A C Primer (5): Arrays and Pointer Basics

Ion Mandoiu
Laurent Michel
Revised by M. Khan, J. Shi and W. Wei

Arrays

- New data types
- 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 doubles
  - Arrays of arrays….

Array example


```
int x[5];   // define an array of 5 int's
// accessing array elements is similar to accessing list in Python
// the index starts from 0
x[0] = 1;   x[1] = 2;   x[2] = 3; x[3] = 4; x[4] = x[3] + 1;
// initialize array with a list
int y[5] = {1, 2, 3, 4, 5};
// Number of elements is optional if all elements are listed
int z[] = {1, 2, 3, 4, 5};
// Specify the value of first 2 elements. The rest are set to 0
int a[5] = {1, 2};
// C99. b will have 1, 2, 0, 0, 5.
int b[5] = {1, 2, [4] = 5};
```

Array in memory

- Think about how array elements are stored in memory
- Index starts from 0, the last one is 4 = (5 – 1)

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

	Address	Value
y[4]	1016	5
y[3]	1012	4
y[2]	1008	3
y[1]	1004	2
y[0]	1000	1

String Initialization

- A string is a char array that ends with a 0 (null character)
  - Memory that stores 0 is part of the string
- It can be initialized with a list of characters
- or a string (double-quoted literal)

```c
#include <stdio.h>
int main()
{
   char s[6] = {'H','e','l','l','o','\0'};
   char t[6] = "Hello";
   char u[] = "Hello";
   printf("Array is: %s\n", s);
}
```

	Addr.	Value
s[5]	505	0
s[4]	504	'o'
s[3]	503	'l'
s[2]	502	'l'
s[1]	501	'e'
s[0]	500	'H'

Arrays as Automatic Variables

- You can declare arrays inside *any function or block*
  - Destroyed when exiting from the function or block
- Variable length arrays(VLA, C99) The size of your array can depend on *function arguments or other known values*

```
int foo(int n,int k) {
    int x[n];  // The value of n is known at this time
               // Like other auto variables, x is kept on
               // the stack and is NOT initialized

for (int i = 0; i < n; i++)
        x[i] = 0;
    ……
    return -1;
}
```

Array Assignment

- You ***cannot*** assign a whole array at once to another array
  - Even when the types match

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

```
a.c: In function 'main':
a.c:5:7: error: assignment to expression with array type
     x = y;
       ^
a.c:6:7: error: assignment to expression with array type
     x = z;
       ^
```

Arrays and Functions

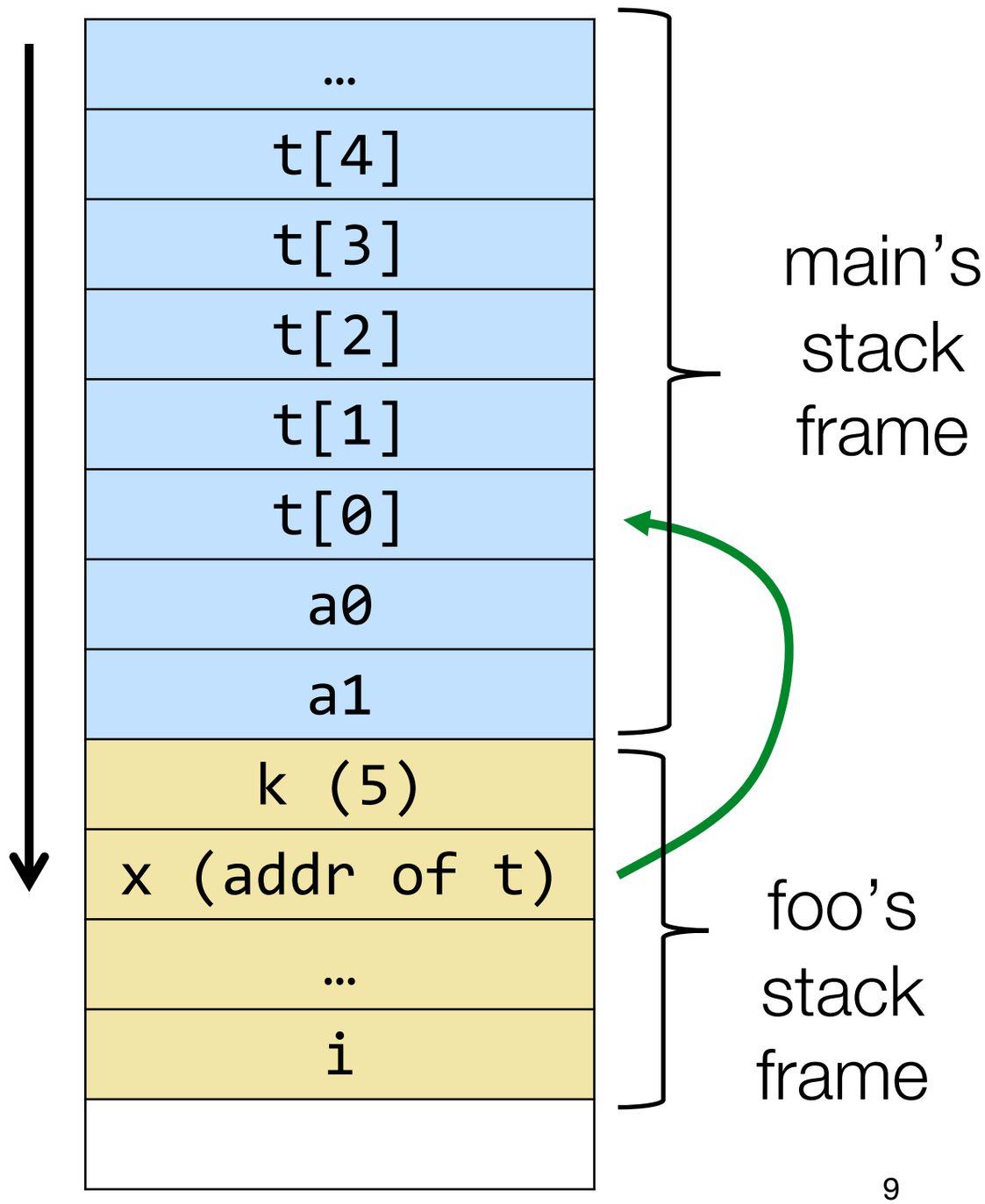
- Arrays can be passed to functions!
  - With one big caveat…
- Calling convention in C
  - BY VALUE for everything….
    - EXCEPT arrays…
- Arrays are always passed BY REFERENCE
  - Passed as “pointers” – we‘ll look at pointers soon
- Functions cannot return arrays
  - No easy assignments

