

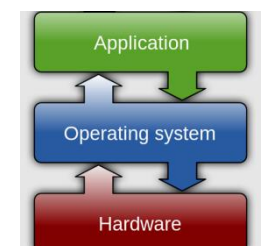
Refreshing C

Hartmut Kaiser

<https://teaching.hkaiser.org/fall2026/csc7103>

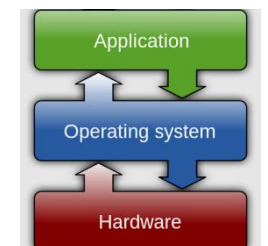
C History

- Developed in the 1970s – in conjunction with development of UNIX operating system
 - When writing an OS kernel, efficiency is crucial
 - This requires low-level access to the underlying hardware:
 - e.g. programmer can leverage knowledge of how data is laid out in memory, to enable faster data access
- UNIX originally written in low-level assembly language – but there were problems:
 - No structured programming (e.g. encapsulating routines as “functions”, “methods”, etc.) – code hard to maintain
 - Code worked only for particular hardware – not portable



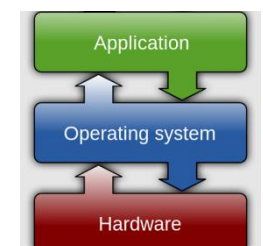
C Characteristics

- C takes a middle path between low-level assembly language...
 - Direct access to memory layout through pointer manipulation
 - Concise syntax, small set of keywords
- ... and a high-level programming language like Java:
 - Block structure
 - Some encapsulation of code, via functions
 - Type checking (pretty weak)



C Dangers

- C is not object oriented!
 - Can't “hide” data as “private” or “protected” fields
 - You can follow standards to write C code that looks object-oriented, but you have to be disciplined – will the other people working on your code also be disciplined?
- C has portability issues
 - Low-level “tricks” may make your C code run well on one platform – but the tricks might not work elsewhere
- The compiler and runtime system will rarely stop your C program from doing stupid/bad things
 - Compile-time type checking is weak
 - No run-time checks for array bounds errors, etc. like in Java



Compiling your C program

- `gcc intro.c -o intro`
 - and run the program using
 - `./intro arg1!`
- If we just run `./intro`
 - We may get a segfault!

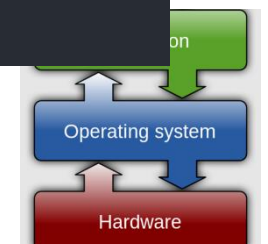
```
#include <stdio.h>

int main(int argc, char *argv[]) {
    // printf prints out a formatted string
    printf("%s\n", "Hello world!")

    // Print out the first command line argument
    printf("%s\n", argv[1])

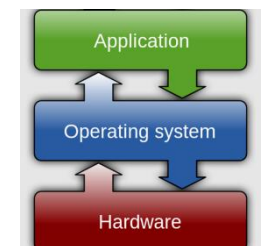
    // A return value of 0 means there were no errors
    return 0;
}
```

```
Output:
Hello world!
arg1! // Assuming we ran ./intro arg1!
```



Compiling your C program

- All C programs begin with a `main` function
 - The first argument `argc` denotes the number of elements in `argv`
 - The second argument `argv` is a list of string arguments passed from the command line to the program
 - The return value of a function indicates the exit code where 0 means successful (by convention)
- At the top of the file we have an `#include` statement to include `stdio.h`, a library that makes available functions such as `printf`.
- We can use `printf` to print formatted strings. In this case `%s` treats the input as a string.
 - `printf` is your friend for debugging!
 - `printf` doesn't emit a newline, so you'll have to add a `\n` if you want a newline



C Syntax

- At the top of the file we have an `include` statement to include `stdio.h`, a library that contains functions such as `printf`.
- Next is an array declaration using the `int {name}[{size}]` syntax.
- This array is declared on the stack. The distinction between declaration on the stack and heap is important to keep in mind!
- We can initialize the values by using the `{elem1, elem2, ...}` initializer syntax.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    const int NUMS_SIZE = 3;
    // Declare an array on the stack of length 3 in one line
    int nums[NUMS_SIZE] = {1, 2, 5};

    // Alternatively initialize an array like this
    nums[0] = 1;
    nums[1] = 2;
    nums[2] = 5;

    for (int i = 0; i < NUMS_SIZE; i++) {
        // printf doesn't emit a new line so we add a \n
        printf("Num at index %d is %d\n", i, nums[i]);
    }

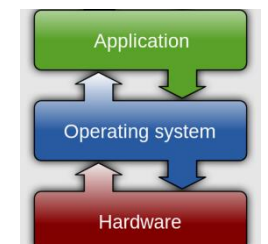
    // We can declare strings and print them out using %s
    char *str = "abcd";
    printf("My string is %s\n", str);

    // A return value of 0 means there were no errors
    return 0;
}
```

The printf() function

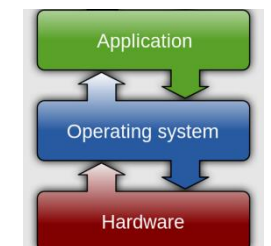
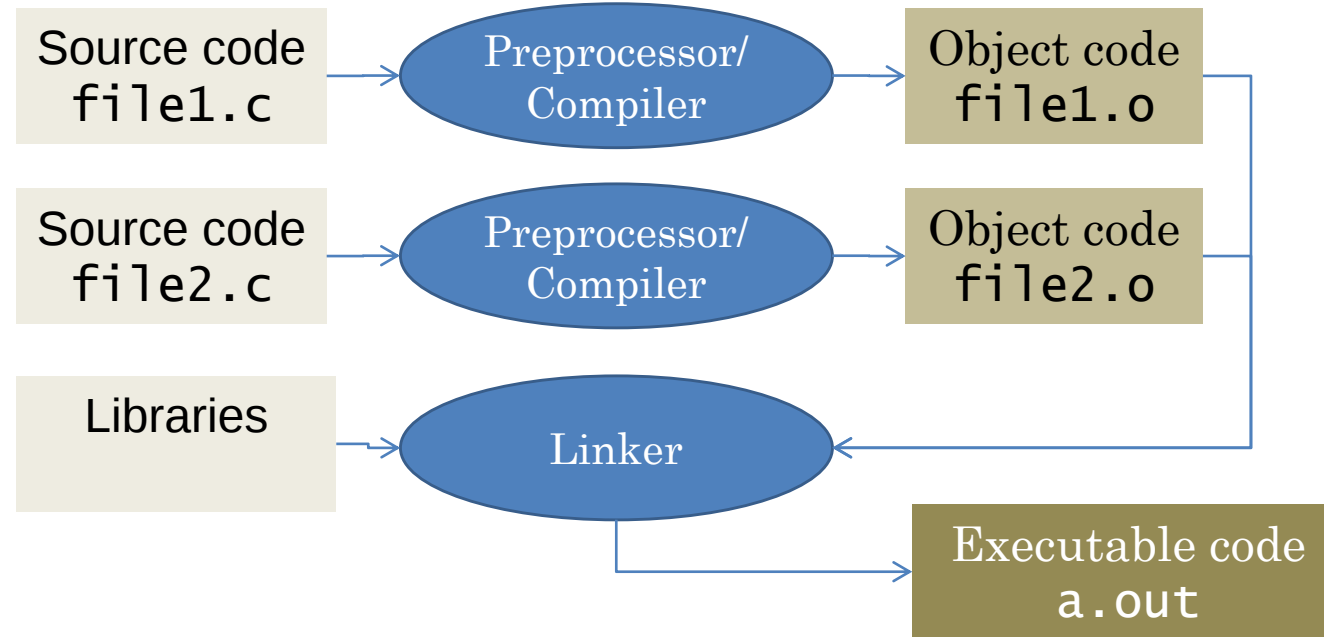
- `printf("Original input: %s\n", input);`
- `printf()` is a library function declared in `<stdio.h>`
- Syntax: `printf(FormatString, Expr...)`
 - `FormatString`: String of text to print
 - `Expr...`: Values to print
 - `FormatString` has placeholders to show where to put the values (note: #placeholders should match #Exprs)
 - Placeholders: `%s` (print as string), `%c` (print as char),
`%d` (print as integer),
`%f` (print as floating-point)
- `\n` indicates a newline character

Make sure you pick
the right one!



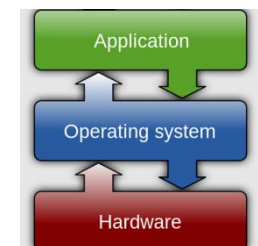
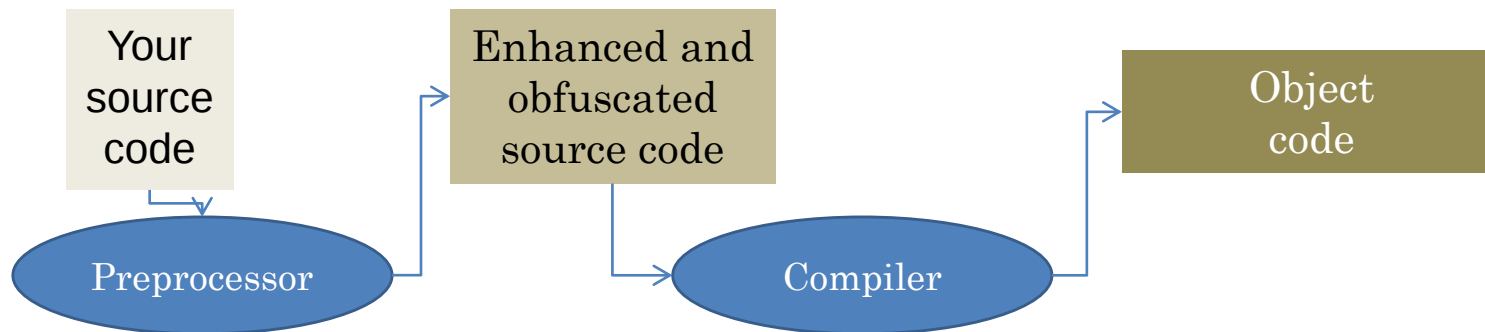
Separate compilation

- A C program consists of source code in one or more files
- Each source file is run through the preprocessor and compiler, resulting in a file containing object code
- Object files are tied together by the linker to form a single executable program



The Preprocessor

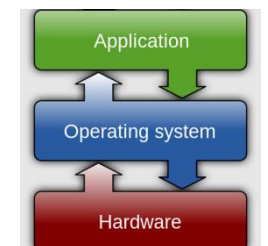
- The preprocessor takes your source code and – following certain directives that you give it – tweaks it in various ways before compilation.
- A directive is given as a line of source code starting with the # symbol
- The preprocessor works in a very crude, “word-processor” way, simply cutting and pasting –
 - it doesn't really know anything about C!



Preprocessor Directives

```
#define MAX_COLS    20  
#define MAX_INPUT   1000
```

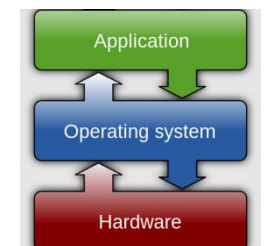
- The `#define` directives perform
 - “global replacements”:
 - every instance of `MAX_COLS` is replaced with `20`, and every instance of `MAX_INPUT` is replaced with `1000`.
- Other directives:
`#include <> / #include ""`
`#ifdef / #else / #endif`



Preprocessor directives

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

- The `#include` directives “paste” the contents of the files `stdio.h`, `stdlib.h` and `string.h` into your source code, at the very place where the directives appear.
- These files contain information about some library functions used in the program:
 - `stdio` stands for “standard I/O”, `stdlib` stands for “standard library”, and `string.h` includes useful string manipulation functions.
- Want to see the files? Look in `/usr/include` (or similar)



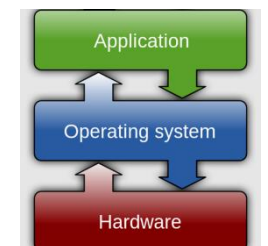
Pointers, Arrays, Strings

- The notions of string, array, and pointer are somewhat interchangeable:
 - An array of characters could be defined, for purposes of holding the input string:

