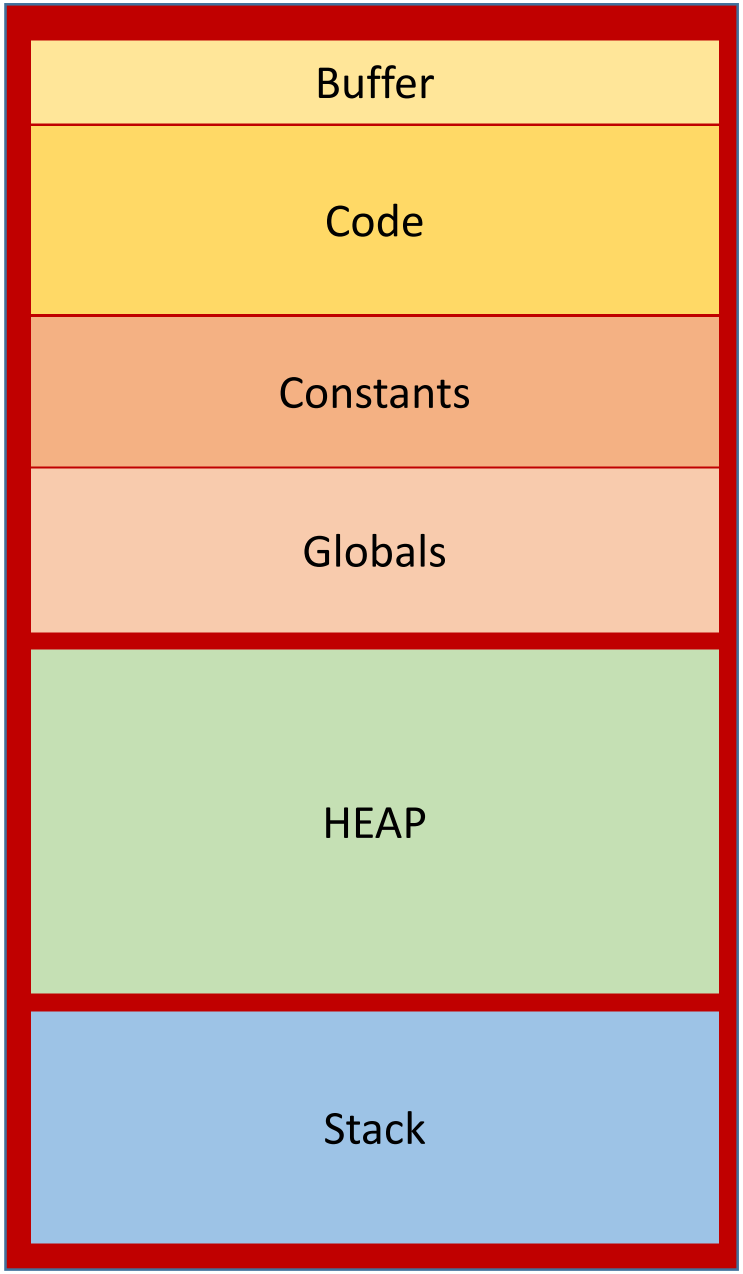
C memory model

Lecture 03.02

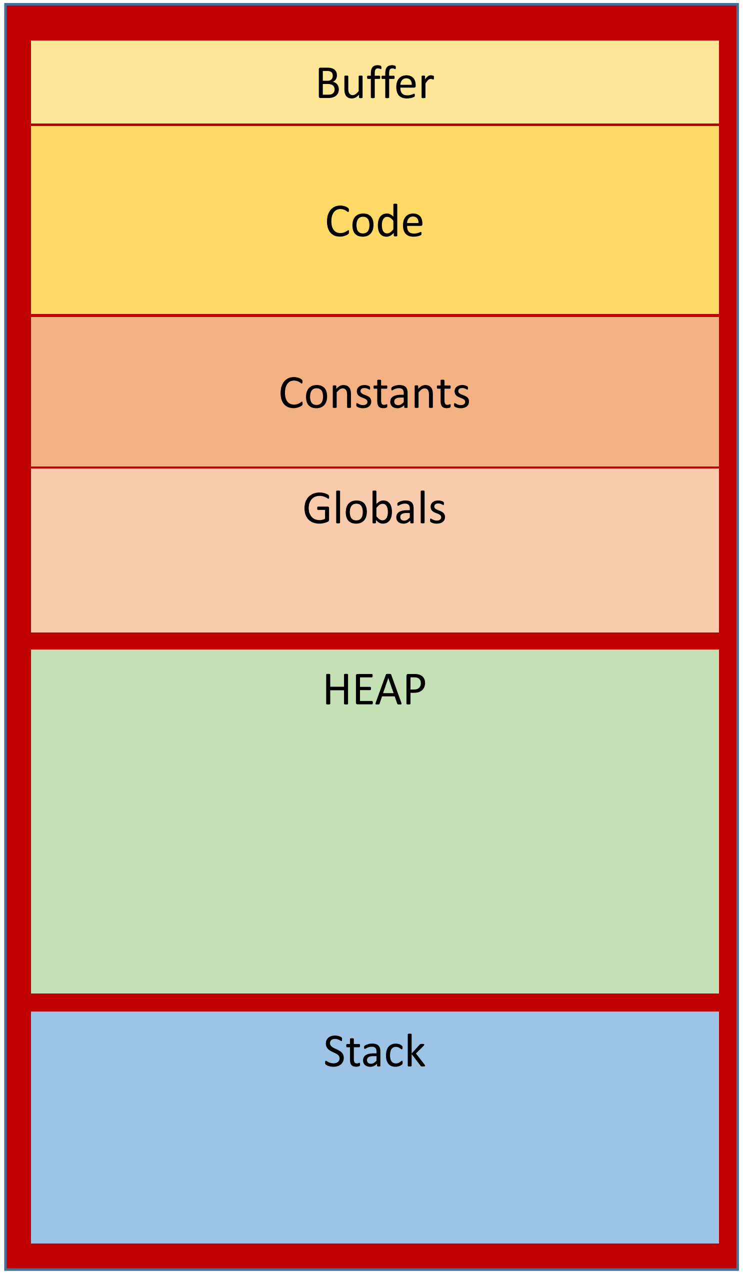
Outline

- Memory of a single process
- Globals and stack
- Heap for dynamic allocation



A. Fun

```
int fun (char a, char b) {
        a++;
        b++;
        return b;
}
char a='a'; //value 97
char b='b';
int main () {
        char c = (char) fun (a, b);
        printf ("%c %c %c\n", a, b, c);
        //what is printed by the way?
}
```

