A C Primer (Part II)

Overview

- Compound Types
  - Arrays
  - Structures
  - Pointers
- Pointer arithmetic
- Memory layout and alignment

Arrays

- Is a type constructor
  - It produces a new type
- As expected
  - Arrays represent a linear, contiguous collection of “things”
  - Each “thing” in the array has the **same fixed** type.
- Examples
  - Array of characters
  - Array of integers
  - Array of booleans
  - Array of structures
  - Arrays of arrays….

Array Example 1

- Simple array of integers

```
int main()
{
   int x[5];
   x[0] = 1;
   x[1] = 2;
   x[2] = 3;
   x[3] = 4;
   x[4] = 5;
   return 0;
}
```

0	1	2	3	4
1	2	3	4	5

Arrays are 0-based
First is @ offset 0
Last is @ offset n-1

A Few Questions

- Where is the array allocated ?
- When is the array allocated ?
- When is the array deallocated ?
- What about recursion ?
- What happens if you try to access x[5] ?
- What happens if you try to access x[-1] ?
- What if you do not know the size at *compile time*?

A Few Questions

- Where is the array allocated ?

  On the stack, so it is automatic
- When is the array allocated ?

  It is allocated when you enter the function
- When is the array deallocated ?

  When the function returns
- What about recursion ?

  Each invocation gets its own copy! ;-)
- What happens if you try to access x[5] ?

  Oooooh…… you are accessing memory that is not yours!
- What happens if you try to access x[-1] ?

  Same as above!
- What if you do not know the size at *compile time*?

  We can deal with this later on …. [dynamic allocation]

Array Example 2

- A character array

```c
#include <stdio.h>
int main()
{
    char s[6];
    s[0] = 'H';
    s[1] = 'e';
    s[2] = 'l';
    s[3] = 'l';
    s[4] = 'o';
    s[5] = '\0';
}
```

0	1	2	3	4	5
H	e	l	l	o	0

Same as int array!
First is @ offset 0
Last is @ offset n-1
Last character is \0

Array Example 2

- A character array

```c
#include <stdio.h>
int main()
{
   char s[6];
   s[0] = 'H';
   s[1] = 'e';
   s[2] = 'l';
   s[3] = 'l';
   s[4] = 'o';
   s[5] = '\0';
   printf("Array is: %s\n",s);
}
```

0	1	2	3	4	5
H	e	l	l	o	0

A 0-terminated character array is…

A String!

Array Example 2

- A character array. Convenience initialization.

```
#include <stdio.h>
int main()
{
   char s[6] = {‘H’,’e’,’l’,'l','o','\0'};
   printf("Array is: %s\n",s);
}
```

0	1	2	3	4	5
H	e	l	l	o	0

You can provide an initializer list…

It works for all types

Array Example 2

- A character array. Convenience initialization **again!**

```
#include <stdio.h>
int main()
{
   char s[6] = “Hello”;
   printf("Array is: %s\n",s);
}
```

0	1	2	3	4	5
H	e	l	l	o	0

Even better for **Strings**…

You can give the list of characters in a double-quoted literal

Array Example 2

- A character array. Convenience initialization **again!**

```c
#include <stdio.h>
int main()
{
   char s[] = “Hello”;
   printf("Array is: %s\n",s);
}
```

0	1	2	3	4	5
H	e	l	l	o	0

Even better for **Strings**…

You can even drop the size.
C will compute it from the initializer
(also true for int arrays)

Array Indexing

- Works as in Java
  - You can read or write anywhere inside the array
- Unlike Java
  - You can also read or write *outside the array*. Be very careful!
- Indexing
  - Must evaluate to an integer
  - Can be an expression
- Each definition yields a type

Array Assignment

- Unlike Java
  - You ***cannot*** assign a whole array at once to another array
  - Even when the types match
- Note [again]
  - In Java, arrays are heap allocated
  - In C, arrays can be stack allocated

```c
int main() {
    int x[10];
    int y[20];
    int z[10];
    x = y;
    x = z;
}
```

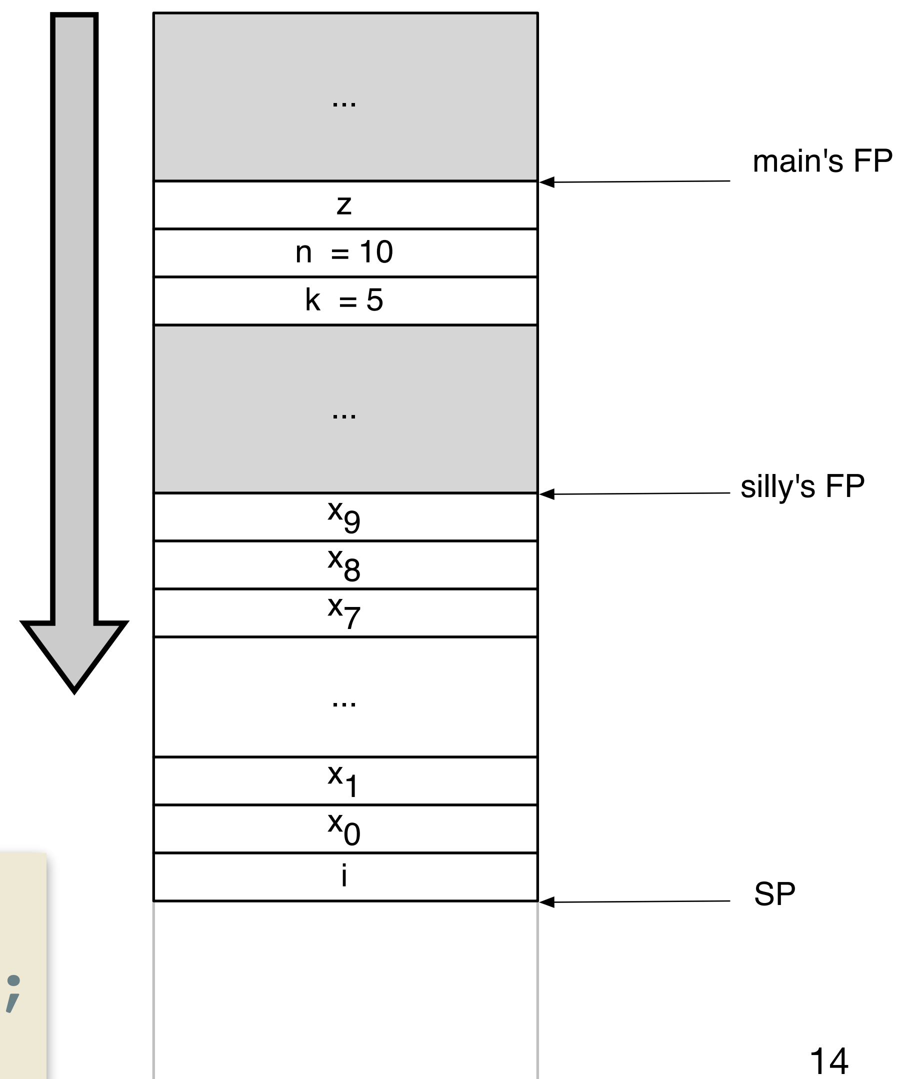
```
src (master) $ cc char3.c
char3.c:5:6: error: array type 'int [10]' is not assignable
   x = y;
   ~ ^
char3.c:6:6: error: array type 'int [10]' is not assignable
   x = z;
   ~ ^
2 errors generated.
```

Arrays as Automatic Variables

- Two Key Facts
    - You can declare arrays inside *any function*
    - The size of your array can depend on *function arguments* [dynamic!]

```
int silly(int n,int k) {
  int x[n];
  for(int i = 0;i < n;i++)
    x[i] = 0;
  x[0] = 0;
  x[1] = 1;
  for(int i = 2;i < n;i++)
    x[i] = x[i-1] + x[i-2];
  if (0 <= k && k < n)
    return x[k];
  else return -1;
}
```

```
int main() {
    int z = silly(10,5);
}
```



Automatic Array Summary

- Local Arrays
  - Allocated when entering the function [automatic]
  - Deallocated when leaving the function [automatic]
  - **NOT** initialized
  - Exist directly on the stack like other variables.
- Size
  - Can be static [a constant]
  - Can be dynamic [an argument to the function]
  - Cannot be too big since it is on the stack!

Arrays as Arguments to functions

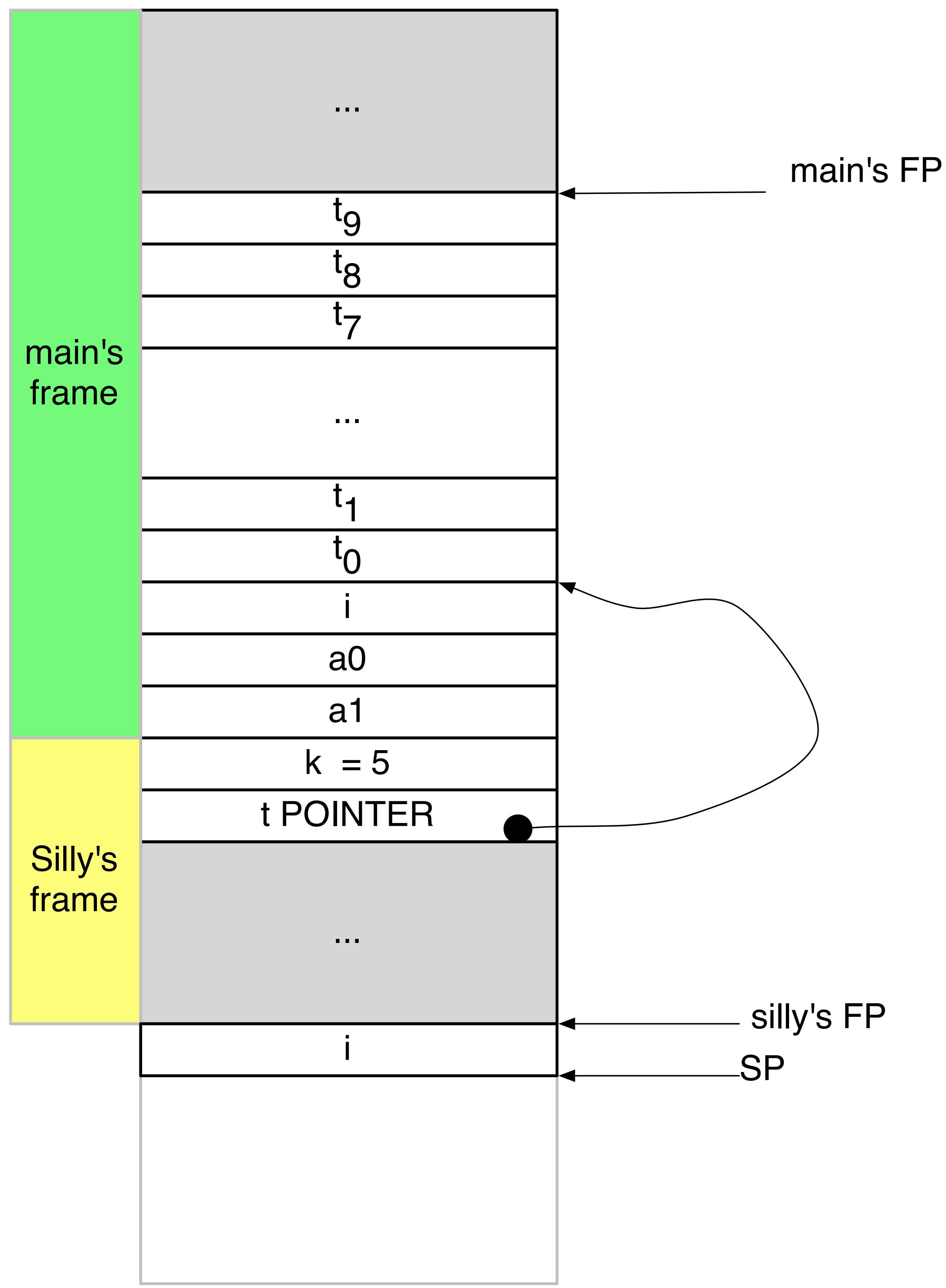
- Arrays can be passed to functions!
  - With one big caveat…
- Calling convention in C
  - BY VALUE for everything….
  - EXCEPT arrays…
- Arrays are always passed as “pointers”.
  - We have to look at pointers soon!