Array argument example

- Address of t is passed to foo()
- Modifications to x are visible in main!

```
int foo(int x[], int k) {
    for (int i = 0; i < k; i++)
        x[i] = i;
    return x[0];
}

int main() {
    int t[5];
    int a0 = foo(t, 5);
    int a1 = t[4];
    printf("%d %d\n", a0, a1);
    // more code
}
```



Multidimensional arrays

```
// declaration and initialization
int   h[2][3] = { {0, 1, 2}, {10, 11, 12} };
```

	0	1	2
0	0	1	2
1	10	11	12

	Address	Value
	1024	
h[1][2]	1020	12
h[1][1]	1016	11
h[1][0]	1012	10
h[0][2]	1008	2
h[0][1]	1004	1
h[0][0]	1000	0
	996	

Array layout in memory:
(assuming an int has four bytes)

- Row 0 first, then Row 1, ...
- In each row: column 0 first, then column 1, …

Pointers

- Perhaps the scariest part of C
- Yet….
  - The most useful part of C!
- A pointer is simply….
  - A value
  - Denoting the address of a memory cell

Variables and Memory

- The memory is an array of bytes
- Every byte in memory is numbered: the address!
  - An address is just an unsigned integer
- Every variable is kept in memory, and is associated with two numbers:

-The address

-The value stored at that address

Implicit Address Use

Addresses are used implicitly by the compiler all the time…

```
int foo (int v)
{
    int a, b;  // allocate storage space for a and b

    a = 1;     // store 1 to memory, at a's address
    b = a;     // load value from a's address, write to b's address
    v = v + b; // load value from v's address, add to value at
               // b's address, and write result to v's address
    return v;
}
```

	Address	Value
	100012	
b	100008	
a	100004	1

Explicit Use: Pointers!

- A pointer is a variable that holds an address of something
- Declaration

```
// declare p to be a pointer to an int
int * p;
```

- The value of p is an address of an int
- p itself has an address

Referencing and dereferencing

- Two new operators

```
&       Reference: "get" the address of something
*       Dereference: "use" the address
```

Example:

```
int x = 10, y;

// px is a pointer to int, i.e., the address of an integer
int   *px;      // px itself has an address

px = &x;        // &x is the address of x. Save it to px

// *px: use px as an address to get the value at that location
y = *px;        // and save the value in y

// save 20 to location pointed to by px (use px as an address)
*px = 20;       // px has x's address, so x becomes 20
```

Picture it

```
// assume 32 bits int and address

int x = 10, y;

int  *px;

px = &x;

y = *px;

*px = 20;
```

	Address	Value
	1028	
	1024	
x	1020	10
y	1016	?
px	1012	?
	1008	
	1004	
	1000	

Picture it

```
// assume 32 bits int and address

int x = 10, y;

int  *px;

px = &x;

y = *px;

*px = 20;
```

	Address	Value
	1007	
	1006	
x	1020	10
y	1016	?
px	1012	1020
	1008	
	1004	
	1000	

Picture it

```
// assume 32 bits int and address

int x = 10, y;

int  *px;

px = &x;

y = *px;

*px = 20;
```

	Address	Value
	1007	
	1006	
x	1020	10
y	1016	10
px	1012	1020
	1008	
	1004	
	1000	

Picture it

```
// assume 32 bits int and address

int x = 10, y;

int   *px;

px = &x;

y = *px;

*px = 20;
```

	Address	Value
	1007	
	1006	
x	1020	20
y	1016	10
px	1012	1020
	1008	
	1004	
	1000	

Revisit the example

```
int foo(int x[], int k) {
    for (int i = 0; i < k; i++)
        x[i] = i;
    return x[0];
}
```

x is the starting address of an array, which is the same as the address of element 0.

```
// x is the address of an int
int foo(int * x, int k) {
    for (int i = 0; i < k; i++)
        x[i] = i; // use the pointer as an array
    return x[0];
}
```

Array Question 1

```
// how many integers are in a, and in b?
int   a[] = {1, 2, 3, 4};
int   b[4] = {1, 2, 3};
```

How many integers are in array a?

Array Question 2

```
// how many bytes (characters) in c? and in d?
char  c[] = {'a', 'b', 'c', 'd'};
char  d[] = "abcd";
```

What is sizeof(d)?

Array Question 3

```
int   a[10];
```

```
If a[0]'s address is 1000, what is a[4]'s address?
```

Array Question 4

```
char  t[10][20];
```

```
If t[0][0]'s address is 1000, what is t[1][1]'s address?
```

Study the remaining slides yourself

Questions on Array

- Where is the array allocated ?
  - It is automatic, so it is on the stack.
- When is the array 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!

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

Automatic Arrays 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 [expression] (C99)
  - Cannot be too big since it is on the stack!