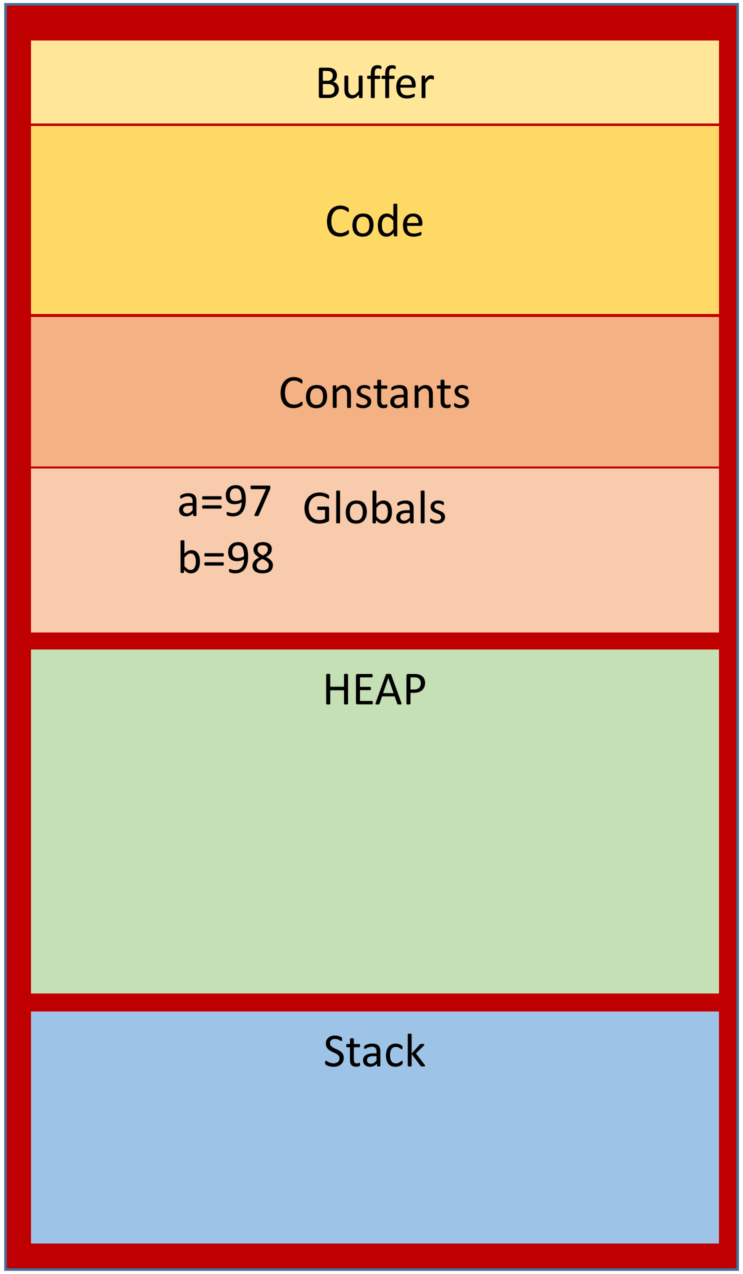


A. Fun

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

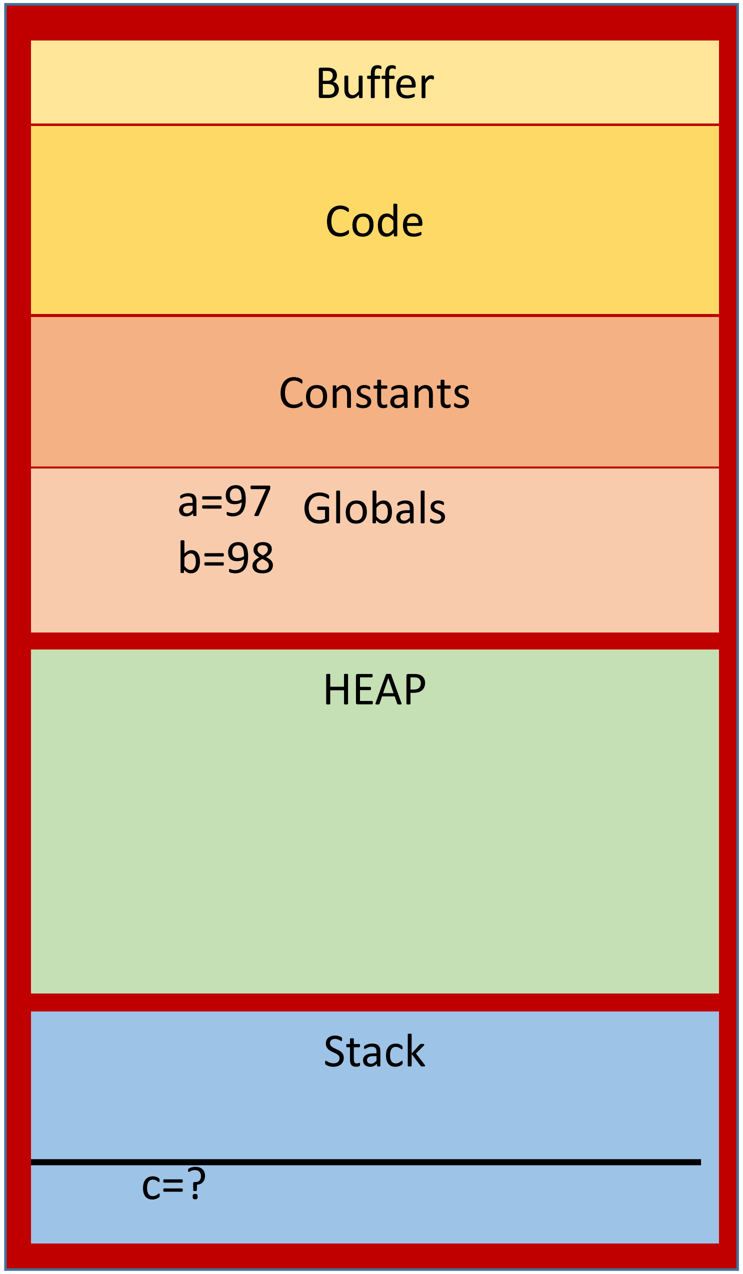
- char a='a'; //value 97
- char b='b';

```
int main () {
        char c = (char) fun (a, b);
        printf ("%c %c %c\n", a, b, c);
        //what is printed by the way?
}
```



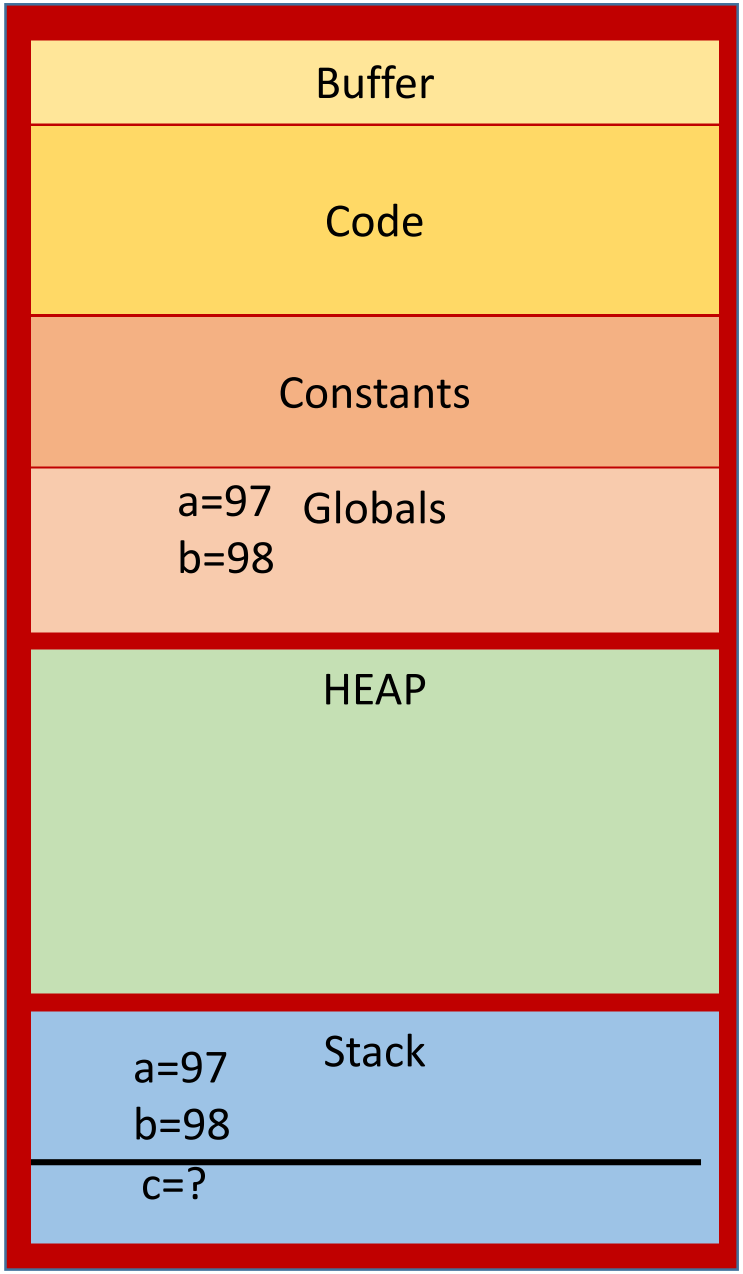
A. Fun

```
int fun (char a, char b) {
        a++;
        b++;
        return b;
}
char a='a'; //value 97
char b='b';
int main () {
        char c = (char) fun (a, b);
        printf ("%c %c %c\n", a, b, c);
        //what is printed by the way?
}
```



