CSE1142 – Pointers in C

Sanem Arslan Yılmaz

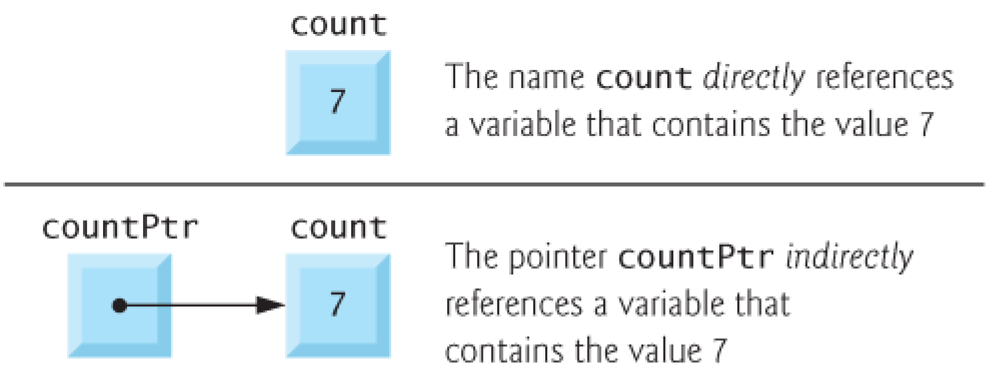
Some of the slides are from:
CMPE150 – Boğaziçi University
Deitel & Associates

Agenda

- Pointer
  - Variable definitions
  - Initialization
- Pointer Operators
- How to use pointers in Functions
- Passing Arguments to Functions by Reference
- Pointer Expressions and Pointer Arithmetic
- Using the `const` Qualifier with Pointers
- Relationship Between Pointers and Arrays
- Array of Pointers

Pointer Variable Definitions and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)
  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value



The name `count` *directly* references a variable that contains the value 7

The pointer `countPtr` *indirectly* references a variable that contains the value 7

Pointer Variable Definitions and Initialization – cont.

- Pointer definitions
  - `*` used with pointer variables

    ```
    int *myPtr;
    ```

  - Defines a pointer to an `int` (pointer of type `int *`)

  - Multiple pointers require using a `*` before each variable definition

    ```
    int *myPtr1, *myPtr2;
    ```

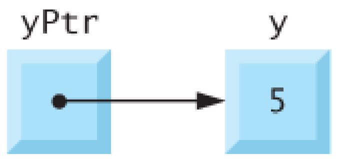
  - Can define pointers to any data type

  - Initialize pointers to `0`, `NULL`, or an address
    - 0 or `NULL` – points to nothing (`NULL` preferred)
    - A pointer with the value NULL points to *nothing*.

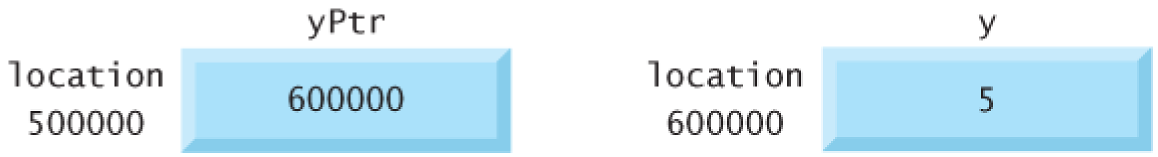
Pointer Operators

- & (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;     /* yPtr gets address of y */
yPtr “points to” y
```



Graphical representation of a pointer pointing to an integer variable in memory.



Representation of y and yPtr in memory.

Pointer Operators – cont.

- `*` (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - `*yptr` returns `y` (because `yptr` points to `y`)
  - `*` can be used for assignment
    - Returns alias to an object

      ```
      *yptr = 7;  /* changes y to 7 */
      ```
  - Dereferenced pointer (operand of `*`) must be an lvalue (no constants)

- `*` and `&` are inverses
  - They cancel each other out

How to use pointers in Functions