VLA support in C

- Variable length arrays (VLAs)
  - Introduced in C99
  - Optional in C11
    - In some applications, it is not desirable to put large amounts of data on stack
  - If it is not supported, you have to use heap to keep the array
    - You will learn how to do it soon

Another array example

```
void        foo (int n)
{
    int     i;
    int     a[n];               // define an array of n integers
    int     b[2][n];            // 2 by n array
    // indexes start from 0. a[0], a[1], …, a[n-1].     Not a[n] !
    for (i = 0; i < n; i ++)// typical loop of n times
        a[i] = i;               // write to array a
   // copy a to row 0 of b. You cannot do b[0] = a
    for (i = 0; i < n; i ++)
        b[0][i] = a[i];
}
```

Pointer declarations

- Word to the wise…
  - The following declarations are equivalent

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

- They all declare…
  - p to be a pointer to an integer
- But
  - First one makes the above statement clear
  - Second one is “non-committing”
  - Third says that what p points to is an integer (classic C style)

Pitfalls

- Consider the following declarations:

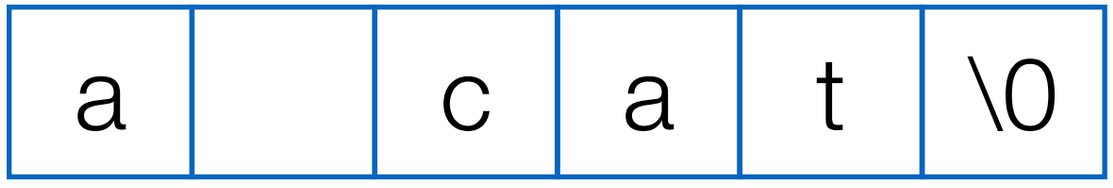
```
int  *a, b;
int* c, d;
int  e, *f;
int  *g, *h;
```

- What are the types of the variables?
  - a, c, f, g, h are int *
  - b, d, e are int

Strings

- In C, strings are just char arrays with a NUL (‘\0’) terminator
- A literal string (“a cat”)
  - compiler allocates memory space to contain all characters you can see and the terminating '\0'
  - can’t be changed by the program (common bug!)

“a cat” in memory



String literals cannot be changed

```
#include <ctype.h>

char *makeBig(char *s) {
  s[0] = toupper(s[0]);
  return s;
}

int main()
{
  makeBig("a cat");
}
```

Output

```
[$ make string
cc      string.c   -o
string
$ ./string
Segmentation fault (core
dumped)
```

Strings

- They are char arrays, but must end with '\0'

```
char line[100];
…

for (i = 0; i < 100 && line[i]; i++) {
  if (isalpha(line[i])
     putchar(line[i]);  // print only one character
}
```

index must be range

Not the end of the string yet

Strings

- We refer to a string via a pointer to its first character
  - Or the address of its first character

```
// str is the address of a char,
// which is 'm', the first character in the string
// str[0] is 'm', str[1] is 'y', and so on
char *str = “my string”;

// s is a pointer to char
char *s;

// s will refer to the same string as str.
s = str;
```

Copying strings

- Use strcpy() (in string.h) to copy a string

```
// buf is a char array
char buf[100];
// str refers to the string literal
char *str = "a cat";
// now, buf has a string. strcpy copies '\0'!
strcpy(buf, str);
// s refers to the string in buf[100],
// which is different from str
char *s = buf;
```

A beginner's implementation of strcpy

- Idea: Copy a character each time until '\0' is copied

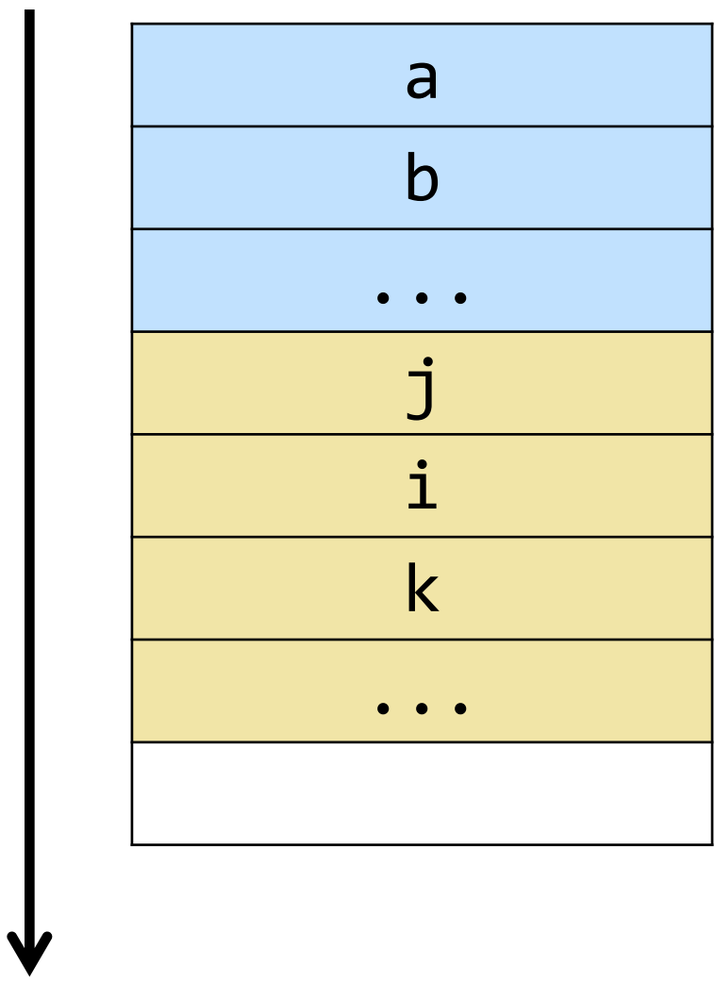
```
char * strcpy(char *dest, char *src)
{
    unsigned int i = 0;
    // copy any character that is not '\0'
    for (i = 0; src[i] != '\0'; i++)
        dest[i] = src[i];
    dest[i] = src[i]; // copy '\0'
    return dest;
}
```