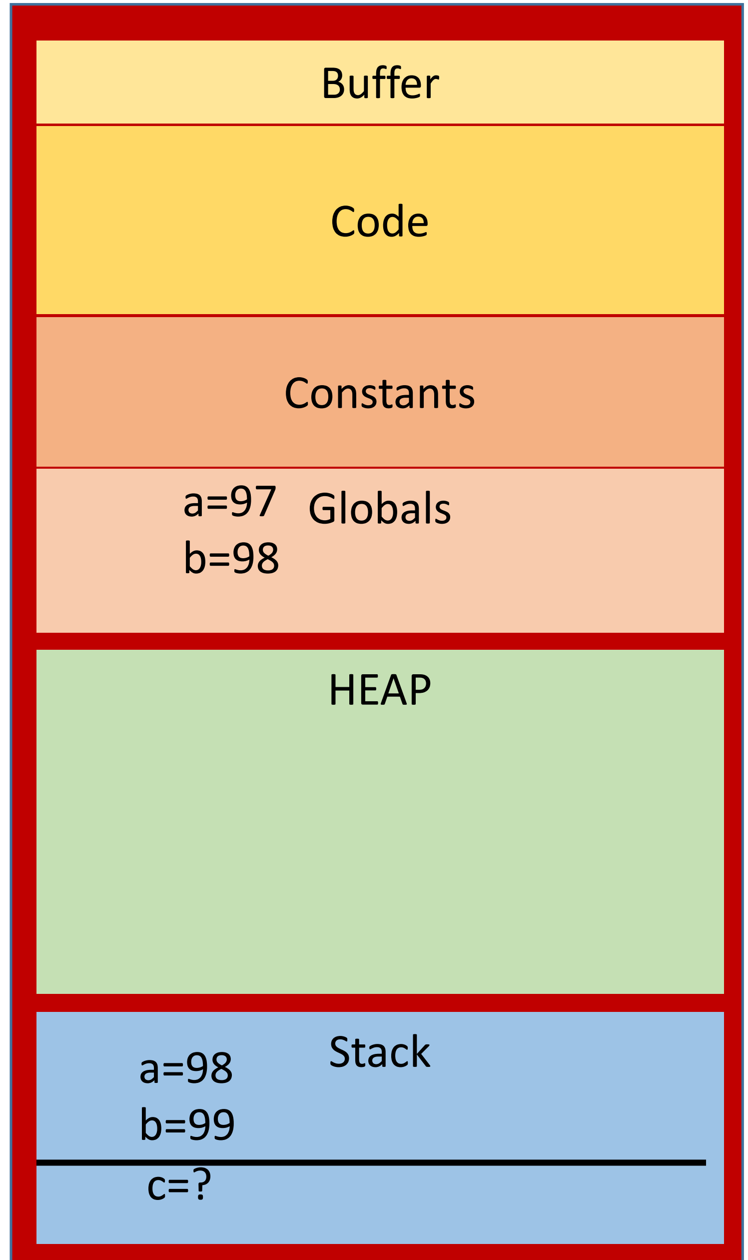
A. Fun

```
int fun (char a, char b) {
        a++;
        b++;
        return b;
}
char a='a'; //value 97
char b='b';
int main () {
        char c = (char) fun (a, b);
        printf ("%c %c %c\n", a, b, c);
        //what is printed by the way?
}
```



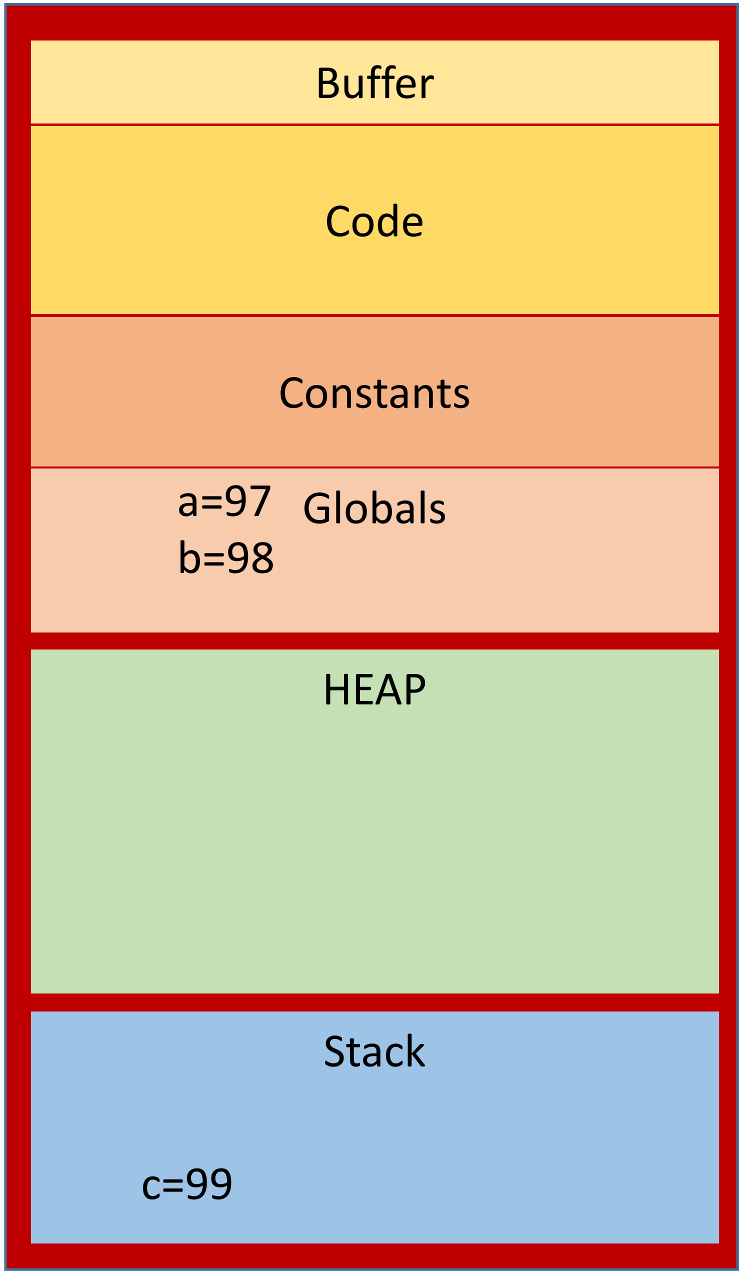
A. Fun

```
int fun (char a, char b) {
        a++;
        b++;
        return b;
}
char a='a'; //value 97
char b='b';
int main () {
        char c = (char) fun (a, b);
        printf ("%c %c %c\n", a, b, c);
        //what is printed by the way?
}
```



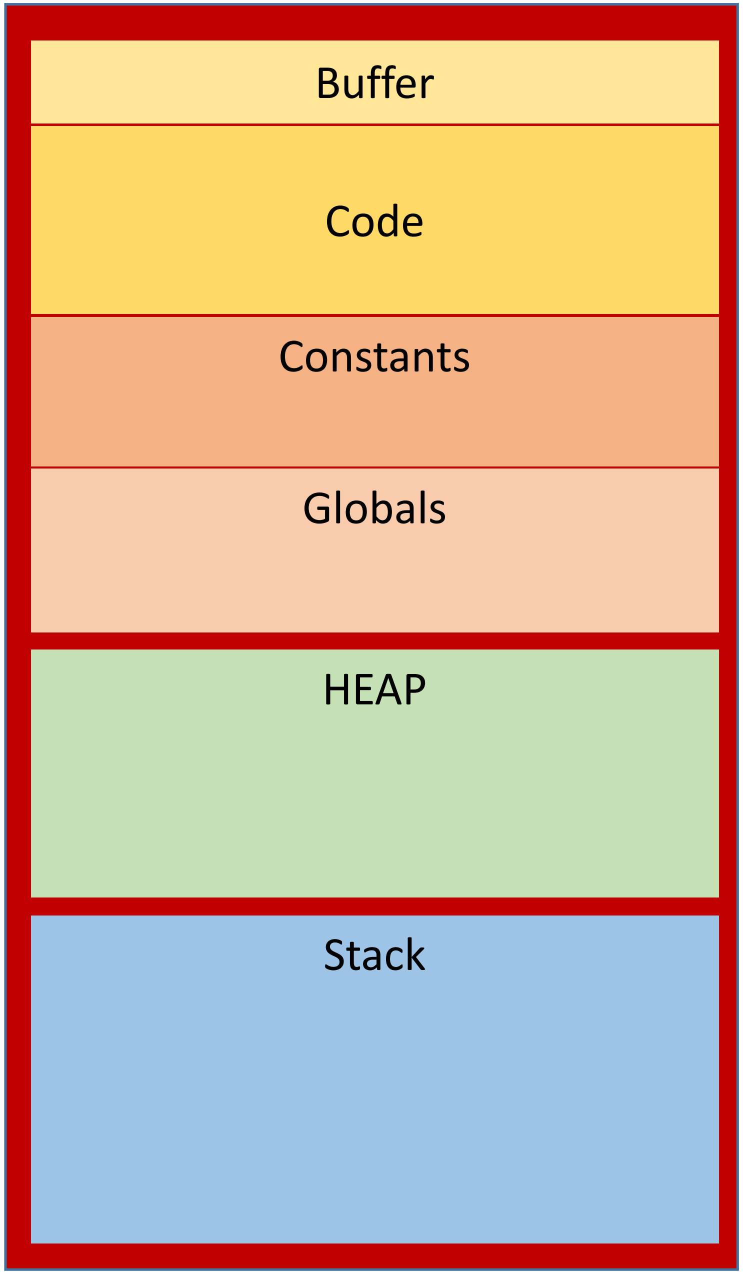
A. Fun

```
int fun (char a, char b) {
        a++;
        b++;
        return b;
}
char a='a'; //value 97
char b='b';
int main () {
        char c = (char) fun (a, b);
        printf ("%c %c %c\n", a, b, c);
        //what is printed by the way?
}
```



