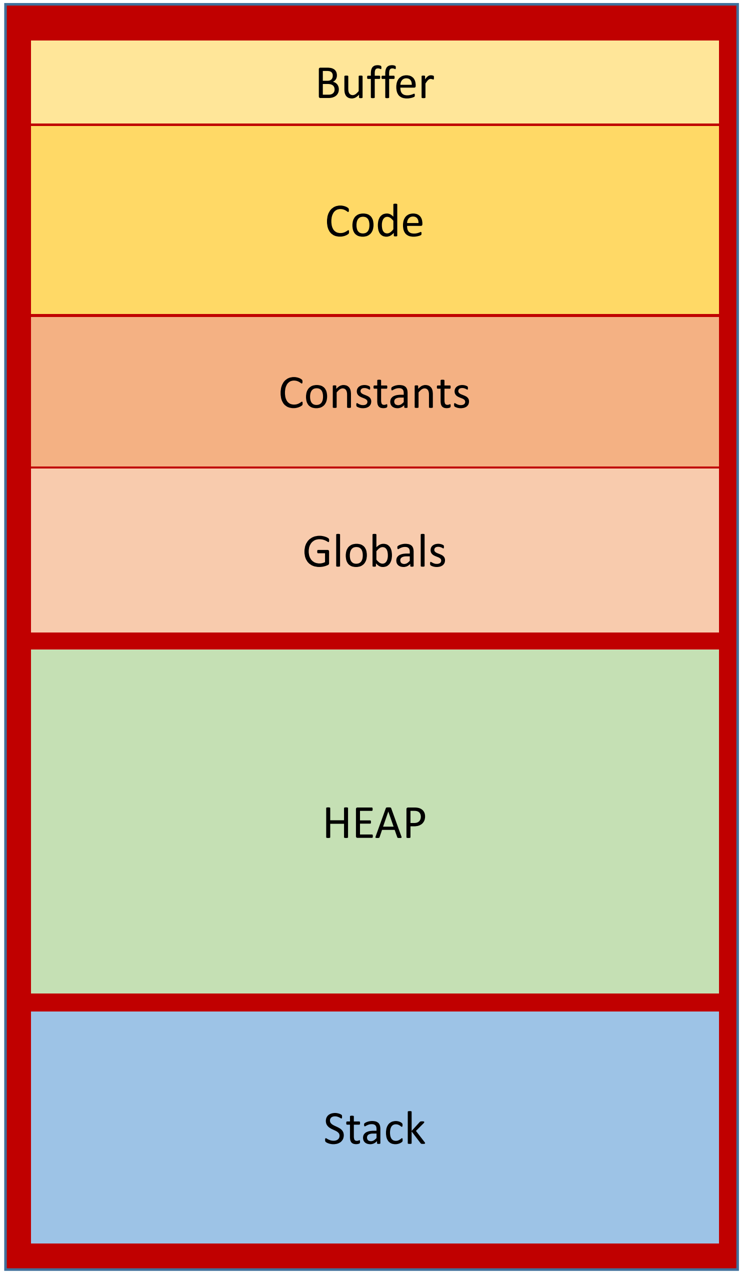
# C memory model

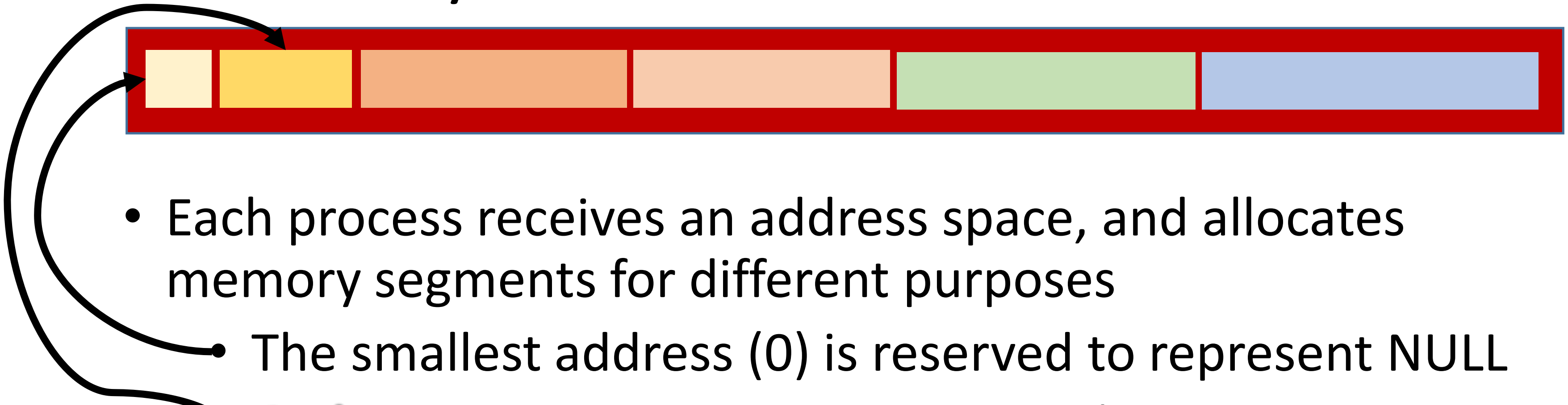
Lecture 03.02

# Outline

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

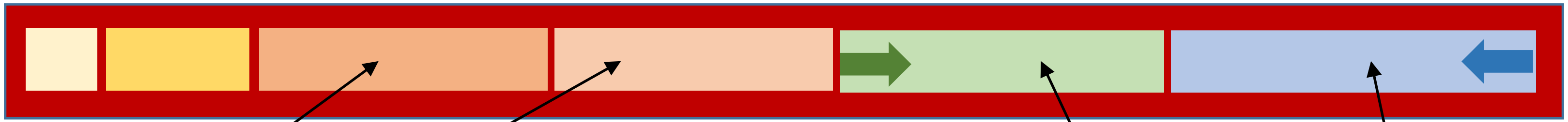


# Memory memorizer



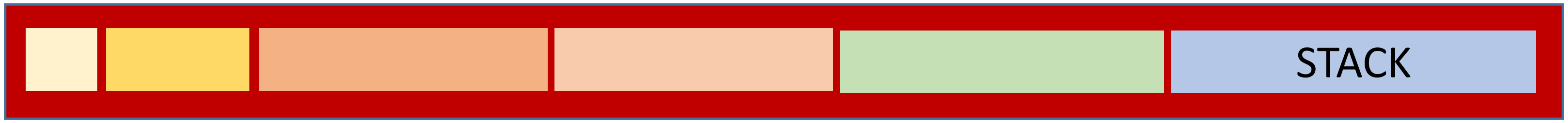
- Each process receives an address space, and allocates memory segments for different purposes
  - The smallest address (0) is reserved to represent NULL
  - **Code segment** stores program code (we can also have pointers to places in code – function pointers)
  - **Constants** stores all the constants. 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

