Pointers

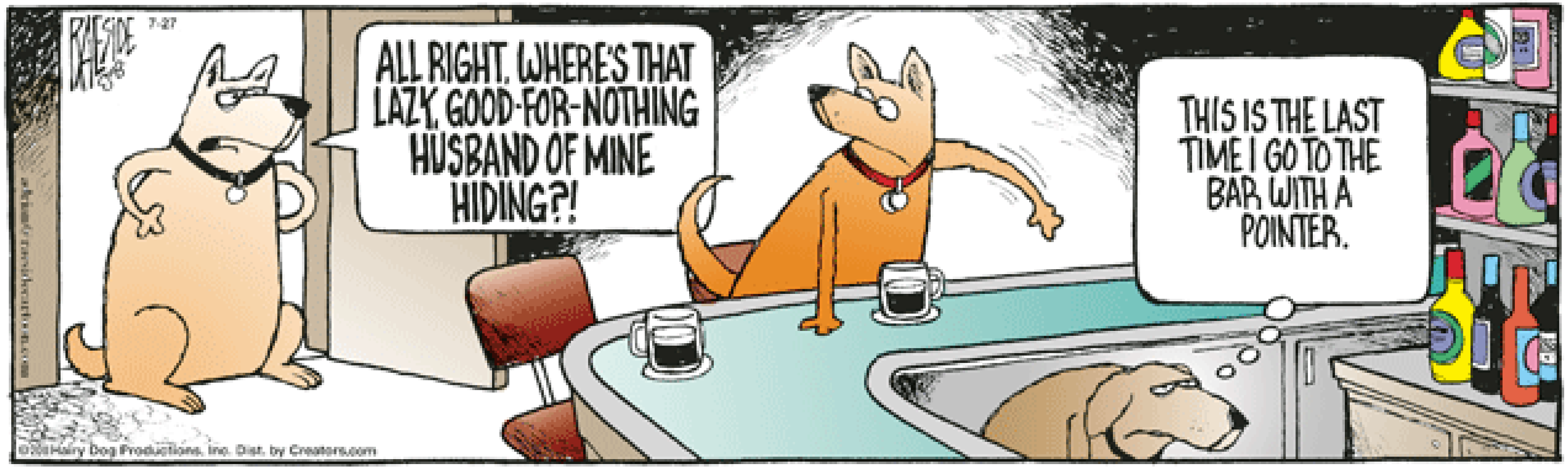
PC FAF 2019

By indirection find direction out.

—William Shakespeare

Many things, having full reference to one consent, may work contrariously.

—William Shakespeare


ALL RIGHT, WHERE'S THAT LAZY, GOOD-FOR-NOTHING HUSBAND OF MINE HIDING?!
THIS IS THE LAST TIME I GO TO THE BAR WITH A POINTER.
RAESIDE
7-27
©2018 Hairy Dog Productions, Inc. Dist. by Creators.com

Why pointers?

Pointers are among C’s most difficult capabilities to master. Pointers enable programs to simulate pass-by-reference, to pass functions between functions, and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees.

Pointer.