Arrays as Arguments to functions (Statically sized)

```
int silly(int k,int x[10]) {
  for(int i = 0;i < 10;i++)
    x[i] = 0;
  x[0] = 0;
  x[1] = 1;
  for(int i = 2;i < 10;i++)
    x[i] = x[i-1] + x[i-2];
  if (0 <= k && k < 10)
    return x[k];
  else return -1;
}

int main() {
  int t[10];
  int a0 = silly(5,t);
  int a1 = t[8];
  printf("got 2 values: %d ::  %d\n",a0,a1);
  return 0;
}
```

It looks like a declaration array passed by "address" modifications are visible!

Pictorially



Arrays as Arguments to functions (dynamically)

```
int silly(int n,int k,int x[n]) {
  for(int i = 0;i < n;i++)
    x[i] = 0;
  x[0] = 0;
  x[1] = 1;
  for(int i = 2;i < n;i++)
    x[i] = x[i-1] + x[i-2];
  if (0 <= k && k < n)
    return x[k];
  else return -1;
}

int main() {
  int nb = 10;
  int t[nb];
  int a0 = silly(10,5,t);
  int a1 = t[8];
  printf("got 2 values: %d ::  %d\n",a0,a1);
  return 0;
}
```

Very similar
array size is known though!
Still passed by “address”
modification still visible!

Arrays? Are we done?

- Nope…
  - More on arrays once we start working with pointers.

Compound Type

- Is a type *constructor*
  - It builds a brand new type

Compound Types

- Also known as “structures”

```c
#include <stdlib.h>

struct Person {
   int      age;
   char gender;
};

int main()
{
   struct Person p;

   p.age = 44;
   p.gender = 'M';
   return 0;
}
```

A structure declaration!   (a type)

A structure *definition*! (a value)

Compound Types

- Also known as "structures"

```
#include <stdlib.h>

struct Person {
   int      age;
   char gender;
};

int main()
{
   struct Person p = {44,'M'};

   return 0;
}
```

A structure declaration! (a type)

A structure *definition*! (a value) and *initialization*

Structures

- Declaration
  - Structure have a type **name**
  - Can have multiple fields
  - Fields can have any legal type
    - Basic types
    - Structures
    - Arrays
    - [and the other types left to discover!]
- Definition
  - Define a value.
  - Value lives on the stack (automatically deallocated on return)

Composing Types

- You can compose
  - Basic Types
  - Structure Types [Compounds]
  - Array Types
- A more realistic struct example

Realistic Example

- Embed an array in the structure for the person’s name
- Make an array of structures of type Person for the whole family.
- Nest initializers
- Caveats
  - Names cannot be > 32 long.
  - Four persons in family
    - Indexed 0..3

```
#include <stdlib.h>

struct Person {
   int         age;
   char     gender;
   char name[32];
};

int main()
{
   struct Person family[4] = {
      {50,’M',"Darth Vader"},
      {49,’F',"Padmé"},
      {21,'F',"Leia"},
      {19,'M',"Luke"}
   };
   int kidAge = family[3].age;
   return 0;
}
```

Type Definitions

- Type names can become long
- C provide the ability to define type abbreviations
    - typedef declaration
        - Give existing type
        - Give new type name
    - Use the new type anywhere
- Useful to make code even more readable

```c
struct Person {
   int         age;
   char    gender;
   char name[32];
};

typedef struct Person TPerson;

int main()
{
   TPerson family[4] = {
      {50,’M’,"Darth Vader"},
      {49,’F’,"Padmé"},
      {21,'F',"Leia"},
      {19,'M',"Luke"}
   };

   return 0;
}
```

Overview

- Basic Types
- Compound Types
  - Arrays
  - Structures
  - Pointers
- Pointer arithmetic
- Memory layout and alignment

Pointers

- Perhaps the scariest part of C
- Yet….
  - The most useful part of C!

- Pointer is simply….
  - A value
  - Denoting the address of a memory cell

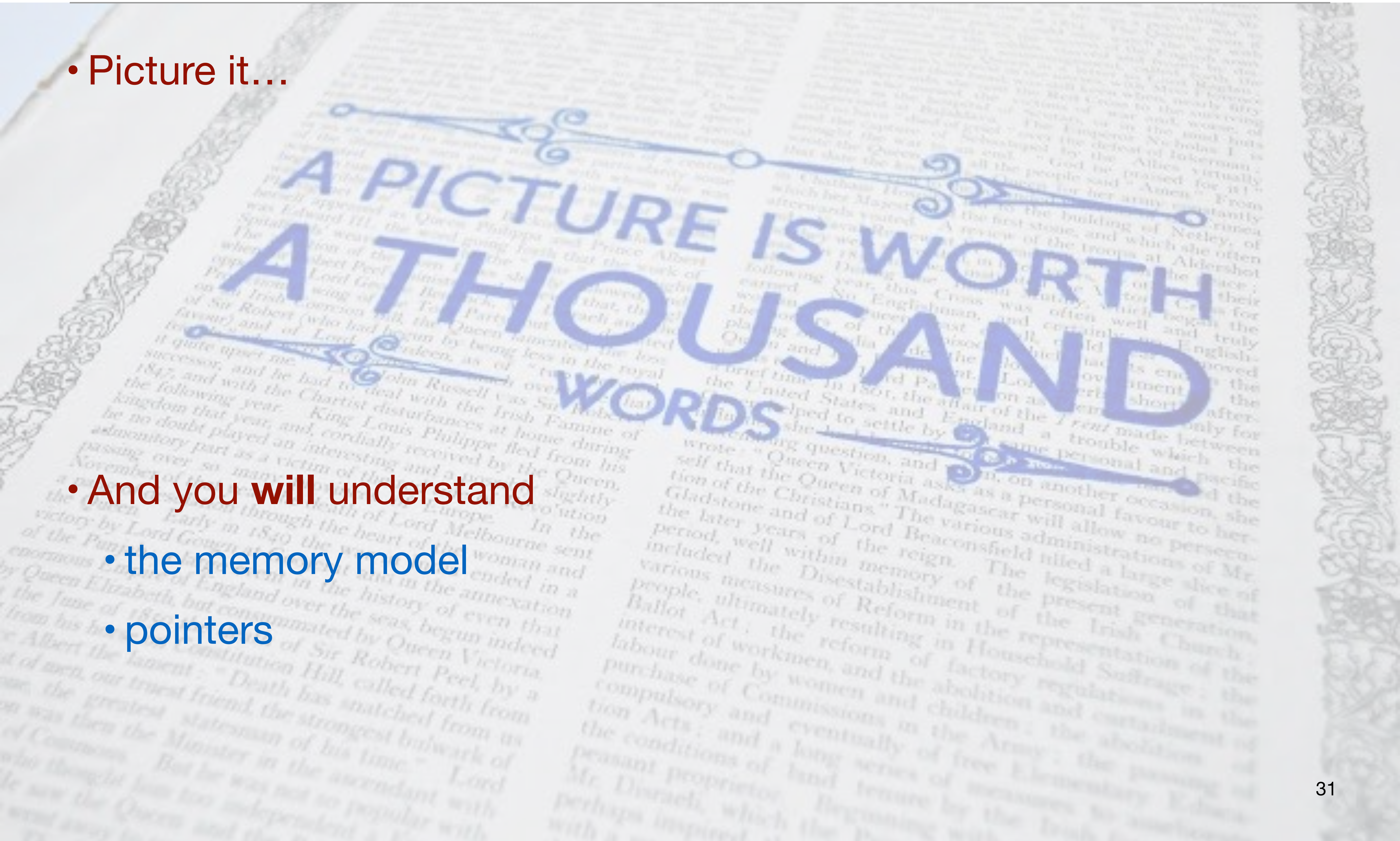
Bottom Line

The idea

“*Pointers are a difficult concept*” is a

MYTH

Key Insight…



- Picture it…
- And you **will** understand
  - the memory model
  - pointers

Pointers Usage

- What can you do ?
  - Get the “address” of something
  - Dereference an “address” to get to something
  - Compute the address of something

Memory Model...

- Three pools of memory
  - Static
  - Stack
  - Heap
- Each pool features
  - Different lifetime
  - Different allocation/deallocation policy

Why does it matter?

Static Memory Pool

- This is where
  - All constants are held
  - All strings in the program are held
  - All variables declared “static” are held
- Allocated when
  - The program start
- Deallocated when
  - The program terminates
- Bottom line
  - FIXED SIZE

Rule for this class

DO NOT USE STATIC

NEVER, EVER, EVER

(repeat on a T. Swift tune…)

Automatic Memory Pool

- This is where….
  - Memory comes from for ***local variables*** of functions!
  - Allocated automatically when entering the function
  - De-allocated automatically when you leave the function.
- Recursion ?
  - Each recursive invocation gets its own set of local variables!
- Bottom line
  - It’s easy to manage (automatic!)
  - It’s variable over time
  - Scope is that of function.

Dynamic Memory Pool

- This is where…
  - Memory comes from for manual "on-the-fly" allocations
  - Two simple APIs (at the core)
    - malloc(b) [allocate a block of 'b' bytes]
    - free(p) [frees a previously allocated block]
- Who is in charge ?
  - The programmer for both allocation / deallocation
- Lifetime of memory blocks ?
  - As long as they are not freed!

Pointers &

- Taking the address of….
  - A static ?
    - The address is never going to go “bad”
    - The static lives as long as the program!
  - A stack [automatic] variable ?
    - The address is valid as long as the variable is!
    - When the function returns…. The address is bogus
  - A heap variable ?
    - The address is valid as long as the variable is!
    - The variable disappear when explicitly de-allocated (freed)

Pointer examples

- Simple deal
  - & “makes” an address
  - * “dereferences” an address
- It works for all types

```
int x = 10;

int* px = &x;

*px = 20;
```

Declaration

Notation

- Word to the wise…
  - The following three are identical

```
int*     p;
int   *  p;
int     *p;
```

- They all declare…
  - p to be a pointer to an integer
- But
  - First one m
  - Second on
  - Third says

The first is very clear
But…
The third is classic C

Pitfall

- Read the following !

```
int    *a,b;
```

- What is...
  - a ?
  - b ?

How Much Space ?

- How do you determine the amount of space for some type?
  - You need this to dynamically allocate space!
- Easy!

```
sizeof(T)
```

- Returns the number of bytes to hold a value of type T

Pointer and Dynamic Memory

- Pointers are useful when
  - Allocating memory at run time!