```
char input[MAX_INPUT];
```

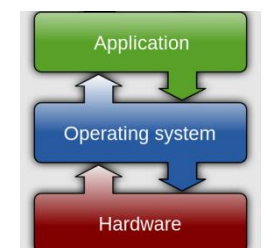
- Yet when it's passed in as an argument to a function, `input` has morphed into a pointer to a character (`char *`):

```
void some_function(char const *input, ...)
```

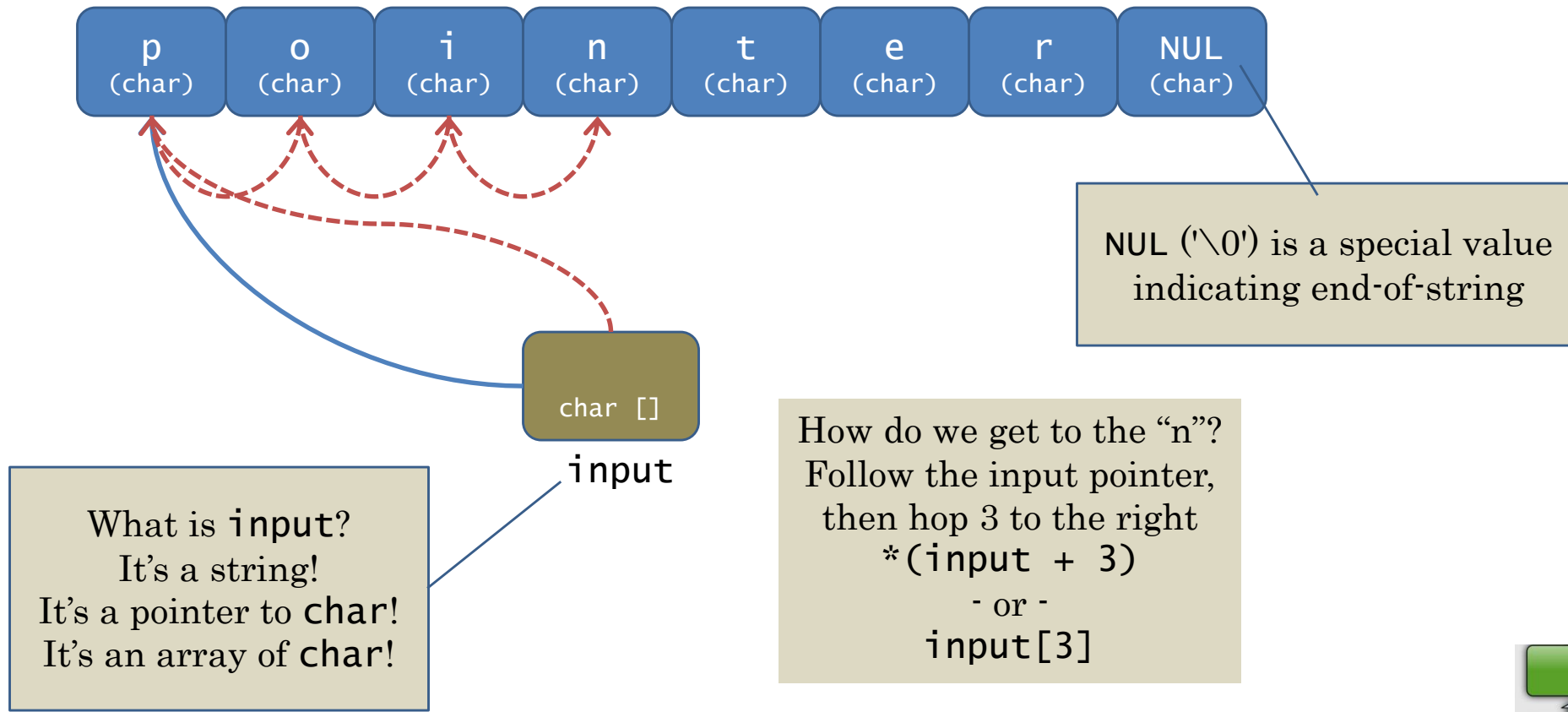


Pointers, Arrays, Strings

- In C, the three concepts are indeed closely related:
 - A pointer is simply a memory address. The type `char*` (i.e. “pointer to character”) signifies that the data at the address the pointer is holding is to be interpreted as a character.
 - An array is simply a pointer – of a special kind:
 - The array ‘name’ refers to the first of a sequence of data items stored sequentially in memory
 - How do you get to the other array elements? By incrementing the pointer value
 - A string is simply an array of characters – unlike Java, which has a predefined String class.



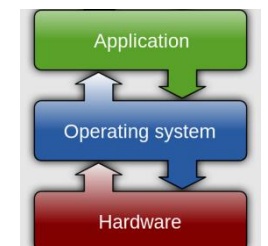
String Layout and Access



Data Types in C

Four basic data types

- Integers: `char`, `short int`, `int`, `long int`, `enum`
 - Also corresponding unsigned integral types: `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`
- Floating-point types: `float`, `double`, `long double`
- Pointers
- Aggregates: `struct`, `union`
- Integer and floating-point types stand for themselves, but pointers and aggregate types combine with other types, to form a virtually limitless variety of types

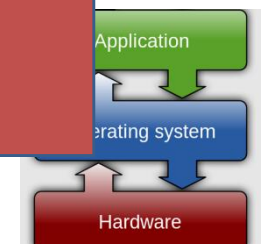


Characters are of integer Type

- From a C perspective, a character is indistinguishable from its numeric ASCII value –
 - the only difference is in how it's displayed
- Example: converting a character digit to its numeric value
 - The value of '2' is not 2 – it's 50 (hexadecimal 0x32)
 - To convert, subtract the ASCII value of '0' (which is 48, or 0x30)

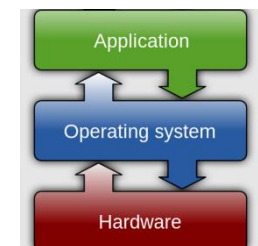
```
char digit, digit_num_value;  
...  
digit_num_value = digit - '0';
```

Behaviorally,
this is identical to
 $\text{digit} - 48$
Why is
 $\text{digit} - '0'$
preferable?



Integer Values play the Role of “Booleans”

- There is no “Boolean” type
 - Relational operators (`==`, `<`, etc.) return either `0` or `1`
 - Boolean operators (`&&`, `||`, etc.) return either `0` or `1`,
 - and take any `int` values as operands
- How to interpret an arbitrary `int` as a Boolean value:
 - `0` $\rightarrow$ `false`
 - Any other value $\rightarrow$ `true`



The infamous = Blunder

- Easy to confuse equality with assignment
 - In C, the test expression of an if statement can be any `int` expression — including an assignment expression

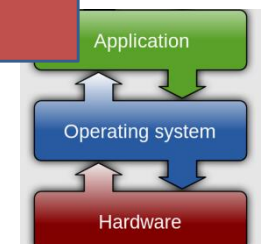
```
if (y = 0)
    printf("Sorry, can't divide by zero.\n");
else
    result = x / y;
```

Assignment performed;
y set to 0 (oops)

Expression returns
result of assignment:
0, or "false"

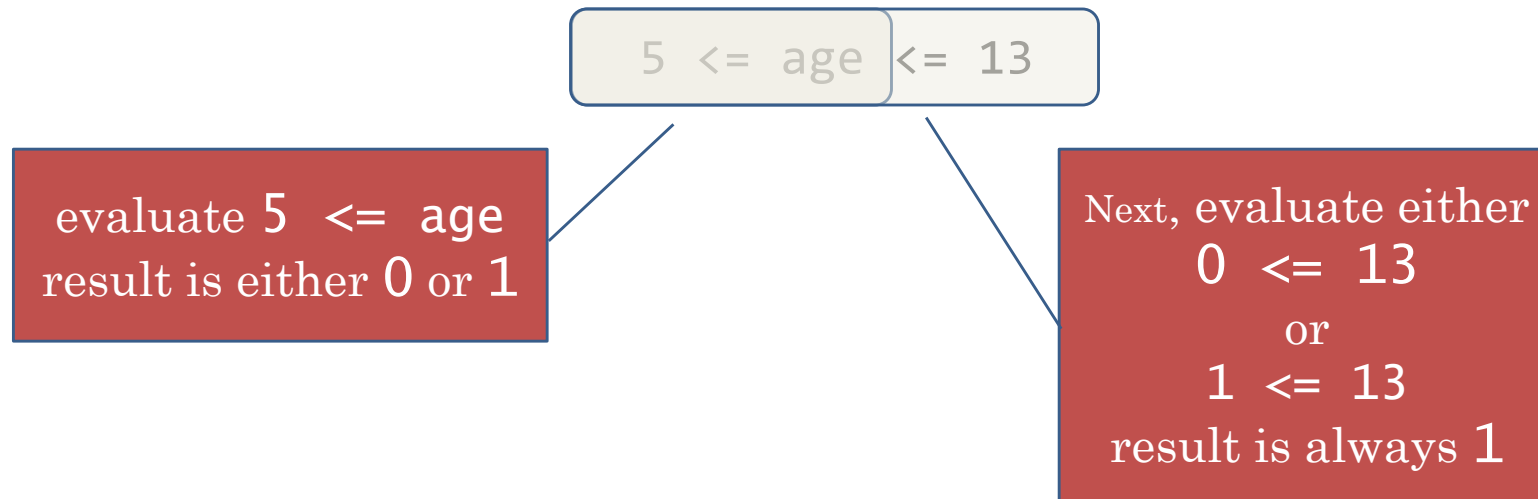
`else` clause executed:
divide by 0!

- The compiler will not catch this bug!
 - Some compilers will issue a warning

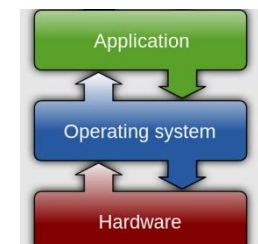


The less infamous “relational chain” Blunder

- Using relational operators in a “chain” doesn't work
- Example: “is age between 5 and 13”

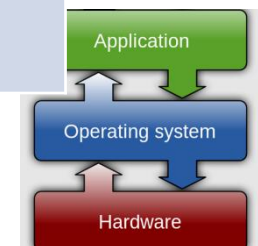


- A correct solution: `5 <= age && age <= 13`



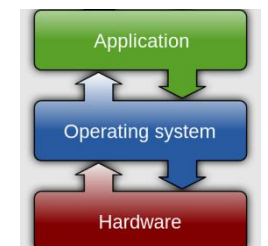
Ranges of Integer Types

Type	Min value	Max value
char	0	UCHAR_MAX (≥ 127)
signed char	SCHAR_MIN (≤ -127)	SCHAR_MAX (≥ 127)
unsigned char	0	UCHAR_MAX (≥ 255)
short int	SHRT_MIN (≤ -32767)	SHRT_MAX (≥ 32767)
unsigned short int	0	USHRT_MAX (≥ 65535)
int	INT_MIN (≤ -32767)	INT_MAX (≥ 32767)
unsigned int	0	INT_MAX (≥ 65535)
long int	LONG_MIN (≤ -2147483647)	LONG_MAX (≥ 2147483647)
unsigned long int	0	ULONG_MAX (≥ 4294967295)



Ranges of Integer Types

- Ranges for a given platform can be found at `/usr/include/limits.h`
- `char` can be used for very small integer values
- Plain `char` may be implemented as `signed` or `unsigned` on a given platform – safest to “assume nothing” and just use the range 0...127
- `short int` “supposed” to be smaller than `int` —
 - but it depends on the underlying platform



Ranges of Floating-Point Types

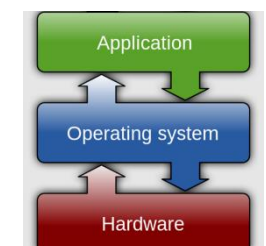
Type	Min value	Max value
float	FLT_MIN ($\leq -10^{37}$)	FLT_MAX ($\leq -10^{37}$)
double	DBL_MIN ($\leq -FLT_MIN$)	DBL_MAX ($\geq FLT_MAX$)
long double	LDBL_MIN ($\leq -DBL_MIN$)	LDBL_MAX ($\geq DBL_MAX$)

Floating-point literals must contain a decimal point, an exponent, or both.

3.14159

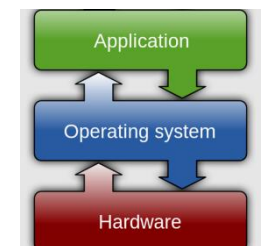
25.

6.023e23



Danger: Precision of Floating-Point Values

- Testing for equality between two floating-point values: almost always a bad idea
 - One idea: instead of simply using `==`, call an “equality routine” to check whether the two values are within some margin of error.

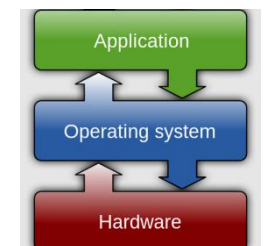


Casting: Converting one Type to Another

- The compiler will do a certain amount of type conversion for you (silently):