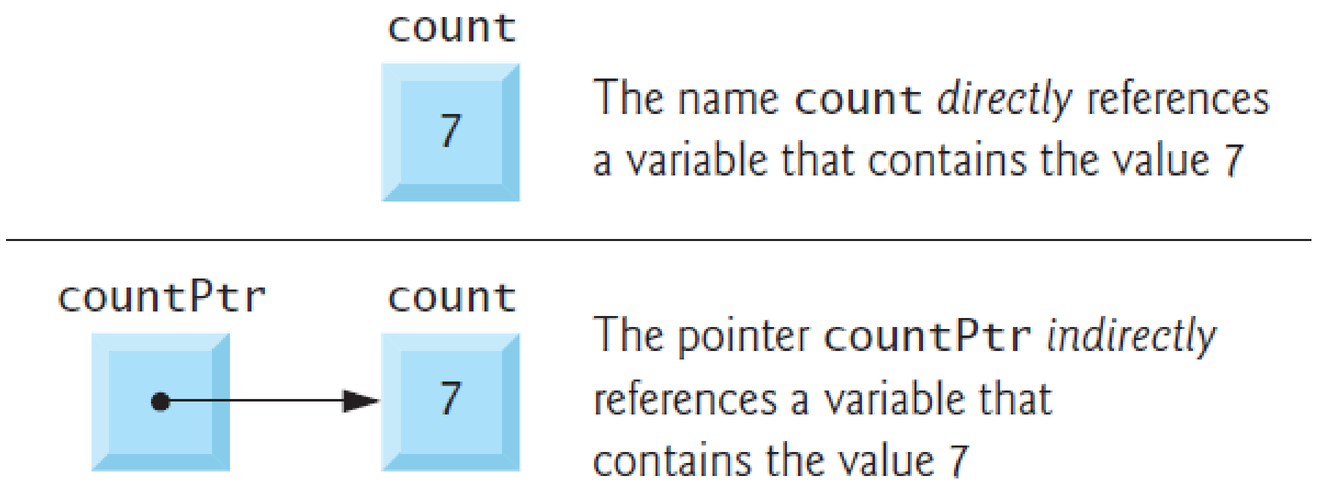
Philosophy

Pointers are variables whose values are *memory addresses*.

Normally, a variable directly contains a specific value. A pointer, on the other hand, contains an *address* of a variable that contains a specific value.

In this sense, a variable name *directly* references a value, and a pointer *indirectly* references a value.

Referencing a value through a pointer is called indirection.


count
7
The name count *directly* references a variable that contains the value 7
countPtr
count
7
The pointer countPtr *indirectly* references a variable that contains the value 7

Pointers.

Declaration

```
int *countPtr, count;
```

Pointers, like all variables, must be defined before they can be used. The definition specifies that variable `countPtr` is of type int * (i.e., a pointer to an integer) and is read (right to left), "`countPtr` is a pointer to int" or "`countPtr` points to an object of type int."

Also, the variable `count` is defined to be an int, *not* a pointer to an int. The * applies *only* to `countPtr` in the definition. When * is used in this manner in a definition, it indicates that the variable being defined is a pointer.

Initializing and Assigning Values to Pointers

Pointers should be initialized when they're defined, or they can be assigned a value. A pointer may be initialized to `NULL`, `0` or an address.

A pointer with the value `NULL` points to *nothing*. `NULL` is a *symbolic constant* defined in the <stdio.h>.

Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred. When `0` is assigned, it's first converted to a pointer of the appropriate type. The value `0` is the *only* integer value that can be assigned directly to a pointer variable.

Pointer Operators

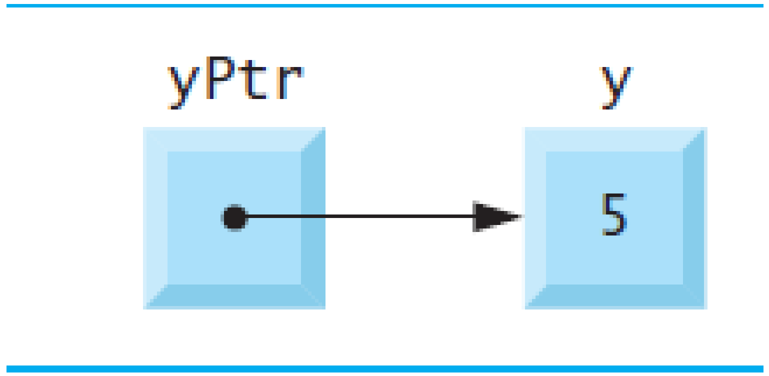
&

The **&**, or **address operator**, is a unary operator that returns the *address* of its operand. For example, assuming the definitions

```
int y = 5;
int *yPtr;
```

the statement

```
yPtr = &y;
```



assigns the *address* of the variable `y` to pointer variable `yPtr`. Variable `yPtr` is then said to "point to" `y`.

Examples

(old slides in RO)

Declarare:

```
     <tip> *<nume variabilă>;
```

Exemple:

```
char *c, d, *e, f;
int  *tab[6], k, *y;
float *z, q, *tab_2[34];
```