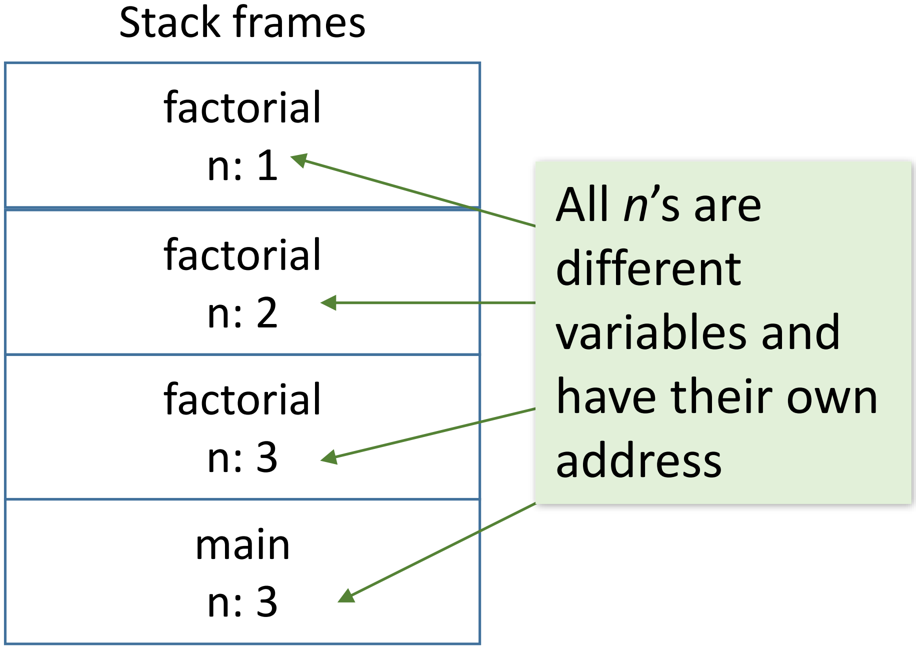


- **Constants** stores all the constants. This memory is read-only
- **Globals** stores global variables – variables visible to all functions
- **Stack** stores variables of a currently executing function
- **Heap** is reserved for dynamic memory allocation

# 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 frames

factorial
n: 1

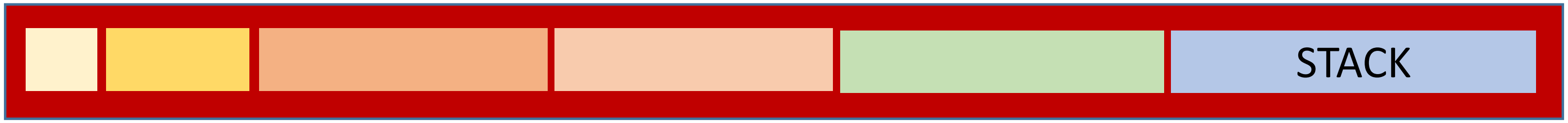
factorial
n: 2

factorial
n: 3

main
n: 3

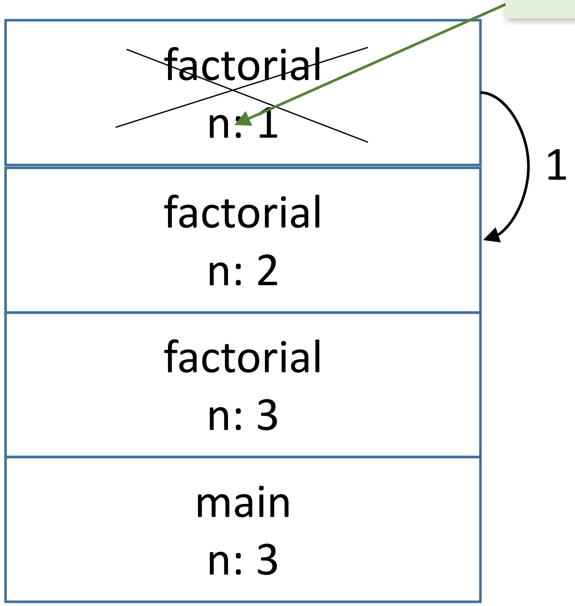
All *n*’s are different variables and have their own address

# Stack variables, automatic variables, temporary variables



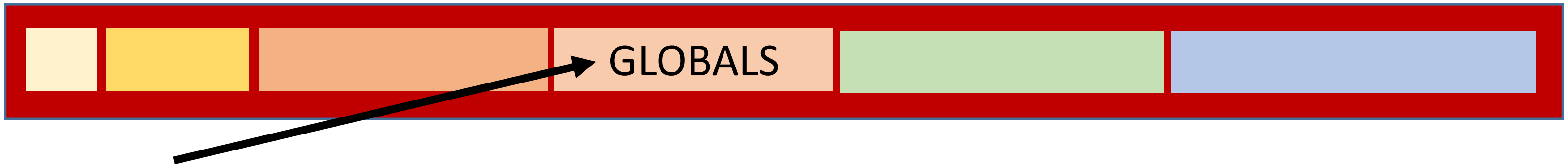
```
int factorial(int n) {
  if(n <= 1)   {
    return 1;
  }  else   {
    return n * factorial(n - 1);
  }
}
int main () {
      int n = 3;
      int f = factorial (n);
}
```