- Example
  - You need n integers (n known at runtime)….

```
void doSomething(int n)
{
  int* pox;

  pox = (int*)malloc(sizeof(int)*n);

  *pox = 0;   // What happens?

  free(pox);
}
```

Shorthand

- If you need to allocate an array
  - There is another library function called calloc
  - calloc is implemented in term of malloc
  - calloc also initialize the content to 0


```
void doSomething(int n)
{
  int* pox;

  pox = (int*)calloc(n,sizeof(int));

  *pox = 0;   // What happens?

  free(pox);
}
```

Deallocation

- Straightforward
  - Simply call the library function “free”
  - Takes a pointer to the block to free


```
free(ptr)
```

Best Practice

- Always remember two key rules

RULE #1:

Everything you malloc should eventually be freed

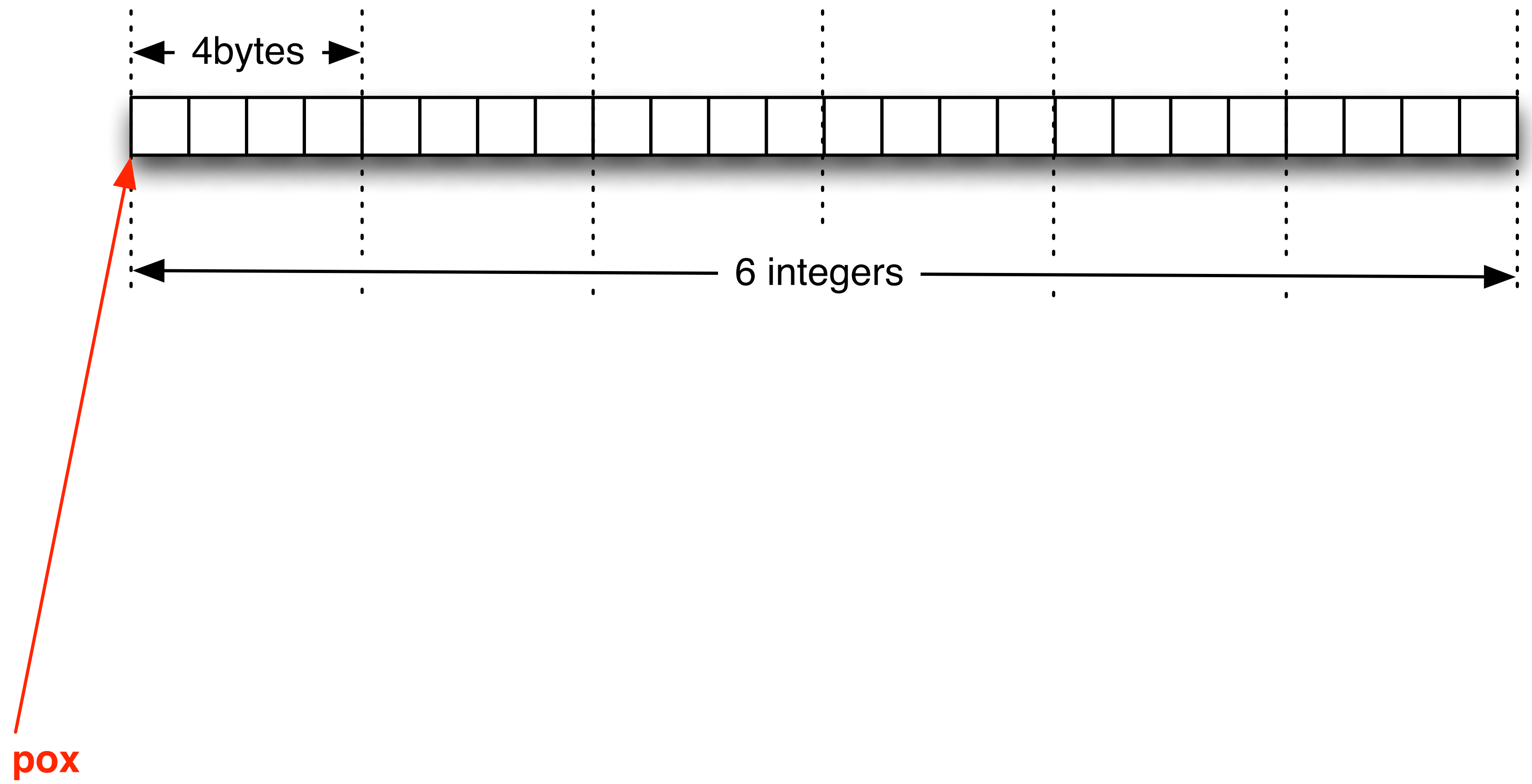
RULE #2:

Only free what is allocated via malloc / calloc

- Consequences of not remembering the rules
  - Memory "Leaks" [you will eventually run out of memory]
  - Double deletion and horrible crashes

Pointers and Arrays

- What really happened….



- Wait!!!!
  - That looks like an array!

Pointers and Arrays

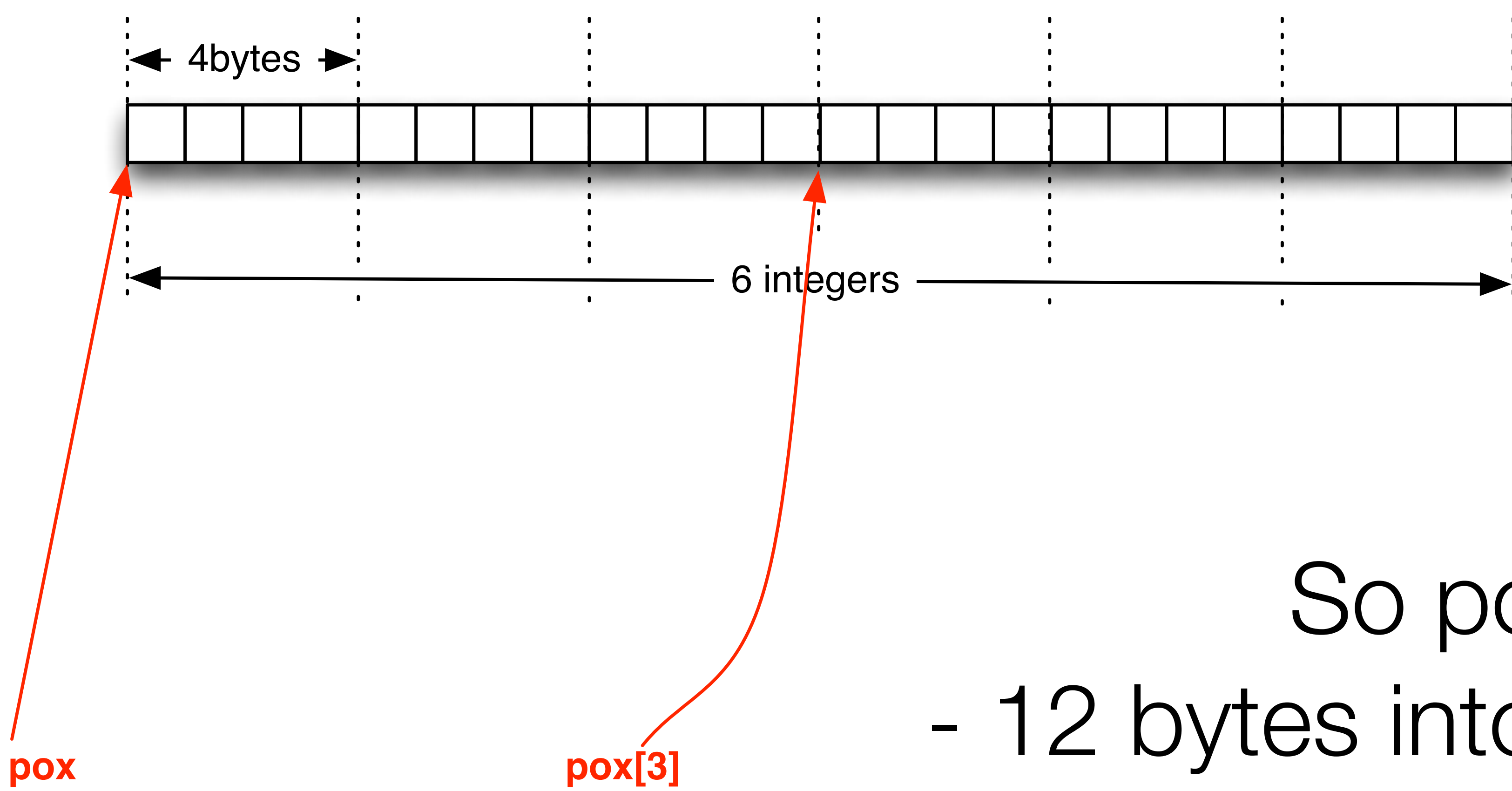
- Yes, pointers and arrays are the same thing
  - Array is represented by the address of 0th element!

```
void doSomething(int n) {
  int* pox;

  pox = (int*)malloc(sizeof(int)*n);

  //*pox = 0;   // What happens?

  pox[0] = 0;
  pox[1] = 1;
  …
  pox[n-1] = n-1;

  free(pox);
}
```

Pointer Arithmetic

- Since pointers and arrays are the same thing….
- What is going on when we write:

```
pox[3] = ...;
```

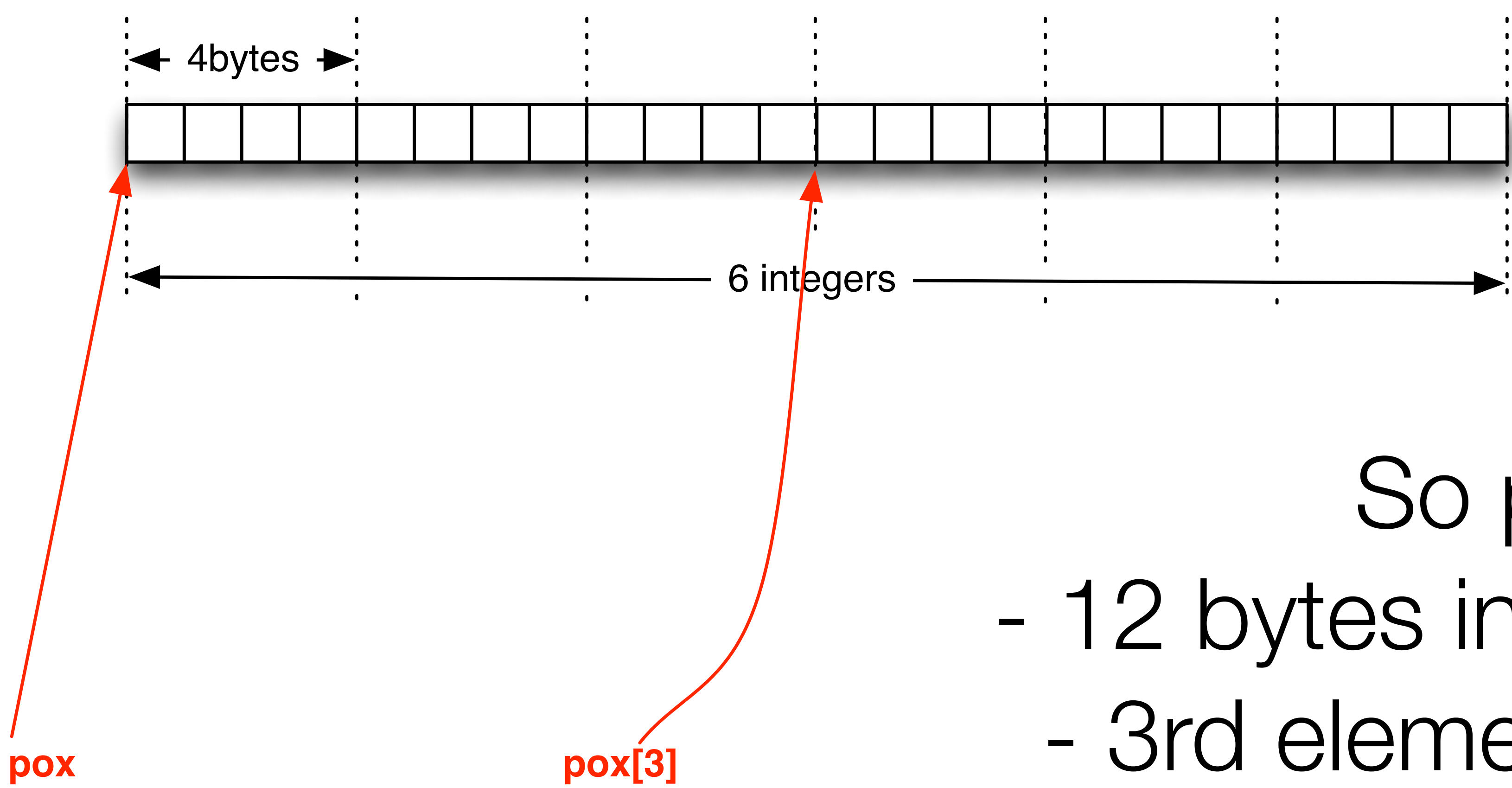


So pox[3] is
- 12 bytes into the pox “array”
- 3rd element of pox “array”

Pointer Arithmetic

- Since pointers and arrays are the same thing….
- What is going on when we write:

```
pox[3] = ...;
```



So pox[3] is

- 12 bytes into the pox "array"
- 3rd element of pox "array"

```
*(pox+3) = ...;  // arithmetic is type driven
```

Implication

- You can compute the address of anything
- You only need to
  - Know the base address
  - Know the offset
  - Know the types

Bounds

- When doing…

```
int n = 6;
int* pox;
pox = (int*)malloc(sizeof(int)*n);
```

- Index of
  - First element is 0
  - Last element is n-1 (6-1 = 5)
- Yet…
  - C allows reads and write
    - Before 0
    - After n-1