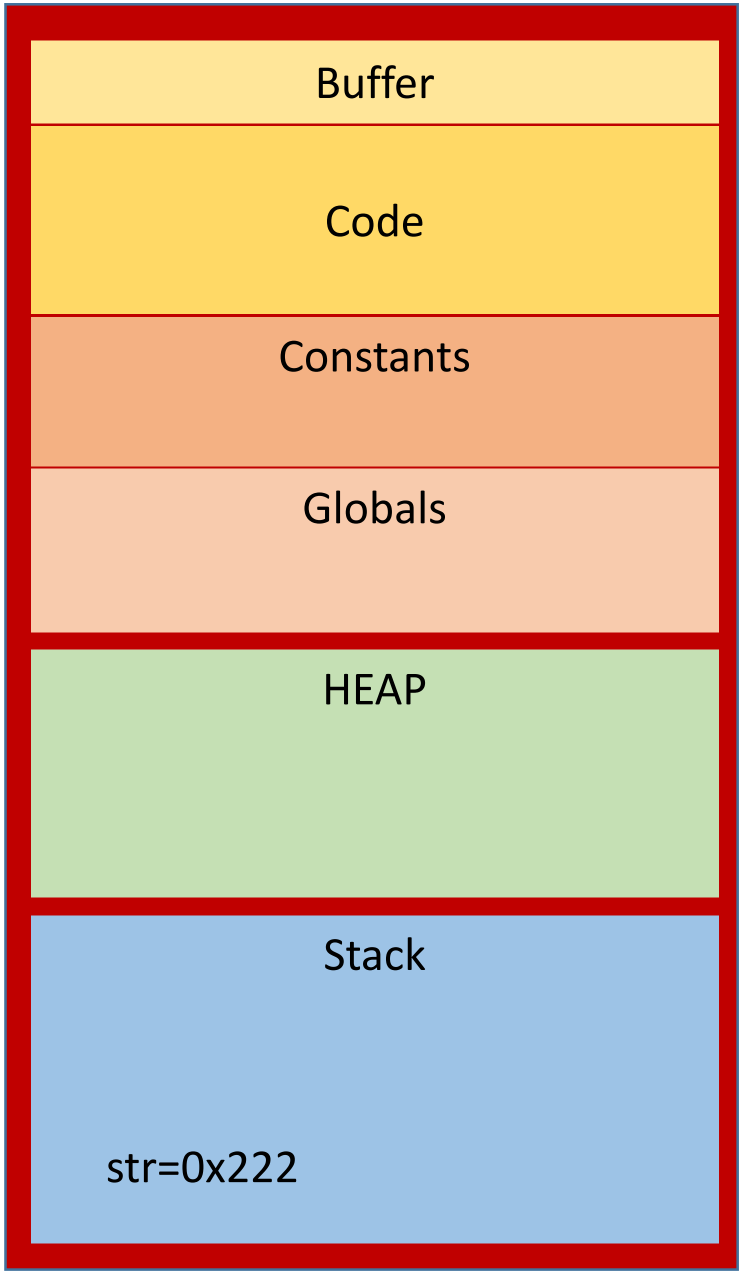
B. More fun

```
int * more_fun (char *a) {
        a = malloc (5);
        *a = 'a';
        *(a+1) = 'b';
        *(a+2) = 0;
        int result[] = {1,2};
        return result;
}

int main () {
        char *str;
        int *ip = more_fun (str);
        printf ("%d %s\n", *ip, str);
}
```



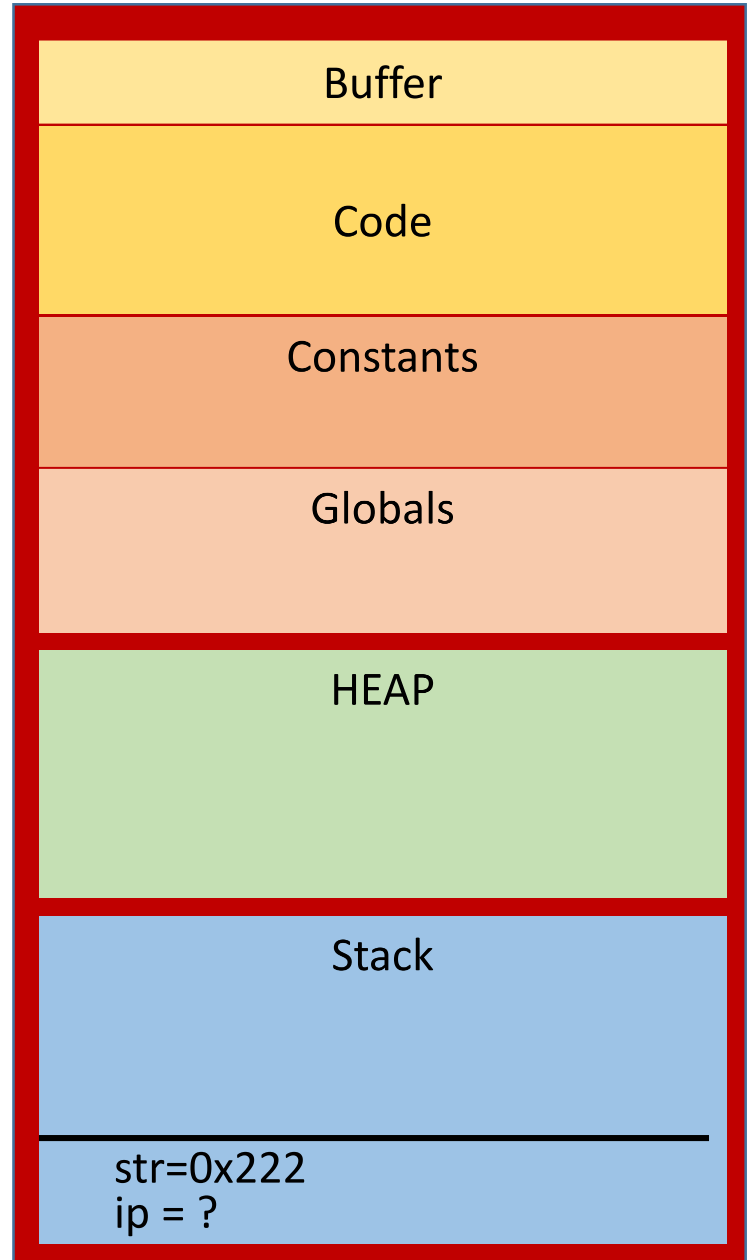
B. More fun

```
int * more_fun (char *a) {
        a = malloc (5);
        *a = 'a';
        *(a+1) = 'b';
        *(a+2) = 0;
        int result[] = {1,2};
        return result;
}

int main () {
▶       char *str;
        int *ip = more_fun (str);
        printf ("%d %s\n", *ip, str);
}
```