Example: pass argument by value

- Suppose we want to swap the values of two int variables
- The following won't work since parameters are passed by value

```
void swap(int i, int j)
{
    int k = j;
    j = i;
    i = k;
}

int main()
{
   int a=1, b=2;
    swap(a, b);
   return 0;
}
```

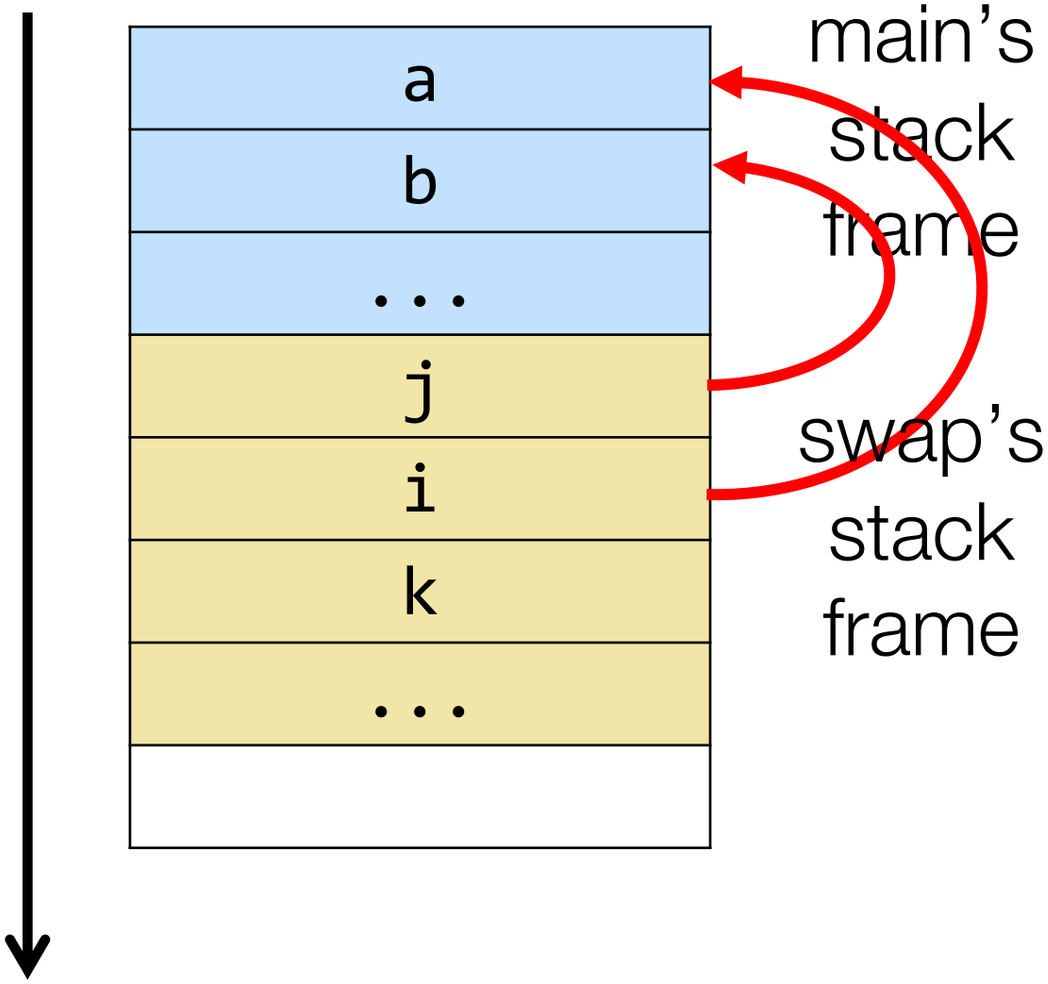


Example: pass argument by reference

- You must pass pointers to the variables instead

```
void swap(int *i, int *j)
{
     int k = *j;
     *j = *i;
     *i = k;
}

int main()
{
   int a=1, b=2;
    swap(&a, &b);
   return 0;
}
```