```
int a = 'A';    /* char literal converted to int */
```

- In some circumstances, you need to explicitly cast an expression as a different type – by putting the desired type name in parentheses before the expression
 - e.g. `(int)3.14159` will return the `int` value 3

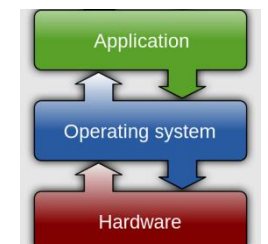


Pointers

Review of Pointers

- A pointer is just a memory location (an address).
- A memory location is simply an integer value, that we interpret as an address in memory.
- The contents at a particular memory location is just a collection of bits - there's nothing special about them that makes them ints, chars, etc.
 - How you want to interpret the bits is up to you (that's what **types** are for).
- Is this... an `int` value?
 - ... a pointer to a memory address?
 - ... a series of char values?

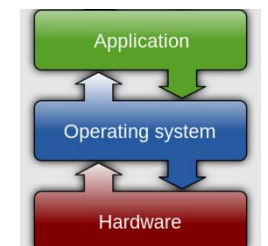
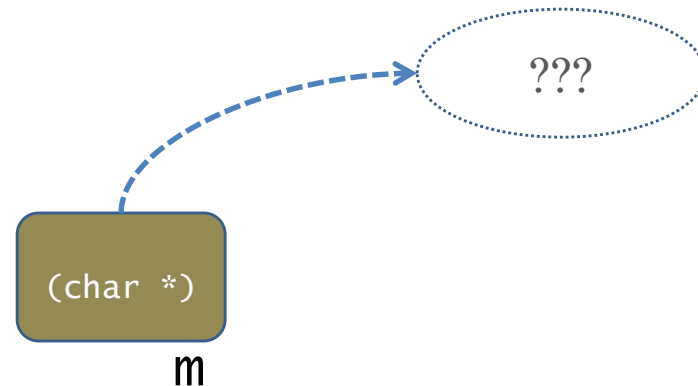
0xfe4a10c5



Review of Pointer Variables

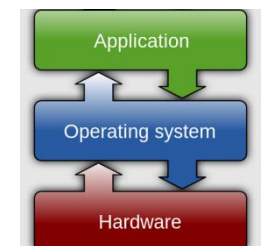
- A pointer variable is just a variable, that contains a value that we interpret as a memory address.
- Just like an uninitialized `int` variable holds some arbitrary “garbage” value,
 - an uninitialized pointer variable points to some arbitrary “garbage address”

```
char* m;
```



Following a “garbage” pointer

- What will happen? Depends on what the arbitrary memory address is:
 - If it's an address to memory that the OS has not allocated to our program, we get a segmentation fault
 - If it's a nonexistent address, we get a bus error
 - Some systems require multibyte data items, like `ints`, to be aligned: for instance, an `int` may have to start at an even-numbered address, or an address that's a multiple of 4. If our access violates a restriction like this, we get a bus error
 - If we're really unlucky, we'll access memory that is allocated for our program –
 - We can then proceed to destroy our own data!



How can you Test whether a Pointer points to Something Meaningful?

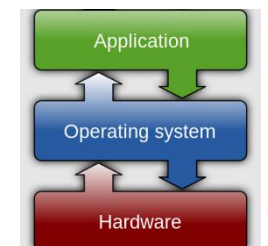
- There is a special pointer value NULL, that signifies “pointing to nothing”. You can also use the value 0.

```
char* m = NULL;
```

```
...
```

```
if (m) { ... safe to follow the pointer ... }
```

- Here, m is used as a boolean value
 - If m is “false”, aka 0, aka NULL, it is not pointing to anything
 - Otherwise, it is (presumably) pointing to something good
 - Note: It is up to the programmer to assign NULL values when necessary

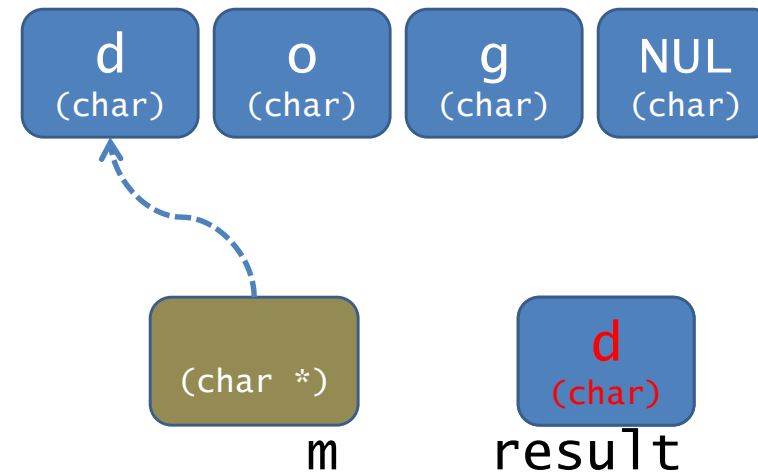


Indirection operator *

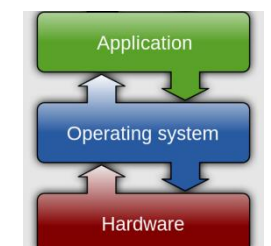
- Moves from address to contents

```
char* m = "dog";
```

```
char result = *m;
```



- `m` is an address of a `char`
 - `*m` instructs us to take the contents of that address
 - `result` gets the value `'d'`

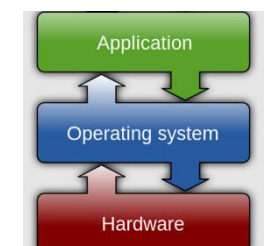
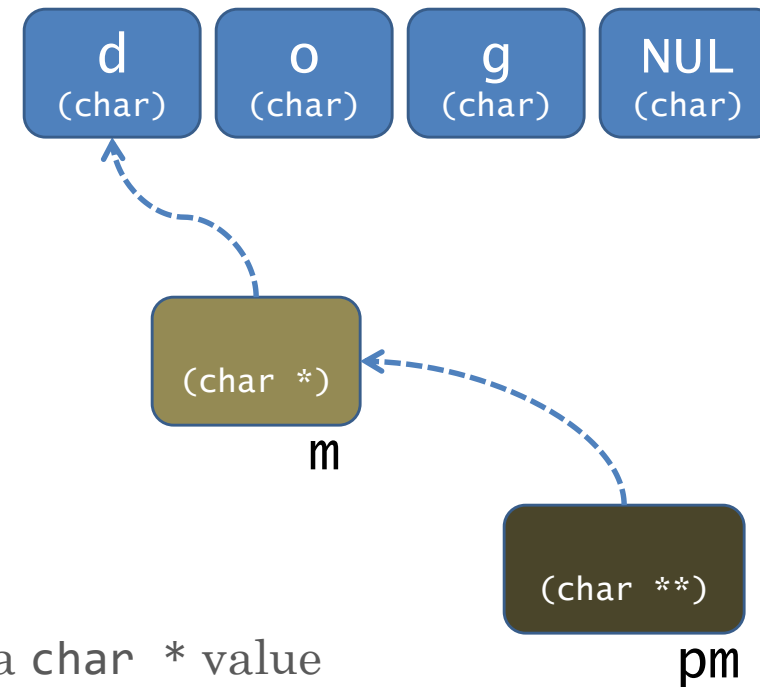


Address operator &

- Instead of contents, returns the address

```
char* m = "dog";  
char** pm = &m;
```

- pm needs a value of type char **
 - Can we assign *m to it? No – type is char
 - Can we assign m to it? No – type is char *
 - &m gives it the right value – the address of a char * value



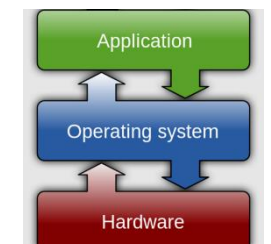
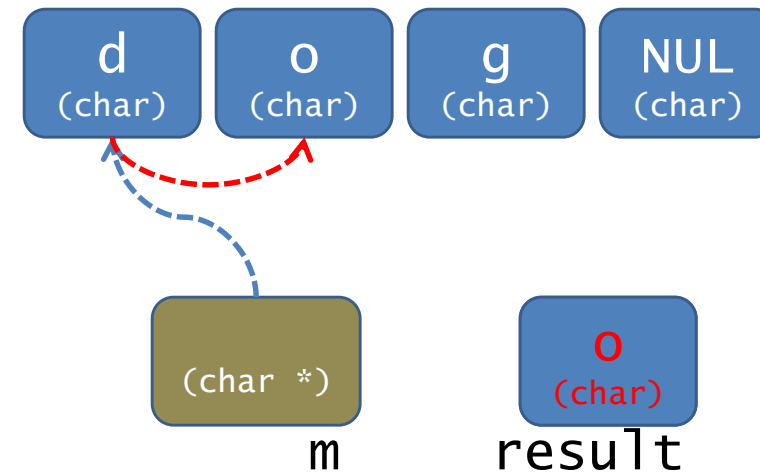
Pointer Arithmetic

- C allows pointer values to be incremented by integer values

```
char* m = "dog";
```

```
char result = *(m + 1);
```

- `m` is an address of a `char`
 - `(m + 1)` is the address of the next `char`
 - `*(m + 1)` instructs us to take the contents of that address
 - `result` gets the value `'o'`



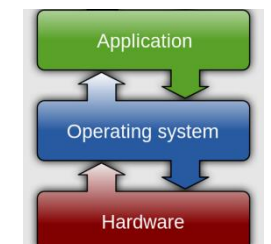
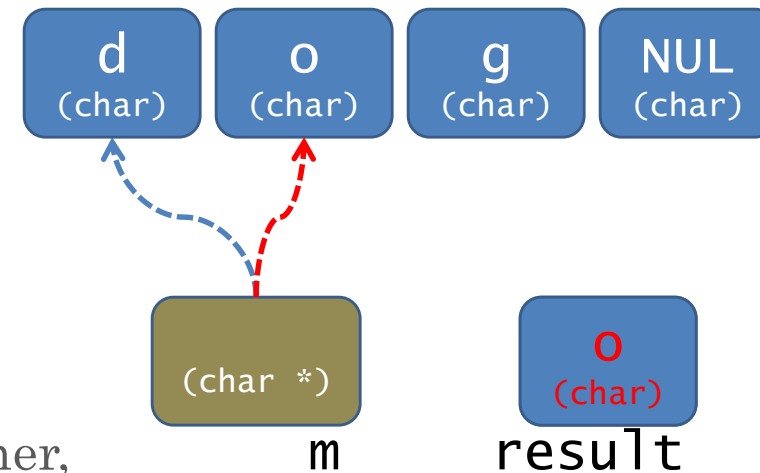
Pointer Arithmetic

- A slightly more complex example:

```
char* m = "dog";
```

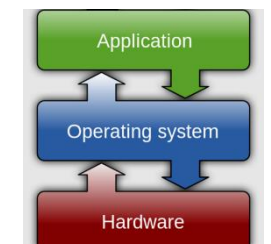
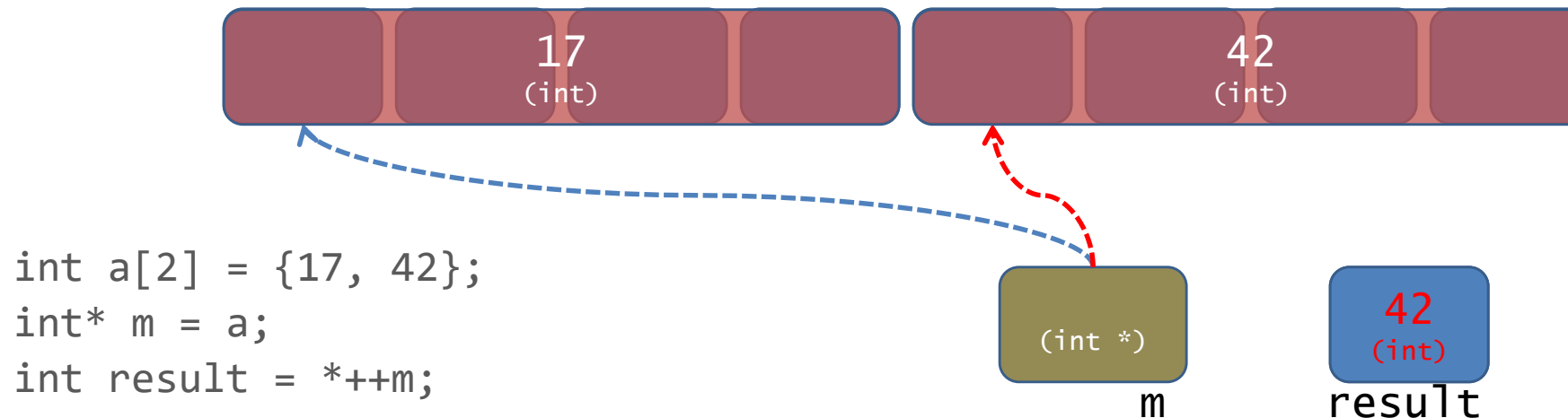
```
char result = *++m;
```

- `m` is an address of a `char`
- `++m` changes `m`, to the address one byte higher,
 - and returns the new address
- `*++m` instructs us to take the contents of that location
- `result` gets the value `'o'`



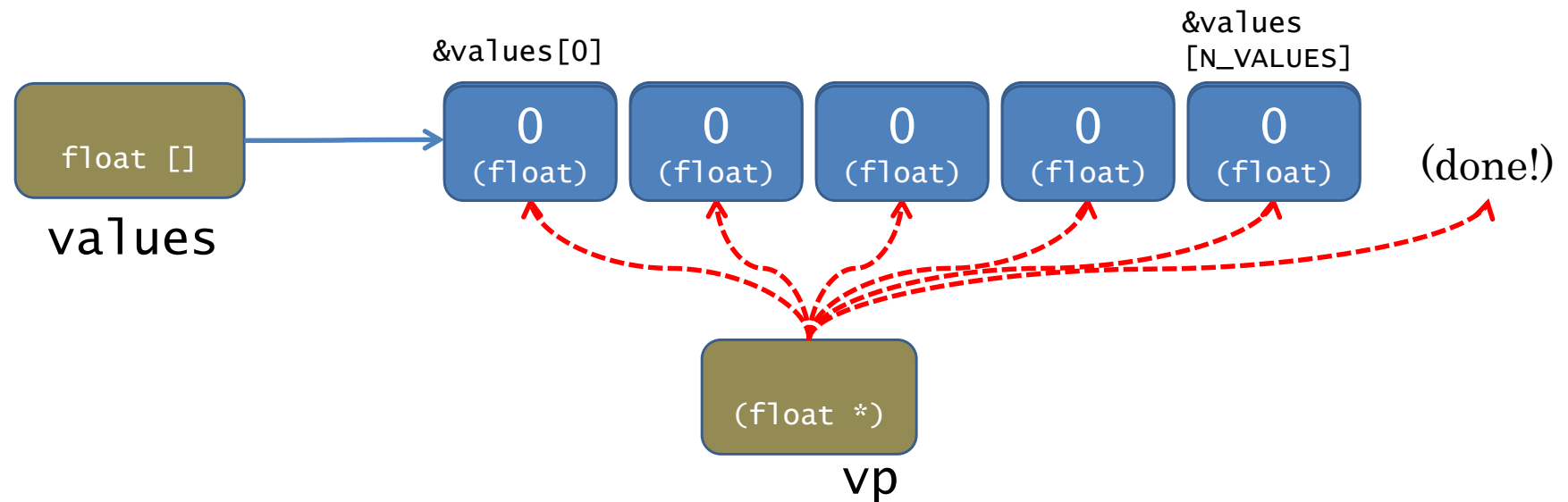
Pointer Arithmetic

- How about multibyte values?
 - Q: Each char value occupies exactly one byte, so obviously incrementing the pointer by one takes you to a new char value...
 - But what about types like `int` that span more than one byte?
 - A: C “does the right thing”: increments the pointer by the size of one `int` value

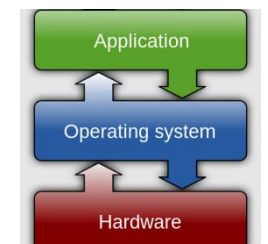


Example: Initializing an Array

```
#define N_VALUES 5  
float values[N_VALUES];
```



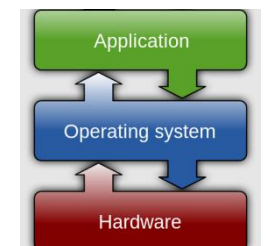
```
float* vp;  
for (vp = &values[0]; vp < &values[N_VALUES]; /**/)  
    *vp++ = 0;
```



Example: strcpy “string copy”

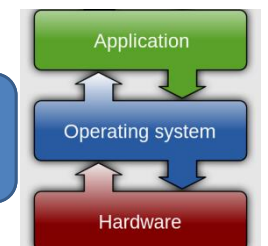
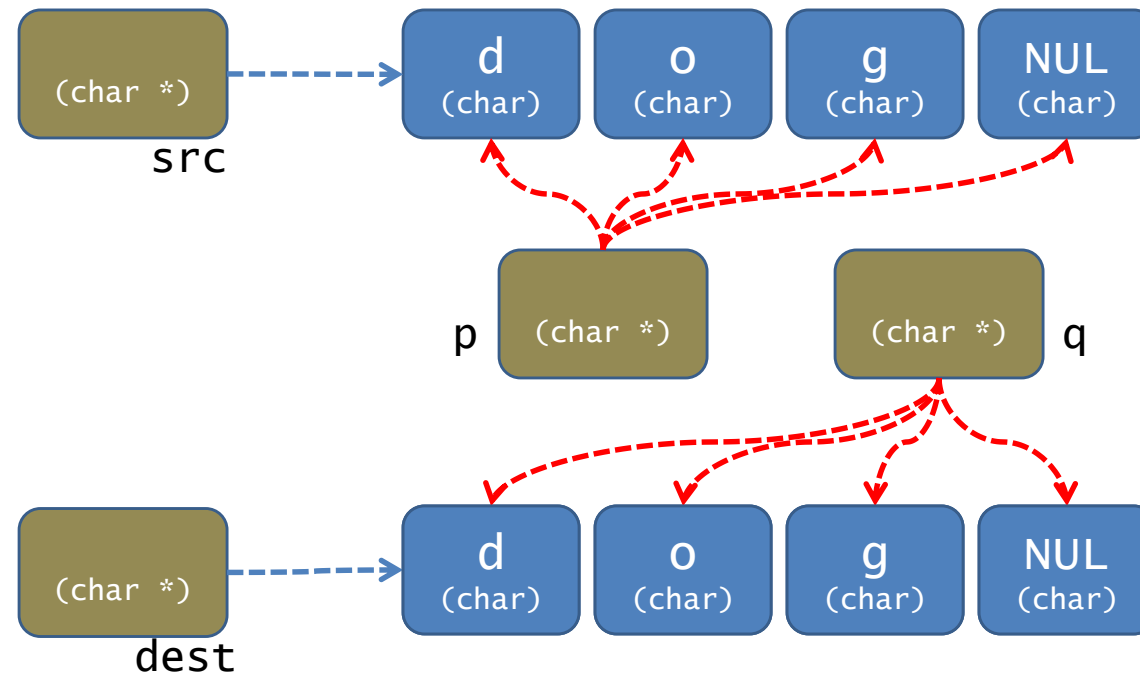
```
char* strcpy(char* dest, const char* src)
```

- (assume that) `src` points to a sequence of `char` values that we wish to copy, terminated by `'\0'`
- (assume that) `dest` points to an accessible portion of memory large enough to hold the copied chars
- `strcpy` copies the `char` values of `src` to the memory pointed to by `dest`
- `strcpy` also gives `dest` as a return value



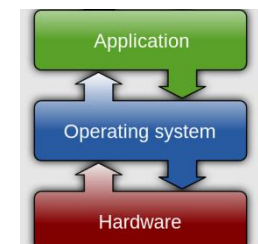
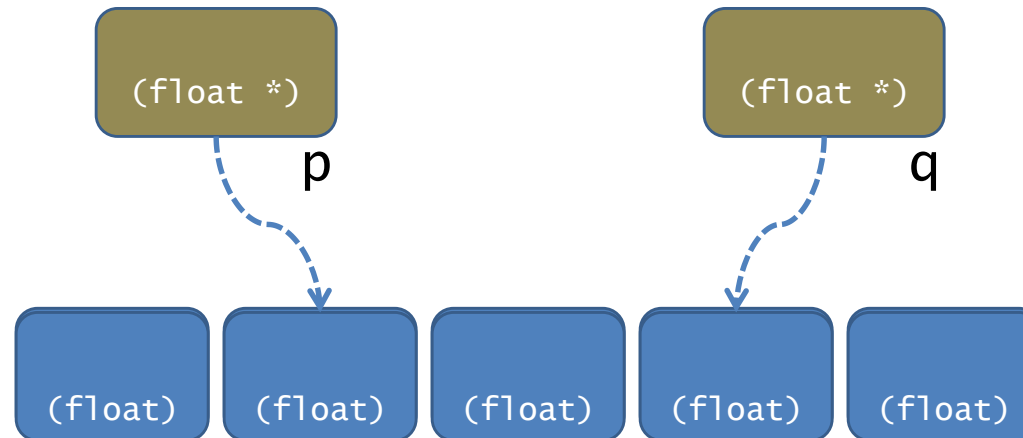
Example: strcpy “string copy”