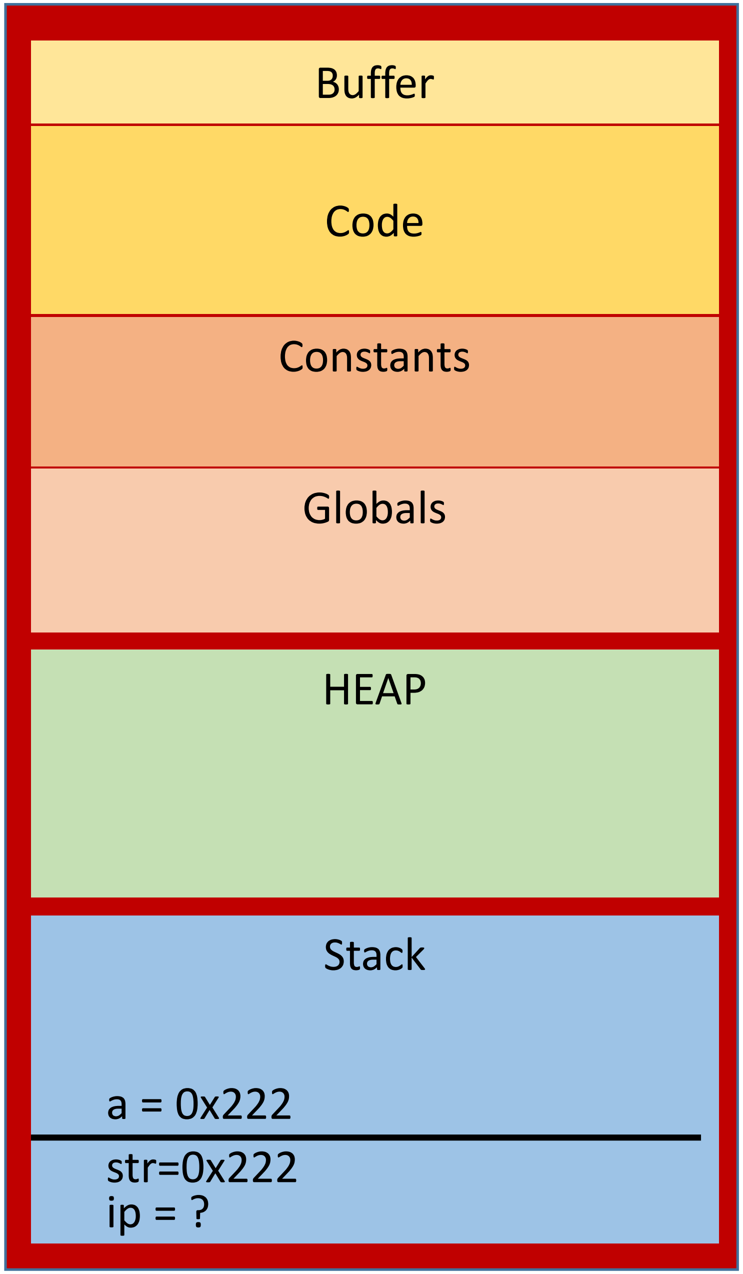
B. More fun

```
int * more_fun (char *a) {
        a = malloc (5);
        *a = ‘a’;
        *(a+1) = ‘b’;
        *(a+2) = 0;
        int result[] = {1,2};
        return result;
}

int main () {
        char *str;
        int *ip = more_fun (str);
        printf (“%d %s\n”, *ip, str);
}
```



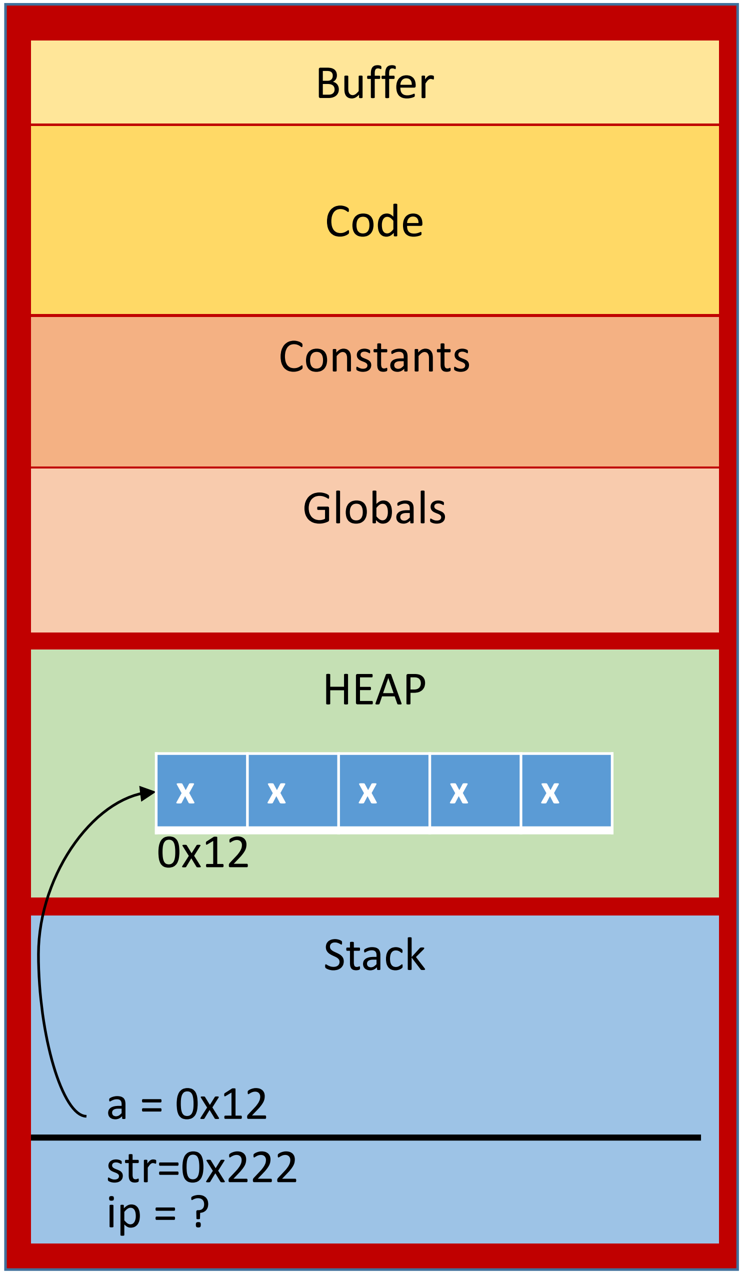
B. More fun

```
int * more_fun (char *a) {
        a = malloc (5);
        *a = ‘a’;
        *(a+1) = ‘b’;
        *(a+2) = 0;
        int result[] = {1,2};
        return result;
}

int main () {
        char *str;
        int *ip = more_fun (str);
        printf (“%d %s\n”, *ip, str);
}
```



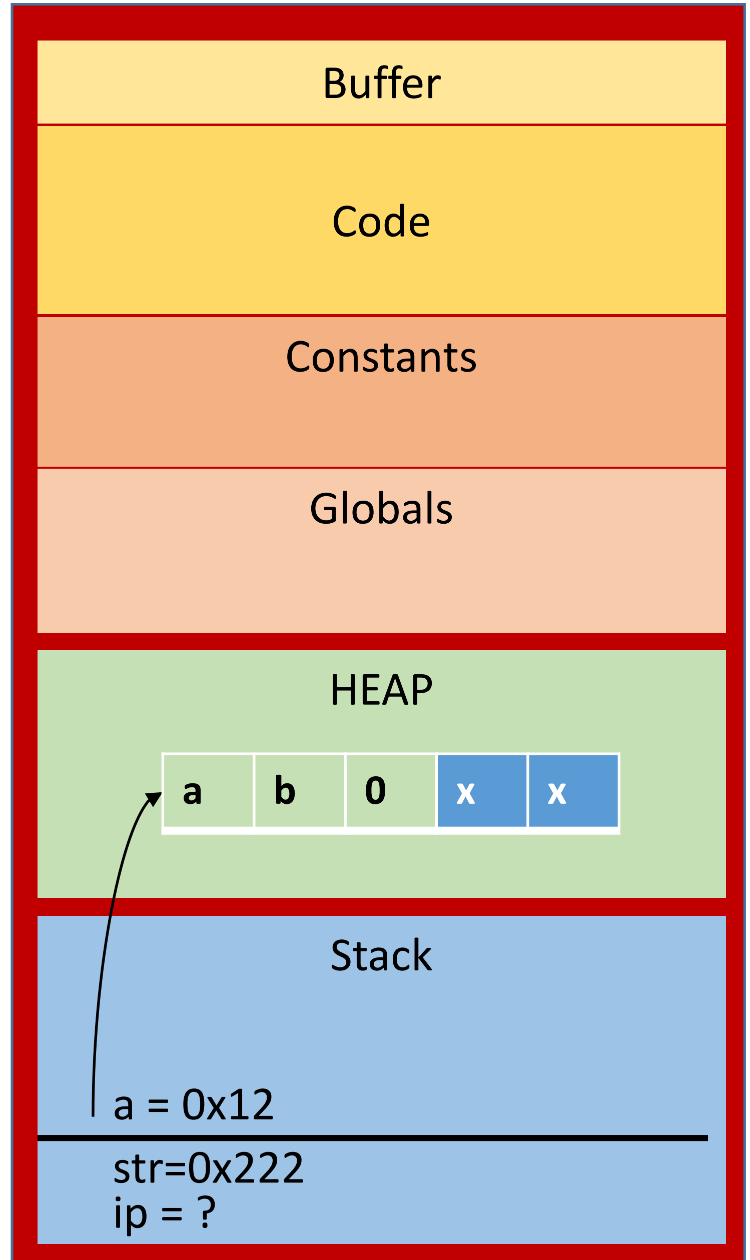
B. More fun

```
int * more_fun (char *a) {
        a = malloc (5);
        *a = 'a';
        *(a+1) = 'b';
        *(a+2) = 0;
        int result[] = {1,2};
        return result;
}

int main () {
        char *str;
        int *ip = more_fun (str);
        printf ("%d %s\n", *ip, str);
}
```