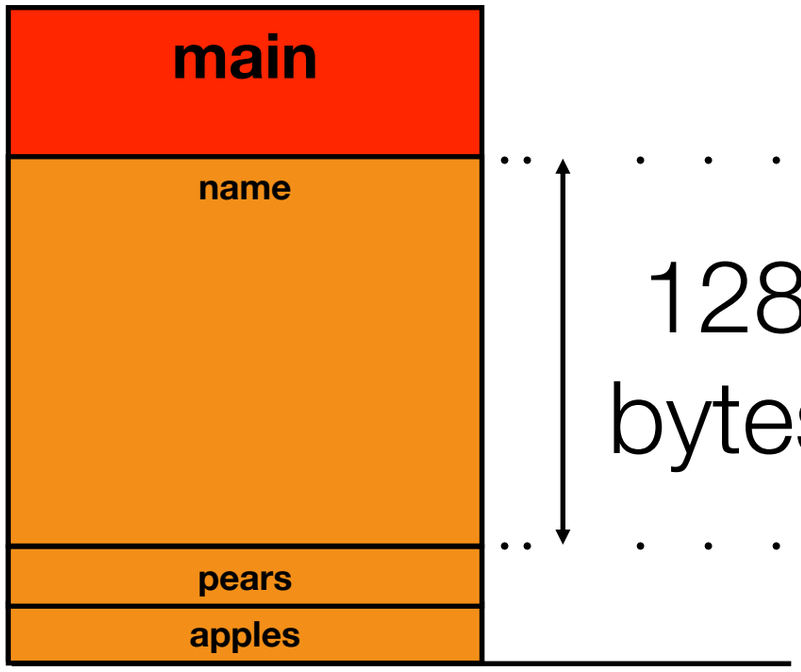
Example: **scanf**

```c
#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 [**&**]

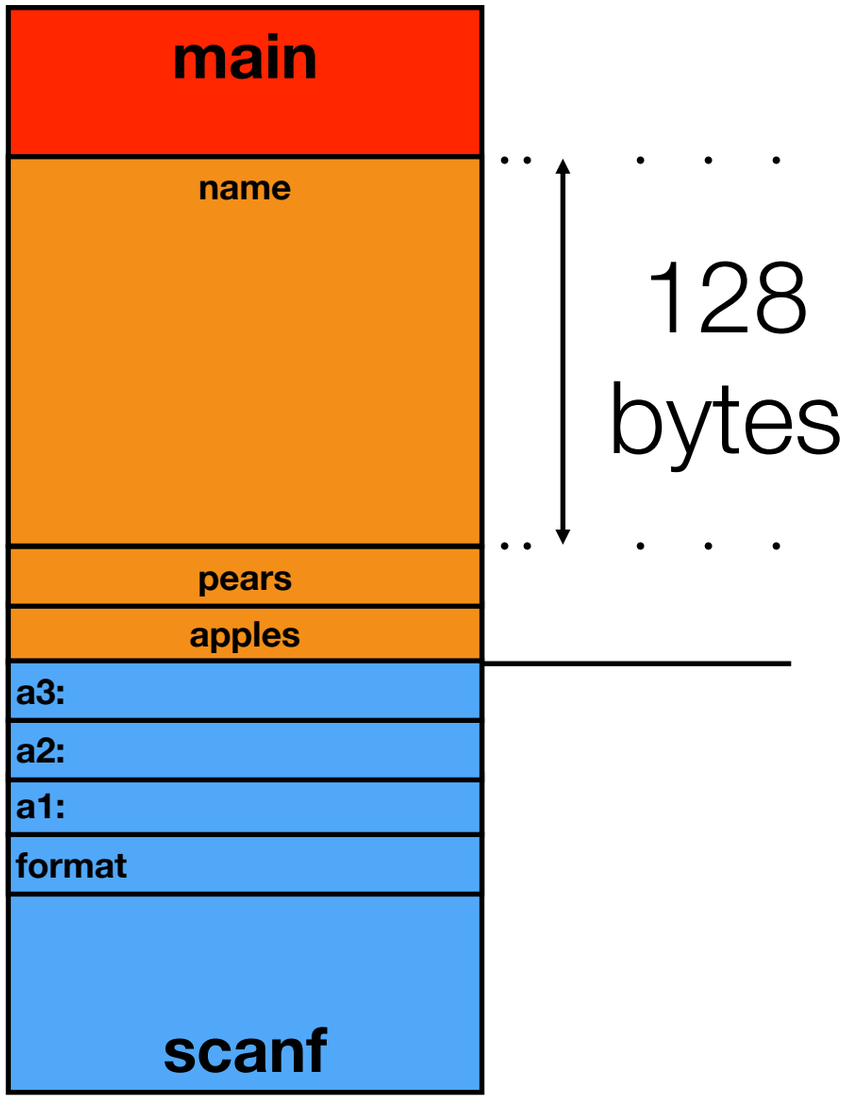
Frame of main … In picture