```
void func(int *num)
{   *num = 5;
}
int main()
{   int count=10;
    func(&count);
}
```

Define the parameter as pointer

Use '*' before the parameter name so that you access the value at the mentioned address

Send the address of the argument

Example

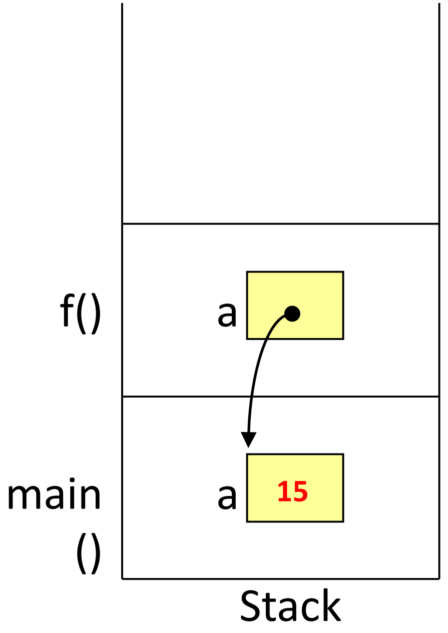
```
#include <stdio.h>

void f(int *a)
{
   *a+=5;
   printf("in function f(): a=%d\n", *a);
}

int main()
{  int a=10;
   printf("in main(), before calling f(): a=%d\n",a);
   f(&a);
   printf("in main(), after calling f(): a=%d\n",a);
}
```

OUTPUT

```
in main(), before calling f(): a=10
in function f(): a=15
in main(), after calling f(): a=15
```



Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function—pass-by-value and pass-by-reference.

- *All arguments in C are passed by value.*
  - a copy of the argument in the function call is made and passed to the function.

- Use pointers and the indirection operator to *simulate* pass-by-reference.

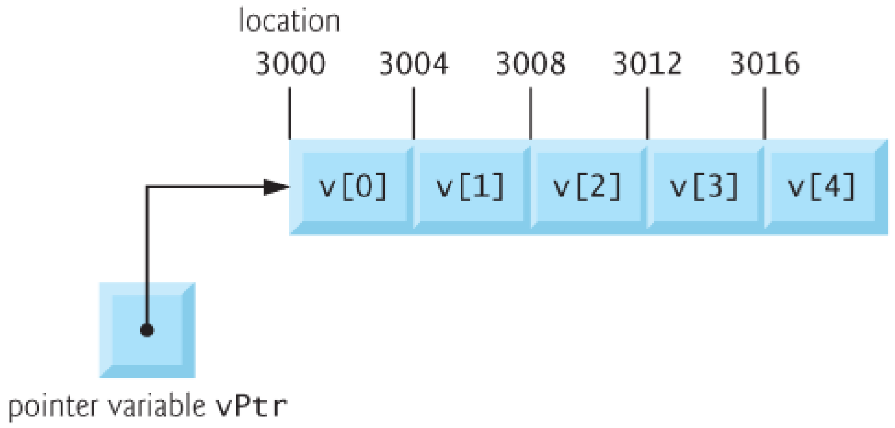
- Arrays are not passed using operator **&**
  - C automatically passes the starting location in memory of the array
  - The name of an array is equivalent to **&arrayName[0]**
  - When the compiler encounters a function parameter for a one-dimensional array of the form **int b[]**, the compiler converts the parameter to the pointer notation **int *b**.

Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or --)
  - Add an integer to a pointer(+ or += , - or -=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array or string

Pointer Expressions and Pointer Arithmetic – cont.

- Assume that array int v[5] has been defined on a machine with 4 byte `ints`
  - `vPtr` points to first element `v[ 0 ]`
    - at location `3000 (vPtr = 3000)`
  - `vPtr += 2;` sets `vPtr` to `3008`
    - `vPtr` points to `v[ 2 ]` (incremented by 2), but the machine has 4 byte `ints`, so it points to address `3008`



Array v and a pointer variable vPtr that points to v.

Pointer Expressions and Pointer Arithmetic – cont.

- In conventional arithmetic, 3000 + 2 yields the value 3002.

- This is normally not the case with pointer arithmetic.

- When an integer is added to or subtracted from a pointer, the pointer is *not* incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers.

- For example, the statement
  - vPtr += **2;**

would produce 3008 (3000 + 2 * 4), assuming an integer is stored in 4 bytes of memory.