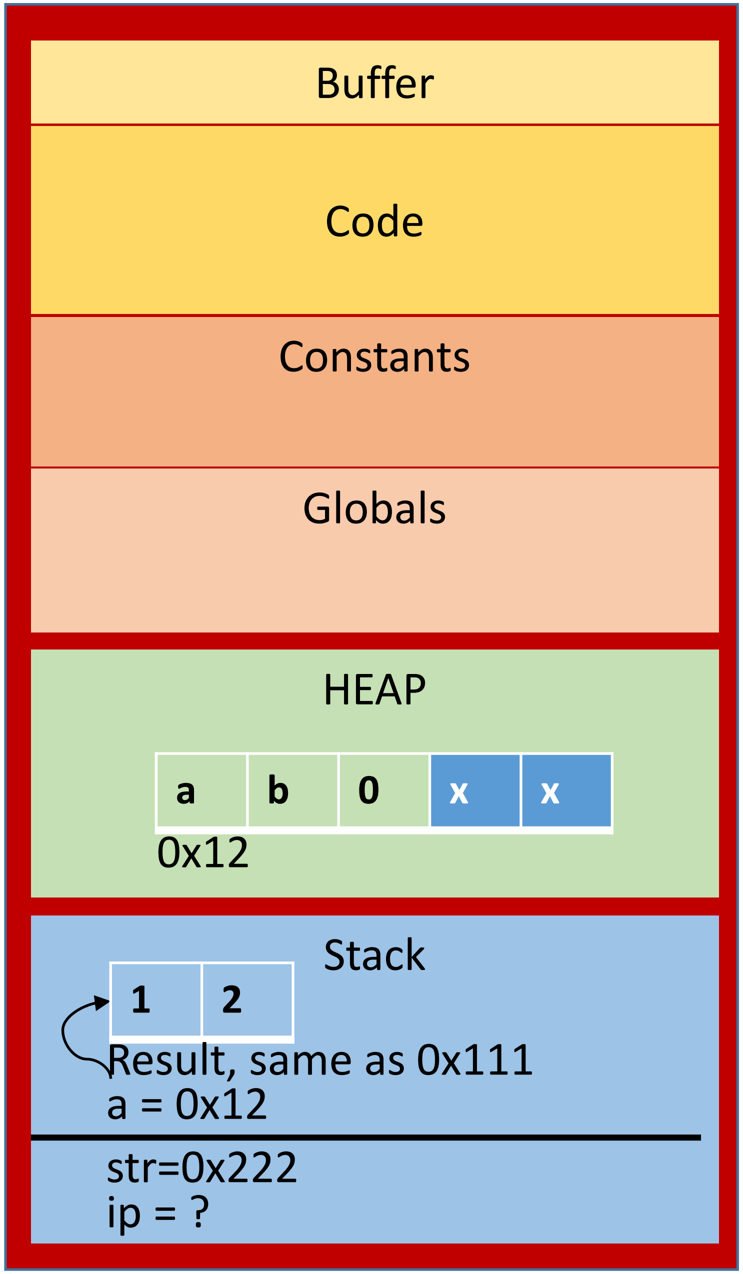
B. More fun

```
int * more_fun (char *a) {
        a = malloc (5);
        *a = 'a';
        *(a+1) = 'b';
        *(a+2) = 0;
        int result[] = {1,2};
        return result;
}

int main () {
        char *str;
        int *ip = more_fun (str);
        printf ("%d %s\n", *ip, str);
}
```



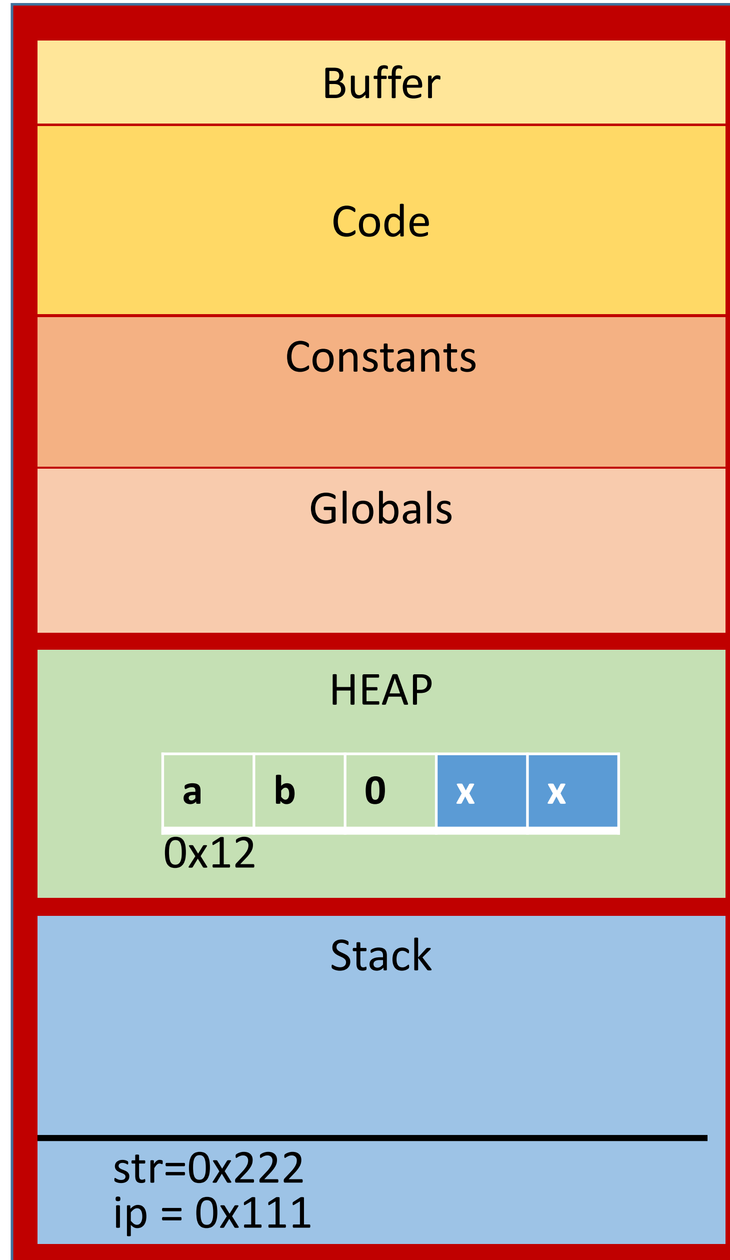
B. More fun

```
int * more_fun (char *a) {
        a = malloc (5);
        *a = 'a';
        *(a+1) = 'b';
        *(a+2) = 0;
        int result[] = {1,2};
        return result;
}

int main () {
        char *str;
        int *ip = more_fun (str);
        printf ("%d %s\n", *ip, str);
}
```

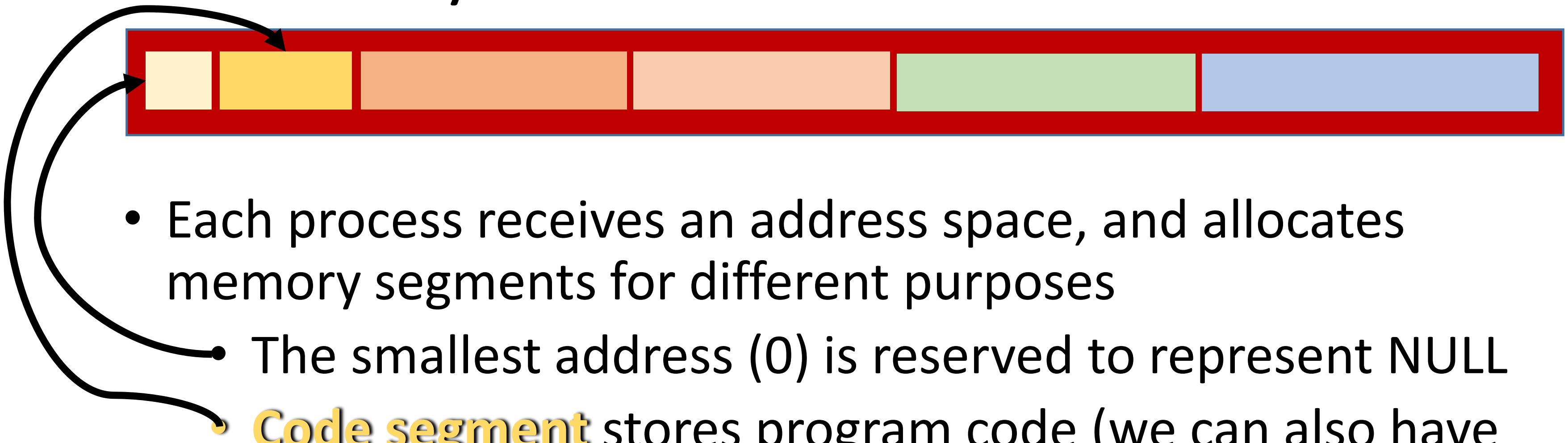


B. More fun

```
int * more_fun (char *a) {
        a = malloc (5);
        *a = ‘a’;
        *(a+1) = ‘b’;
        *(a+2) = 0;
        int result[] = {1,2};
        return result;
}

int main () {
        char *str;
        int *ip = more_fun (str);
        printf (“%d %s\n”, *ip, str);
}
```