```
char* strcpy(char* dest, const char* src) {  
    const char* p;  
    char* q;  
    for(p = src, q = dest; *p != '\0'; ++p, ++q)  
        *q = *p;  
    *q = '\0';  
    return dest;  
}
```



Pointer Subtraction and relational Operations

- Only meaningful in special context: where you have two pointers referencing different elements of the same array
 - $q - p$ gives the difference (in number of array elements, not number of bytes between p and q (in this example, 2))
 - $p < q$ returns 1 if p has a lower address than q ; else 0
 - (in this example, it returns 1)



Arrays

Arrays

- C arrays can be tricky since we can represent them using the [] syntax or as pointers using the * syntax.
- In the case of the [] syntax
 - We can access elements by using the familiar arr1[index] syntax
- In the case of the pointer * syntax
 - malloc returns a pointer to the start of a buffer that is the same size as the argument that is passed in
 - The value of int* arr2 in this case is a pointer to the start of a buffer of size 12
 - Each int is 4 bytes and there are 3 of them

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int NUM_ELEMS = 3;
    // Below are two equivalent ways of initializing an array

    // Declare an array on the stack
    int arr1[NUM_ELEMS];

    for (int i = 0; i < NUM_ELEMS; i++) {
        arr1[i] = i;
    }

    // Declare an array via malloc which is on the heap
    // arr2 is a pointer to the start of the array
    int *arr2 = malloc(sizeof(int) * NUM_ELEMS);

    printf("arr2: %x\n", arr2);

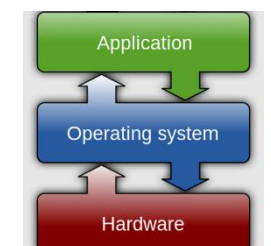
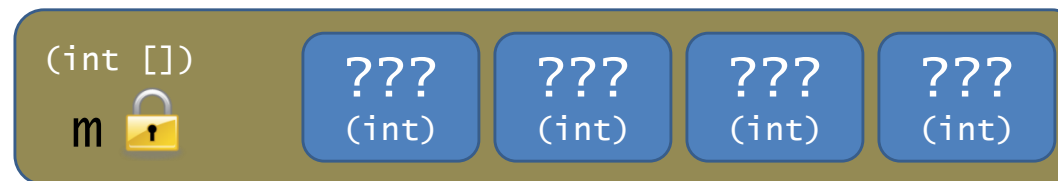
    for (int i = 0; i < NUM_ELEMS; i++) {
        // Add offset i to the start of the array
        // and get the value at that location by dereferencing
        *(arr2 + i) = i;
        // Note there is no "dereference" in the print statement
        printf("(arr2 + %d): %x\n", i, arr2 + i);
    }

    return 0;
}
```

Review of arrays

- There are no array variables in C – only array names
 - Each name refers to a constant pointer (address of first element of array)
 - Space for array elements is allocated at time of definition, can't be changed
- Can't change where the array name refers to...
 - but you can change the array elements,
 - via pointer arithmetic

```
int m[4];
```



Subscripts and pointer arithmetic

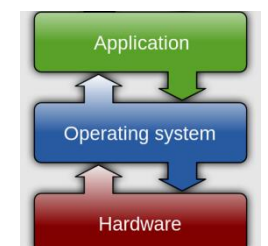
- `array[subscript]` equivalent to `*(array + (subscript))`
- Strange but true: Given earlier declaration of `m`, the expression `2[m]` is legal!
 - Not only that: it's equivalent to:

`2[m]`

`*(2 + m)`

`*(m + 2)`

`m[2]`



Array names and Pointer Variables, playing together

```
int m[3];
```

subscript OK
with pointer
variable

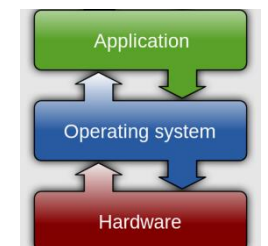
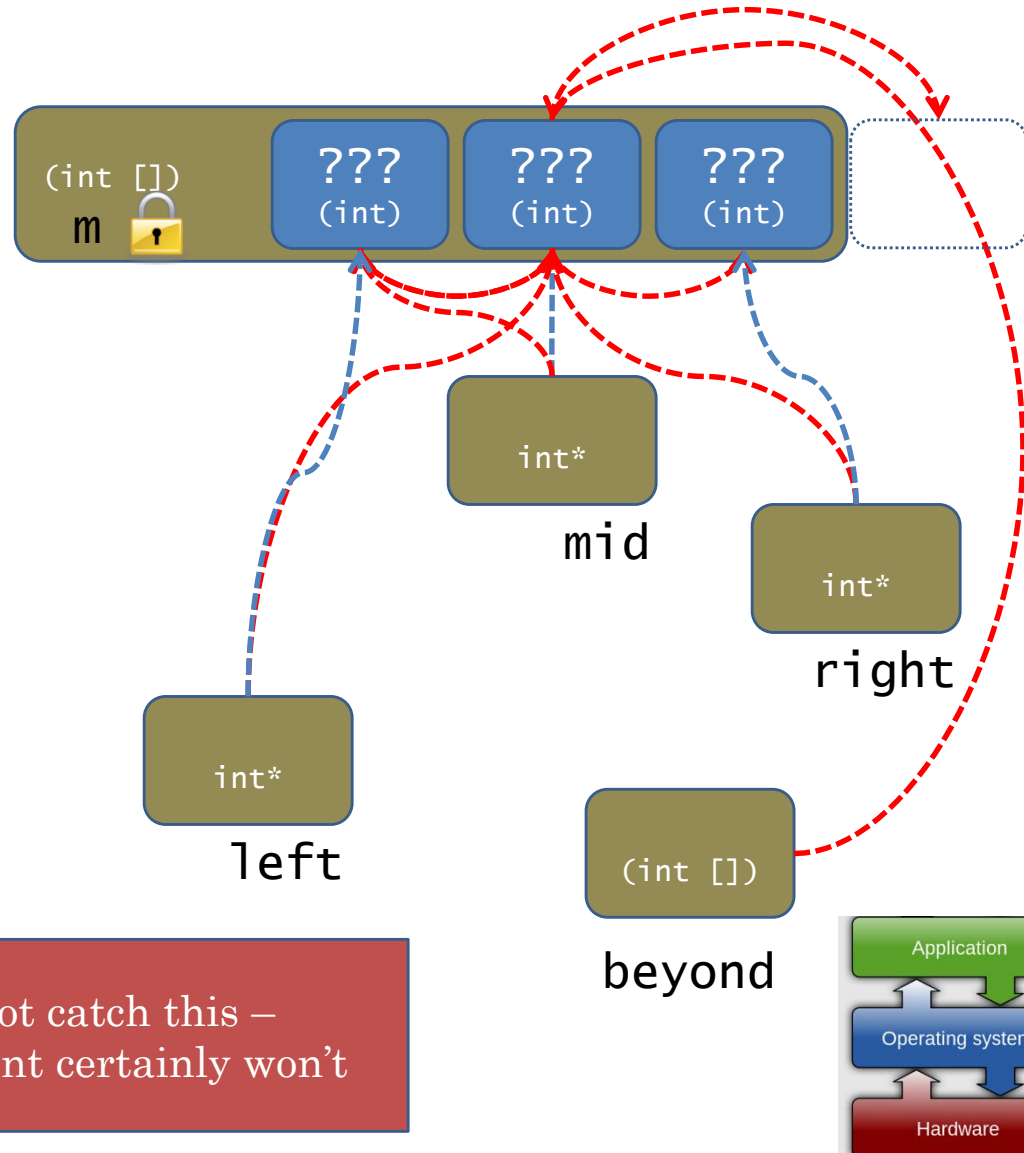
```
int* mid = m + 1;
```

```
int* right = &mid[1];
```

```
int* left = &mid[-1];
```

```
int* beyond = &mid[2];
```

compiler may not catch this –
runtime environment certainly won't

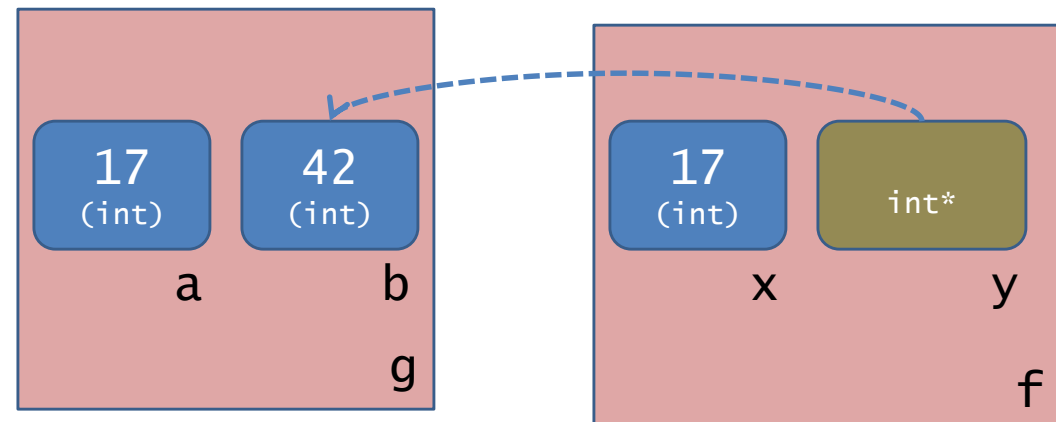


Array names as function arguments

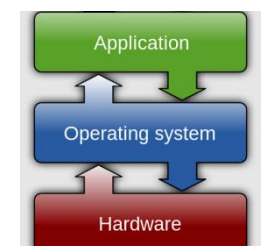
- In C, arguments are passed “by value”
 - A temporary copy of each argument is created, solely for use within the function call:

```
void f(int x, int* y) { ... }
```

```
void g(...) {  
    int a = 17, b = 42;  
    f(a, &b);  
    ...  
}
```

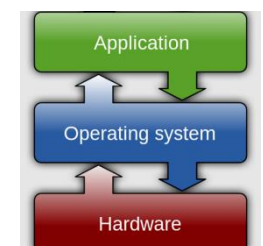


- Pass-by-value is “safe” in that the function plays only in its “sandbox” of temporary variables –
 - can’t alter the values of variables in the callee (except via the return value)



Array Names as Function Arguments

- But, functions that take arrays as arguments can exhibit what looks like “pass-by-reference” behavior, where the array passed in by the callee does get changed
 - Remember the special status of arrays in C
 - They are basically just pointers.
 - So arrays are indeed passed by value
 - but only the pointer is copied, not the array elements!
 - Note the advantage in efficiency (avoids a lot of copying)
 - But - the pointer copy points to the same elements as the callee’s array
 - These elements can easily be modified via pointer manipulation

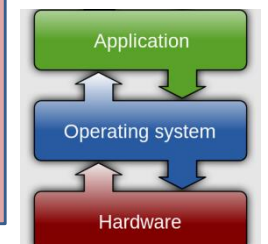
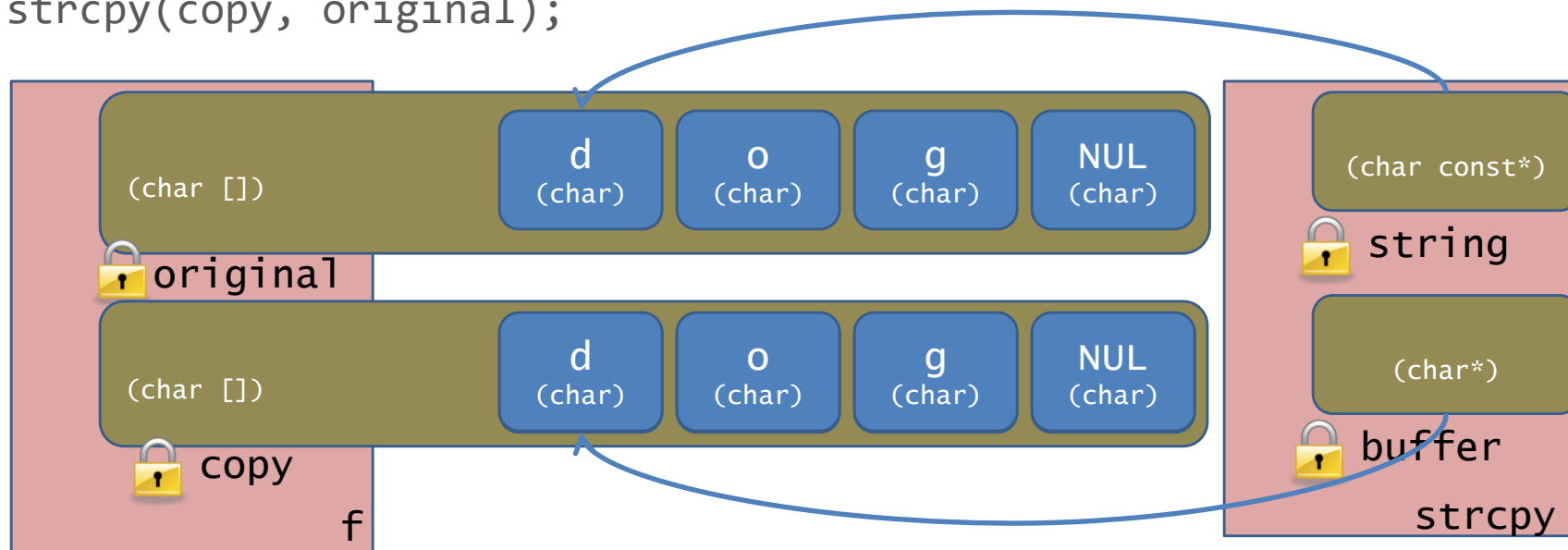


Array names as function arguments

- The strcpy “string copy” function puts this “pseudo” call-by-reference behavior to good use:

```
void strcpy(char* buffer, char const* string);
```

```
void f(...) {  
    char original[4] = "dog";  
    char copy[4];  
    strcpy(copy, original);  
}
```



When can array size be omitted?