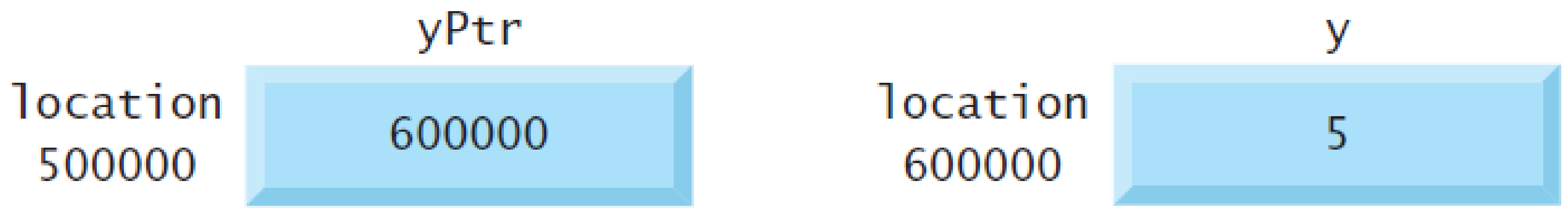
1. Declară doi pointeri către variabile de tip char (c, e)
2. Declară un pointer către o variabilă simplă de tip întreg (y) şi un pointer către un tablou unidimensional (tab)
3. Declară un pointer către o variabilă simplă de tip real (z) şi un pointer către un tablou liniar cu elemente de tip real

Common Programming Error (1)

The asterisk **`(*)`** *notation used to declare pointer variables does* **not** *distribute to all variable names in a declaration. Each pointer must be declared with the* **`*`** *prefixed to the name; e.g., if you wish to declare* **`xPtr`** *and* **`yPtr`** *as int pointers, use int* **`*xPtr, *yPtr;`**.

Pointer Representation in Memory

Figure shows the representation of the pointer in memory, assuming that integer variable `y` is stored at location `600000`, and pointer variable `yPtr` is stored at location `500000`.

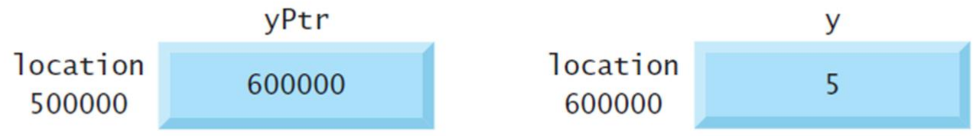


The operand of the address operator must be a variable; the address operator *cannot* be applied to constants or expressions.

Pointer Operators

*

(indirection)



The unary * operator, commonly referred to as the **indirection operator** or **dereferencing operator**, returns the *value* of the object to which its operand (i.e., a pointer) points. For example, the statement

```
printf( "%d", *yPtr );
```

prints the value of variable y, namely 5. Using * in this manner is called **dereferencing a pointer**.

Common Programming Error (2)

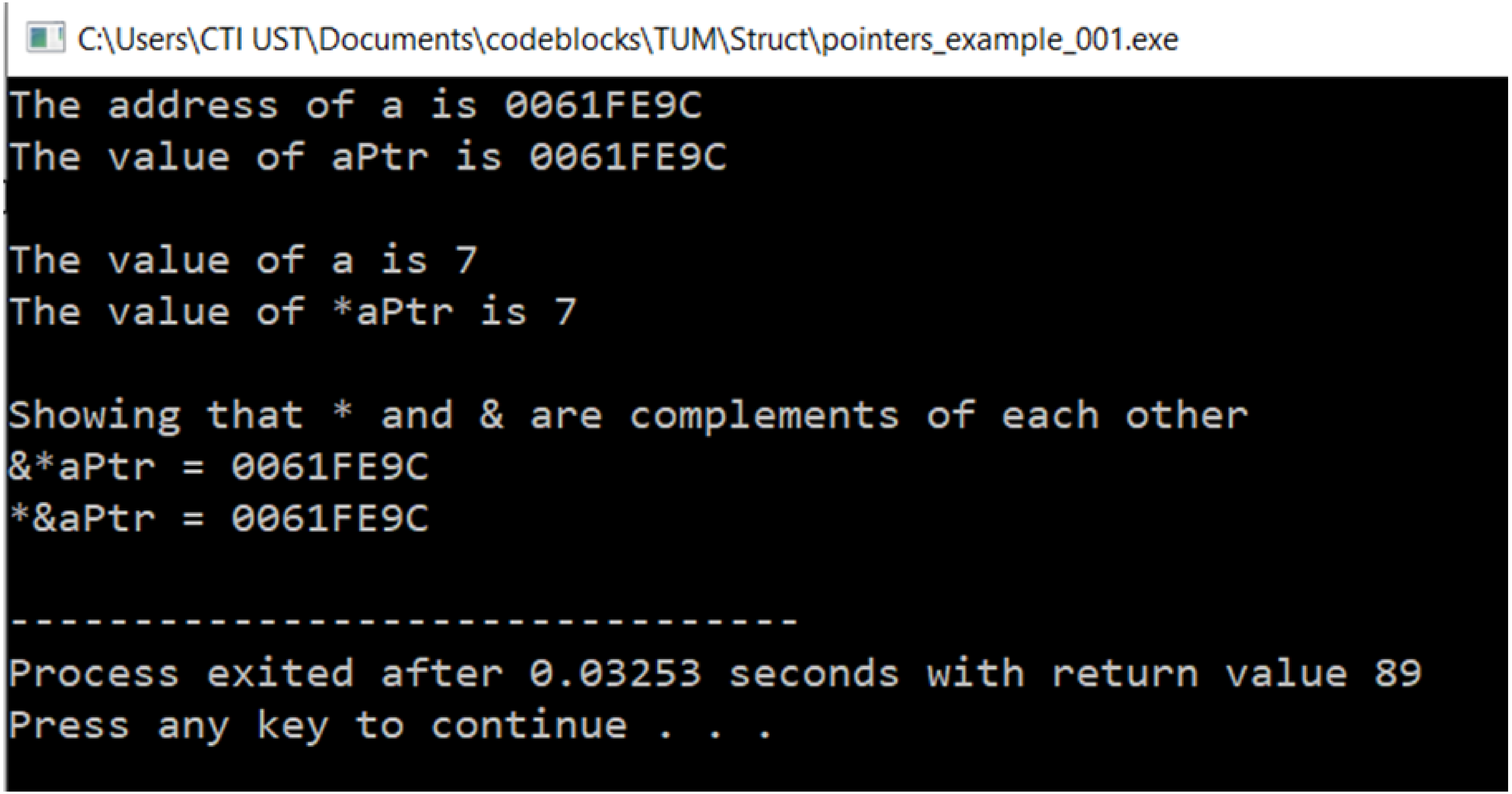
Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results