Initializing Pointers

- Ways to initialize a pointer
  - As a result of malloc

```
int *p  = (int*)malloc(sizeof(int)*10);
```

  - As a result of **&**

```
int a  = 10;
int *p = &a;
```

  - As a result of assigning NULL

```
int* t = NULL;
```

Best Practice

- You should know that….
  - A call to malloc may **fail**
    - Whenever you are out of virtual memory
    - You would get back the NULL value
    - Not much to do except report the error and terminate *nicely*.
- Idiom

```
int* p;
p = (int*)malloc(sizeof(int)* n);
if (p == NULL)
   report error and finish;
```

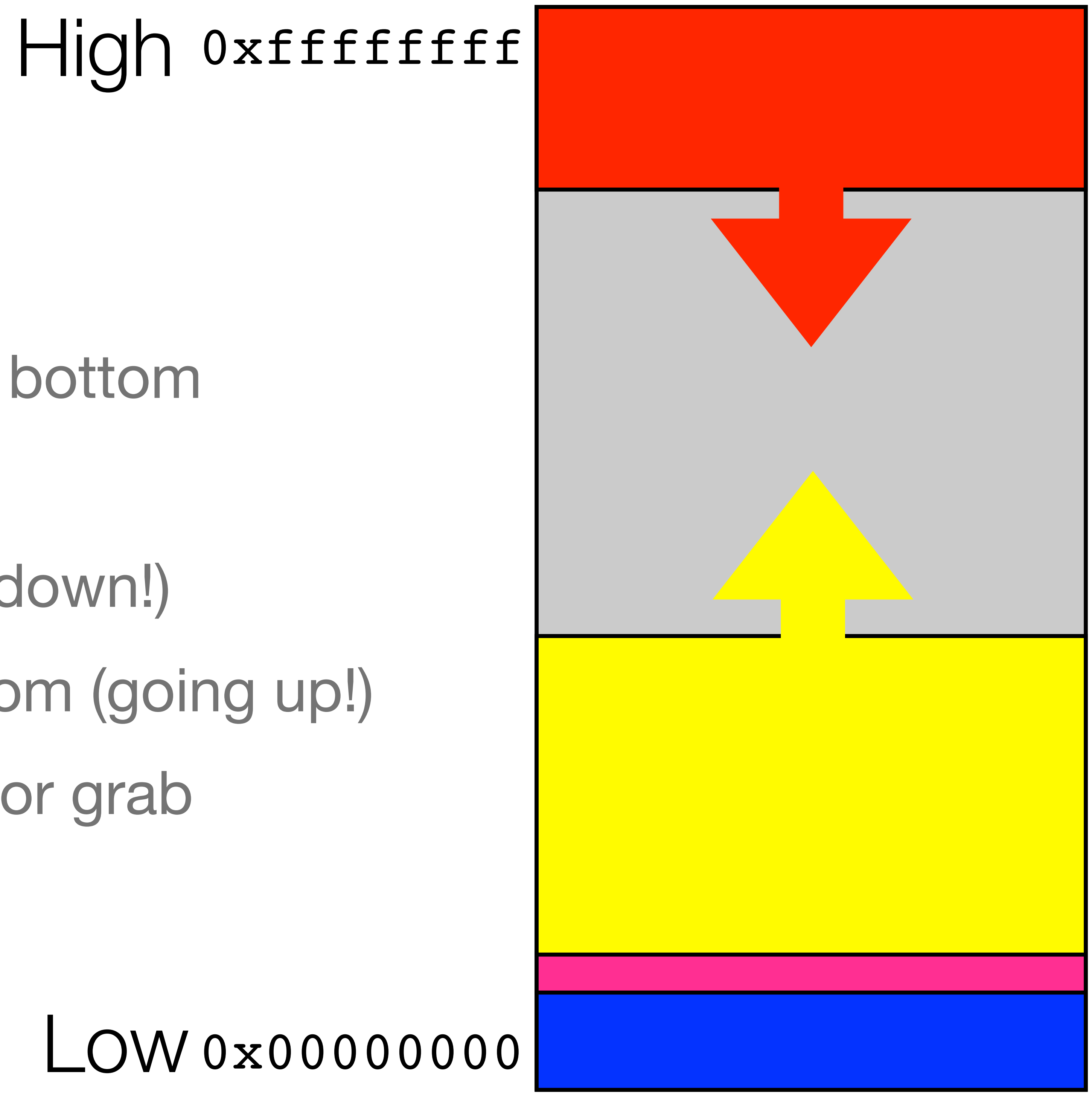
Back to the **scanf** example!

```
#include <stdio.h>

int main()
{
   char name[128];
   int pears  = 0;
   int apples = 0;
   scanf("%s %d %d",name,&pears,&apples);
   printf("%s ate %d apples and %d pears.\n",name,apples,pears);
   return 0;
}
```

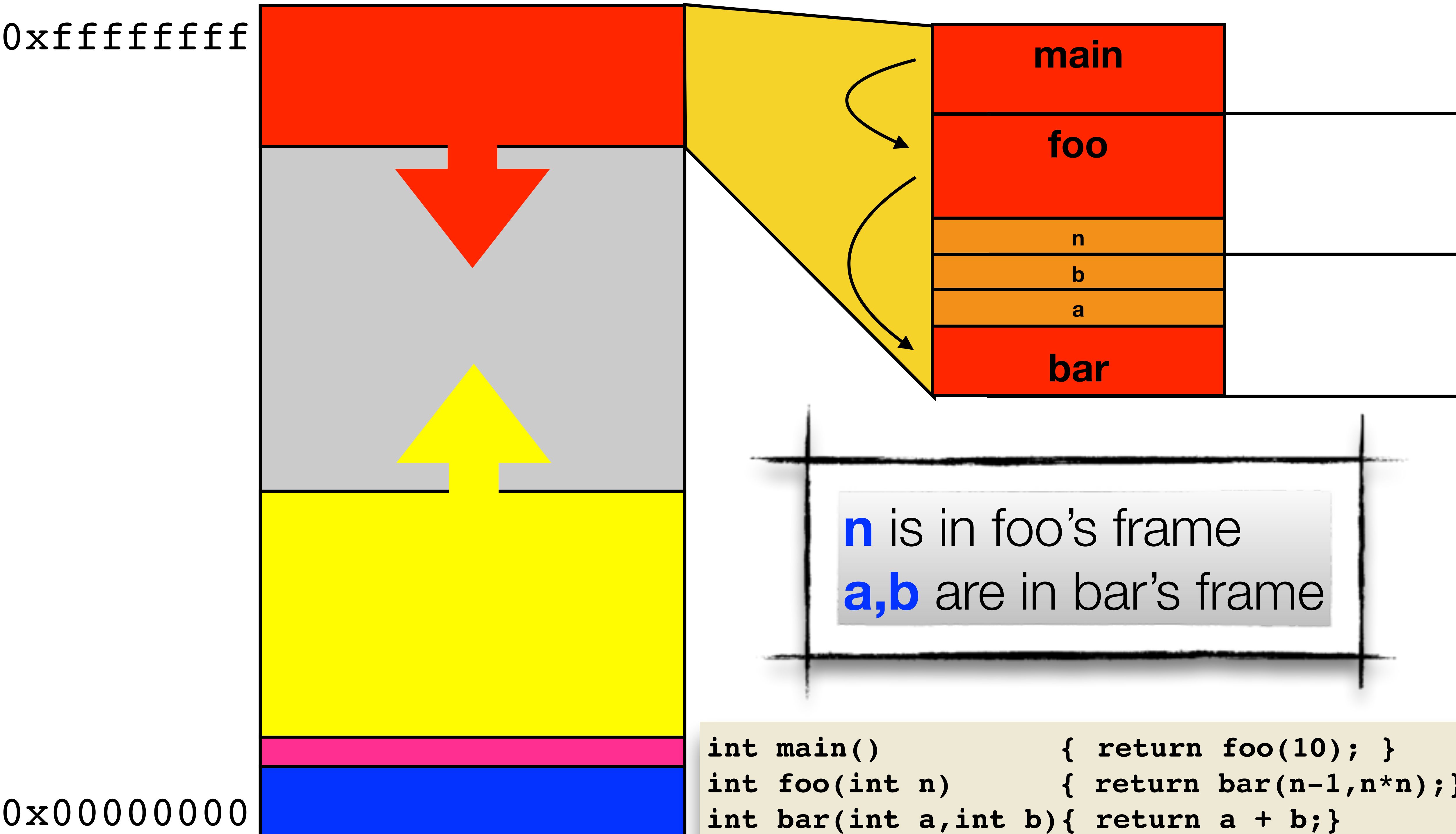
- What is going on ?
  - Pass to scanf three **VALUES**
    - **name**
    - **address of pears [&]**
    - **address of apples [&]**

In Picture (Refresher!)

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** are just above
  - **Stack** is at the top (going down!)
  - **Heap** grows from the bottom (going up!)
  - Gray no-man's land is up for grab



Zooming in [on the stack!]



n is in foo's frame
a,b are in bar's frame

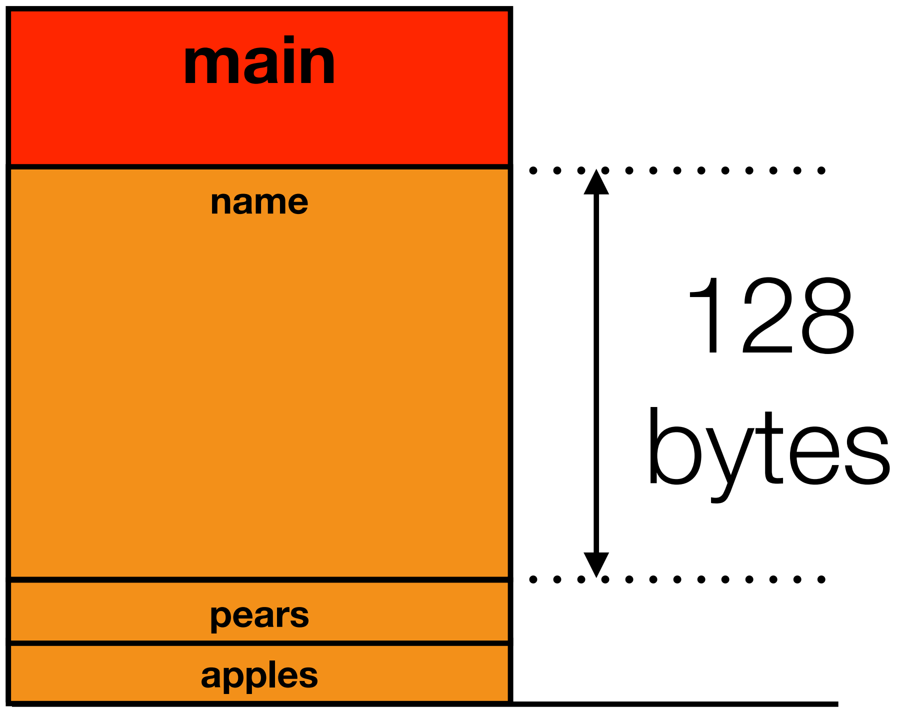
```
int main()           { return foo(10); }
int foo(int n)       { return bar(n-1,n*n);}
int bar(int a,int b){ return a + b;}
```

`scanf` Code

- The symbols are coming from ?
  - name ?
  - pears ?
  - apples ?
- What are their **types** ?
  - name ?
  - pears ?
  - apples ?