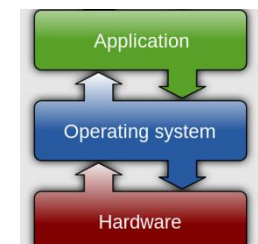
- There are a couple of contexts in which an array declaration need not have a size specified:
 - Parameter declaration:

```
int strlen(char string[]); // same as: int strlen(char* string);
```

 - As we've seen, the elements of the array argument are not copied, so the function doesn't need to know how many elements there are.
 - Array initialization:

```
int vector[] = {1, 2, 3, 4, 5};
```

 - In this case, just enough space is allocated to fit all (five) elements of the initializer list

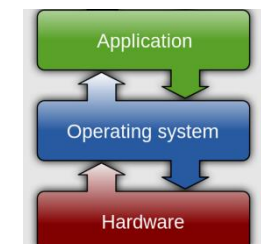
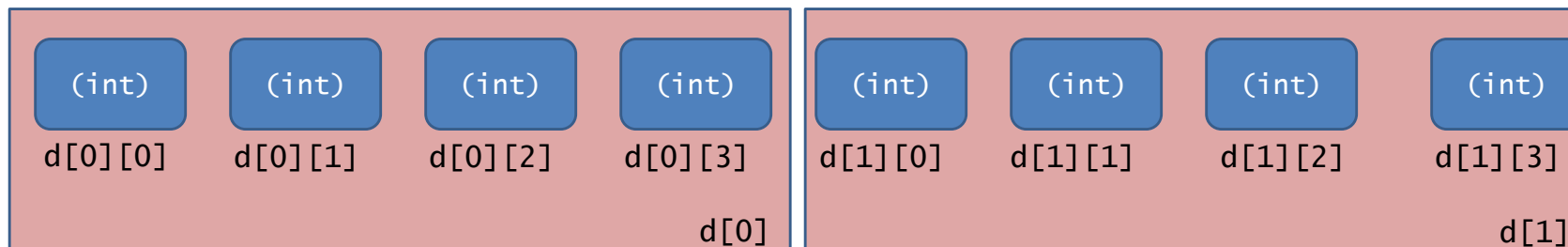


Multidimensional arrays

- How to interpret a declaration like:

```
int d[2][4];
```

- This is an array with two elements:
 - Each element is an array of four `int` values
- The elements are laid out sequentially in memory, just like a one-dimensional array
 - Row-major order: the elements of the rightmost subscript are stored contiguously



Subscripting in a multidimensional array

```
int d[2][4];
```

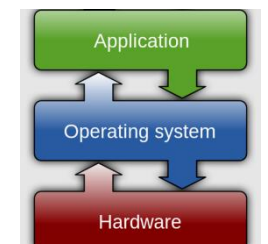
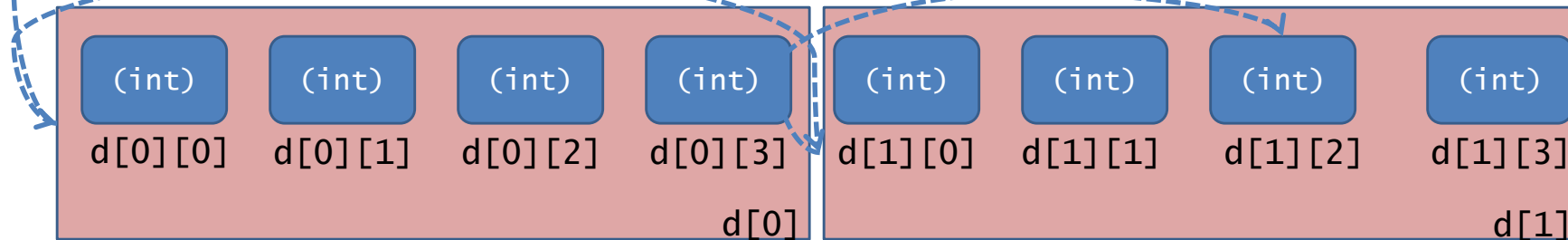
`d[1][2]`



$*(*(d+1)+2)$

Then increment by the
size of 2 `ints`

Increment by the size of
1 array of 4 `ints`



Why do we care about Storage Order?

- If you stay within the “paradigm” of the multidimensional array, the order doesn’t matter...
- But if you use tricks with pointer arithmetic,
 - it matters a lot

- It also matters for initialization

- To initialize d like this:

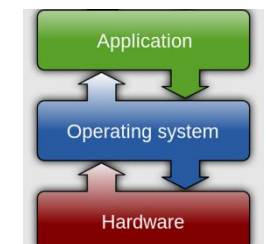
- use this:

```
int d[2][4] = {0, 1, 2, 3, 4, 5, 6, 7};
```

- rather than this

```
int d[2][4] = {0, 4, 1, 5, 2, 6, 3, 7};
```

0	1	2	3
4	5	6	7

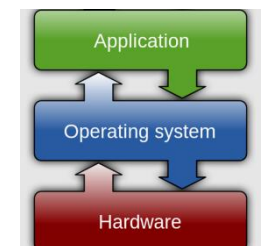


Multidimensional Arrays as Parameters

- Only the first subscript may be left unspecified:

```
void f(int matrix[][10]);    /* OK */  
void g(int (*matrix)[10]);  /* OK */  
void h(int matrix[][]);     /* not OK */
```

- Why?
 - Because the other sizes are needed for scaling when evaluating subscript expressions (see previous slides)
 - This points out an important drawback to C:
 - Arrays do not carry information about their own sizes!
 - If array size is needed, you must supply it somehow
 - (e.g., when passing an array argument, you often have to pass an additional “array size” argument) – bummer



Malloc

Using malloc

- malloc returns a pointer to the start of a region of memory on the heap. It takes in the number of bytes to allocate.
- Knowing the differences between malloc-ing data on the heap and declaring data on the stack is important for CSC7103.
- Consider the commented out code
`char copied[length + 1]`
 - If we were to use this line of code instead of the line with malloc, what would happen?
 - It's possible we get a segfault or the returned string is garbage!
 - This happens because we declared our string on the stack inside the `str_copier` function frame and returned a pointer to the string located in the function frame
 - But when we return from `str_copier`, the stack frame is deallocated so now we have a dangling pointer to a location in the deallocated function frame!
 - So never declare things on the stack and then return pointers to them!

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

int main(int argc, char *argv[]) {
    // Declare a string
    char *str = "Help";
    printf("Copied str: %s\n", str_copier(str));
    // What's missing here? free(str);
    return 0;
}

// Returns a malloced copy of the string
char *str_copier(char *str) {
    int length = strlen(str);

    char *copied = malloc(sizeof(char) * (length + 1));
    // Consider the following commented out code
    // char copied[length + 1];

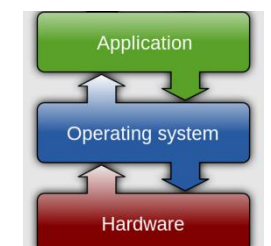
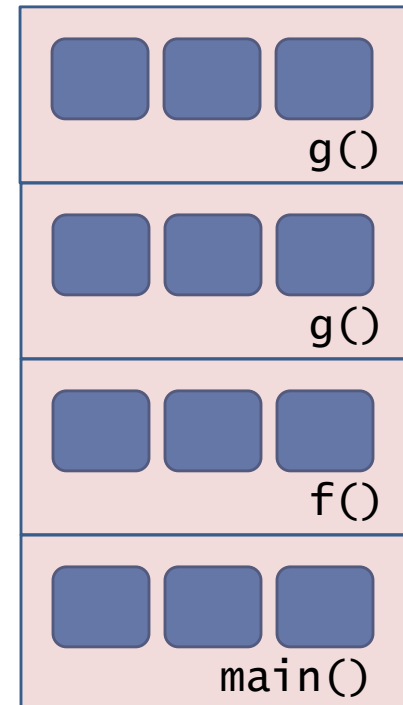
    strcpy(copied, str);
    return copied;
}
```



Overview of Memory Management

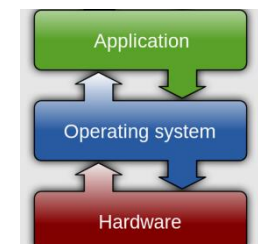
- Stack-allocated memory
 - When a function is called, memory is allocated for all of its parameters and local variables.
 - Each active function call has memory on the stack (with the current function call on top)
 - When a function call terminates,
 - the memory is deallocated (“freed up”)

Ex: `main()` calls `f()`,
 `f()` calls `g()`
 `g()` recursively calls `g()`



Overview of Memory Management

- Heap-allocated memory
 - This is used for persistent data, that must survive beyond the lifetime of a function call
 - global variables
 - dynamically allocated memory – C statements can create new heap data (similar to new in Java/C++)
 - Heap memory is allocated in a more complex way than stack memory
 - Like stack-allocated memory, the underlying system determines where to get more memory – the programmer doesn't have to search for free memory space!



Allocating new Heap Memory

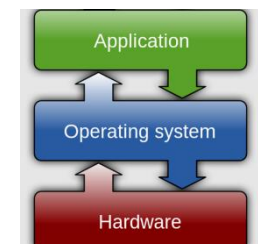
```
void* malloc(size_t size);
```

- Allocate a block of size bytes,
 - return a pointer to the block
 - (NULL if unable to allocate block)

Note: `void *` denotes a generic pointer type

```
void* calloc(size_t num_elements, size_t element_size);
```

- Allocate a block of `num_elements * element_size` bytes,
 - initialize every byte to zero,
 - return pointer to the block
 - (NULL if unable to allocate block)

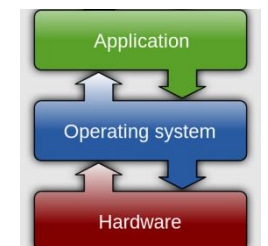


Allocating new Heap Memory

```
void* realloc(void* ptr, size_t new_size);
```

- Given a previously allocated block starting at ptr,
 - change the block size to new_size,
 - return pointer to resized block
 - If block size is increased, contents of old block may be copied to a completely different region
 - In this case, the pointer returned will be different from the ptr argument, and ptr will no longer point to a valid memory region
- If ptr is NULL, realloc is identical to malloc
- Note: may need to cast return value of malloc/calloc/realloc:

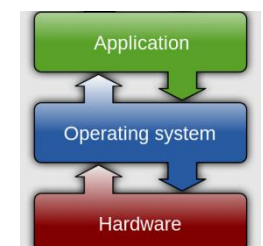
```
char* p = (char*) malloc(BUFFER_SIZE);
```



Deallocating Heap Memory

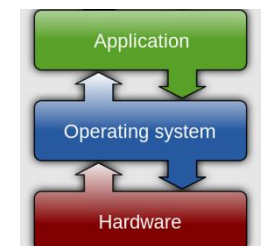
```
void free(void* pointer);
```

- Given a pointer to previously allocated memory,
 - put the region back in the heap of unallocated memory
- Note: easy to forget to free memory when no longer needed...
 - especially if you're used to a language with “garbage collection” like Java
 - This is the source of the notorious “memory leak” problem
 - Difficult to trace – the program will run fine for some time, until suddenly there is no more memory!



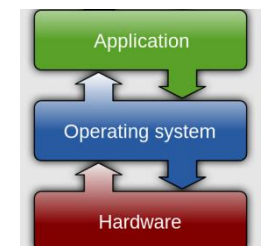
Checking for successful Allocation

- Call to `malloc` might fail to allocate memory, if there's not enough available
- Easy to forget this check, annoying to have to do it every time `malloc` is called...



Memory errors

- Using memory that you have not initialized
- Using memory that you do not own
- Using more memory than you have allocated
- Using faulty heap memory management



Structs

Structs

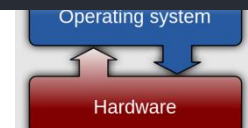
- Structs organize and group variables in a container so that they're easily accessible by a single pointer.
- As in other languages, creating objects is extremely helpful in keeping your abstractions clean!
- Let's analyze the code
 - We can declare a struct type by using the `struct name {fields};` syntax.
 - To access fields of a struct value, we can use the `.` (dot) syntax.
 - To access fields of a struct pointer, we have two choices
 - We can dereference the pointer to get a struct value and then use the `.` (dot) notation
 - Or we can use the arrow notation `->` to quickly do the first option
 - The arrow notation is probably the cleaner and quicker method