Stack frames



This *n* does not exist after function returns

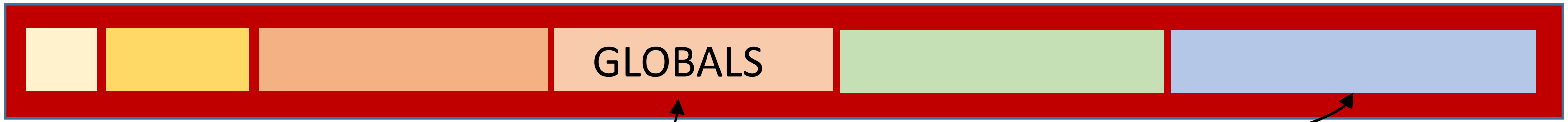
# Global variables



```
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

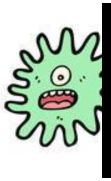
# Three-card trick

```
#include <stdio.h>
int main() {
    char *cards = "JQK";
    char a_card = cards[2];
    cards[2] = cards[1];
    cards[1] = cards[0];
    cards[0] = cards[2];
    cards[2] = cards[1];
    cards[1] = a_card;
    puts(cards);
    return 0;
}
```

Where is the Queen?

What is printed?

# Compile and run: Linux



```
gcc -o trick trick.c && ./trick
bus error
```

- On different machines and operating systems:

```
trick.exe has stopped working
```

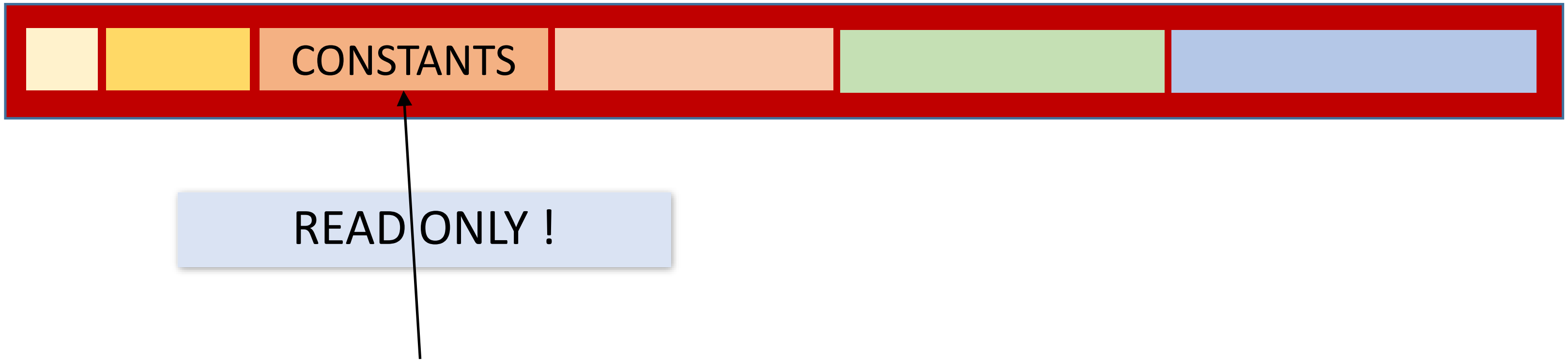
```
segmentation error
```

```
segmentation fault
```

# What do you think the problem is?

A. The string can’t be updated
B. We’re swapping characters outside the string
C. The string isn’t in memory
D. Something else

# String literals live in a different place: constants

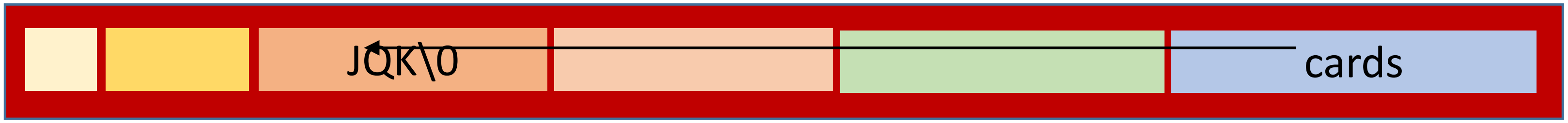


READ ONLY !

