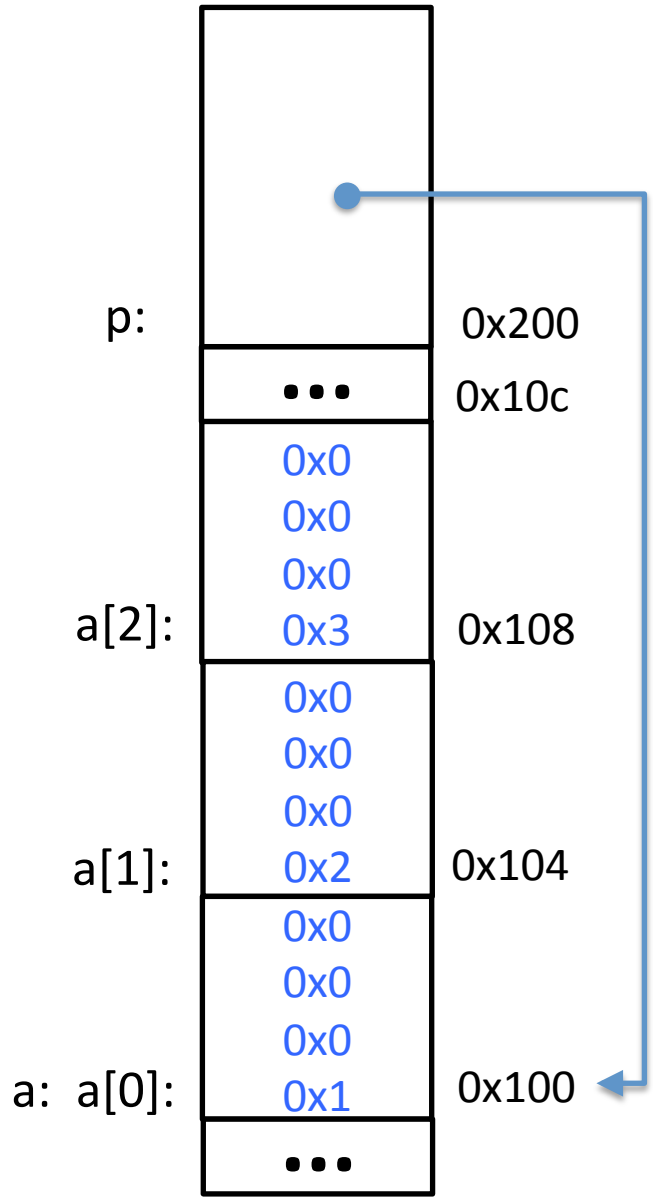


Array

```
int a[3] = {1, 2, 3};
```

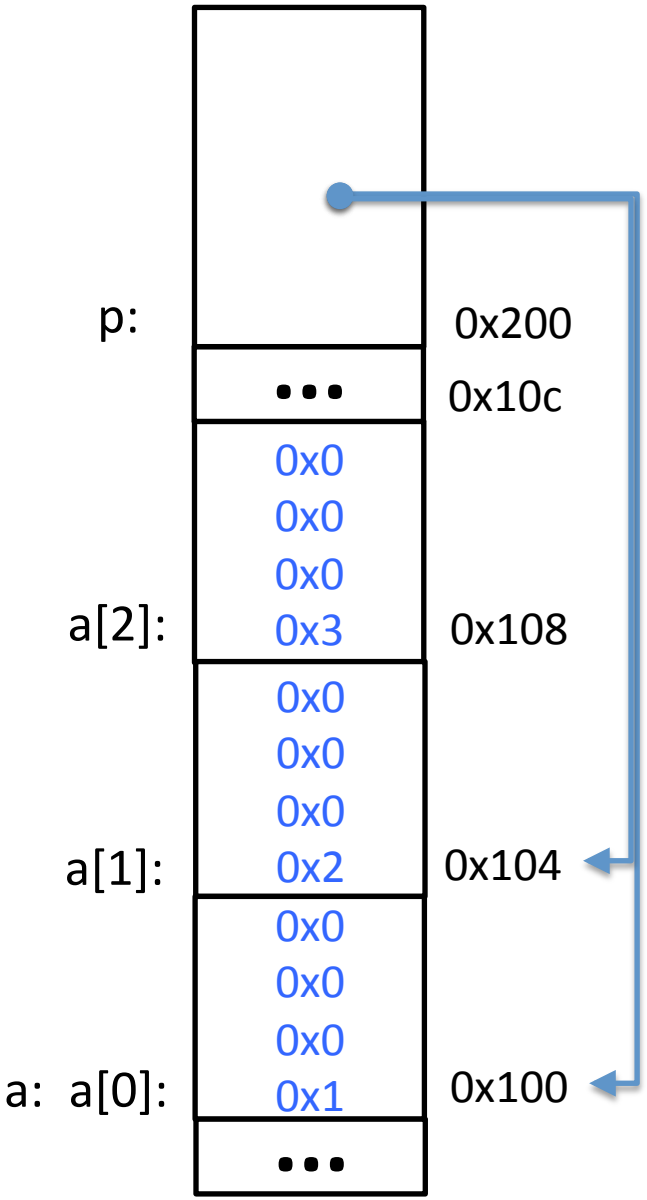


Array




```
int a[3] = {1, 2, 3};
int *p;
p = &a[0]; //equivalent to p = a;
printf("%p\n", p); //output?  0x100
printf("%d\n", *p); //output?  1
```

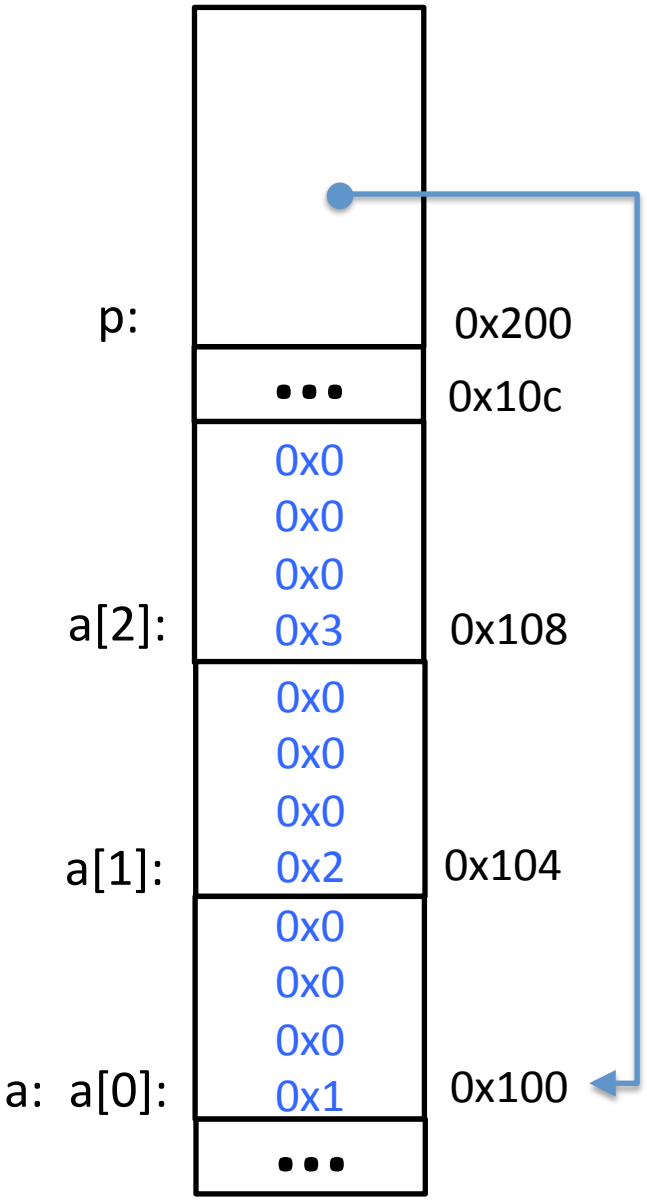
Pointer arithmetic




```
int a[3] = {1, 2, 3};

int *p;
p = &a[0];

p = p + 1; //equivalent to p++

printf("%p\n", p); //output?  0x104

printf("%d\n", *p); //output?  2
```

Rule of pointer arithmetic:
p+i has address of *i*-th object after p, i.e.
p+i's value is p's value plus i*sizeof(object)