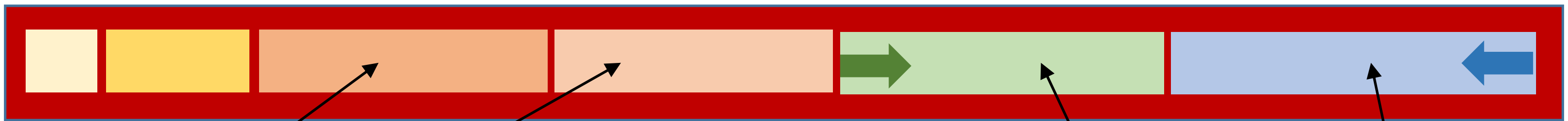


Memory memorizer



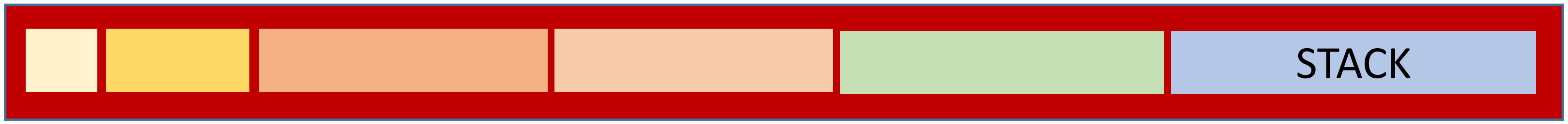
- Each process receives an address space, and allocates memory segments for different purposes
  - The smallest address (0) is reserved to represent NULL
  - **Code segment** stores program code (we can also have pointers to places in code – function pointers)
  - **Constants** stores all the constants. This memory is read-only
  - **Globals** stores global variables – variables visible to all functions
  - **Stack** stores variables of a currently executing function
  - **Heap** is reserved for dynamic memory allocation

Memory memorizer

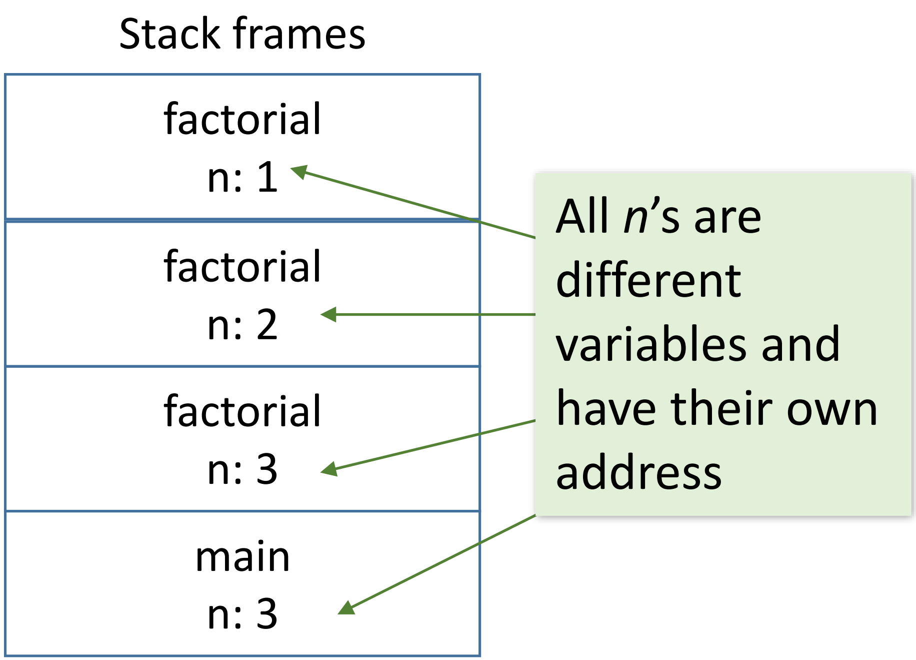


- **Constants** stores all the constants. This memory is read-only
- **Globals** stores global variables – variables visible to all functions
- **Stack** stores variables of a currently executing function
- **Heap** is reserved for dynamic memory allocation

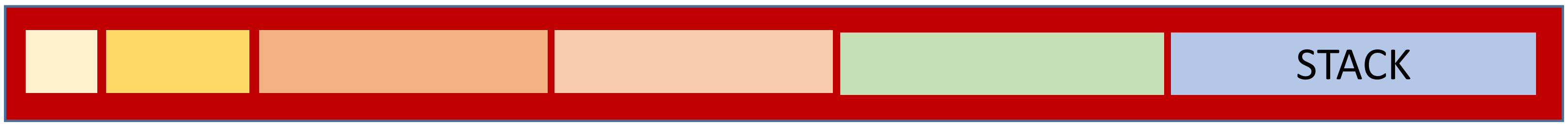
Stack variables (automatic variables, temporary variables)



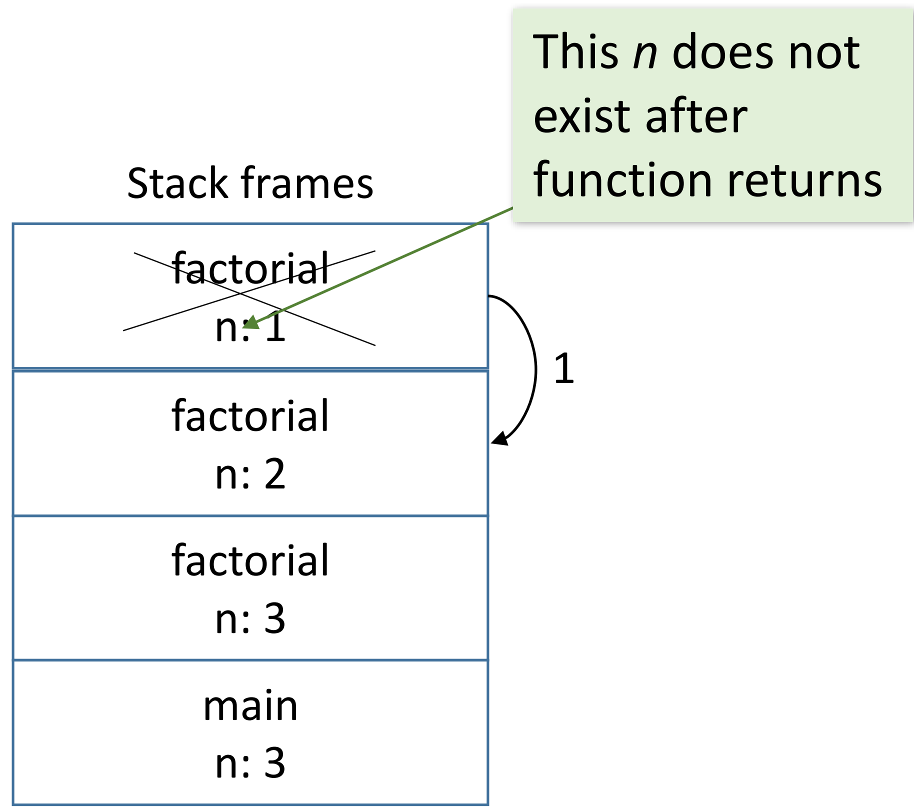

```
int factorial(int n) {
  if(n <= 1)  {
    return 1;
  }  else  {
    return n * factorial(n - 1);
  }
}
int main () {
      int n = 3;
      int f = factorial (n);
}
```