```
// Declare a struct type
struct coord {
    int x;
    int y;
};

int main(int argc, char *argv[]) {
    // Declare a struct on the stack
    struct coord c1;
    // We can access and assign fields using the . syntax
    c1.x = 3;
    c1.y = 4;

    // Declare a struct on the heap
    struct coord *c2 = malloc(sizeof(struct coord));
    // This dereferences the struct pointer to get the struct
    // and accesses its field x
    (*c2).x = 3;
    // When we have a pointer to a struct we can use
    // the arrow syntax to quickly reference its fields
    c2->y = 4;

    // No change because a copy of the struct is passed
    modify1(c1);
    modify1(*c2);
    printf("modify1\n");
    printf("x: %d, y: %d\n", c1.x, c1.y);
    printf("x: %d, y: %d\n", c2->x, c2->y);
}
```



Structs

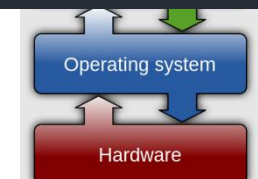
- When we pass struct values into functions (such as `modify1`), they are copied
 - This means any changes we make to that struct are not reflected in the original struct
- When we pass struct pointers into functions (such as `modify2`), the original struct may be modified
 - Since we have a pointer, we can go to the location of the struct and modify that struct

```
// Change because we passed a pointer to the struct
modify2(&c1);
modify2(c2);
printf("modify2\n");
printf("x: %d, y: %d\n", c1.x, c1.y);
printf("x: %d, y: %d\n", c2->x, c2->y);

return 0;
}

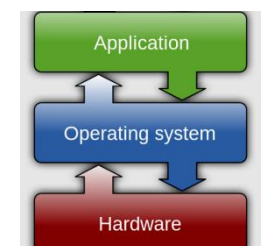
// structs are copied if given the struct itself
void modify1(struct coord c) {
    c.x = 5;
    c.y = 6;
}

void modify2(struct coord *c) {
    c->x = 5;
    c->y = 6;
}
```



C structures: aggregate, yet scalar

- aggregate in that they hold multiple data items at one time
 - named members hold data items of various types
 - like the notion of class/field in C++
 - but without the data hiding features
- scalar in that C treats each structure as a unit
 - as opposed to the “array” approach: a pointer to a collection of members in memory
 - entire structures (not just pointers to structures) may be passed as function arguments, assigned to variables, etc.
 - Interestingly, they cannot be compared using ==
 - (rationale: too inefficient)

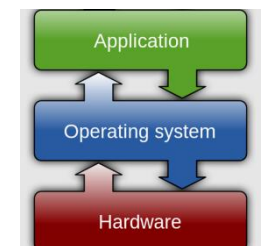


Structure Declarations

- Combined variable and type declaration
`struct tag {member-list} variable-list;`
- Any one of the three portions can be omitted

```
struct { int a, b; char* p; } x, y; /* omit tag (type name) */
```

- variables x, y declared with members as described:
 - int members a, b and char pointer p.
- x and y have same type, but differ from all others –
 - even if there is another declaration:
 - `struct { int a, b; char *p; } z;`
 - `/* z has different type from x, y */`



Structure Declarations

```
struct S { int a, b; char* p; }; /* omit variables */
```

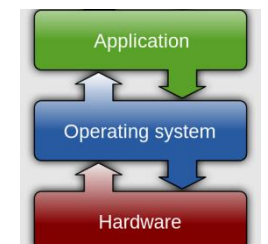
- No variables are declared, but there is now a type struct S that can be referred to later

```
struct S z; /* omit members */
```

- Given an earlier declaration of struct S, this declares a variable of that type

```
typedef struct { int a, b; char* p; } S;  
/* omit both tag and variables */
```

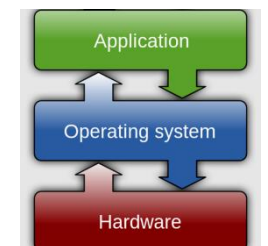
- This creates a simple type name S
 - (more convenient than struct S)



Recursively defined Structures

- Obviously, you can't have a structure that contains an instance of itself as a member – such a data item would be infinitely large
- But within a structure you can refer to structures of the same type, via pointers

```
struct TREENODE {  
    char *label;  
    struct TREENODE *leftchild, *rightchild;  
}
```

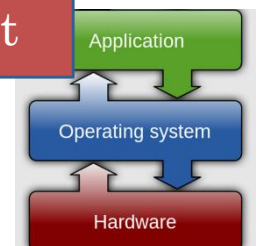


Recursively defined Structures

- When two structures refer to each other, one must be declared in incomplete (prototype) fashion

```
struct HUMAN;  
struct PET {  
    char name[NAME_LIMIT];  
    char species[NAME_LIMIT];  
    struct HUMAN* owner;  
} fido = {"Fido", "Canis lupus familiaris", NULL};  
struct HUMAN {  
    char name[NAME_LIMIT];  
    struct PET pets[PET_LIMIT];  
} sam = {"Sam", {fido}};
```

We can't initialize the **owner** member at this point, since it hasn't been declared yet

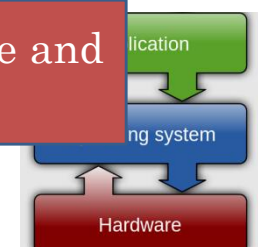


Member Access

- Direct access operator `s.m`
 - subscript and dot operators have same precedence and associate left-to-right, so we don't need parentheses for `sam.pets[0].species`
- Indirect access `s->m`: equivalent to `(*s).m`
 - Dereference a pointer to a structure, then return a member of that structure
 - Dot operator has higher precedence than indirection operator, so parentheses are needed in `(*s).m`
 - `(*fido.owner).name` or `fido.owner->name`

`.` evaluated first: access **owner** member
`*` evaluated next: dereference pointer to **HUMAN**

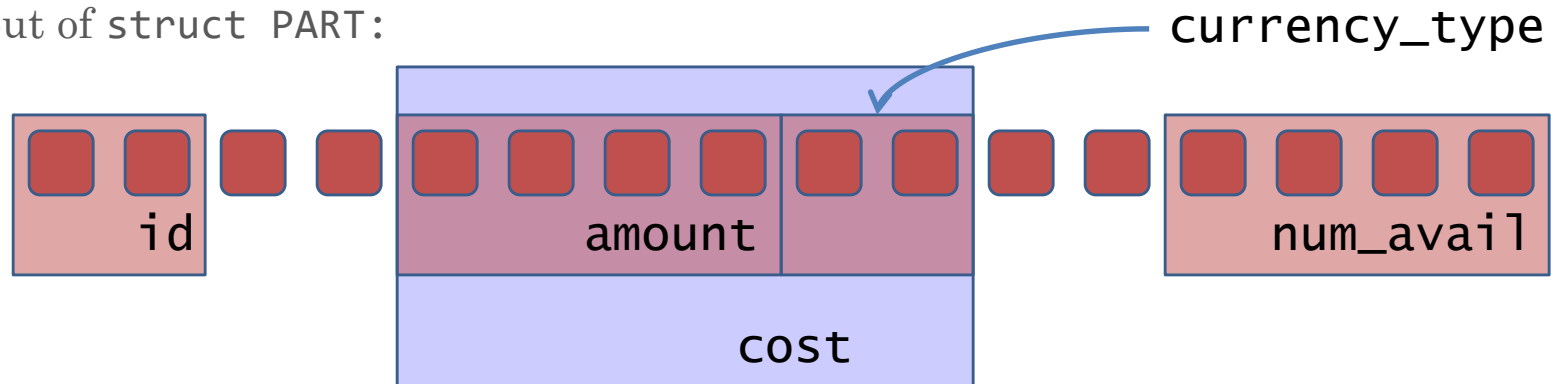
`.` and `->` have equal precedence and associate left-to-right



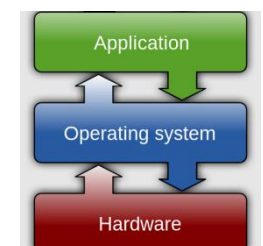
Memory Layout

```
struct COST { int amount;  
              char currency_type[2]; };  
struct PART { char id[2];  
              struct COST cost;  
              int num_avail; };
```

- layout of struct PART:



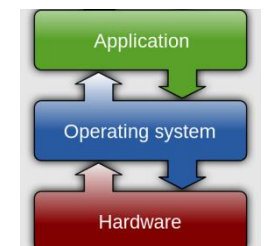
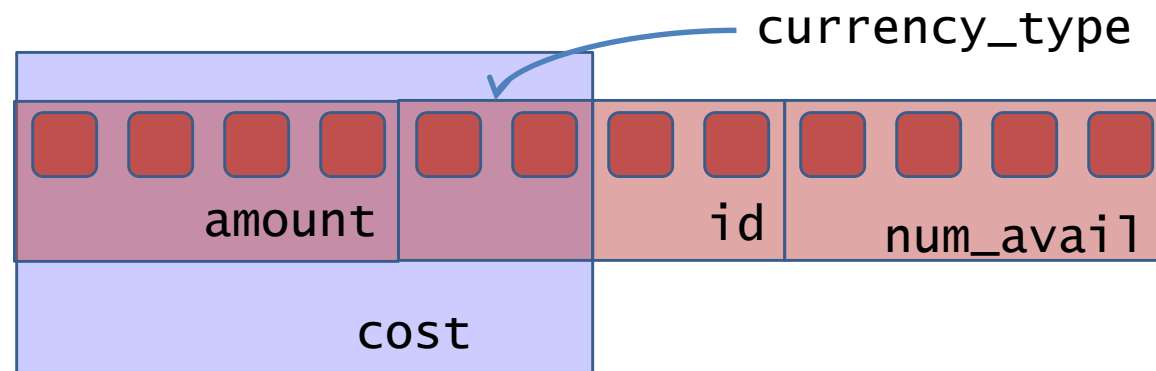
- Here, the system uses 4-byte alignment of integers,
 - so amount and num_avail must be aligned
- Four bytes wasted for each structure!



Memory layout

- A better alternative (from a space perspective):

```
struct COST { int amount;  
              char currency_type[2]; };  
  
struct PART { struct COST cost;  
              char id[2];  
              int num_avail; };
```



Structures as Function Arguments

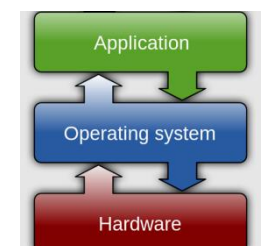
- Structures are scalars, so they can be returned and passed as arguments – just like ints, chars

```
struct BIG changestruct(struct BIG s);
```

- Call by value: temporary copy of structure is created
- Caution: passing large structures is inefficient
 - – involves a lot of copying
- avoid by passing a pointer to the structure instead:

```
void changestruct(struct BIG* s);
```
- What if the struct argument is read-only?
 - Safe approach: use const