Pointer Expressions and Pointer Arithmetic – cont.

- Subtracting pointers
  - If `vPtr` had been incremented to `3016`, which points to `v[4]`, the statement
    - `vPtr -= 4;`

    would set `vPtr` back to `3000`—the beginning of the array.

  - Increment/decrement operators
    - Either of the statements

      -
        ```
        ++vPtr;
        vPtr++;
        ```

      increments the pointer to point to the *next* location in the array.
    - Either of the statements

      -
        ```
        --vPtr;
        vPtr--;
        ```

      decrements the pointer to point to the *previous* element of the array

Pointer Expressions and Pointer Arithmetic – cont.

- Pointer variables may be subtracted from one another.
  - Returns number of elements from one to the other.

    ```
    vPtr2 = &v[ 2 ];  (vPtr2 = 3008)
    vPtr = &v[ 0 ];   (vPtr = 3000)
    ```

  - `x = vPtr2 - vPtr` // would assign to x the value of 2 (not 8)
- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - A common use of pointer comparison is determining whether a pointer is NULL.

Pointer Expressions and Pointer Arithmetic – cont.

- A pointer can be assigned to another pointer if both have the same type.
- The exceptional case:
  - `void *` (pointer to void)
  - A generic pointer that can represent *any* pointer type.
    - A memory location for an *unknown* data type.
  - All pointer types can be assigned a pointer to `void`, and a pointer to `void` can be assigned a pointer of any type.
  - In both cases, a cast operation is not required.
  - A pointer to `void` *cannot* be dereferenced.

Using the `const` Qualifier with Pointers

- `const` qualifier
  - Variable cannot be changed
  - Use `const`
    - If a variable should not change in the body of a function to which it's passed, the variable should be declared **const** to ensure that it's not accidentally modified.
  - Attempting to change a `const` variable produces an error
- `const` pointers
  - Always point to the same memory location
  - Must be initialized when defined

Const pointers

The declaration is read from *right to left.*

- `int *const myPtr = &x;`
  - Type `int *const` – constant pointer to an `int`
- `const int *myPtr = &x;`
  - Regular pointer to a `const int`
- `const int *const Ptr = &x;`
  - `const` pointer to a `const int`
  - `x` can be changed, but not `*Ptr`

Example – Constant Data