Pointer arithmetic

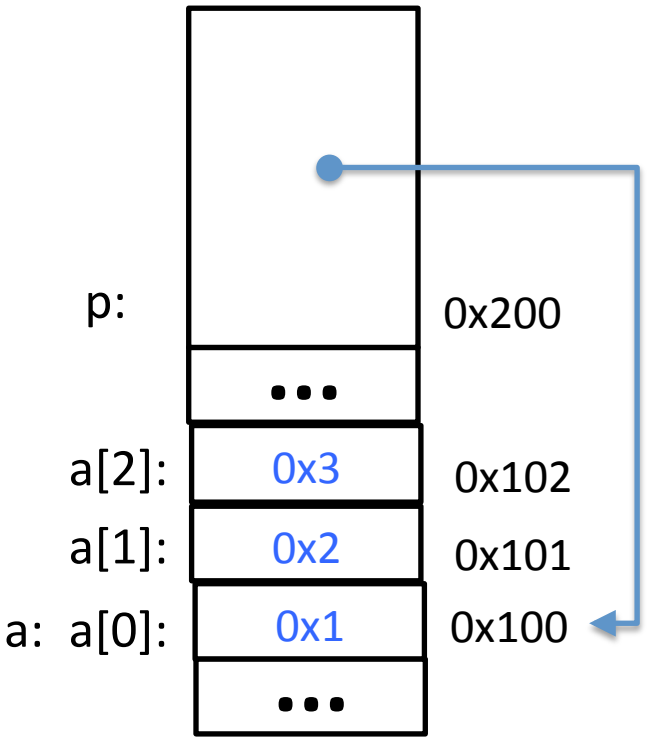



```
int a[3] = {1, 2, 3};

int *p;
p = &a[0];

printf("%p\n", p+2); //output?      0x108

printf("%d\n", *(p+2)); //output?   3
```

Rule of pointer arithmetic:
p+i has address of *i*-th object after p, i.e.
p+i's value is p's value plus i*sizeof(object)

Pointer arithmetic

```
char a[3] = {1, 2, 3};

char *p;
p = &a[0];

printf("%p\n", p+2); //output?      0x102

printf("%d\n", *(p+2)); //output?   3
```



Rule of pointer arithmetic:
p+i has address of *i*-th object after p, i.e.
p+i's value is p's value plus i*sizeof(object)

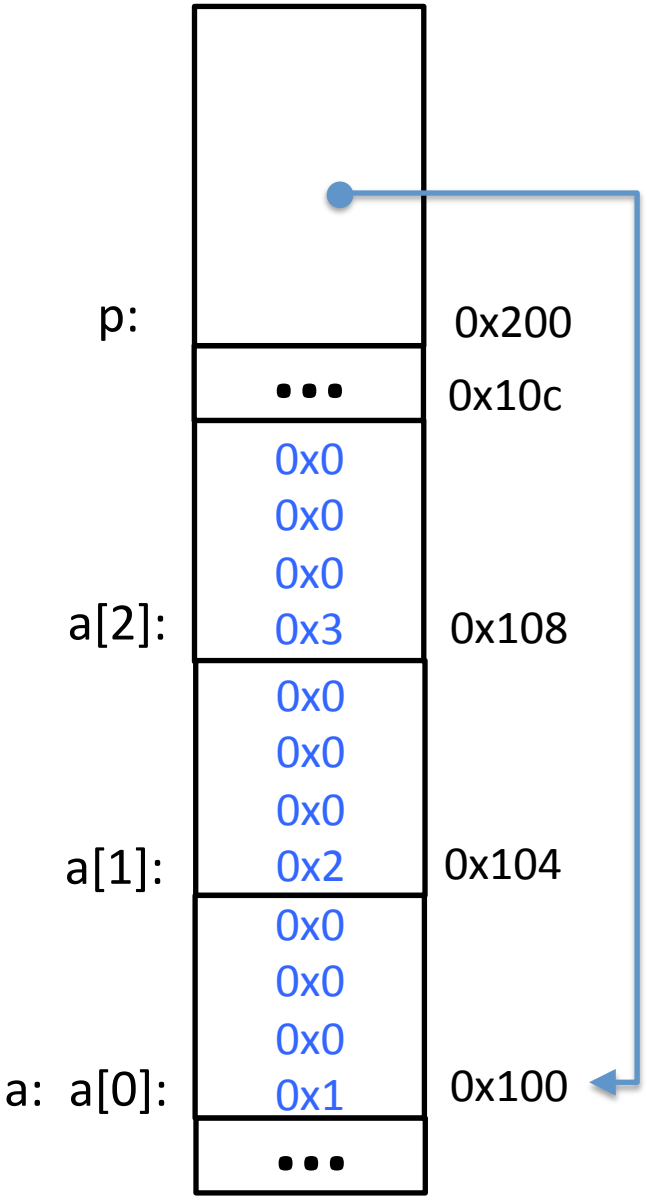
Array and pointer

```
int a[10];
int *p;

p = &a[0]; // a is alias for &a[0];

for (int i = 0; i < 10; i++) {

   *(p+i) = 0; // p[i] is alias for *(p+i)

}
```

Array and pointer

```
int a[10];
int *p;

p = a;      // a is alias for &a[0];

for (int i = 0; i < 10; i++) {

    p[i] = 0;   // p[i] is alias for *(p+i)

}
```

Out of bound access?




```
int a[3] = {1, 2, 3};

int *p;
p = &a[0];

printf("%p\n", p+3); //output?

printf("%d\n", *(p+3)); //output?

printf("%p\n", &a[3]); //output?

printf("%d\n", a[3]); //output?
```

0x10c

??
can be any value
or segmentation fault

0x10c

??
can be any value
or segmentation fault

Example

```
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;
  *p = 400;
  for (int i=0; i<3; i++) {
    printf("%d ", a[i]);
  }
  printf("\n");
}
```

same as:
int *p;
p = a;

same as:
p[0] = 400;

Output? 400 200 300

Another Example

```
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;
  p++;
  *p = 400;
  for (int i=0; i<3; i++) {
    printf("%d ", a[i]);
  }
  printf("\n");
}
```

equivalent to
*(++p) = 400;

Output? 100 400 300

Pass array to function via pointer

```
// multiply every array element by 2
void multiply2(int *a) {
    for (int i = 0; i < ???; i++) {
        a[i] *= 2;
    }
}

int main() {
    int a[2] = {1, 2};
    multiply2(a);
    for (int i = 0; i < 2; i++) {
        printf("a[%d]=%d", i, a[i]);
    }
}
```

Pass array to function via pointer

```
// multiply every array element by 2
void multiply2(int *a, int n) {
    for (int i = 0; i < n; i++) {
        a[i] *= 2; // (*(a+i)) *= 2;
    }
}

int main() {
    int a[2] = {1, 2};
    multiply2(a, 2);
    for (int i = 0; i < 2; i++) {
        printf("a[%d]=%d", i, a[i]);
    }
}
```

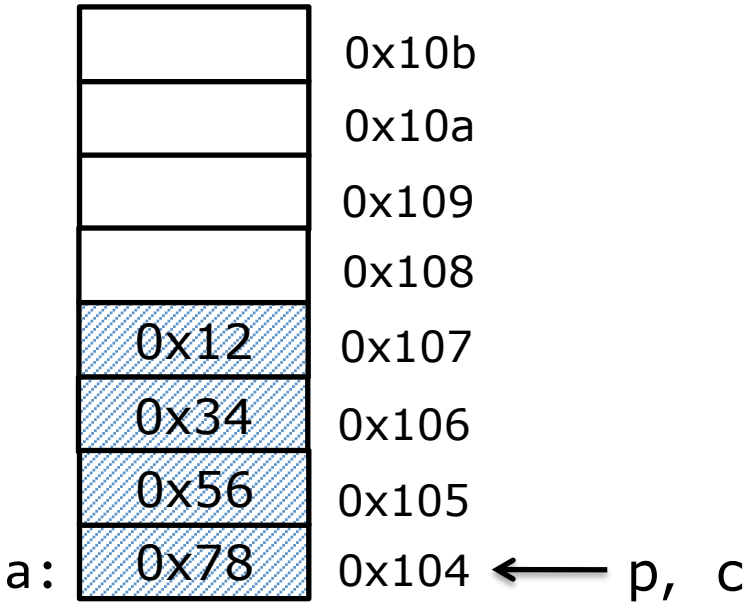
Pointer casting

```
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
printf("%x\n", *c);
```

Output? (when running on Intel laptop)