```
void changestruct(struct BIG const* s);
```



OOP with C

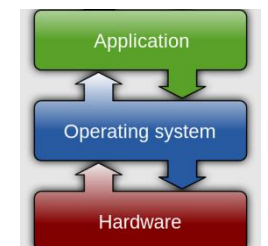
```
struct cost { int amount; char currency_type; };
```

```
struct cost* cost_init() { return malloc(sizeof(cost)); }
```

```
void cost_free(struct cost* c) { free(c); }
```

```
void cost_set(struct cost* c, int amount, char type)  
    { c->amount = amount; c->currency_type = type; }
```

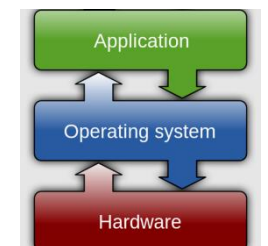
```
struct cost* c = cost_init();  
if (c == NULL) { ... error ... }  
cost_set(c, 42, '$');  
cost_free(c);
```



Unions

- Like structures, but every member occupies the same region of memory!
 - Structures: members are placed consecutively in memory
 - Unions: members are place in the same spot in memory

```
union VALUE {  
    float f;  
    int i;  
    char *s;  
};  
/* either a float or an int or a char* */
```

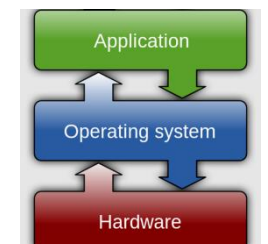


Unions

- Up to programmer to determine how to interpret a union (i.e. which member to access)
- Often used in conjunction with a “type” variable that indicates how to interpret the union value
 - Called ‘discriminated union’

```
enum TYPE { INT, FLOAT, STRING };  
struct VARIABLE {  
    enum TYPE type;  
    union VALUE value;  
};
```

Access **type** to determine
how to interpret **value**



Strings

Strings

- A string is an array of characters ending with a null terminator `\0`.
- So, we can represent strings as a `char *` type (an array of chars).
- Let's analyze the code
 - We can declare a string as an array using either syntax, in this case we chose the pointer syntax.
 - In `print_chars`, we can iterate through the string by using the same array dereferencing and pointer arithmetic method.
 - In `str_copier`, notice that `strlen` returns the length of the string excluding the null terminator.
 - If we didn't have the `+ 1` in our `malloc` call, then we wouldn't have enough space to fit both help and the `\0`

```
int main(int argc, char *argv[]) {
    // Declare a string
    char *str = "Help";
    print_chars(str);
    printf("Copied str: %s\n", str_copier(str));
    return 0;
}

// Iterates through the string and prints out each char
void print_chars(char *str) {
    int length = strlen(str);
    int i = 0;
    while (i < length) {
        // We can get a given char
        // by dereferencing the string like an array
        printf("Char: %c\n", *(str + i));
    }
}

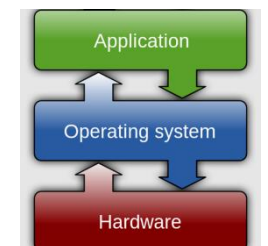
// Returns a malloced copy of the string
char *str_copier(char *str) {
    // strlen gets the length of the string
    // WITHOUT the null terminator
    int length = strlen(str);

    // Let's create the new buffer, NOTICE THE + 1
    // We need the +1 to make room for the null terminator
    char *copied = malloc(sizeof(char) * (length + 1));

    // Copies str into copied
    strcpy(copied, str);
    return copied;
}
```

Review of strings

- Sequence of zero or more characters, terminated by NUL (literally, the integer value `'\0'`)
- NUL terminates a string, but isn't part of it
 - important for `strlen()` – length doesn't include the NUL
- Strings are accessed through pointers/array names
- `string.h` contains prototypes of many useful functions



String literals

- Evaluating "dog" results in memory allocated for three characters 'd', 'o', 'g', plus terminating NUL

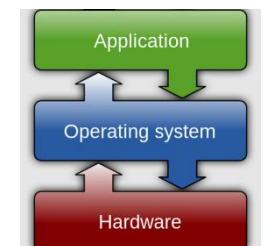
```
char* m = "dog";
```

- Note: If m is an array name, subtle difference:

```
char m[10] = "dog";
```

10 bytes are allocated for this array

This is not a string literal;
It's an array initializer in disguise!
Equivalent to
{ 'd', 'o', 'g', '\0' }



String manipulation functions

- Read some “source” string(s), possibly write to some “destination” location

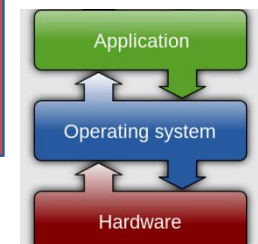
```
char* strcpy(char* dst, char const* src);
```

```
char* strcat(char* dst, char const* src);
```

- Programmer’s responsibility to ensure that:
 - destination region large enough to hold result
 - source, destination regions don’t overlap
 - “undefined” behavior in this case
 - according to C spec, anything could happen!

```
char m[10] = "dog";  
strcpy(m+1, m);
```

Assuming that the implementation of **strcpy** starts copying left-to-right without checking for the presence of a terminating NUL first, what will happen?



strlen() and size_t

```
size_t strlen(char const* string);  
/* returns length of string */
```

- `size_t` is an unsigned integer type, used to define sizes of strings and (other) memory blocks
 - Reasonable to think of “size” as unsigned...
 - But beware! Expressions involving `strlen()` may be unsigned (perhaps unexpectedly)

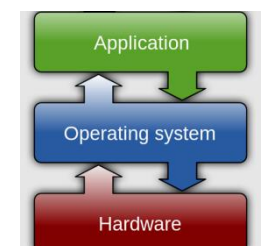
```
if (strlen(x) - strlen(y) >= 0) ...
```

- avoid by casting:

```
((int) (strlen(x) - strlen(y)) >= 0)
```

- Problem: what if `x` or `y` is a very large string?
- a better alternative: `(strlen(x) >= strlen(y))`

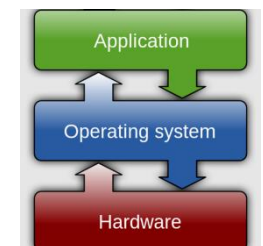
always true!



strcmp() “string comparison”

```
int strcmp(char const* s1, char const* s2);
```

- returns a value less than zero if `s1` precedes `s2` in lexicographical order;
 - returns zero if `s1` and `s2` are equal;
 - returns a value greater than zero if `s1` follows `s2`.
- Source of a common mistake:
 - seems reasonable to assume that `strcmp` returns “true” (nonzero) if `s1` and `s2` are equal; “false” (zero) otherwise
 - In fact, exactly the opposite is the case!



Restricted vs. unrestricted string functions

- Restricted versions: require an extra integer argument that bounds the operation

```
char* strncpy(char* dst, char const* src, size_t len);
```

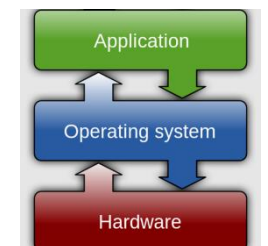
```
char* strncat(char* dst, char const* src, size_t len);
```

```
int strncmp(char const* s1, char const* s2, size_t len);
```

- “safer” in that they avoid problems with missing NUL terminators
- safety concern with strncpy:
 - If bound isn’t large enough, terminating NUL won’t be written
 - Safe alternative:

```
strncpy(buffer, name, BSIZE);
```

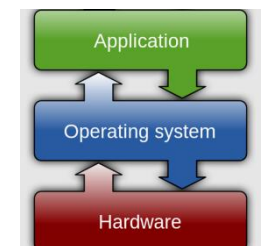
```
buffer[BSIZE-1] = '\0';
```



String searching

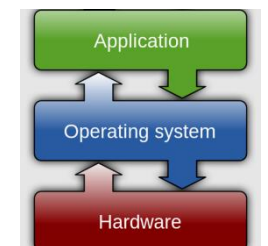
```
char* strpbrk(char const* str, char const* group);  
/* return a pointer to the first character in str  
   that matches *any* character in group;  
   return NULL if there is no match */
```

```
size_t* strspn(char const* str, char const* group);  
/* return number of characters at beginning of str  
   that match *any* character in group */
```



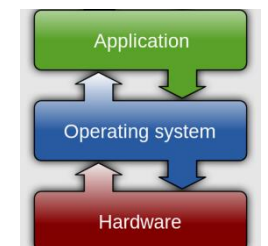
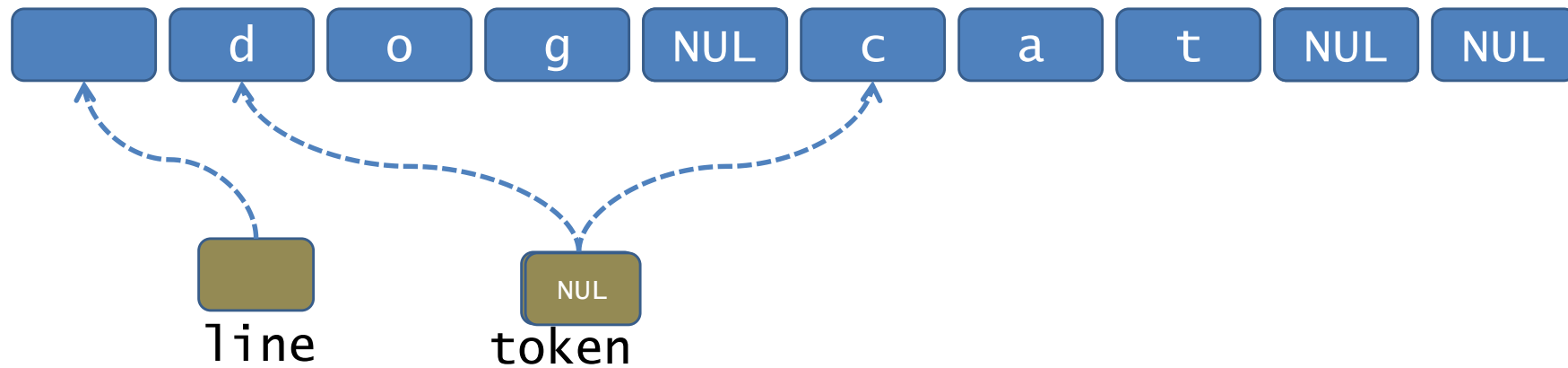
strtok “string tokenizer”

```
char* strtok(char* s, char const* delim);  
/* delim contains all possible "tokens":  
   characters that separate "tokens".  
  
   if delim non-NULL:  
       return ptr to beginning of first token in s,  
       and terminate token with NUL.  
  
   if delim is NULL:  
       use remainder of untokenized string from the  
       last call to strtok */
```



strtok in action

```
for (token = strtok(line, whitespace);  
    token != NULL;  
    token = strtok(NULL, whitespace))  
    printf("Next token is %s\n", token);
```



An implementation of strtok

```
char* strtok(char* s, const char* delim) {  
    static char *old = NULL;  
    char *token;  
    if (! s) { s = old; if (! s) return NULL; }  
    if (s) {  
        s += strspn(s, delim);  
        if (*s == 0) { old = NULL; return NULL; }  
    }  
    token = s;  
    s = strpbrk(s, delim);  
    if (s == NULL) old = NULL;  
    else { *s = 0; old = s + 1; }  
    return token;  
}
```

`old` contains the remains
of an earlier `s` value
(note use of `static`)

`NULL` has been passed in for `s`,
so consult `old`

`strspn` returns number of delimiters
at beginning of `s` – skip past these
characters

`strpbrk` gives the position of the next delimiter.
`s` is updated to this position, but `token` still
points to the token to return.



Memory operations

- Like string operations, work on sequences of bytes
 - but do not terminate when NUL encountered
- ```
void* memcpy(void* dst, void const* src, size_t length);
void* memcmp(void const* a, void const* b, size_t length);
```
- Note: memmove works like memcpy, but allows overlapping source, destination regions
  - Remember, these operations work on bytes
    - If you want to copy N items of type T, get the length right:  
    memcpy(to, from, N \* sizeof(T))