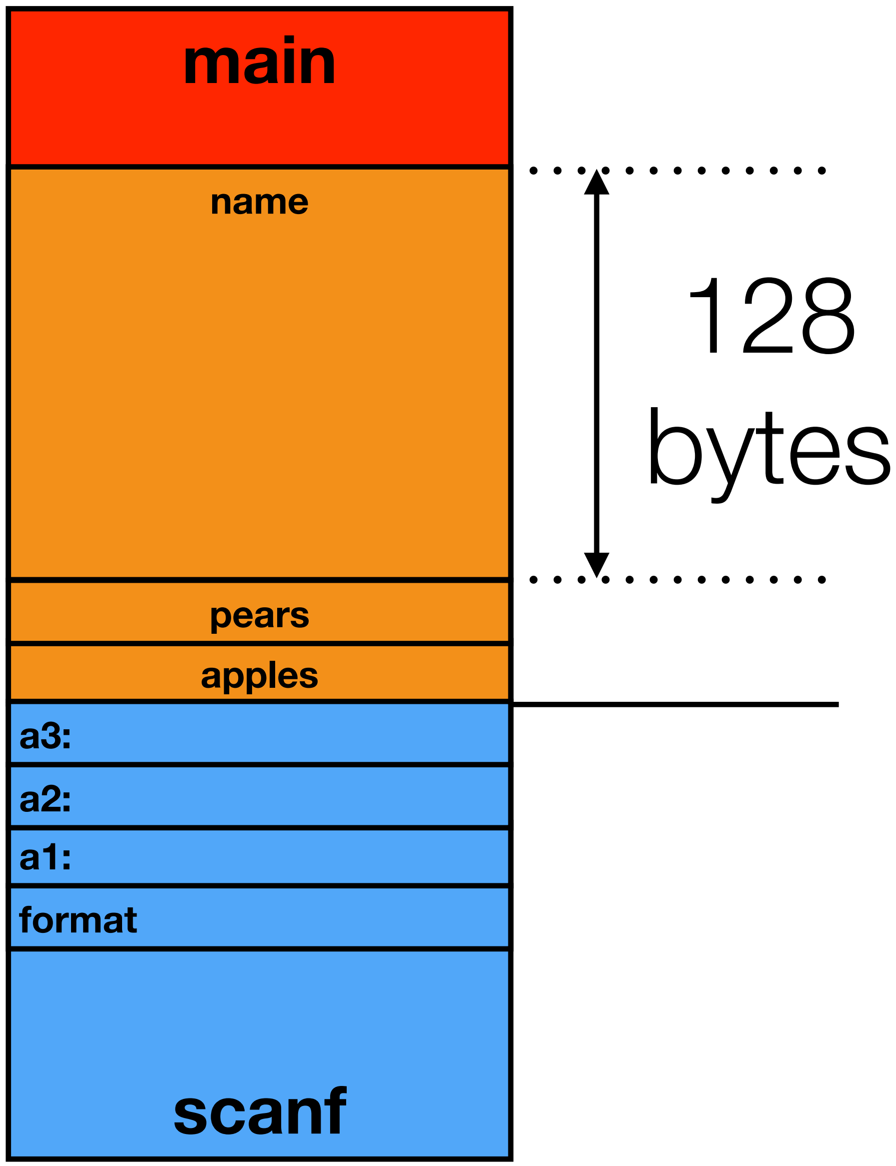
```c
#include <stdio.h>

int main()
{
    char name[128];
    int pears  = 0;
    int apples = 0;
    scanf("%s %d %d",name,&pears,&apples);
    …
}
```

Frame of main … In picture



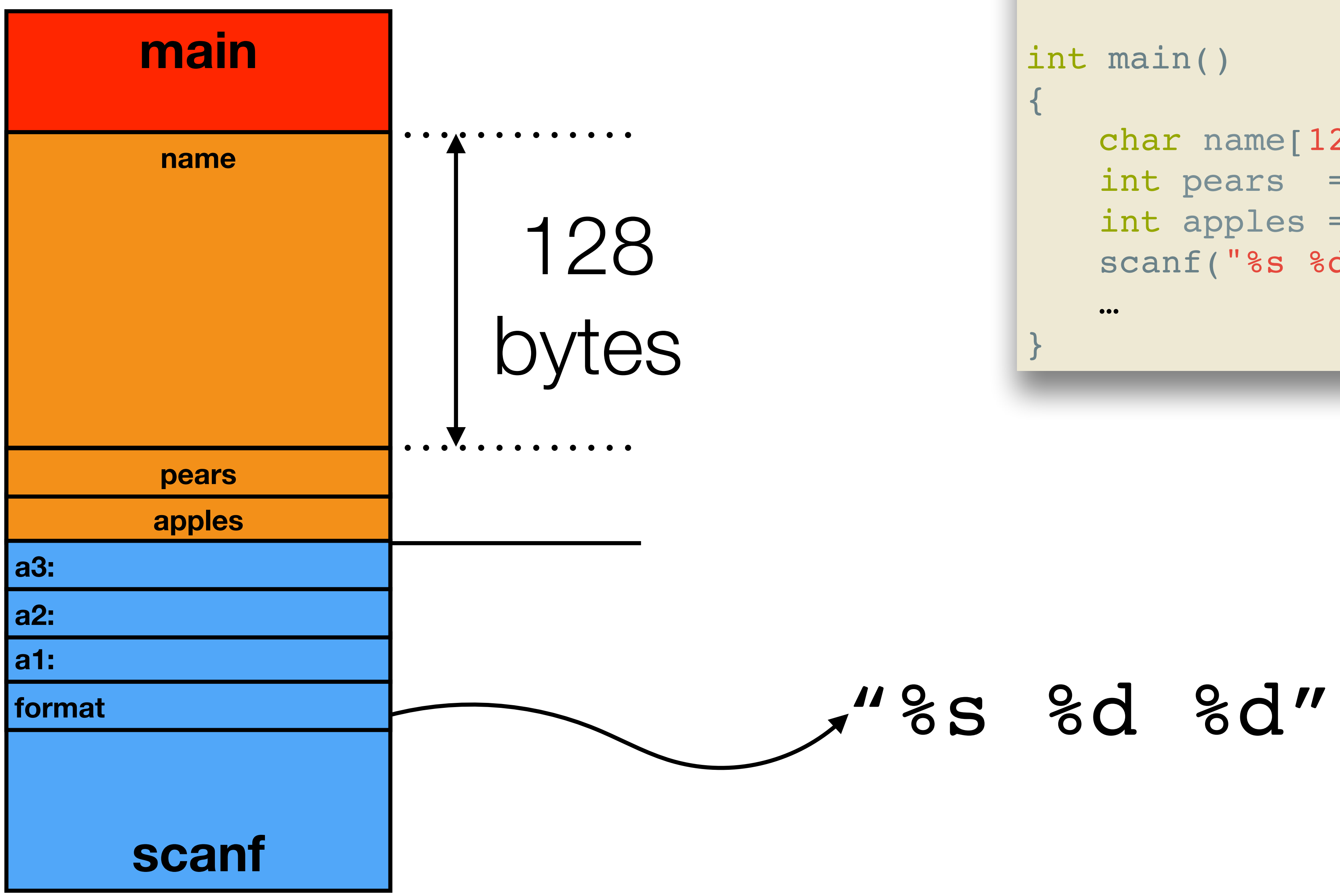

```
#include <stdio.h>

int main()
{
    char name[128];
    int pears  = 0;
    int apples = 0;
    scanf("%s %d %d",name,&pears,&apples);
    …
}
```

Calling **scanf** per se



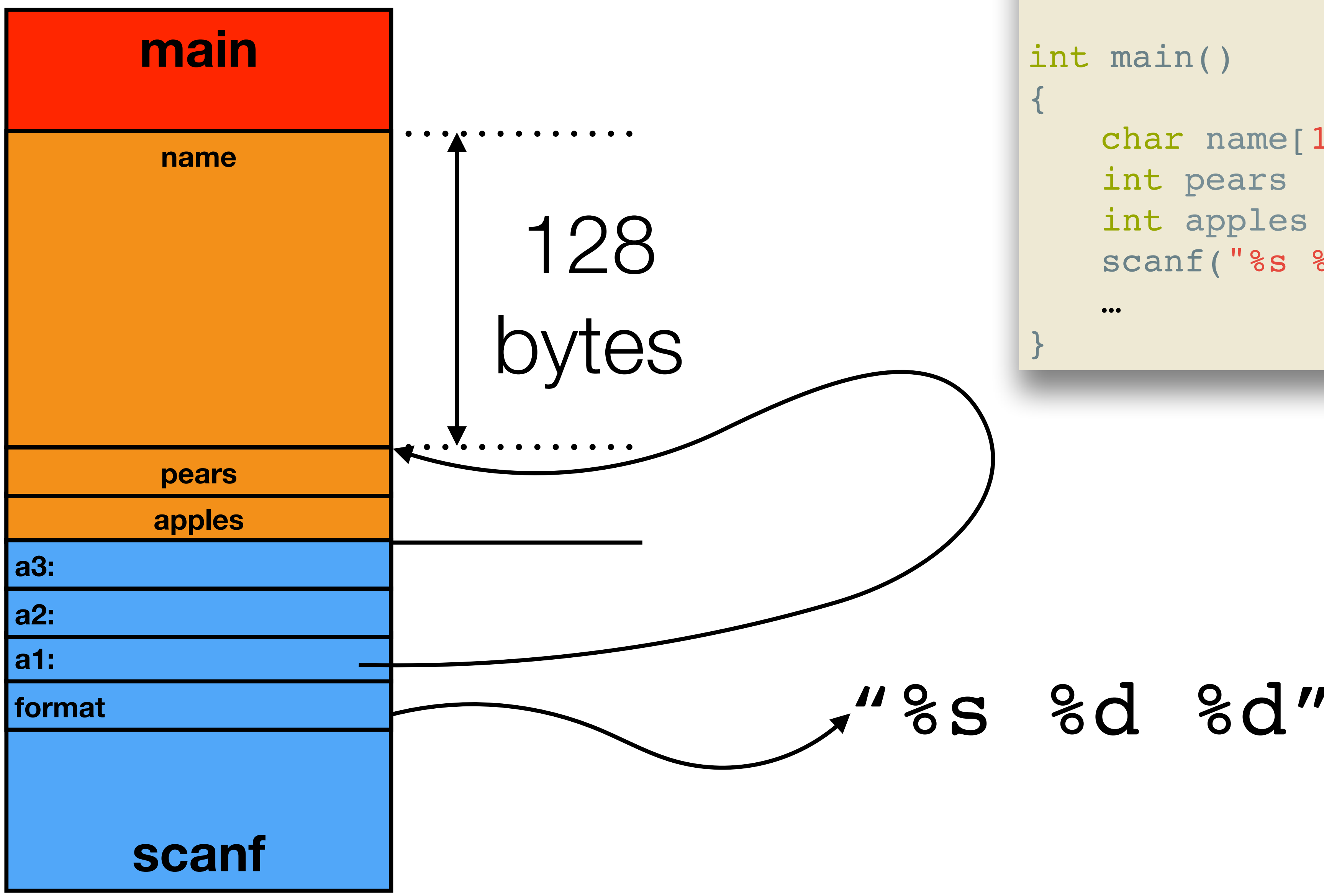

```c
#include <stdio.h>

int main()
{
    char name[128];
    int pears  = 0;
    int apples = 0;
    scanf("%s %d %d",name,&pears,&apples);
    …
}
```