*Demonstrating the & and * Operators*

Notice that the *address* of a and the *value* of `aPtr` are identical in the output, thus confirming that the address of `a` is indeed assigned to the pointer variable `aPtr` (line 11). The `&` and `*` operators are complements of one another— when they're both applied consecutively to `aPtr` in either order (line 21), the same result is printed.

```c
// Fig. 7.4: fig07_04.c
// Using the & and * pointer operators.
#include <stdio.h>

int main( void )
{
   int a; // a is an integer
   int *aPtr; // aPtr is a pointer to an integer

   a = 7;
   aPtr = &a; // set aPtr to the address of a

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
} // end main
```

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```



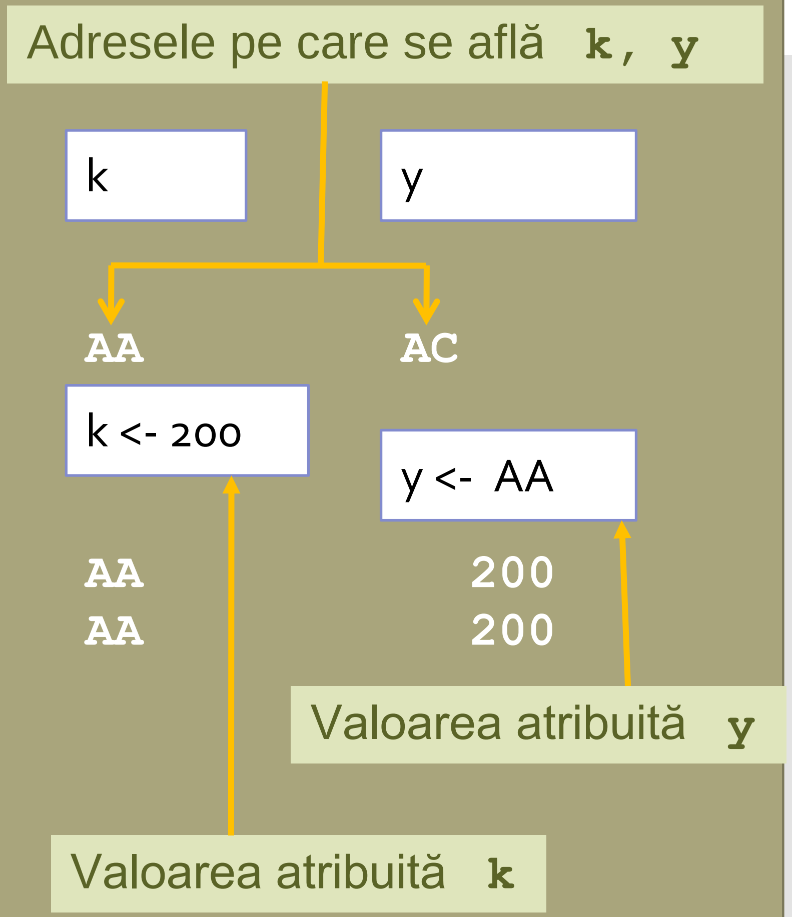
C:\Users\CTI UST\Documents\codeblocks\TUM\Struct\pointers_example_001.exe

```
The address of a is 0061FE9C
The value of aPtr is 0061FE9C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0061FE9C
*&aPtr = 0061FE9C


--------------------------------
Process exited after 0.03253 seconds with return value 89
Press any key to continue . . .
```