Pointer casting

```
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```

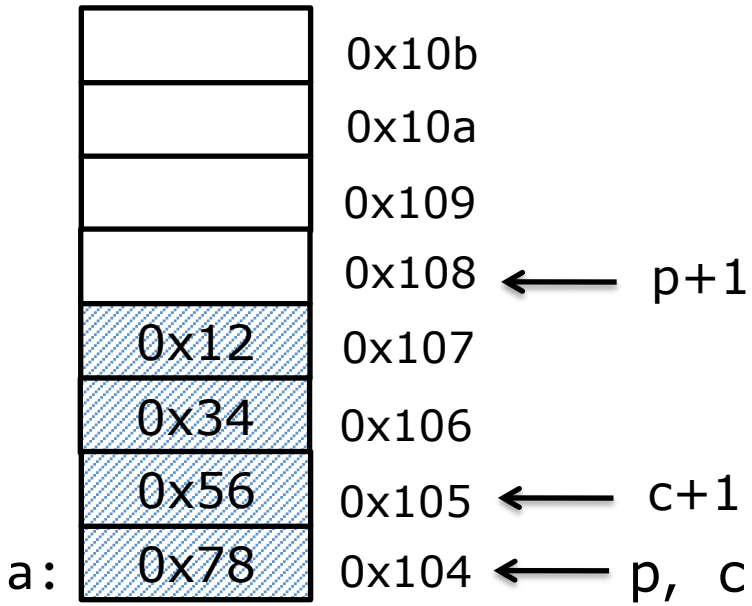


Intel laptop is small endian
*c is 0x78

What is c+1? p+1?

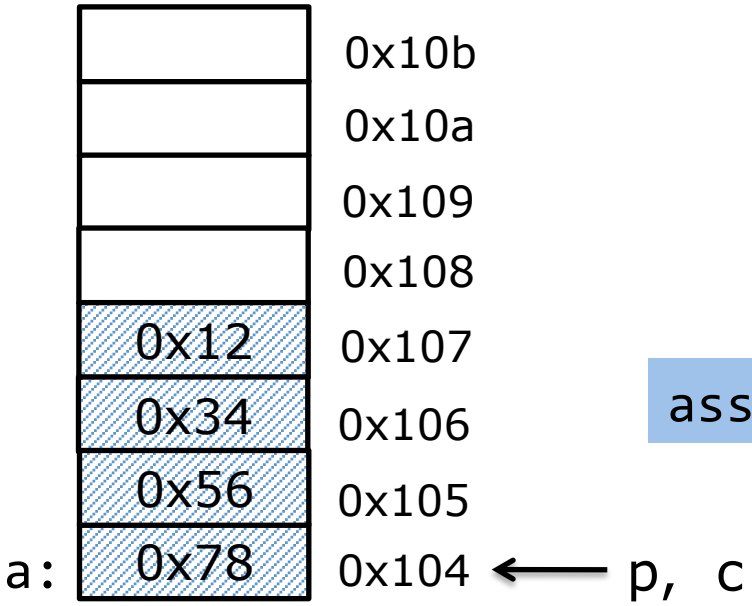
Pointer casting

```
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```



Pointer casting

```
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```

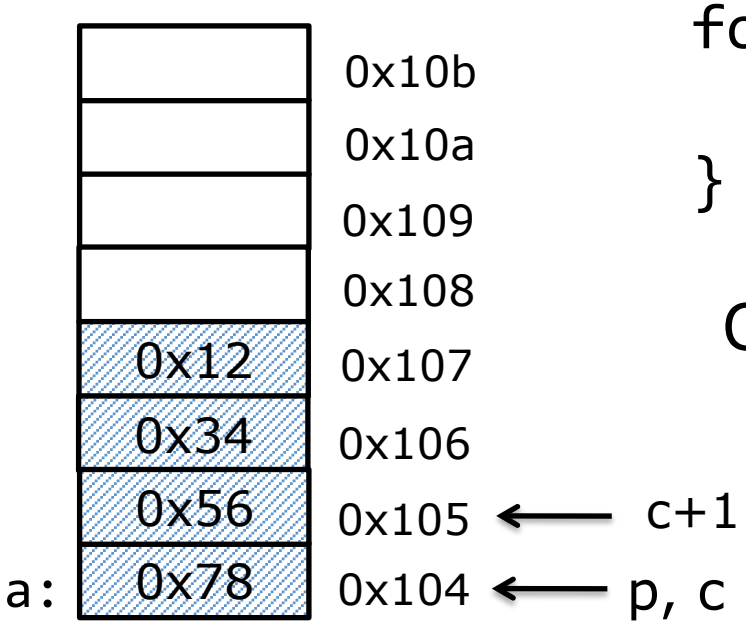



```
assert(p+i == (char *)p + i*sizeof(*p))
```

sizeof(*p), or sizeof(int) is a C built-in that returns size of object/expression

Pointer casting

```
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```




```
for (int i = 0; i < 4; i++) {
     print("%x ", c[i]);
}
```

Output: 0x78 0x56 0x34 0x12

What about big endian?

Another example of pointer casting

```
bool is_normalized_float(float f)
{




}
```

Another example of pointer casting

```
bool is_normalized_float(float f)
{
    unsigned int i;
    i = *(unsigned int *)&f;

    unsigned exp = (i&0x7fffffff)>>23;
    return (exp != 0 && exp != 255);

}
```

function *sizeof*

sizeof(`type`)

- Returns size in bytes of the object representation of type

sizeof(expression)

- Returns size in bytes of the type that would be returned by expression, if evaluated.

function *sizeof*

sizeof()	result (bytes)
sizeof(int)	
sizeof(long)	
sizeof(float)	
sizeof(double)	
sizeof(int *)	

64 bits machine

function *sizeof*

sizeof()	result (bytes)
sizeof(int)	4
sizeof(long)	8
sizeof(float)	4
sizeof(double)	8
sizeof(int *)	8

64 bits machine

function *sizeof*

expr	sizeof()	result (bytes)
int a = 0;	sizeof(a)	
long b = 0;	sizeof(b)	
int a = 0; long b = 0;	sizeof(a + b)	
char c[10];	sizeof(c)	
int arr[10];	sizeof(arr)	
	sizeof(arr[0])	
int *p = arr;	sizeof(p)	

64 bits machine

function *sizeof*

expr	sizeof()	result (bytes)
int a = 0;	sizeof(a)	4
long b = 0;	sizeof(b)	8
int a = 0; long b = 0;	sizeof(a + b)	8
char c[10];	sizeof(c)	10
int arr[10];	sizeof(arr)	10 * 4 = 40
	sizeof(arr[0])	4
int *p = arr;	sizeof(p)	8

64 bits machine

Undefined behavior

In computer programming, undefined behavior (UB) is the result of executing computer code whose behavior is not prescribed by the language specification.

Classic undefined behaviors

- Use an uninitialized variable

```
int a;
int b = a + 1;
```

- out of bound array access

```
int a[2] = {1, 2};
int *p = a
*(p+3) = 3;
```

- Divide by zero

```
int a = 1 / 0;
```

- integer overflow

```
int a = 0x7fffffff
int b = a + 1
```

Why does C have undefined behavior?

Simplify compiler's implementation

Enable better performance

Classic undefined behaviors

- Use uninitialized variables
  - Avoid memory write
- Out-of-bound array access
  - Avoid runtime bound checking
- Divided by zero ?
- integer overflow

Classic undefined behaviors

At instruction set level, different architectures handle them in different ways:

Divided by zero

- X86 raises an exception
- MIPS and PowerPC silently ignore it.

integer overflow

- X86 wraps around (with flags set)
- MIPS raises an exception.

Classic undefined behaviors

Assumption: Unlike Java, C compilers trust the programmer not to submit code that has undefined behavior

The compiler optimizes this code under this assumption

→ Compiler may remove the code or rewrite the code in a way that programmer did not anticipate

Classic undefined behaviors

```
#include <stdio.h>

void foo(int a) {
  if(a+100 < a) {
    printf(”overflowed\n");
     return;
  }

  printf(”normal is boring\n");
}

int main() {
  foo(100);
  foo(0x7fffffff);
}
```

Classic undefined behaviors

```
#include <stdio.h>

void foo(int a) {
  if(a+100 < a) {
    printf("overflowed\n");
    return;
  }

  printf("normal is boring\n");
}

int main() {
  foo(100);
  foo(0x7fffffff);
}
```

gcc removes the check with O3

Characters

How to represent text characters?

- How to associate bit patterns to integers?
  - base 2
  - 2's complement
- How to associate bit patterns to floats?
  - IEEE floating point representation (based on normalized scientific notation)
- How to associate bit patterns to characters?
  - by convention
  - ASCII, UTF

ASCII: American Standard Code for Information Exchange

- Developed in 60s, based on the English alphabet
- use one byte (with MSB=0) to represent each character
- How many unique characters can be represented?