```
char *cards = "JQK";
```

- We cannot update string “JQK” through pointer *cards*

# String literals cannot be updated



- When the computer loads the program into memory, it puts all of the constant values—like the string literal “JQK”—into the constant memory block. This section of memory is **read only**.
- The program creates the cards pointer variable on the stack. The cards variable will contain the address of the string literal “JQK.”
- When the program tries to change the contents of the string pointed to by the cards variable, it can’t: the string is read-only.

# Why compiler did not warn us?

- Because we declared the *cards* as a simple char *, the compiler didn’t know that the variable would always be pointing at a string literal.
- To avoid this problem never write code that sets a simple char pointer to a string literal value like:

```
char *s = "Some string";
```

- There’s nothing wrong with setting a pointer to a string literal - until you try to modify a string literal. Instead, if you want to set a pointer to a literal, use the *const* keyword:

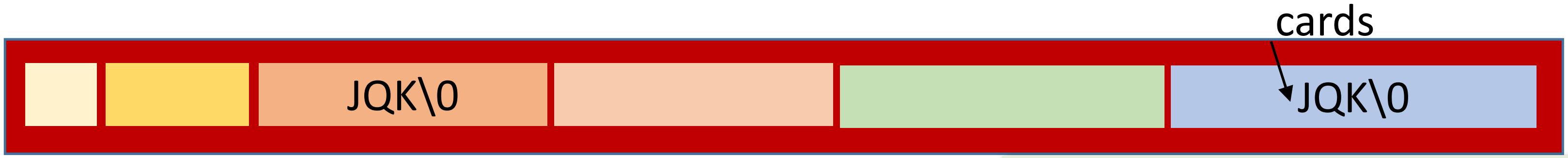
```
const char *s = "some string";
```

- That way, if the compiler sees some code that tries to modify the string, it will give you a compile error:

```
s[0] = 'S';
```

```
trick.c:7: error: assignment of read-only location
```

# Fix: copy literal into char array



```
char cards[] = "JQK";
```

Make a copy of the string in a section of memory that can be amended

- Now cards is not a pointer. Cards is now an array, which lives on the stack. It is filled with copies of characters from the constant when the stack frame for main is loaded
- It's probably not too clear why this changes anything. All strings are arrays. But in the old code, *cards* was just a pointer.
- In the new code, it's an array. If you declare an array called *cards* and then set it to a string literal, the cards array will be a completely new **copy**. The variable isn't just pointing at the string literal. It's a brand-new array that contains a fresh copy of the string literal.

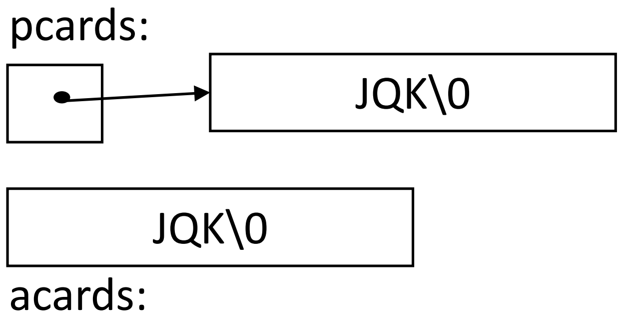
# Again: array is not exactly a pointer

- 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 **/
```

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

# Summary:
# char * vs. char []



- There is an important difference between these definitions:

char acards[] = “JQK"; /* an array */

char *pcards = “JQK"; /* a pointer */

- ***acards*** is an array, just big enough to hold the sequence of characters and ’\0’. Individual characters within the array may be changed but *acards* will always refer to the same storage.
- ***pcards*** is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.