```c
#include <stdio.h>

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

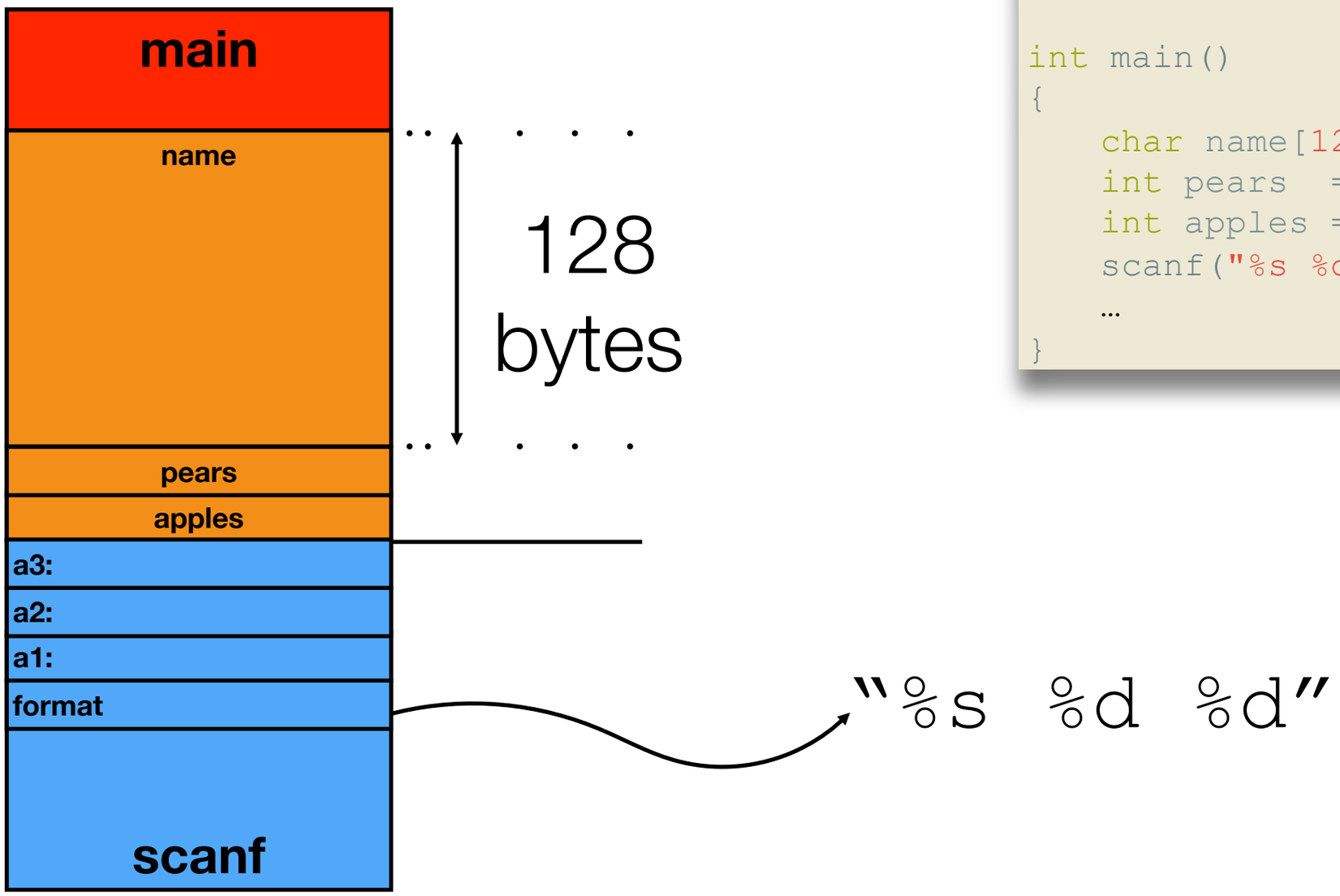
Calling **scanf** per se




```c
#include <stdio.h>

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

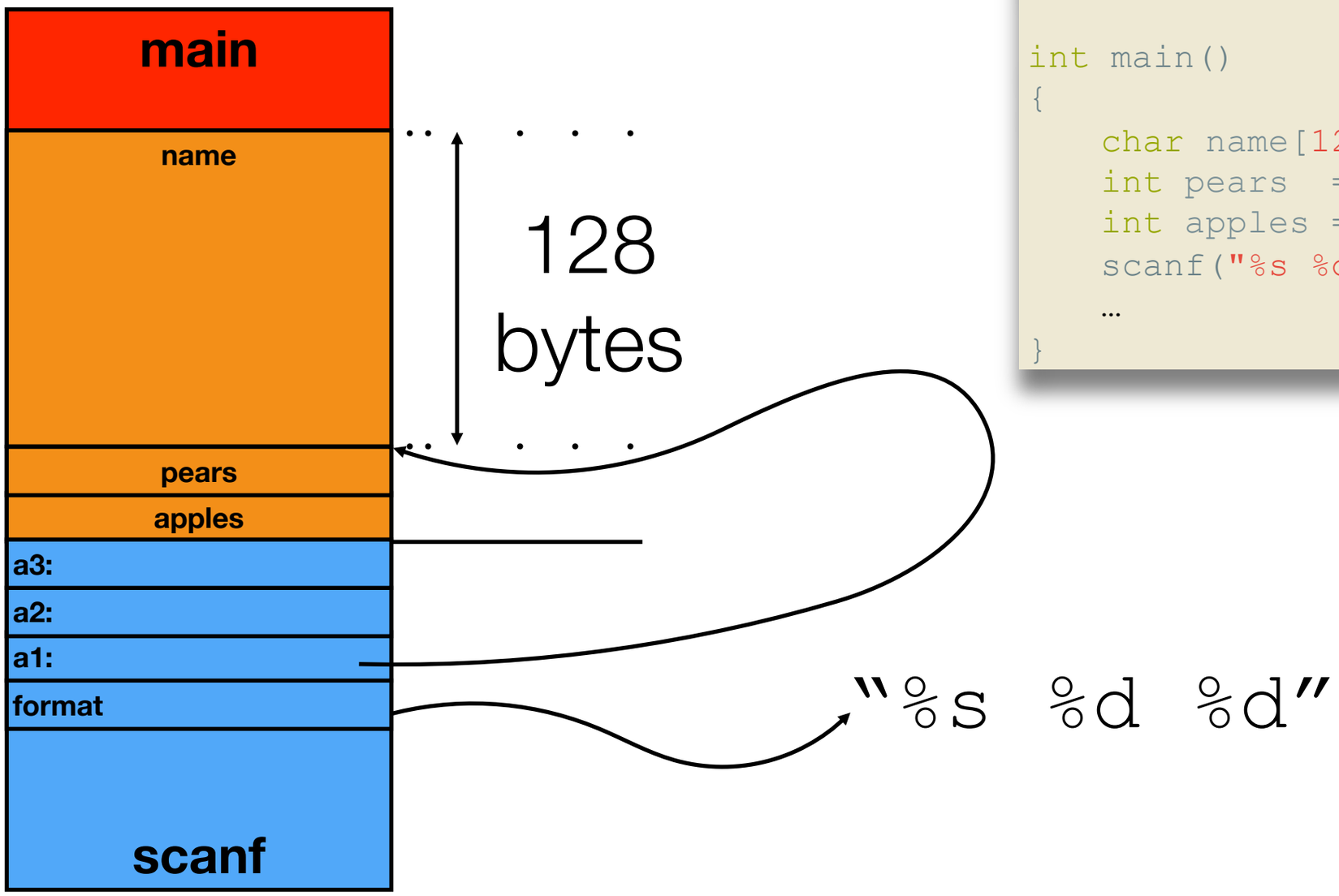
Calling **scanf** per se