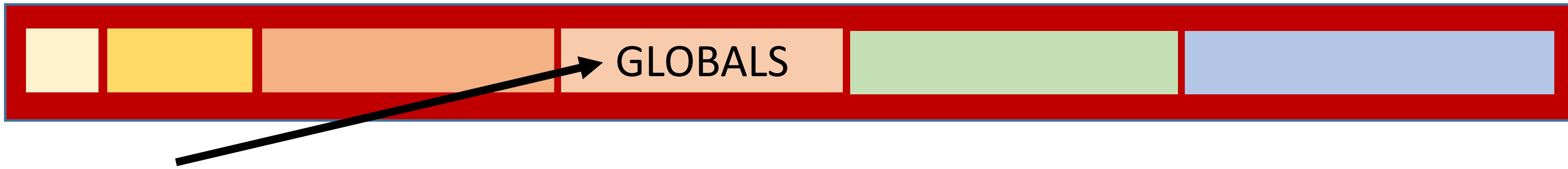
Stack variables, automatic variables, temporary variables




```
int factorial(int n) {
  if(n <= 1)   {
    return 1;
  }  else   {
    return n * factorial(n - 1);
  }
}
int main () {
      int n = 3;
      int f = factorial (n);
}
```



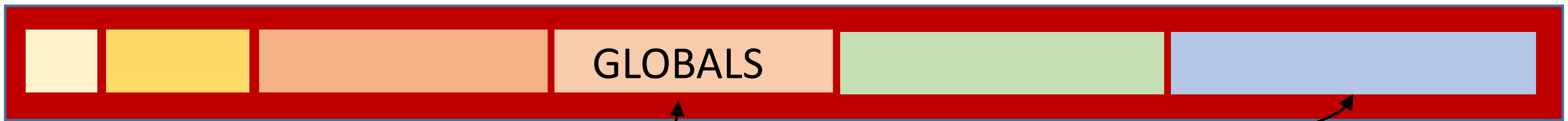
Global variables




```
int depth = 0;
int factorial(int n) {
  depth++;
   if(n <= 1)
      return 1;
   else
      return n * factorial(n - 1);
}
int main () {
        int n = 3;
        int f = factorial (n);
        printf (“%d!=%d recursion depth=%d\n”, n,f,depth);
```

variable *depth* exists in the same address space through the entire program

Static variables




```
void print_plus () {
    int a = 10;
    static int sa = 10;
    a += 5;
    sa += 5;
    printf("a = %d, sa = %d\n", a, sa);
}

int main() {
    int i;
    for (i = 0; i < 10; ++i)
        print_plus();
}
```

A *static* variable inside a function keeps its value between invocations, but unlike global variable is invisible to other functions

Again: array is not exactly a pointer

- An array name is a constant address, while a pointer is a variable:

```
int x[10], *px;
px = x; px++; /** valid **/
x = px; x++; /** invalid, cannot assign a new value **/
```

Array vs. pointer - allocation

- int x[10], *px;

  px = x; px++; /** valid **/

  x = px; x++; /** invalid, cannot assign a new value **/

- Defining the pointer only allocates *memory space for the address*, not for any array elements, and the pointer does not point to anything meaningful.
- Defining an array (x[10]) gives a pointer to a specific place in memory and allocates enough space to hold the array elements.

Stack storage

- Most of the memory we used so far has been in the stack.
- The stack is the area of memory that’s used for local variables.
- Each piece of data is stored in a variable, and each variable disappears as soon as you leave its function.

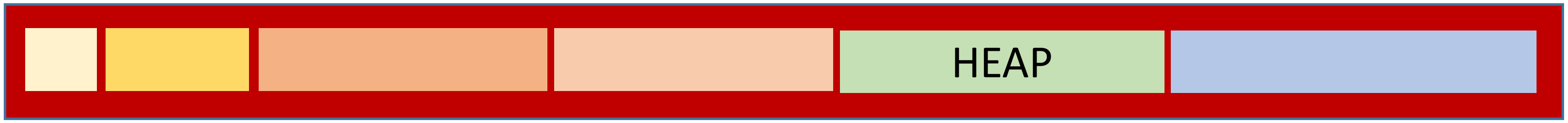
Example: returning an array

- You can't say:

```
int *f() {
    int a[10];
    ...
    return(a);
}
```

- because that 'a' array is deallocated as the function returns.

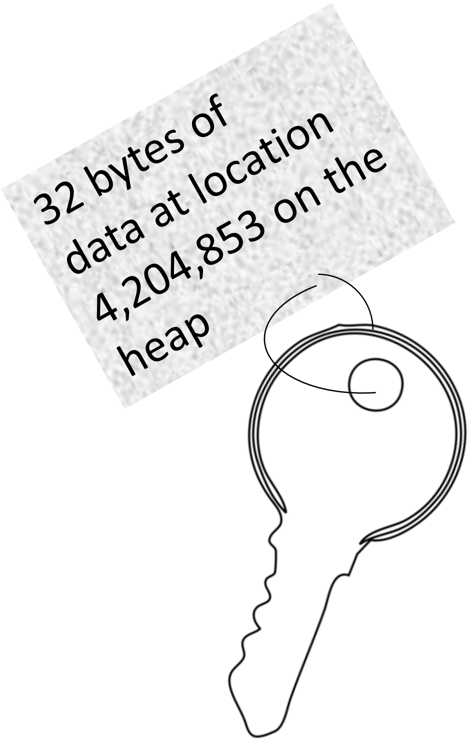
Dynamic storage



- We not always know how much memory we need in advance
- We need to be able to demand and get the memory dynamically, at the point when we need it
- Dynamic memory is allocated on the *heap*

First, get your memory with *malloc*()

- Ask for a large storage locker for the data: *malloc()*
- Tell the *malloc()* function exactly how many bytes of memory you need, and it asks the operating system to set that much memory aside in the heap
- The *malloc()* function then returns a pointer to the new heap space, a bit like getting a key to the locker

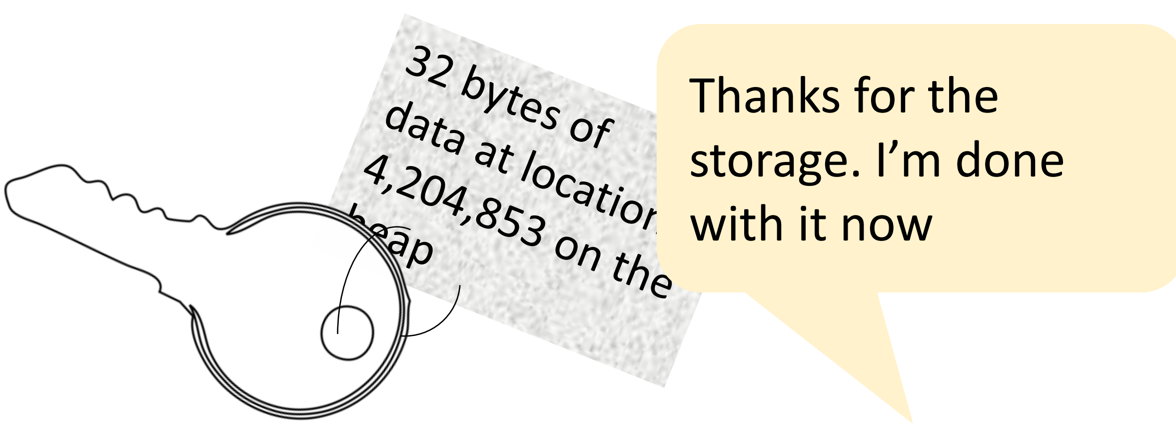


Give the memory back when you're done

- With the stack, you didn't need to worry about returning memory; it all happens automatically: every time you leave a function, the local storage is freed
- The heap is different. Once you've asked for space on the heap, it will never be available for anything else until you explicitly free it.
- There's only so much heap memory available, so if your code keeps asking for more and more heap space, your program will start to develop **memory leaks**

Free memory by calling the *free()* function

- The ***malloc***() function allocates space and gives you a pointer to it
- You'll need to use this pointer to access the data and then, when you're finished with the storage, you need to release the memory using the ***free***() function.
- It's a bit like handing your locker key back to the attendant so that the locker can be reused.



free for each *malloc*

- Every time some part of your code requests heap storage with the *malloc*() function, there should be some other part of your code that hands the storage back with the *free*() function.
- When your program stops running, all of its heap storage will be released automatically, but it's always good practice to explicitly call *free*() on every piece of dynamic memory you've created.

Array as a return value

- Return a pointer to malloc'd memory if you want to return an array:

```
int *f() {
    int *a;
    if ((a = malloc(10 * sizeof(int))) == NULL)
            ...
    ...
    return(a);
}
```

```
int *f() {
    int a[10];
    ...
    return(a);
}
```

- Because the malloc'd memory persists until free() is called on the pointer - its existence is not tied to the duration of the execution of the function.

Summary: heap memory

- Heap memory provides greater control for the programmer — the blocks of memory can be requested in any size, and they remain allocated until they are deallocated explicitly.
- Heap memory can be passed back to the caller function since it is not deallocated on exit
- Heap memory is allocated at run time
- malloc() and free()