```
int main(void){
   int y; // define y

   f(&y);
}

void f(const int *xPtr){
   *xPtr = 100;   // Not allowed since xPtr is a pointer to a constant integer
}
```

Example – Constant Address

```
int main(void){

    int x; // define x
    int y; // define y

   // ptr is a constant pointer to an integer that can be modified  through ptr,
   // but ptr always points to the same memory location
    int *const ptr = &x;

    *ptr = 7;    // allowed: *ptr is not const
    ptr = &y;    // error: ptr is const; cannot assign new address
}
```

Example – Constant Address and Data

```
int main(void){
   int x = 5;   // initialize x
   int y;       // define y

  // ptr is a constant pointer to a constant integer.
  // ptr always points to the same location; the integer at that location cannot be modified
   const int *const ptr = &x; // initialization is OK

   printf("%d\n", *ptr);
   *ptr = 7;     // error: *ptr is const; cannot assign new value
   ptr = &y;     // error: ptr is const; cannot assign new address
}
```

sizeof Operator

- Determine the size in bytes of any data type.

- **sizeof** operator returns the total number of bytes as type **size_t**.
  - **printf("Size of an integer %u", sizeof(int));**

- Consider the following array definition:
  - **double real[22];**

- Determine the number of elements in the array
  - **sizeof(real) / sizeof(real[0])**

Relationship Between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An array name can be thought of as a constant pointer.
- Assume that integer array **b[5]** and integer pointer variable **bPtr** have been defined.
  - `bPtr = b;`
- Array element **`b[ 3 ]`**
  - Can be accessed by **`*( bPtr + 3 )`**
    - Where 3 (or simply n) is the offset. Called pointer/offset notation
  - Can be accessed by **`bptr[ 3 ]`**
    - Called pointer/subscript notation
  - Can be accessed by performing pointer arithmetic on the array itself
    - **`bPtr[ 3 ]`** same as **`b[ 3 ]`**
    - **`*( b + 3 )`**

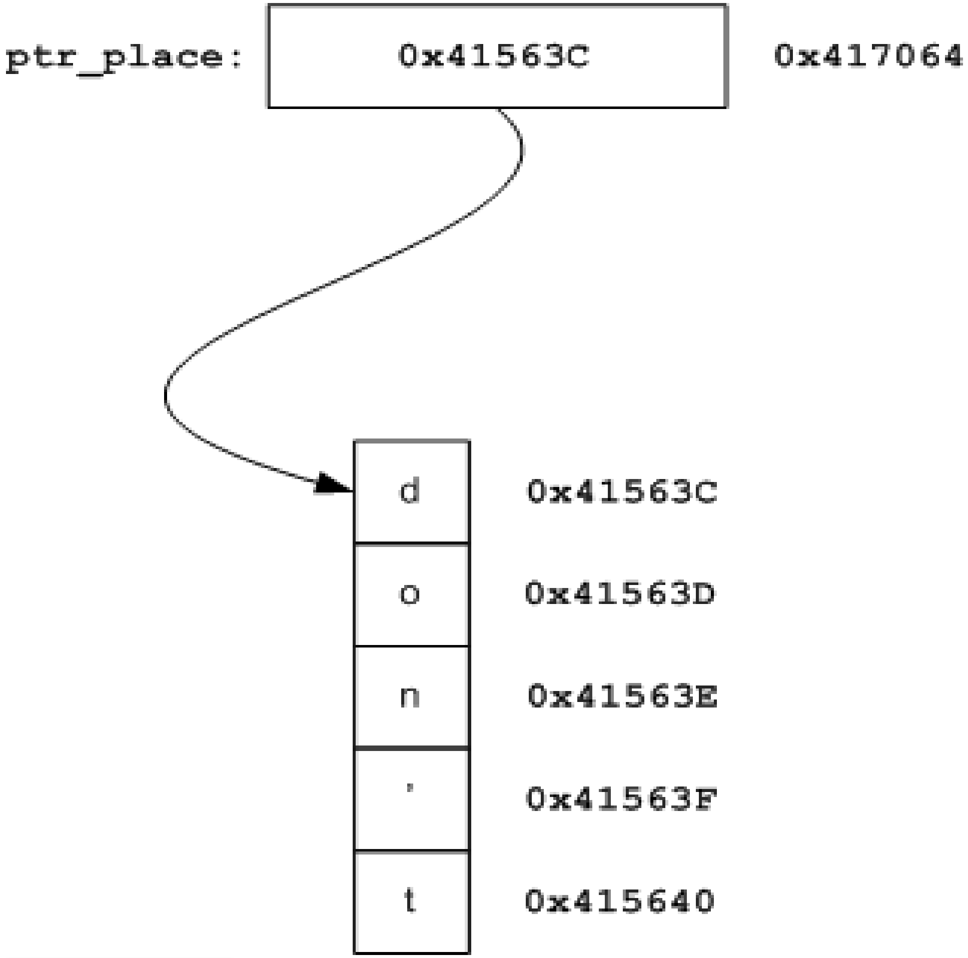
Examples

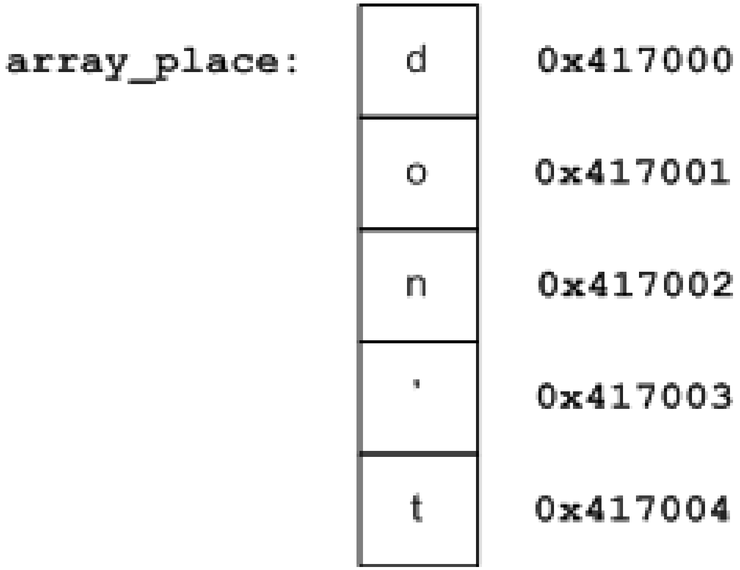
- array_example.c
- string_example.c
- pointers_vs_arrays.c

Pointers vs. Arrays