Calling **scanf** per se



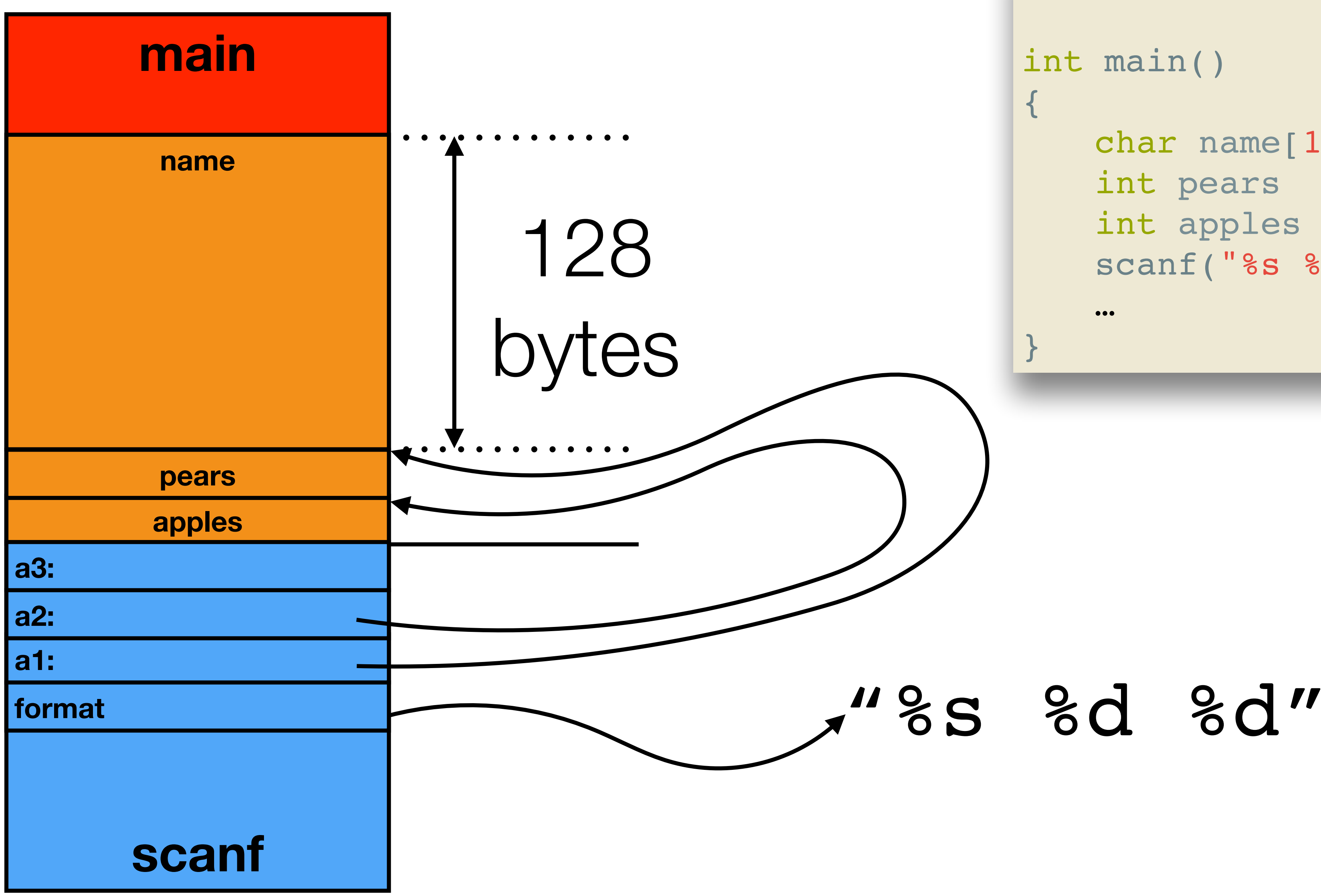

```c
#include <stdio.h>

int main()
{
    char name[128];
    int pears  = 0;
    int apples = 0;
    scanf("%s %d %d",name,&pears,&apples);
    …
}
```

Calling **scanf** per se



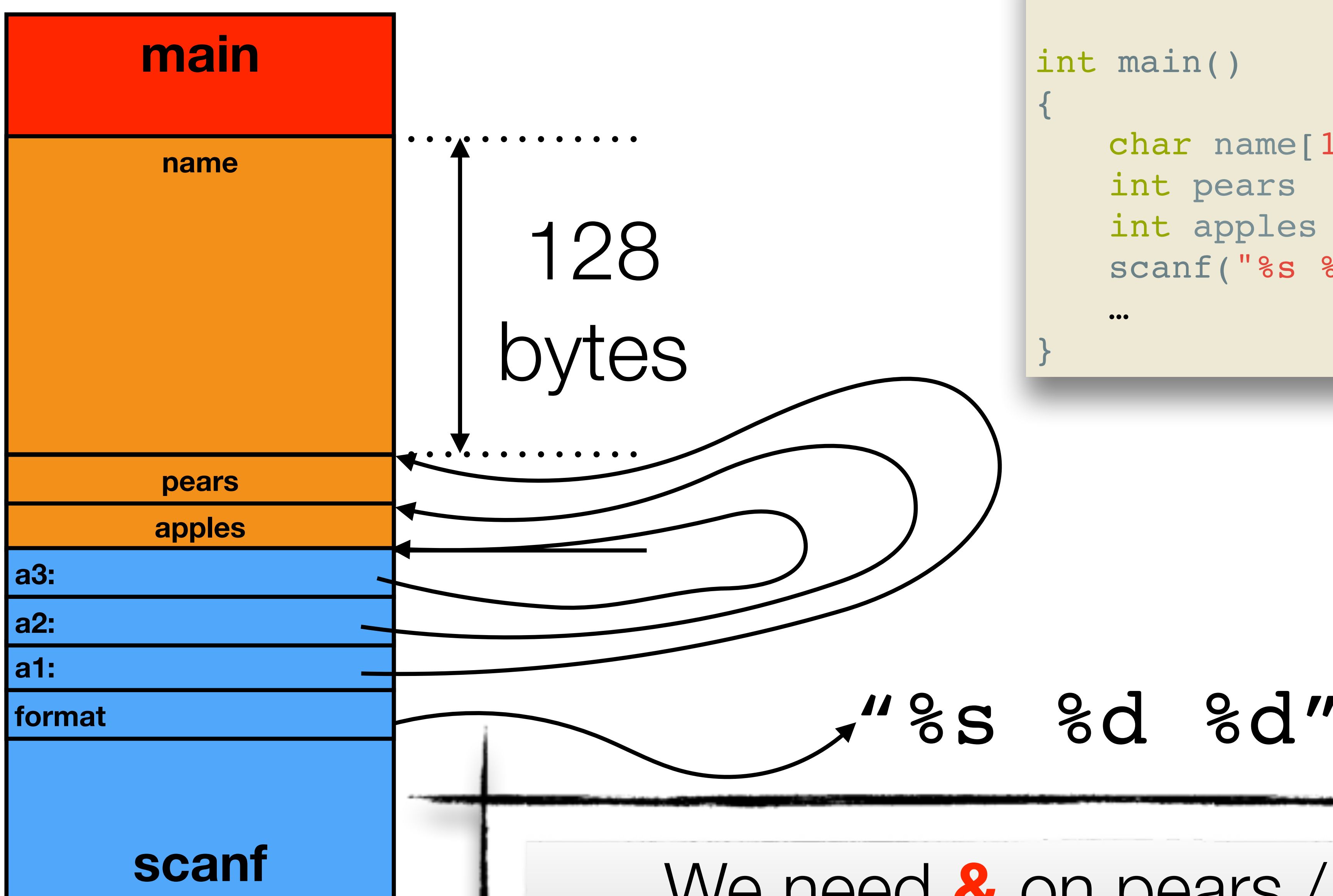

```c
#include <stdio.h>

int main()
{
    char name[128];
    int pears  = 0;
    int apples = 0;
    scanf("%s %d %d",name,&pears,&apples);
    …
}
```

Calling **scanf** per se




```c
#include <stdio.h>

int main()
{
    char name[128];
    int pears  = 0;
    int apples = 0;
    scanf("%s %d %d",name,&pears,&apples);
    …
}
```

Calling **scanf** per se

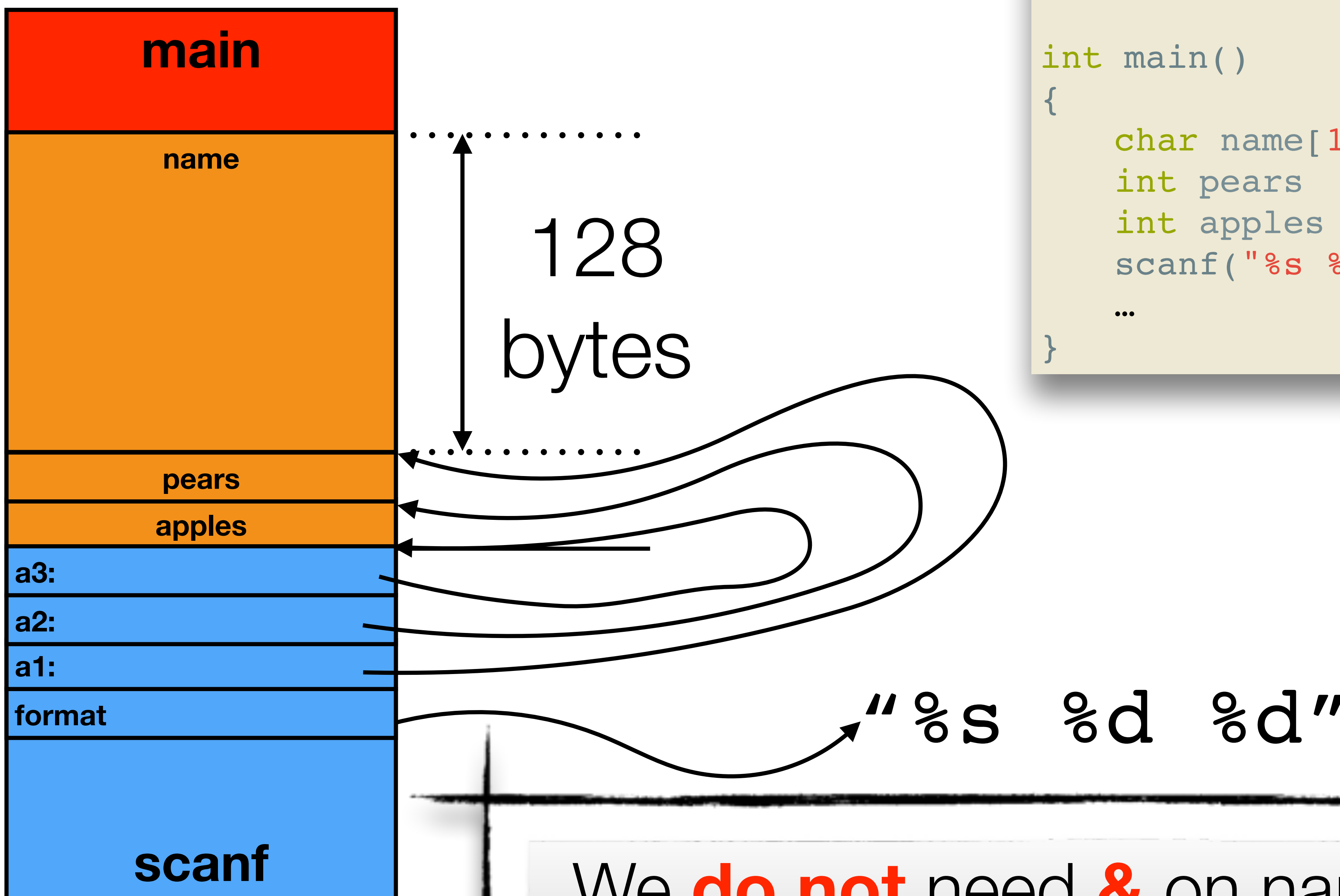



```c
#include <stdio.h>

int main()
{
    char name[128];
    int pears  = 0;
    int apples = 0;
    scanf("%s %d %d",name,&pears,&apples);
    …
}
```

We need & on pears / apples to give their addresses to **scanf** as scanf must **WRITE** there!

Calling **scanf** per se

```c
#include <stdio.h>

int main()
{
    char name[128];
    int pears  = 0;
    int apples = 0;
    scanf("%s %d %d",name,&pears,&apples);
    …
}
```



We **do not** need **&** on name as name is an array and therefore it already is a pointer! Thus, scanf can **WRITE** there too!