128

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

C exercise 1: tolower

```
// tolower returns the corresponding
// lowercase character for c if c is an
// uppercase letter. Otherwise, it returns c.
char tolower(char c) {




}
int main() {
    char c = ‘A’;
    c = tolower(c);
    ...
}
```

C exercise 1: tolower

```
// tolower returns the corresponding
// lowercase character for c if c is an
// uppercase letter. Otherwise, it returns c.
char tolower(char c) {

    // test if c is an uppercase letter
    if (c < 'A' || c > 'Z') {
        return c;
    }


}
```

C exercise 1: tolower

```
// tolower returns the corresponding
// lowercase character for c if c is an
// uppercase letter. Otherwise, it returns c.
char tolower(char c) {

    // test if c is an uppercase letter
    if (c < 'A' || c > 'Z') {
        return c;
    }

    return c + ('a' - 'A');

}
```

C's standard library includes tolower, toupper

C exercise 2: toDigit

```
// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g ‘1’, ‘2’,
// Otherwise, it returns -1.
int toDigit(char c) {




}

int main() {
    int d = toDigit(‘8’);
    printf(“int is %d, multiply-by-2 %d\n”, d, 2*d);
}
```

C exercise 2: toDigit

```
// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g ‘1’, ‘2’,
// Otherwise, it returns -1.
int toDigit(char c) {
     // test if c is a valid character
     if (c < ‘0’ || c > ‘9’) {
         return -1;
     }

}
int main() {
   int d = toDigit(‘8’);
   printf(“int is %d, multiply-by-2 %d\n”, d, 2*d);
}
```

C exercise 2: toDigit

```
// toDigit returns the corresponding integer for c
// if c is a valid digit character, e.g ‘1’, ‘2’,
// Otherwise, it returns -1.
int toDigit(char c) {
     // test if c is a valid character
     if (c < ‘0’ || c > ‘9’) {
          return -1;
     }
     return c – ‘0’;
}
int main() {
   int d = toDigit(‘8’);
   printf(“int is %d, multiply-by-2 %d\n”, d, 2*d);
}
```

The Modern Standard: UniCode

- ASCII can only represent 128 characters
  - How about Chinese, Korean, all of the worlds languages? Symbols? Emojis?
- Unicode standard represents >135,000 characters

U+1F600		grinning face
U+1F601		beaming face with smiling eyes
U+1F602		face with tears of joy
U+1F923		rolling on the floor laughing
U+1F603		grinning face with big eyes

UTF-8

- UTF-8 is one encoding form for Unicode
  - use 1, 2, or 4 byte to represent a character
  - Unicode for ASCII characters have the same ASCII value → UTF-8 one byte code is the same as ASCII
- C has no primitive support for Unicode

What we've learnt and today's plan

- Bitwise operations
- Pointers & arrays
- ASCII characters
- Today's lesson:
  - String
  - Struct
  - Malloc

C Strings

Strings

- String is represented as an array of chars.
  - Array has no space to encode its length.
- How to determine string length?
  - explicitly pass around an integer representing length

```
// tolower_string turns every character in character array s
// into lower case
void tolower_string(char *s, int len) {
     for (int i = 0; i < len; i++) {
          s[i] = tolower(s[i]);
     }
}
```

Strings

- String is represented as an array of chars.
  - Array has no space to encode its length.
- How to determine string length?
  - explicitly pass around an integer representing length
  - C string stores a NULL character to mark the end (by convention)

```
void tolower_string(char *s) {


}
```

Strings

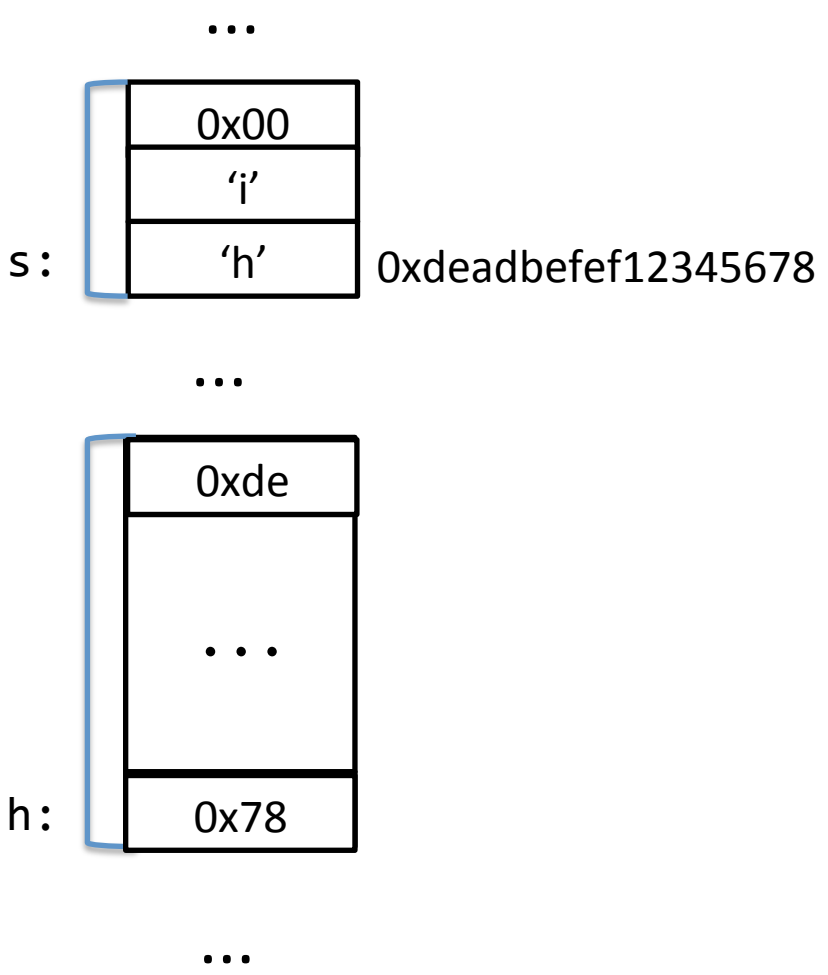
- String is represented as an array of chars.
  - Array has no space to encode its length.
- How to determine string length?
  - explicitly pass around an integer representing length
  - C string stores a NULL character to mark the end (by convention)

```
void tolower_string(char *s) {
    int i = 0;
    while (s[i] != '\0') {
       s[i] = tolower(s[i]);
       i++;
    }
}
```

Copying string?

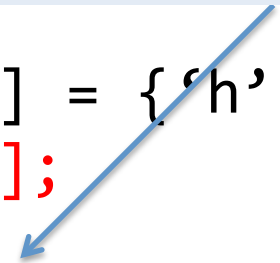
does this make a copy of "hi"?

```
char s[3] = {'h','i','\0'};
char *h;
h = s;
h[0] = 'H';

printf("s=%s h=%s\n",s,h);
```



Copying string?

does this make a copy of "hi"?



```
char s[3] = {'h','i','\0'};
char h[3];
h = s;
h[0] = 'H';

printf("s=%s h=%s\n",s,h);
```

Copying string

```
void strcpy(char *dst, char *src)
{

}

int main()
{
   char s[3] = {‘h’,‘i’,‘\0’};
   char h[3];
   strcpy(h, s);
   h[0] = ‘H’;

   printf(“s=%s h=%s\n”,s,h);
}
```

Copying string

```
void strcpy(char *dst, char *src) {
    int i = 0;
    while (src[i] != '\0') {
       dst[i] = src[i];
       i++;
    }
}

int main() {
   char s[3] = {'h','i','\0'};
   char h[3];
   strcpy(h, s);
   h[0] = 'H';

   printf("s=%s h=%s\n",s,h);
}
```

strcpy is included in C std library.

Copying string

```
void strcpy(char *dst, char *src) {
    int i = 0;
    while (src[i] != '\0') {
       dst[i] = src[i];
       i++;
    }
}

int main() {
   char s[3] = {'h','i', '\0'};
   char h[2];
   strcpy(h, s);
   h[0] = 'H';

   printf("s=%s h=%s\n",s,h);
}
```

Results in out-of-bound write!
Buffer overflow!

Copying string

```
void strncpy(char *dst, char *src, int n) {
    int i = 0;
    while (src[i] != '\0' && i < n) {
       dst[i] = src[i];
       i++;
    }
}

int main() {
   char s[3] = {'h','i','\0'};
   char h[2];
   strncpy(h, s, 2);
   h[0] = 'H';

   printf("s=%s h=%s\n",s,h);
}
```

strncpy is included in C std library.