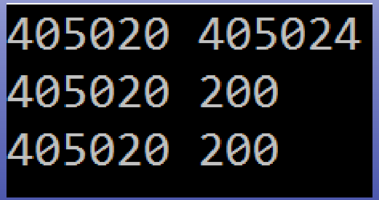
Examples (2)

(old slides in RO)

```c
# include <stdio.h>

int k, *y;
void main()
{
    printf("%X %X\n", &k, &y);
    k = 200;
    y = &k;
    printf("%X %d\n", y, *y);
    printf("%X %d\n", y, k);
}
```



k	–	variabila k de tip întreg
&k	–	adresa variabilei k
y = &k;	–	variabila y primeşte adresa variabilei k
y	–	adresa variabilei k,
***y**	–	valoarea, care se conține pe adresa din **y**



```
405020 405024
405020 200
405020 200
```

Pointers and functions

Pass arguments to a function

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. `Return` may be used to return one value from a called function to a caller (or to return control from a called function without passing back a value). Many functions require the capability to *modify variables in the caller* or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the time and memory overheads of making a copy of the object).

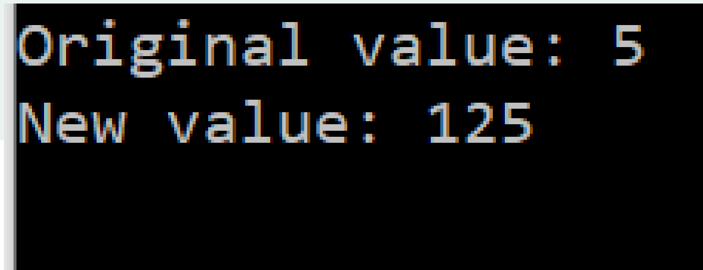
In C, you use pointers and the indirection operator to *simulate* pass-by-reference. When calling a function with arguments that should be modified, the *addresses* of the arguments are passed. This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified. As you know, arrays are *not* passed using operator & because C automatically passes the starting location in memory of the array (the name of an array is equivalent to `&arrayName[0]`). When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller’s memory.

Pass by value

Next program passes the variable `number` by value to function `cubeByValue`

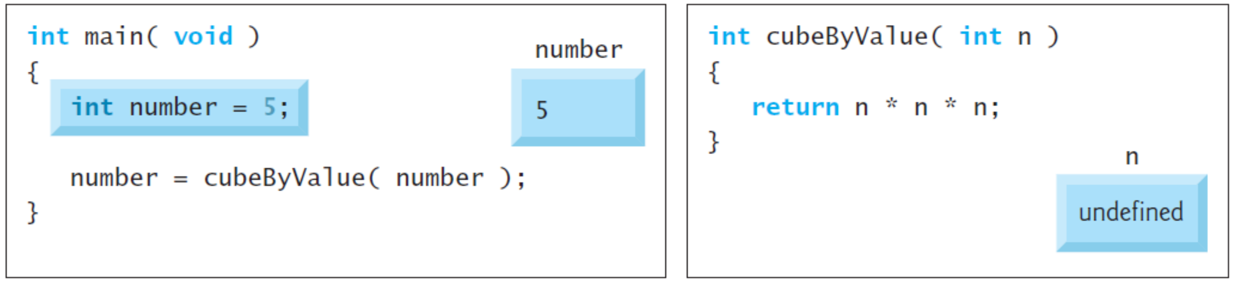
The `cubeByValue` function cubes its argument and passes the new value back to main using a `return` statement. The new value is assigned to `number` in main.

```
// Cube a variable using pass-by-value.

#include <stdio.h>

int cubeByValue( int n ); // prototype

int main( void )
{
    int number = 5;
    printf( "Original value: %d", number );
// pass number by value to cubeByValue
    number = cubeByValue( number );
    printf( "\nNew value: %d\n", number );
}

// return cube of integer argument
int cubeByValue( int n )
{
    return n * n * n;
}
```



```
Original value: 5
New value: 125
```