```
char array_place[12] = "don't panic";
char* ptr_place = "don't panic";

char a = array_place[7];
char b = ptr_place[7];
```

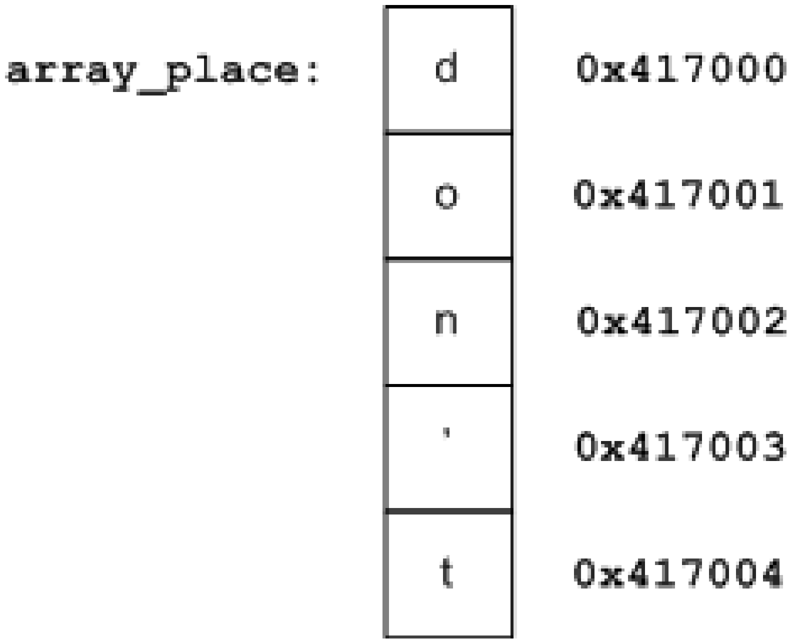


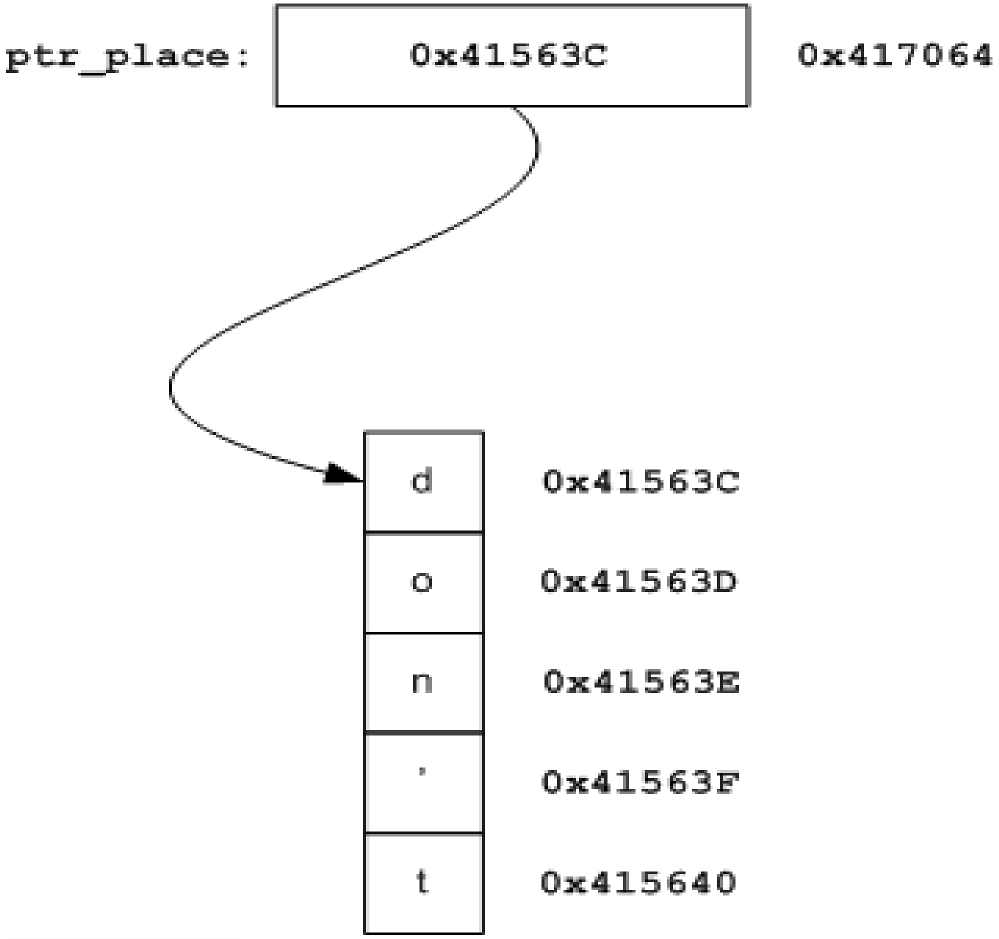


Pointer and Array Traversal