A different way of initializing string …

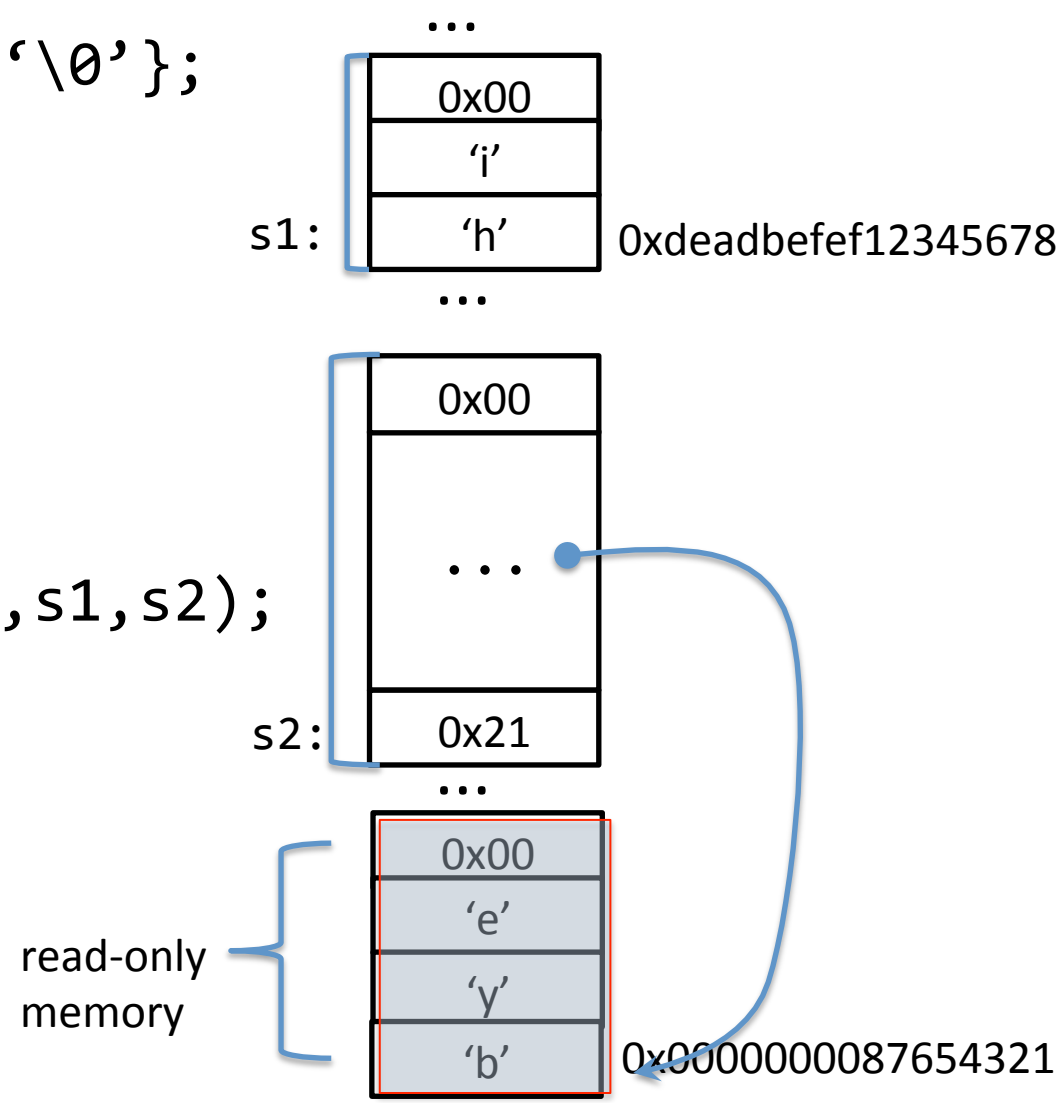
```
char s1[3] = {‘h’,‘i’,‘\0’};
//equivalent to
//char s1[3] = “hi”;
char *s2 = “bye”;
s1[0] = ‘H’;
s2[0] = ‘B’;

printf(“s1=%s s2=%s\n”,s1,s2);
```

OK

Segmentation fault (bus error)

A different way of initializing string

```
char s1[3] = {'h','i','\0'};
//equivalent to
//char s1[3] = "hi";
char *s2 = "bye";
s1[0] = 'H';
s2[0] = 'B';

printf("s1=%s s2=%s\n",s1,s2);
```

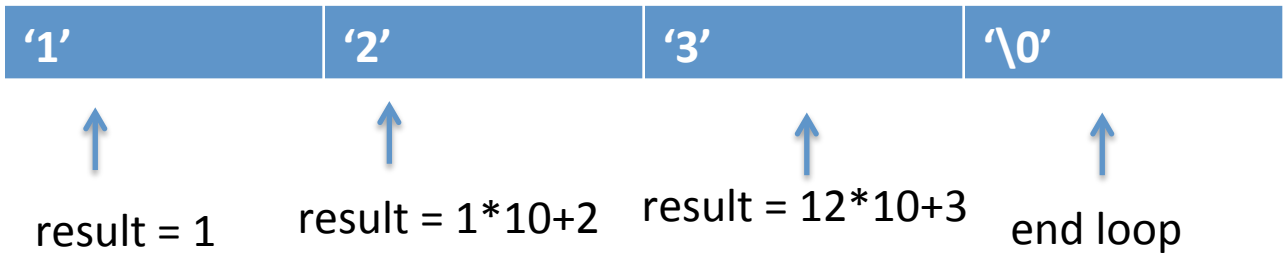


The Atoi function

```
// atoi returns the integer
// corresponding to the string of digits
int atoi(char *s)
{

}

int main()
{
   char *s= “123”;
   printf(“integer is %d\n”, atoi(s));
}
```

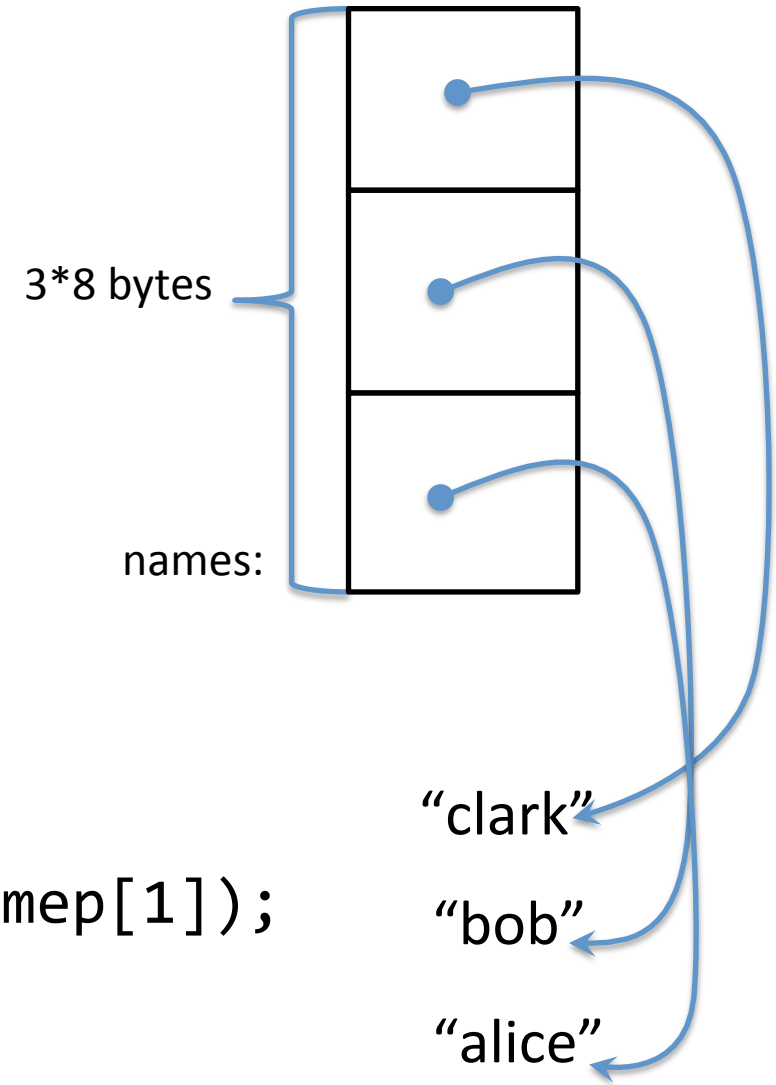
The Atoi function

```
// atoi returns the integer
// corresponding to the string of digits
int atoi(char *s) {
   int result = 0;
   int i = 0;
   while (s[i] >= '0' && s[i] <= '9') {
            result = result * 10 + (s[i] -'0');
            i++;
    }
    return result;
}
```



Array of pointers

```
char* names[3] = {
    “alice”,
    “bob”,
    “clark”
};
```




```
char **namep;
namep = names;

printf(“name is %s”, namep[1]);
```

The most commonly used array of pointers: argv

```
int main(int argc, char **argv)
{
   for (int i = 0; i < argc; i++) {
      printf("%s\n", argv[i]);
   }
}
```

```
$ ./a.out 1 2 3
./a.out 1 2 3
```

argv[0] is the name of the executable

Structs

Struct stores fields of different types contiguously in memory

Structure

- Array: a block of n consecutive elements of the same type.
- Struct: a collection of elements of diffferent types.