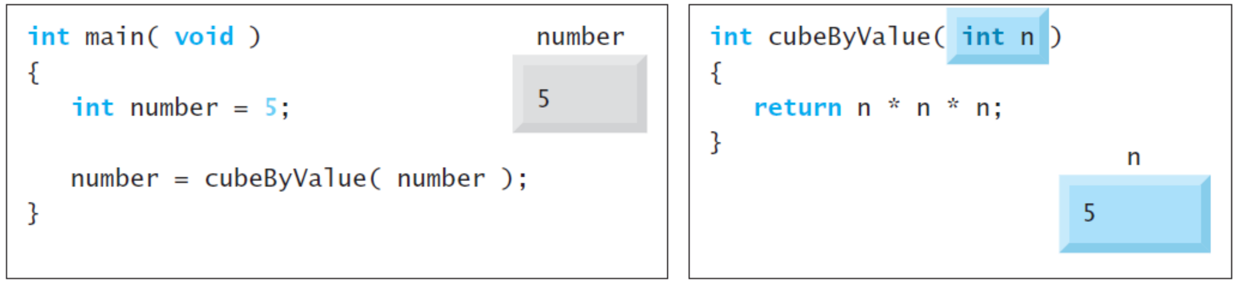
Step 1: Before `main` calls `cubeByValue`:



```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

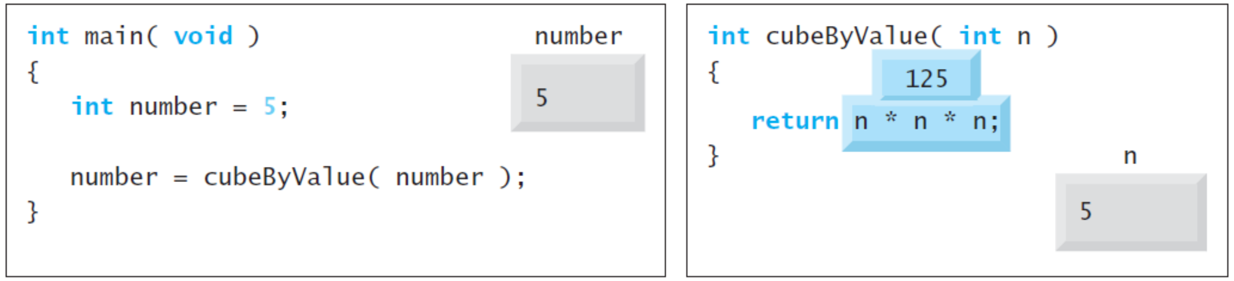
Step 2: After `cubeByValue` receives the call:



```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

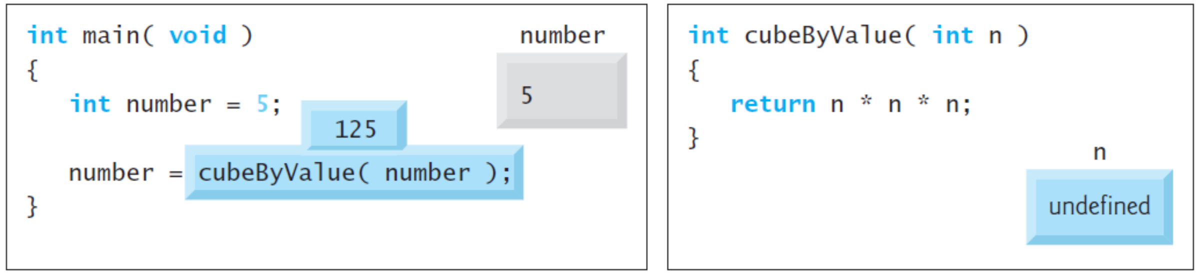
Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