# 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.

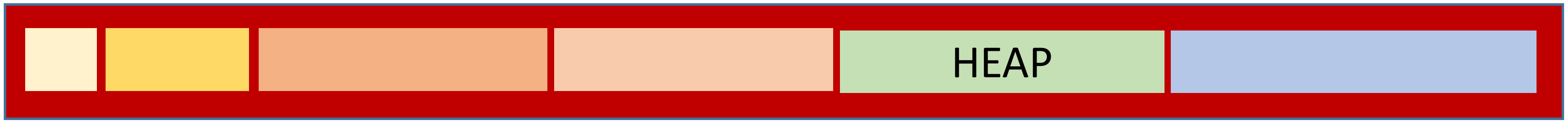
# Example: returning an array

- You can't say:

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

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

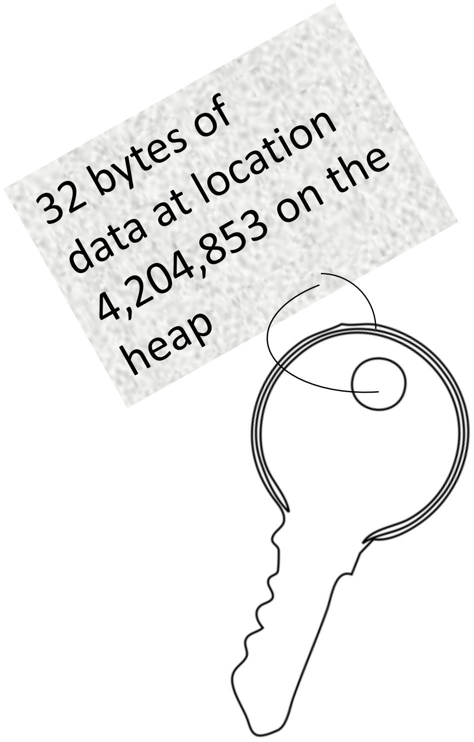
# Dynamic storage



- 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*()

- Imagine your program suddenly finds it has a large amount of data that it needs to store at runtime. This is a bit like asking for a large storage locker for the data: *malloc()*
- You tell the *malloc()* function exactly how much 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

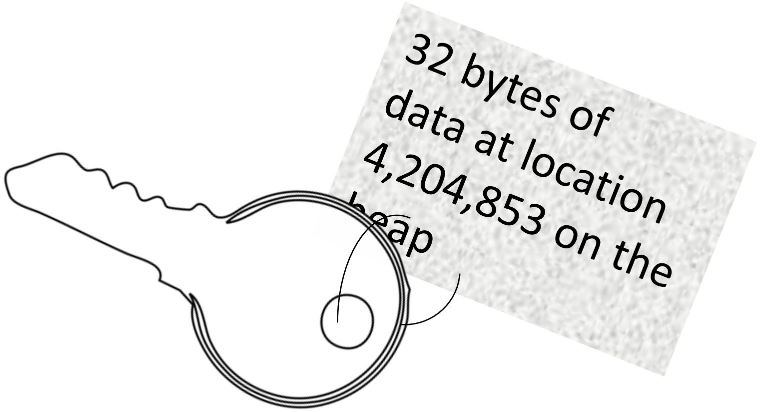


# Give the memory back when you’re done

- The good news about heap memory is that you can keep hold of it for a really long time. The bad news is…you can keep hold of it for a really long time
- 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.



Thanks for the storage. I’m done with it now

# *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.

# Example: creating and returning copy of the string

```
/*Given a C string, return a heap-allocated copy of the string.
Allocates a block on the heap of the appropriate size, copies the string into the block,
and returns a pointer to the block.
The caller takes over ownership of the block and is responsible for freeing it.*/
char* string_copy (const char* string) {
	char* newString;
	int len;
	len = strlen(string) + 1; // +1 to account for the '\0'
	newString = malloc(sizeof(char)*len); // elem-size * number-of-elements

	strcpy (newString, string); // copy the passed-in string to the block
	return(newString); // return a ptr to the block
}
```

# 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()

# Exercise malloc and strings