```c
#include <stdio.h>

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

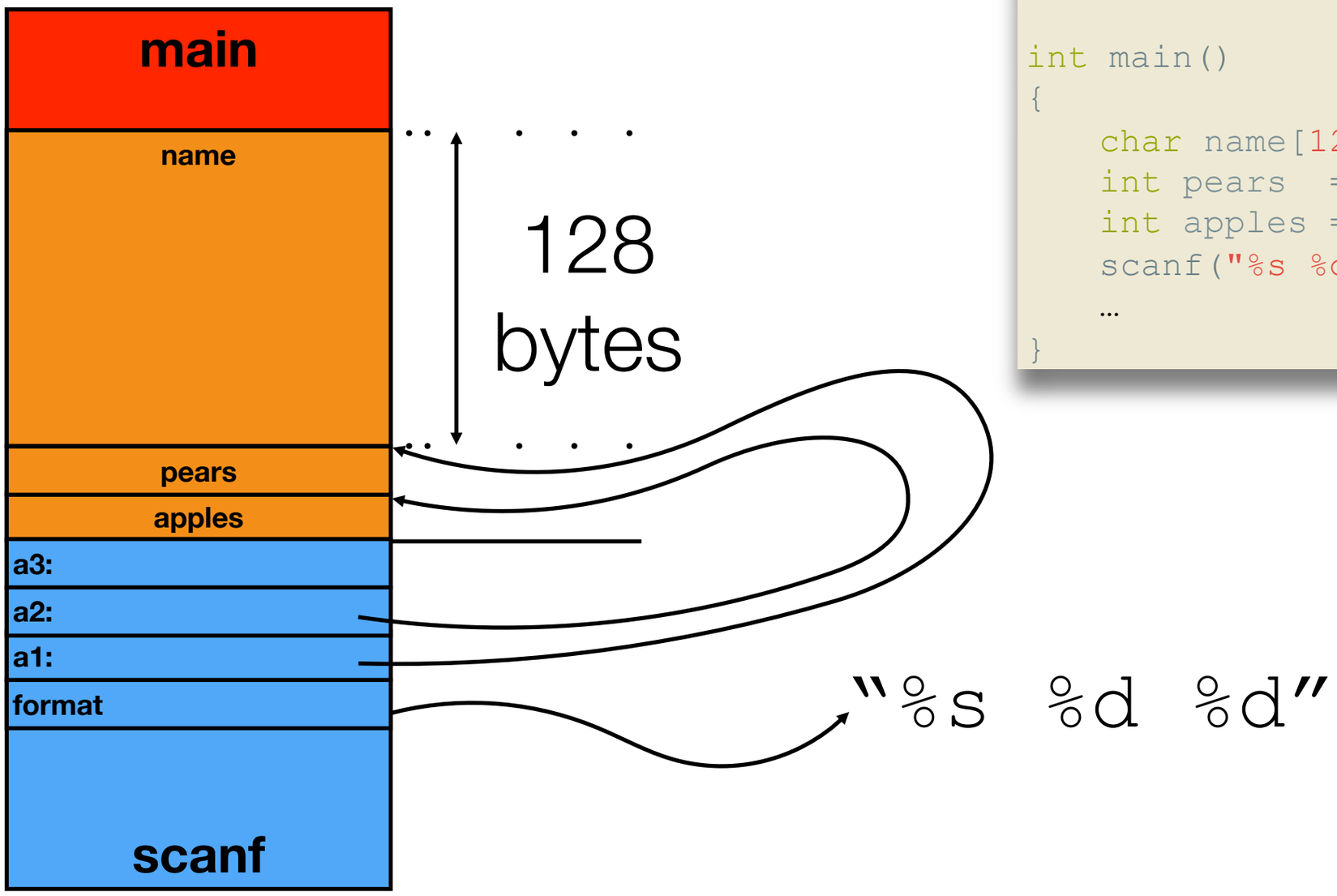
Calling **scanf** per se




```c
#include <stdio.h>

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

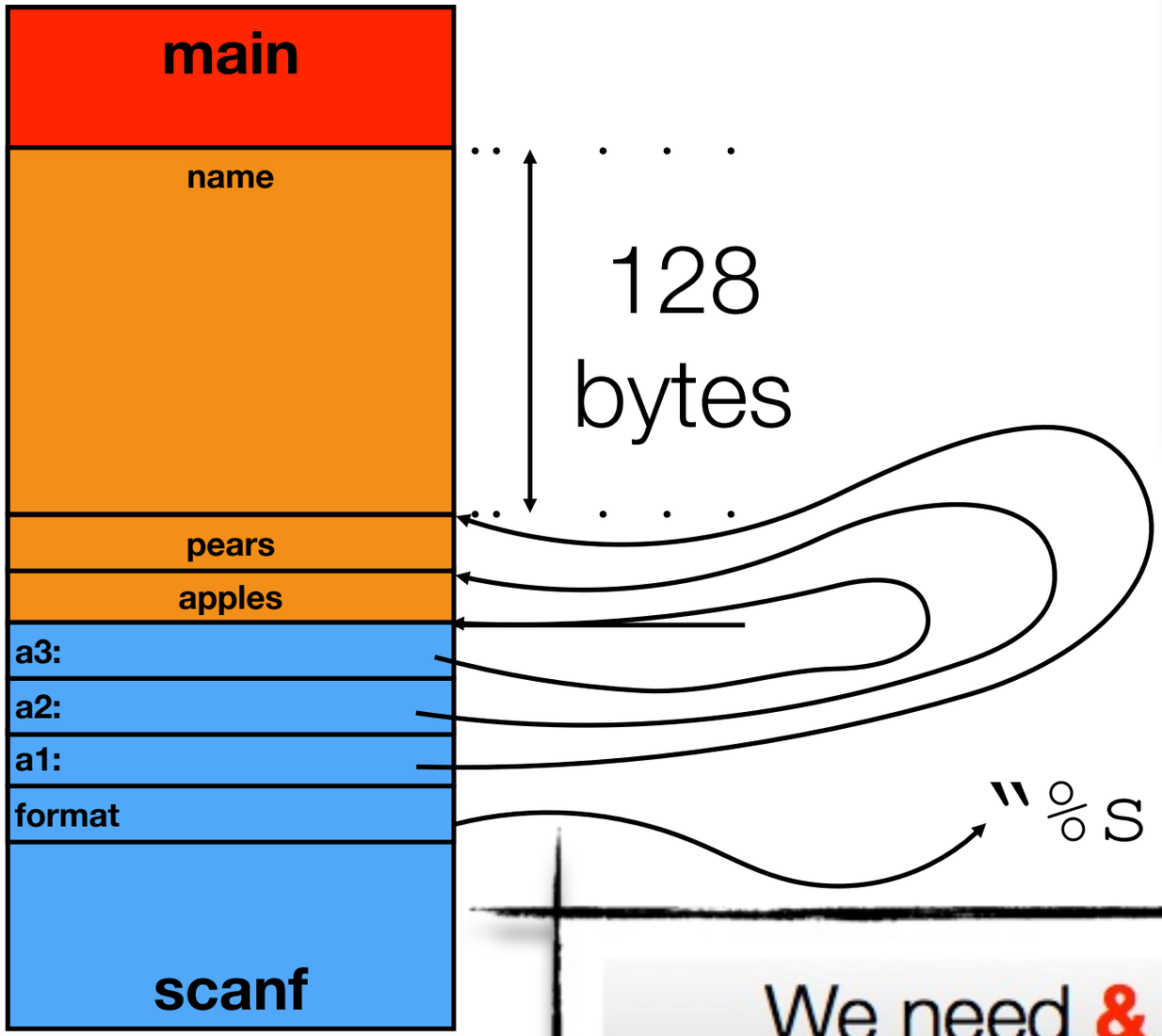
Calling **scanf** per se