Structure

```
struct student {
    int id;
    char *name;
};
```

Fields of a struct are allocated next to each other, but there may be gaps (padding) between them.

Structure

```
struct student {
    int id;
    char *name;
};

struct student t;
```

define variable t with type “struct student”

Structure

```
struct student {
    int id;
    char *name;
};

struct student t;

t.id = 1024;
t.name = “alice”;
```

Access the fields of this struct

Typedef

```
typedef struct {
    int id;
    char *name;
} student;

~~struct~~ student t;
```

Pointer to struct

```
typedef struct {
    int id;
    char *name;
} student;

student t = {1023, “alice”};
student *p = &t;

p->id = 1023;
p->name = “bob”;
printf(“%d %s\n”, t.id, t.name\n”);
```

Mallocs

Allocates a chunk of memory dynamically

Recall memory allocation for global and local variables

- Global variables are allocated space before program execution.
- Local variables are allocated when entering a function and de-allocated upon its exit.

Malloc

Allocate space dynamically and flexibly:

- malloc: allocate storage of a given size
- free: de-allocate previously malloc-ed storage

```
void *malloc(size_t size);
```

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be casted to any type.

```
void free(void *ptr);
```

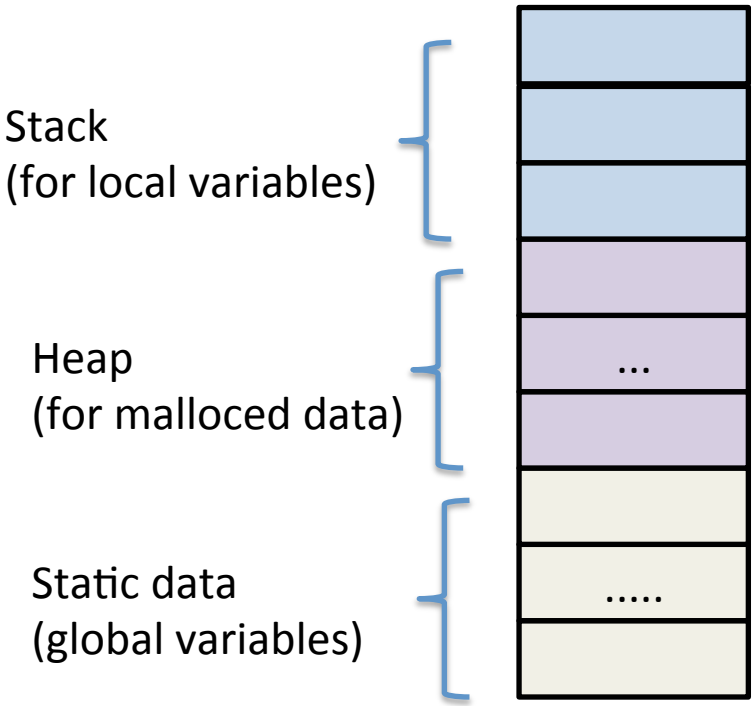
Malloc

Malloc is implemented as a C library

```
#include <stdlib.h>

int *newArray(int n) {
   int *p;
   p = (int*)malloc(sizeof(int) * n);
   return p;
}
```

Conceptual view of a C program's memory at runtime

- Separate memory regions for global, local, and malloc-ed.

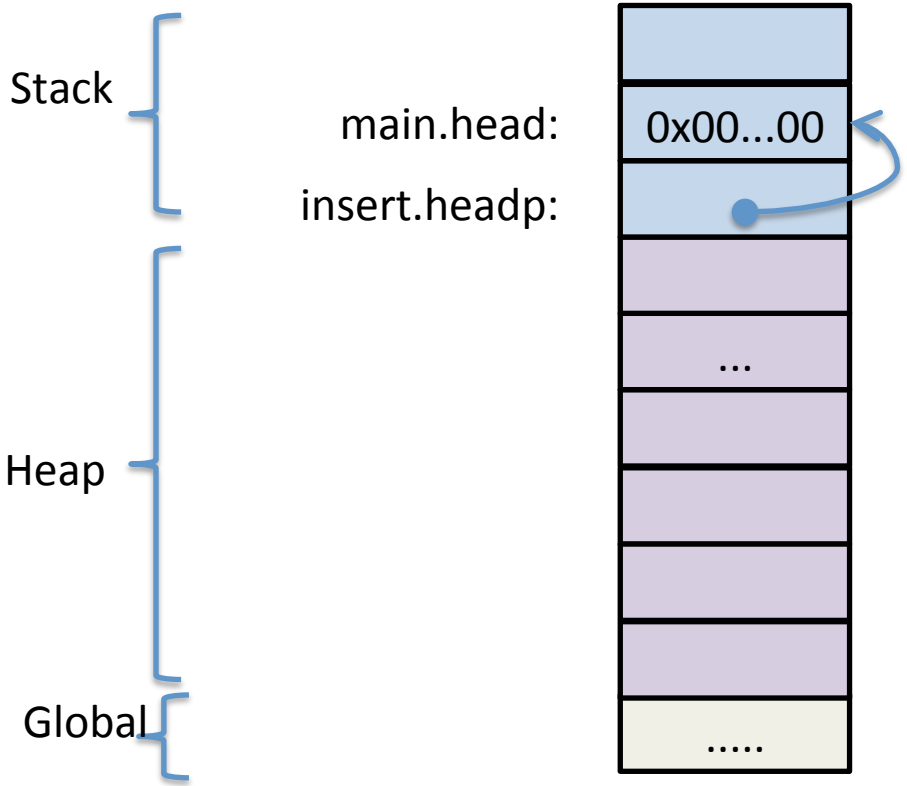


We will refine this simple view in later lectures

Linked list in C: insertion

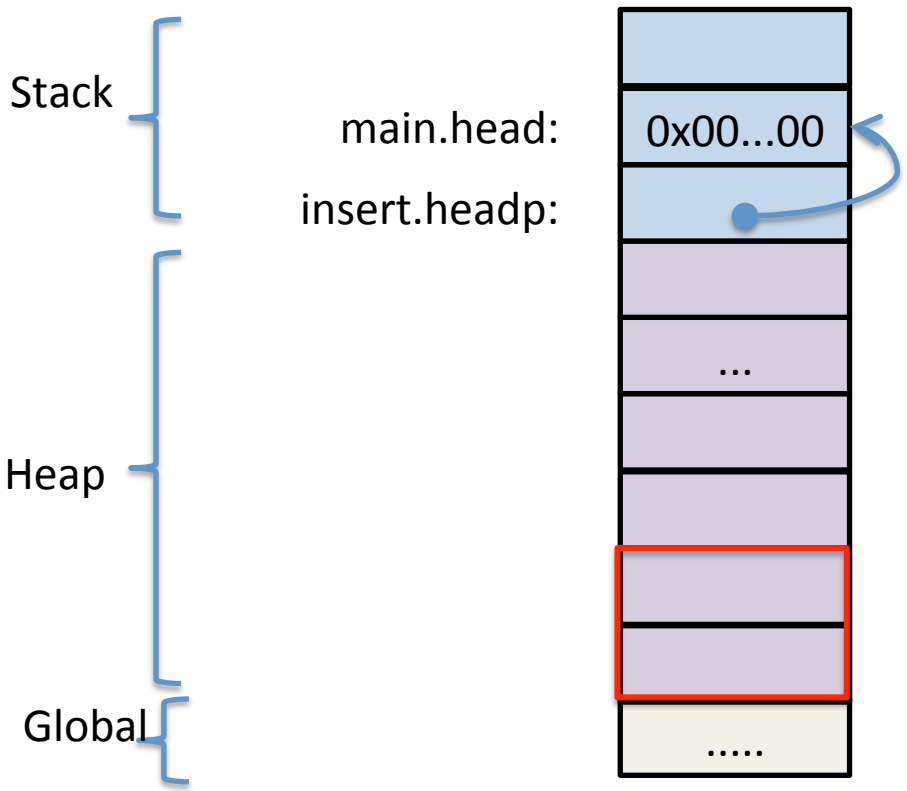
```
typedef struct {
     long val;
     struct node *next;
 }node;

// insert val into linked list to the head of the linked
// list and return the new head of the list in *head
void
insert(node **head, long val) {

}

int main() {
     node *head = NULL;
     for (long i = 10; i < 13; i++)
        insert(&head, i);
}
```