Overview

- Basic Types
- Compound Types
  - Arrays
  - Structures
  - Pointers
- Pointer arithmetic
- Memory layout and alignment

Pointers are addresses

- It’s quite simple
  - The **value** of a pointer is the **address** in the address space
  - Thus, the value of a pointer is an integer between (in hexa, 32-bit)
    - 0x00000000 [on 64-bit: 0x0000000000000000]
    - 0xFFFFFFFF [on 64-bit: 0xFFFFFFFFFFFFFFFF]
- Corollary
  - If a pointer is a integer, you can do ***arithmetic…***
  - To ***compute*** other addresses

Rules of Pointer Arithmetic

- Very similar to integer arithmetic
- You can
  - Add a value to a pointer
  - Subtract two pointers
  - Scale an offset
- Big difference
  - All additions are subject to **automatic scaling** of the constant added

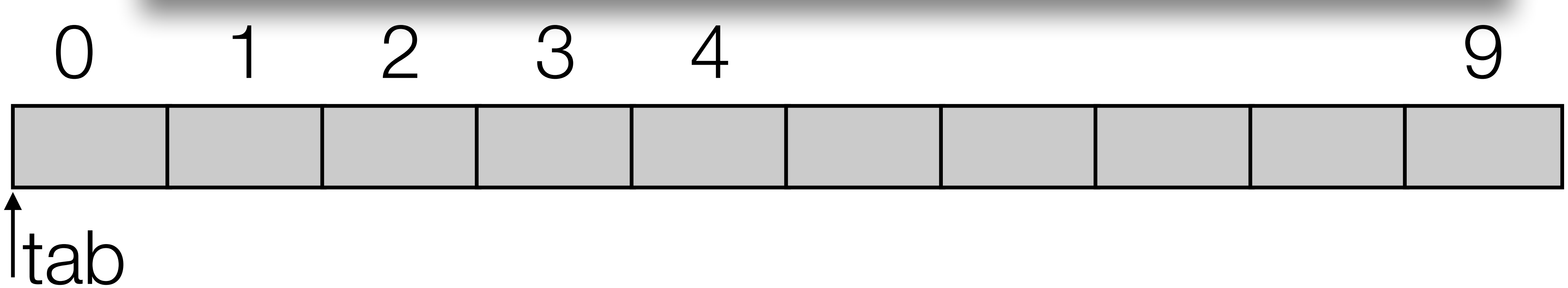
Main Driver in automatic scaling is

The **type** of what is pointed to

Example

- Simple illustration

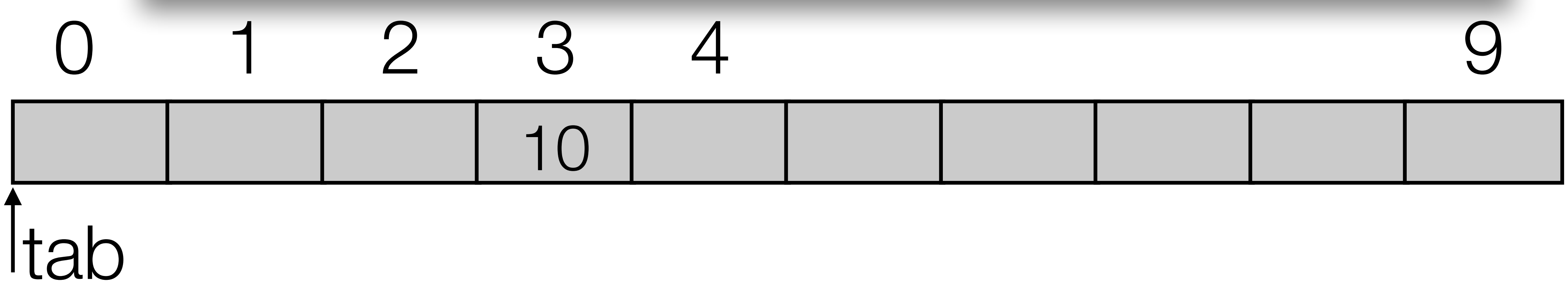
```c
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]   = 10;
   int *p   = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```



Example

- Simple illustration

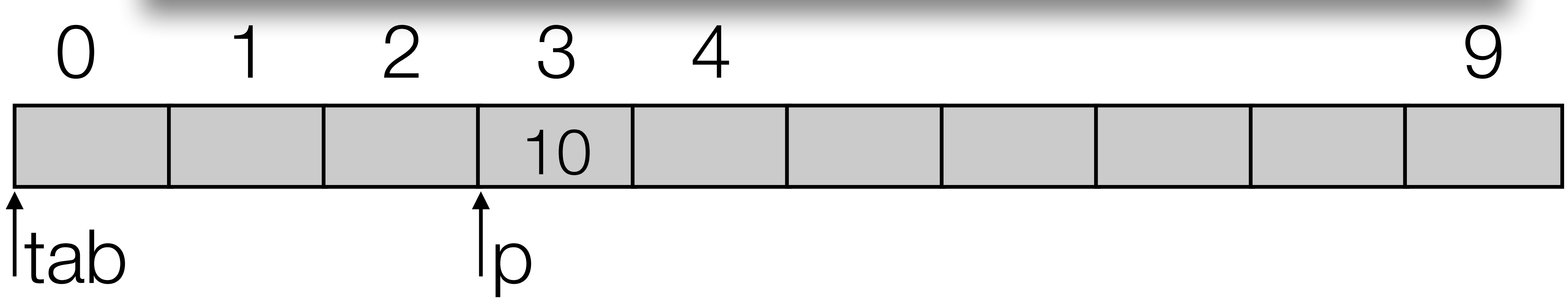
```
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]   = 10;
   int *p   = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```



Example

- Simple illustration

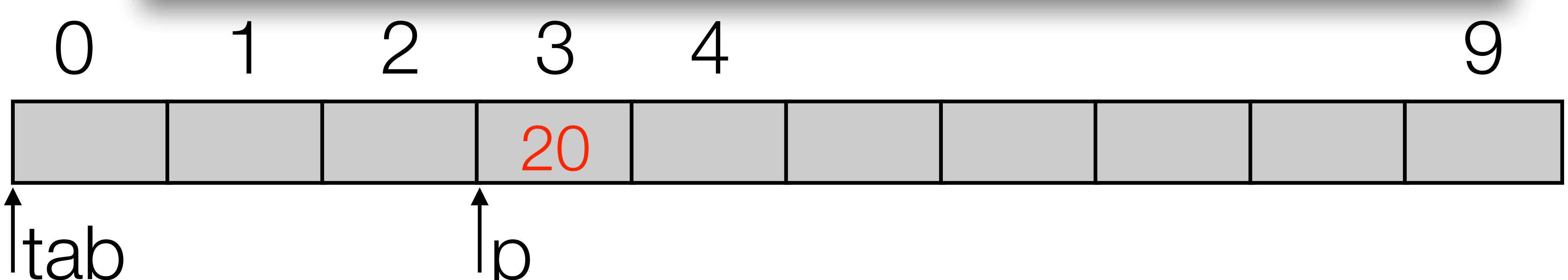
```
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]   = 10;
   int *p   = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```



Example

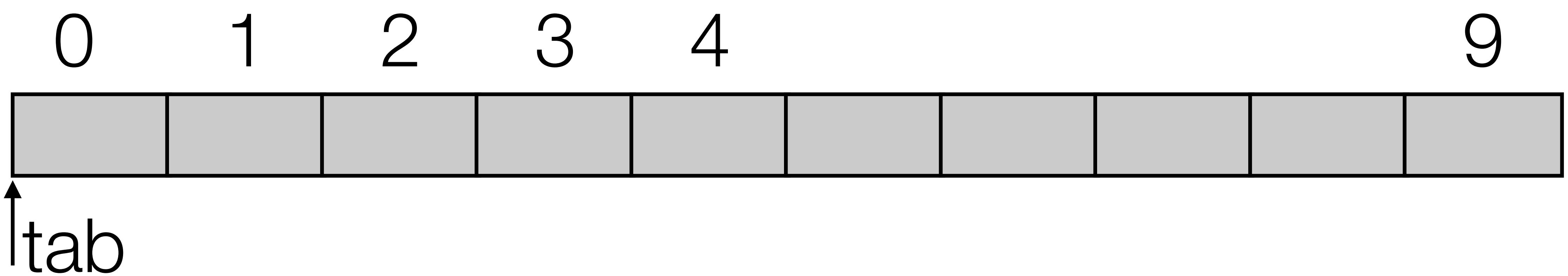
- Simple illustration

```c
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]   = 10;
   int *p   = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```



But what about memory addresses?

- Same story…!



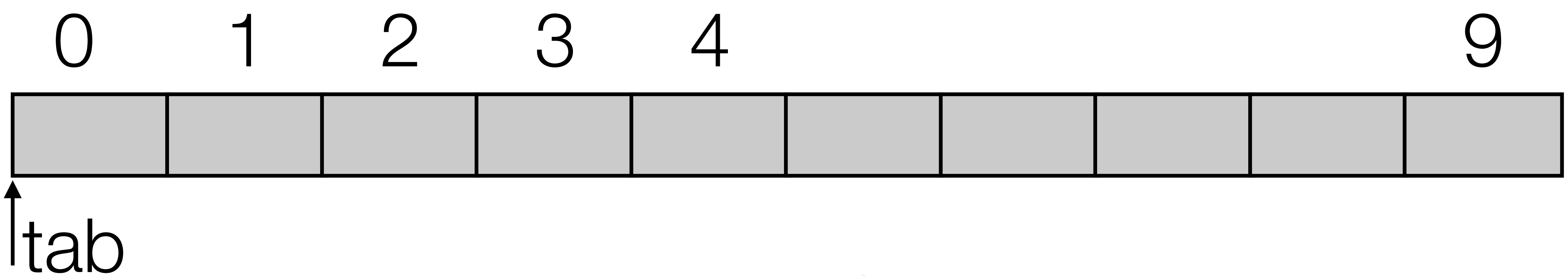
`0x11110000`

tab+1 = ?

`0x11110001 ?`

But what about memory addresses?

- Same story…!



`0x11110000`

tab+1 = ?

~~`0x11110001` ?~~

`0x11110004` ?

WHY?
Simply because tab is a pointer to an int and an int is 4-bytes wide!

Bottom line

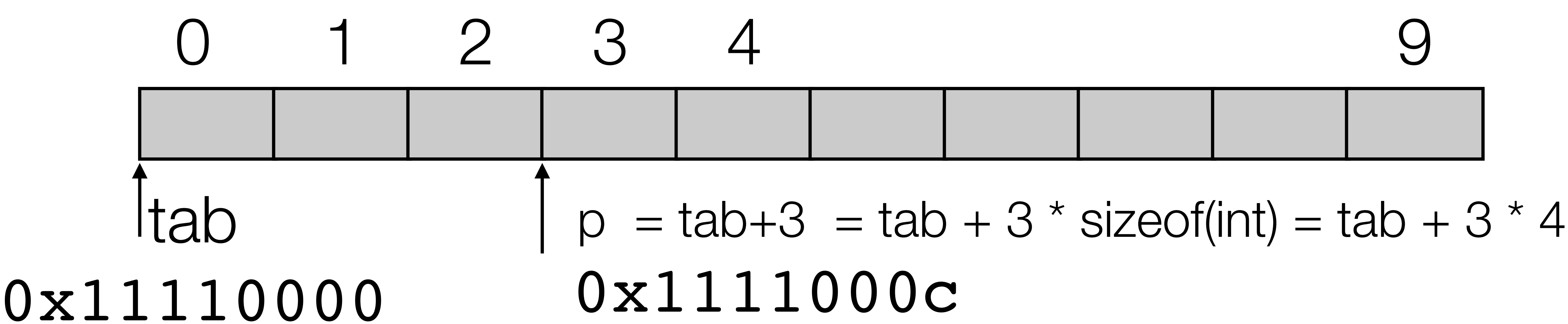
```
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]   = 10;
   int *p   = tab + 3;
   printf("What is at tab+3? = %d\n",*p);
   *p = 20;
   printf("What is at tab[3]? = %d\n",tab[3]);
   return 0;
}
```

The offset **3** is scaled by the compiler with the size of the type to get an address in bytes

Bottom line

```
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(s
   tab[3]    = 10;
   int *p    = tab + 3;
   printf("What is at tab+3?          %d\n", *p);
   *p = 20;
   printf("What is at tab[3]? = %d\n",tab[3]);
   return 0;
}
```

The offset **3** is scaled by the compiler with the size of the type to get an address in bytes



Effect of casting types ?

- If you cast a pointer type…
  - Any subsequent pointer arithmetic will use the type you chose
  - Hence the scaling constants are different
- Note
  - sizeof(char) = 1
- Corollary
  - Casting a pointer to (**char***) results in no scaling!

This is why you need to cast the result from a call to malloc!

Fundamental observation

- C is permissive
  - It will let you cast pointers in any way you like
  - This is called “weak typing”
- Corollary
  - You can “forge” pointers to point wherever you wish….
  - [within your address space of course!]