```c
#include <stdio.h>

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

Calling **scanf** per se

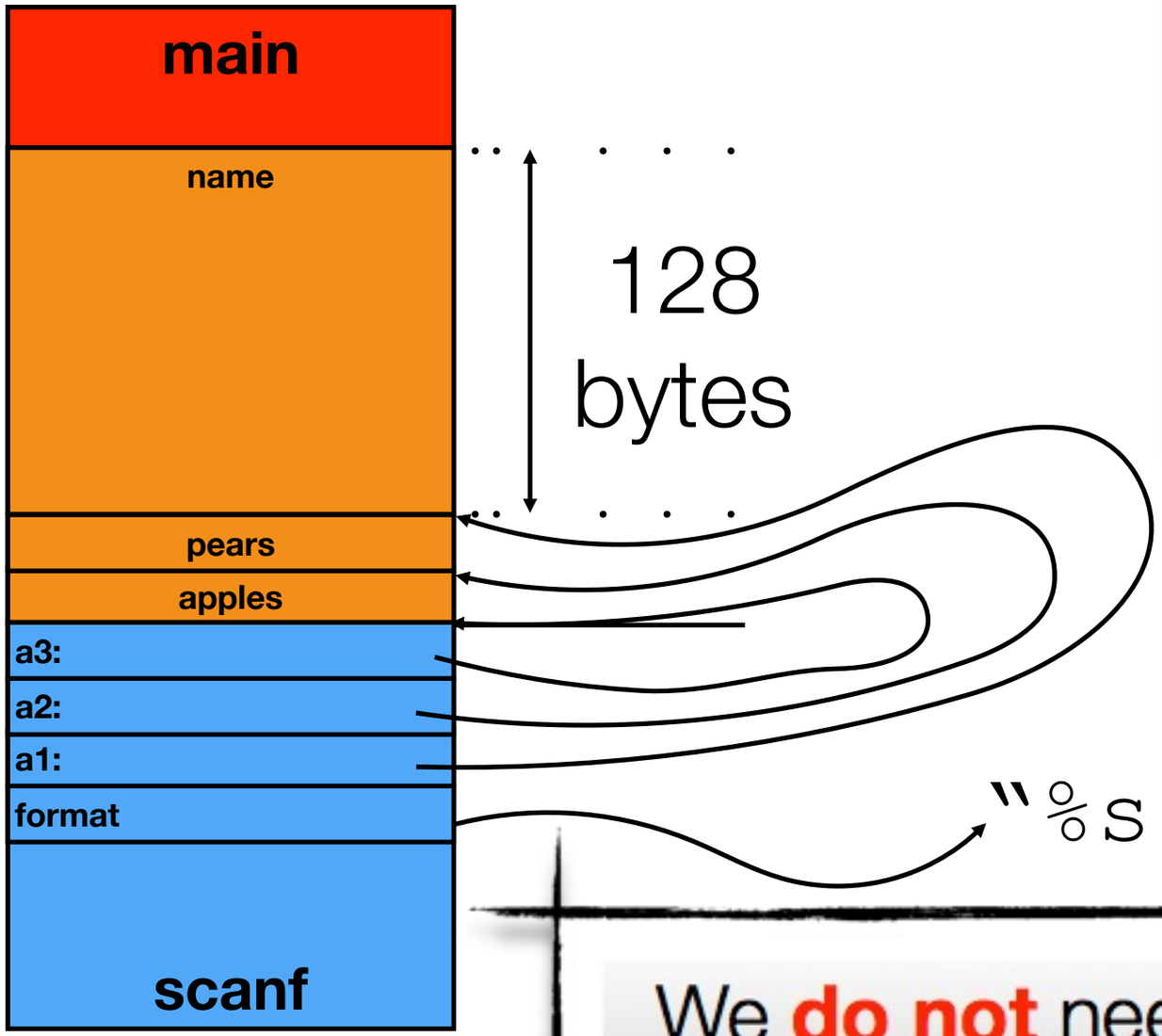



```c
#include <stdio.h>

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!