Inserting into a linked list



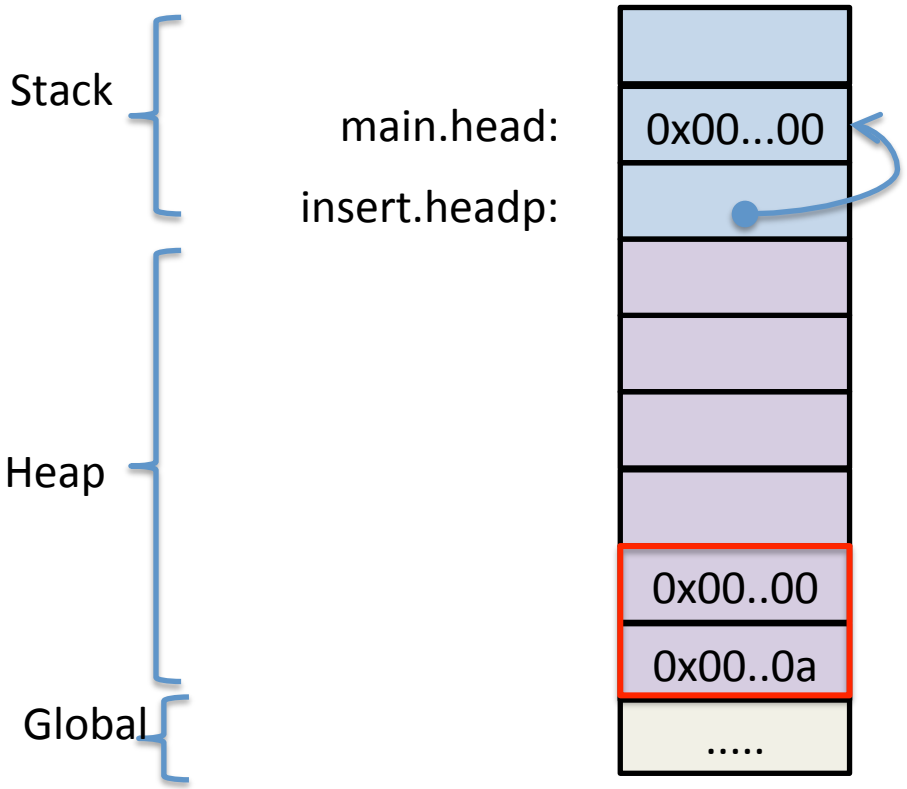

```
void insert(node **headp, long val)
{



}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

1st insert call



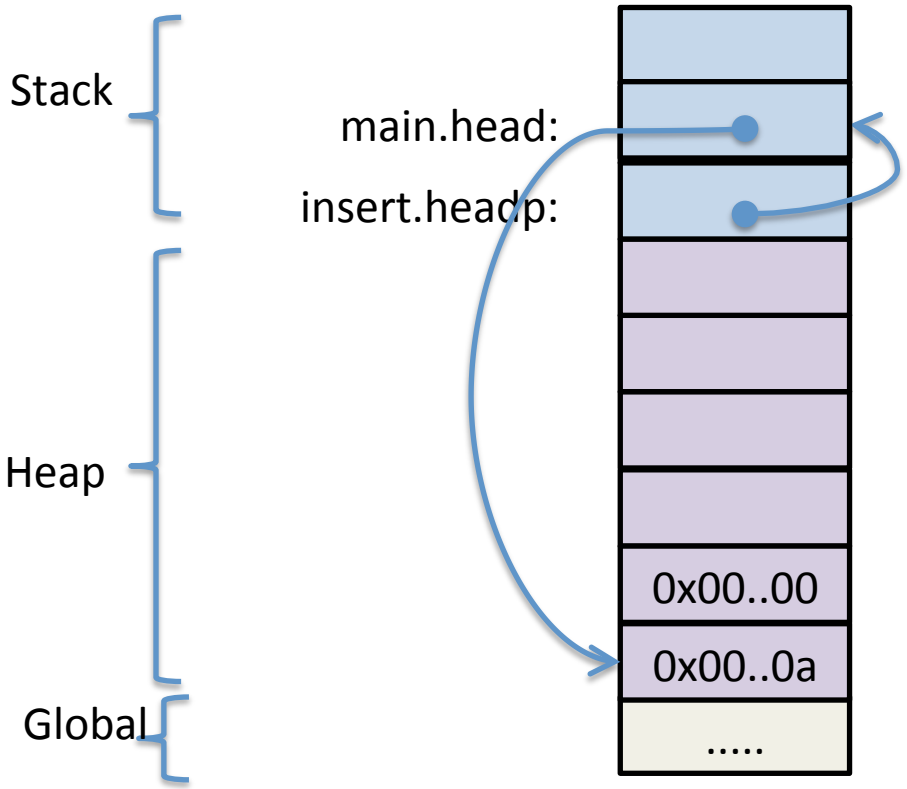

```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));




}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

1st insert call



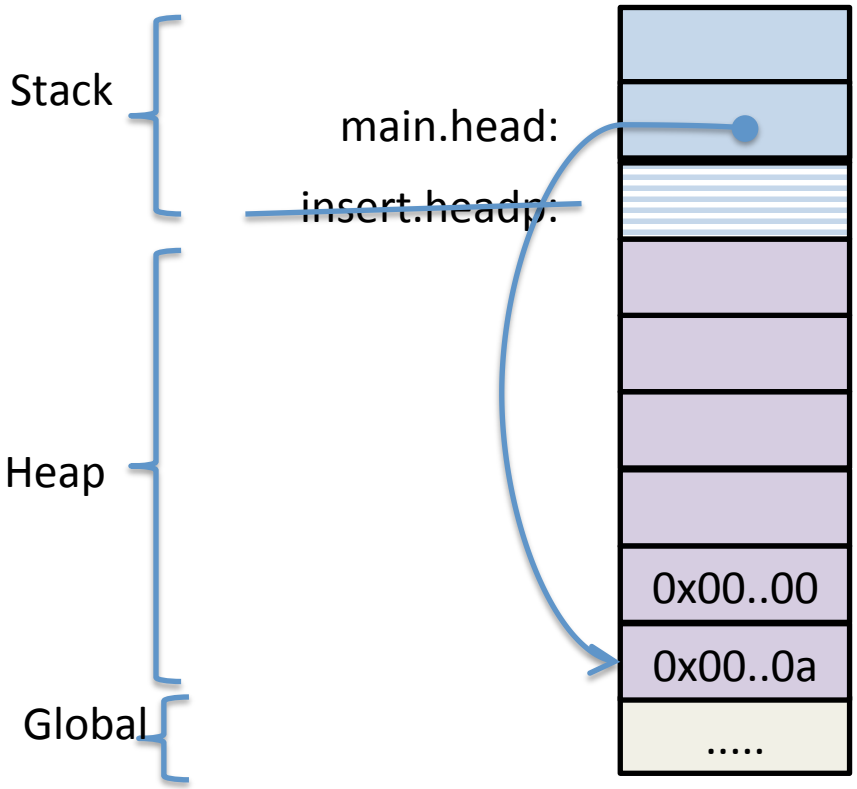

```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;

}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

1st insert call



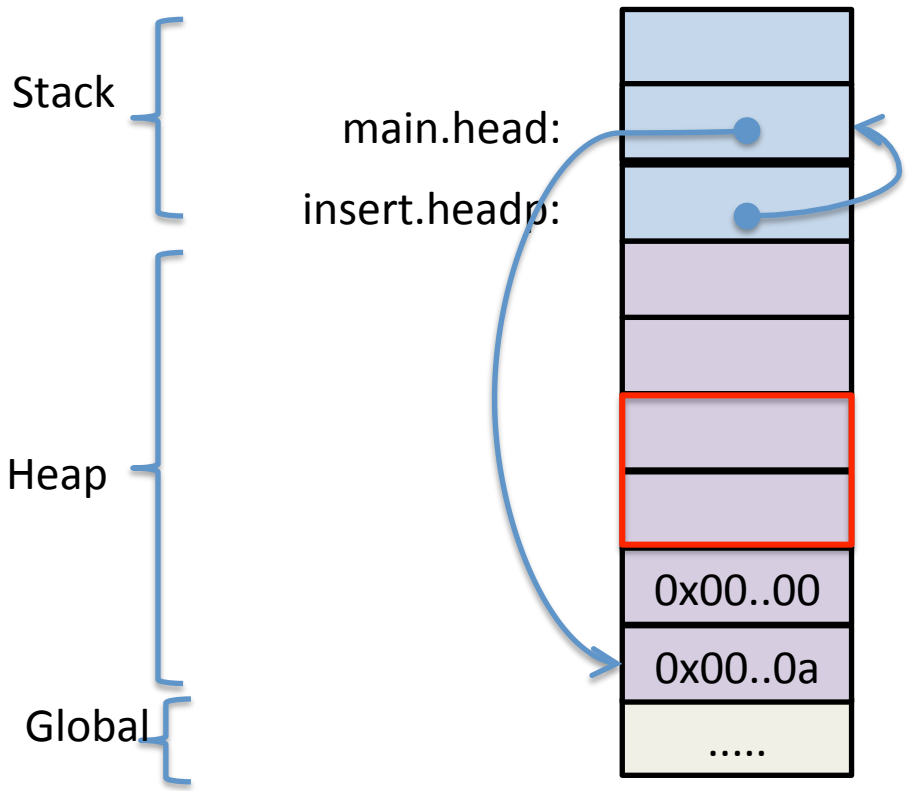