That’s what makes C very attractive for

low-level programming

That is also very **powerful** and thus **dangerous**!

Pointer subtraction?

- Very easy to understand!
  - It computes the offset (distance) between the two pointers!
  - The distance is in number of “Data Items” (again, based on types)
- Requirement
  - Both pointers have the same type
- Check the example!

Example

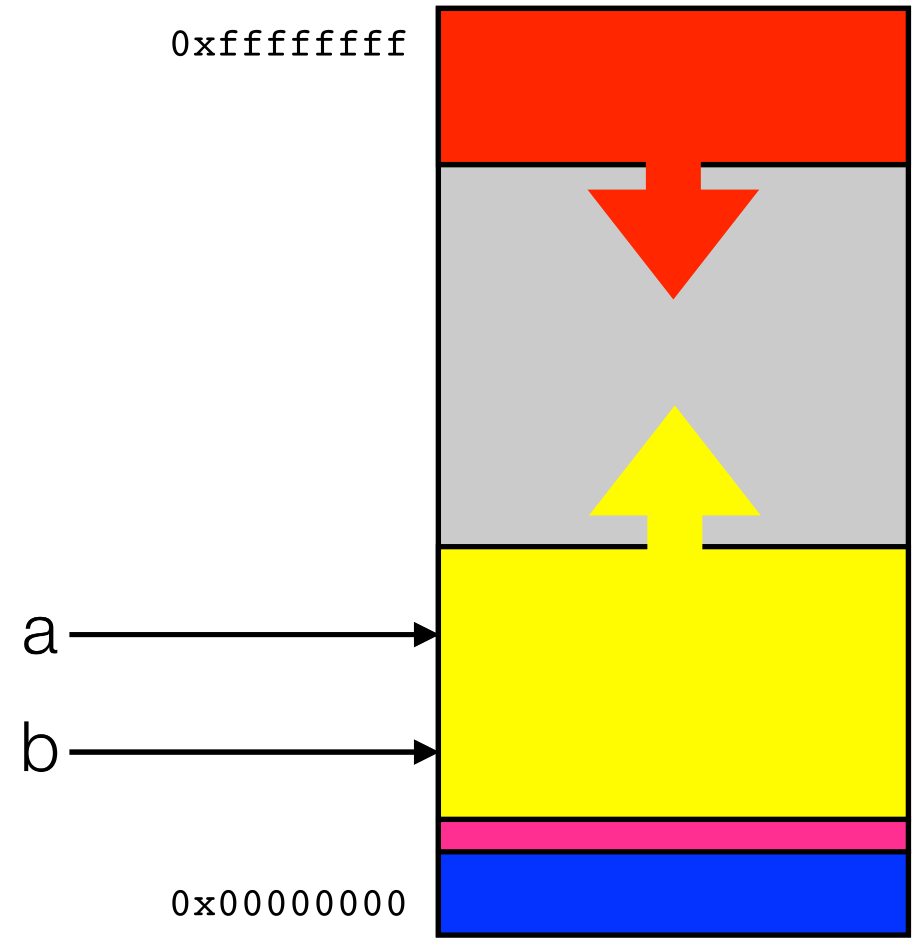
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int* t = (int*)malloc(sizeof(int)*10);
   int* p = t + 9;
   int  dist = p - t;
   printf("Distance is %d\n",dist);
   return 0;
}
```

Output

```
src (master) $ cc ptrsub.c
src (master) $ ./a.out
Distance is 9
src (master) $
```

Pointer comparisons

- You can also compare two pointers
  - < , > , <= , >= , != , ==
- Purpose
  - Check boundary conditions in arrays
  - Manually manage memory blocks
- Semantics
  - Simply based on process layout!
    - ==, != are obvious
    - <,>,<=,>= easy enough!



Generic Pointers

- Purpose
  - Have a pointer to a memory block whose content is “un-typed”
  - Very convenient for raw memory manipulation
- Requires
  - Casting the pointer types before dereferencing for read / write

Generic Pointers Example

- Note
  - Arithmetic is possible
  - Arithmetic is “byte-oriented”

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
   void* p = (void*)malloc(sizeof(int)*10);
   void* a = p  + 4;
   printf("p is %p  a is %p. Distance is: %ld\n",p,a,a - p);
   return 0;
}
```

```
src (master) $ cc void.c
src (master) $ ./a.out
p is 0x7fbe92c03a20  a is 0x7fbe92c03a24. Distance is: 4
```

Overview

- Basic Types
- Compound Types
  - Arrays
  - Structures
  - Pointers
- Pointer arithmetic
- Memory layout and alignment

C Structures

- Strutures are a way to ***package*** related attributes

```
struct IntStack {
  int tab[32];
  int top;
};

…
struct IntStack myStack;
myStack.top = 0;
```

- Fields are *contiguous* in memory
- Fields are *aligned* for the natural types

Alignment

Alignment requirement on a x64 architecture	
char	1
short	2
int	4
long	8
float	4
double	8

Meaning…

- Memory used to store a value of type X **MUST**
  - **be lined-up on a multiple of natural alignment for X**
- Why?
  - Performance!
- If you do not respect alignment requirements…
  - BUS ERROR (sigbus)
  - The O.S. will kill your program

Good news…

The C compiler handles alignment

automatically

99% of the time

Bad news…

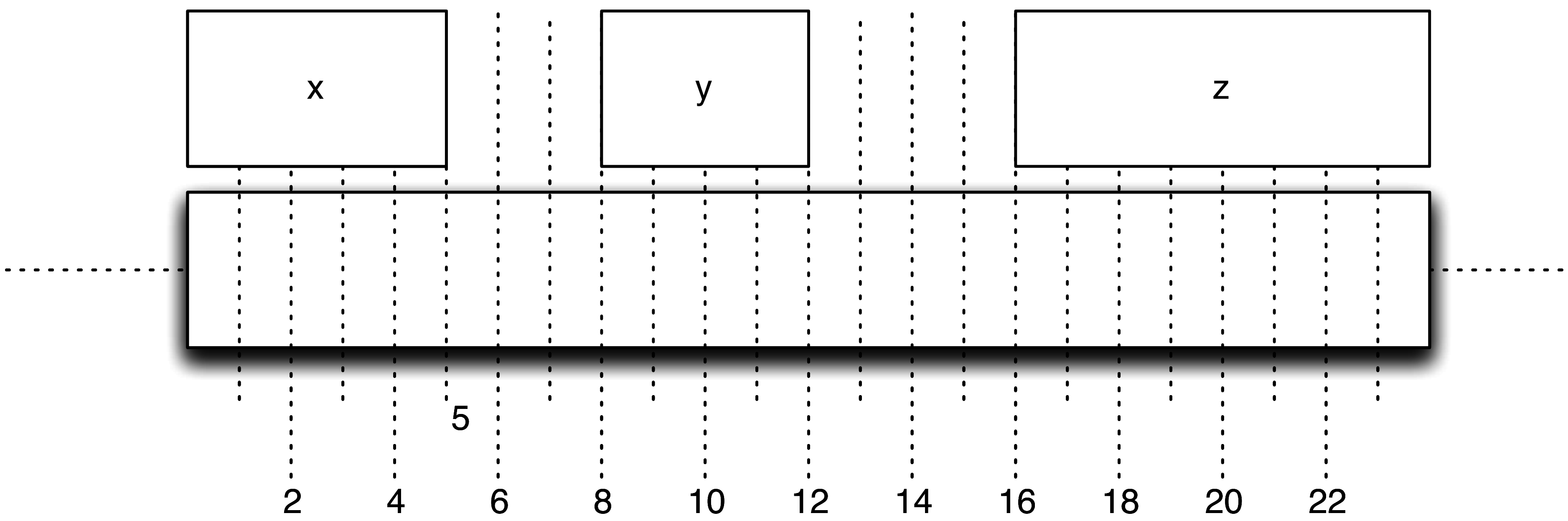
The C compiler handles alignment

automatically

99% of the time

The Programmer must handle the remaining 1%

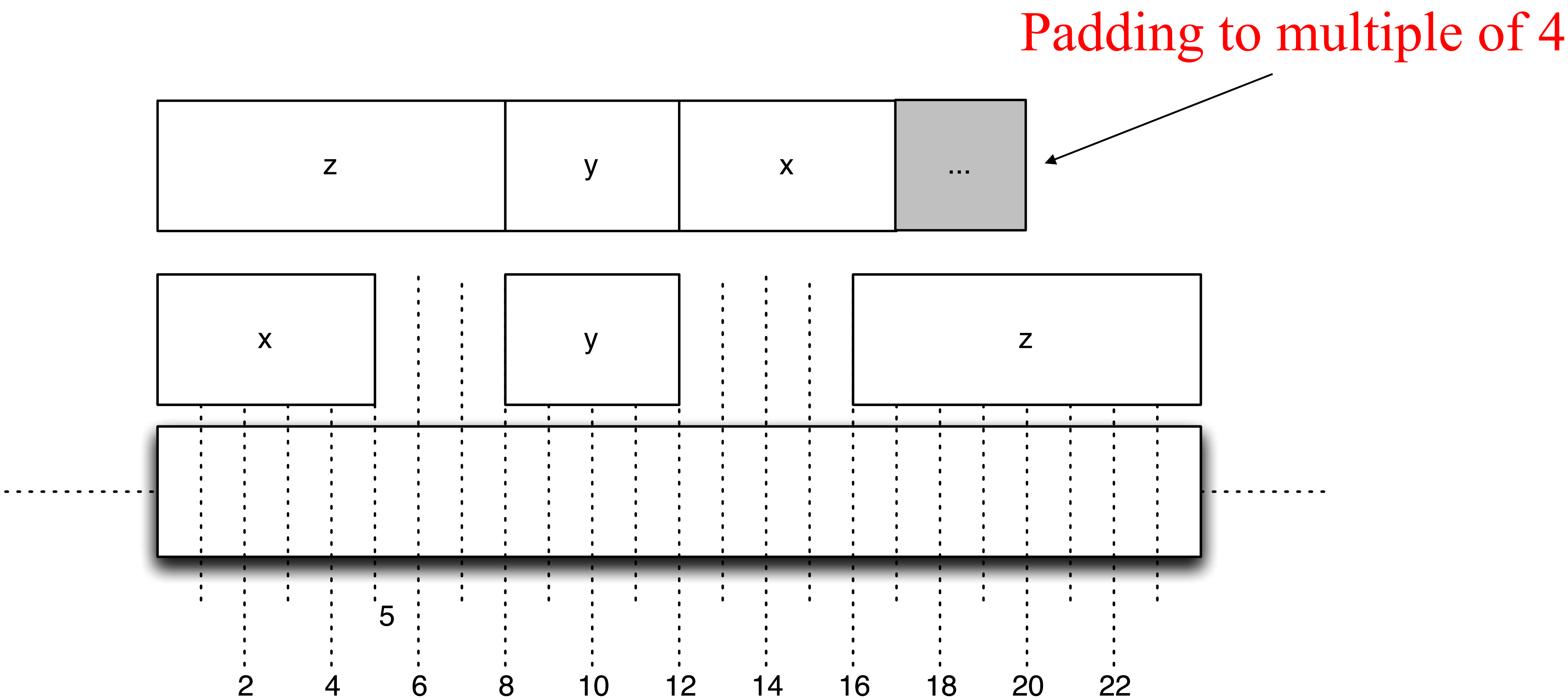
Example




```
struct {
        char x[5];
        int y;
        double z;
}
```

Can you do anything about this to *improve?*

Improvement




```
struct {
        char x[5];
        int y;
        double z;
}
```

```
struct {
        double z;
        int y;
        char x[5];
}
```

Padding ?

- Motivation….
  - What if you wish to create an array of structs ?
  - Second struct in array should start on 4 boundary!
- Conclusion
  - Make it easy
  - All structures have sizes that are multiple of
    - 4 on i32
    - 8 on x64

What is this 1% business?

- When you do pointer arithmetic of course!
  - You must understand the layout (and padding) of structures to correctly compute addresses within the structure!
- When you call system routines with specific alignment needs
  - Your arguments must comply
  - Use compiler annotations to force specific alignments (beyond our scope, simply remember that this exists!)