```
for (i = 0; i < sizeof(array_place); ++i)
    printf("%c ", array_place[i]);

for (; *ptr_place; ++ptr_place)
    printf("%c ", *ptr_place);
```

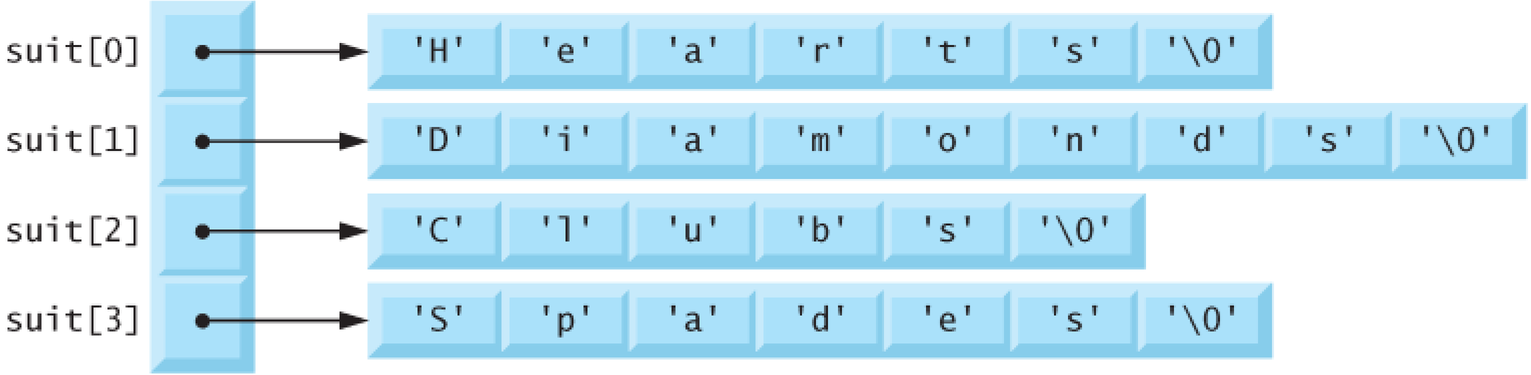




Array of Pointers

- Arrays may contain pointers.
- A common use of an array of pointers is to form an array of strings, referred to simply as a string array.
- Each entry in an array of strings is actually a pointer to the first character of a string.

- `const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};`


suit[0]
suit[1]
suit[2]
suit[3]
'H' 'e' 'a' 'r' 't' 's' '\0'
'D' 'i' 'a' 'm' 'o' 'n' 'd' 's' '\0'
'C' 'l' 'u' 'b' 's' '\0'
'S' 'p' 'a' 'd' 'e' 's' '\0'