```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

after 1st insert call



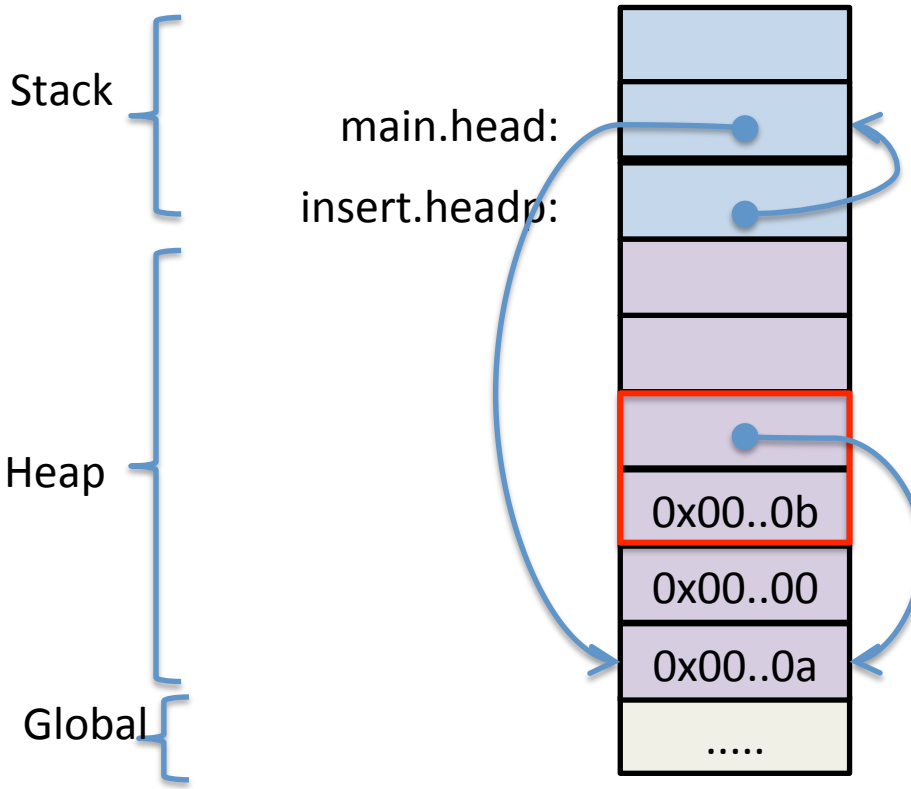

```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

2nd insert call



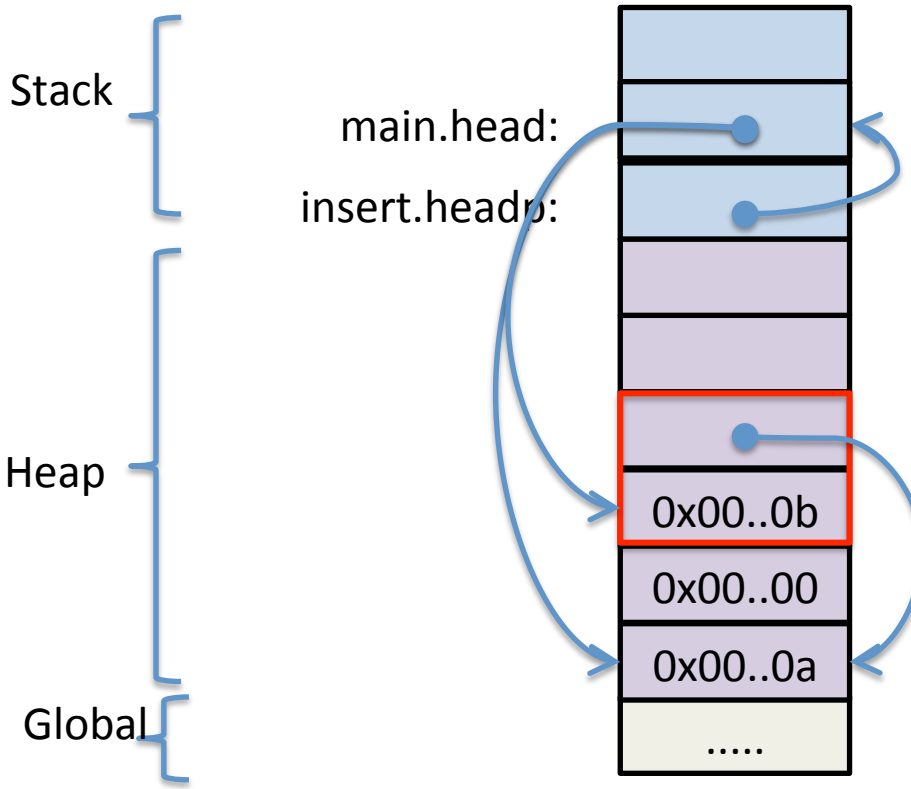

```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

2nd insert call



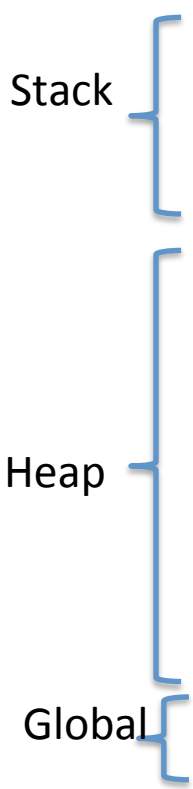

```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

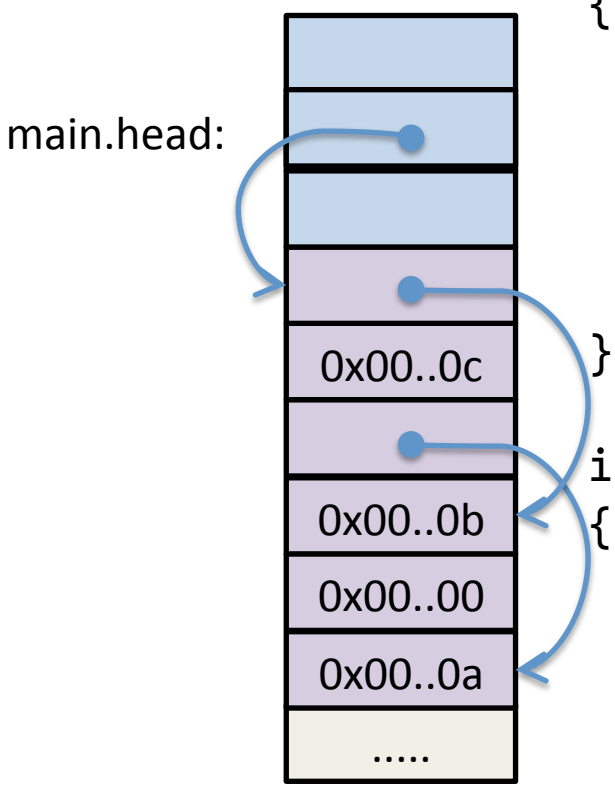
2nd insert call




```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

after 3rd call






```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```