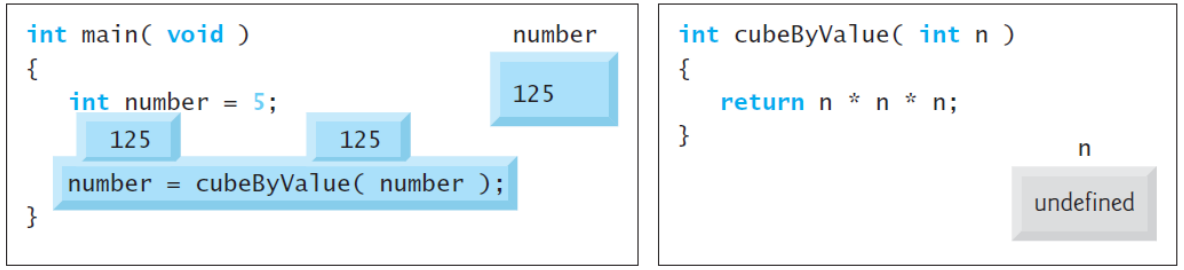
Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:



```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

Step 5: After `main` completes the assignment to `number`:



```
int main( void )
{
   int number = 5;

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

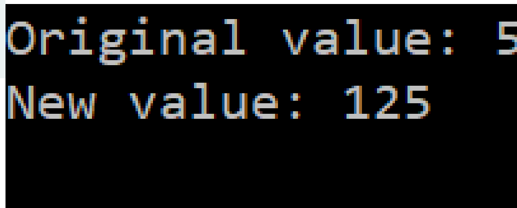
Pass by reference

The header specifies that `cubeByReference` *receives* the *address* of an integer variable as an argument, stores the address locally in `nPtr` and does not return a value.

Next program pass the variable `number` by reference — the address of number is passed — to function `cubeByReference`.

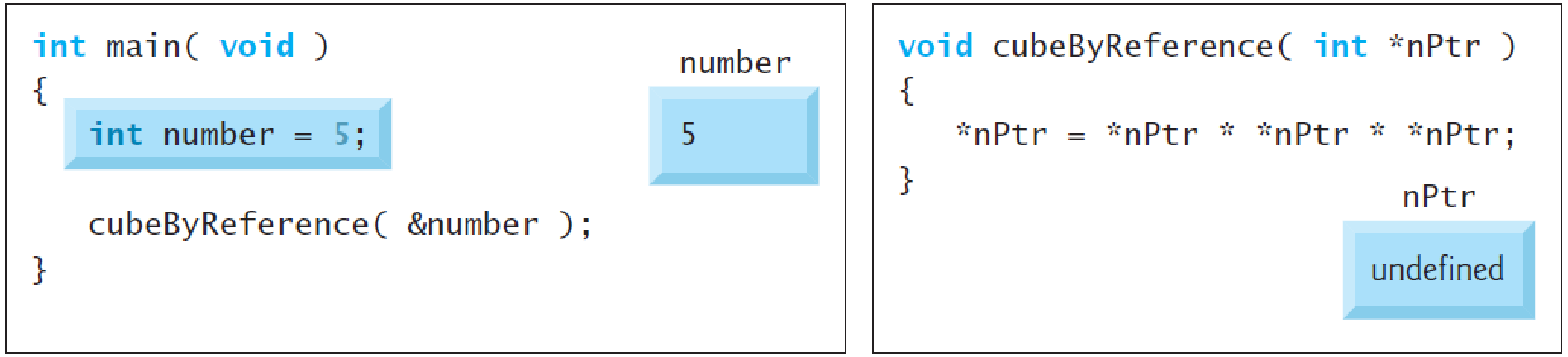
Function `cubeByReference` takes as a parameter a pointer to an int called `nPtr`. The function *dereferences* the pointer and cubes the value to which `nPtr` points, then assigns the result to `*nPtr` (which is really number in main), thus changing the value of number in main.

```
// Cube a variable using pass-by-refernce.

#include <stdio.h>

void cubeByRef( int *nPtr );

int main( void )
{
   int number = 5;
   printf( “Original value: %d", number );
// pass number by ref to cubeByRef
   cubeByRef( &number );
   printf( "\nNew value: %d\n", number );
}

// return cube of integer argument
int cubeByRef( int *nPtr )
{
   return *nPtr * *nPtr * *nPtr;
}
```



```
Original value: 5
New value: 125
```

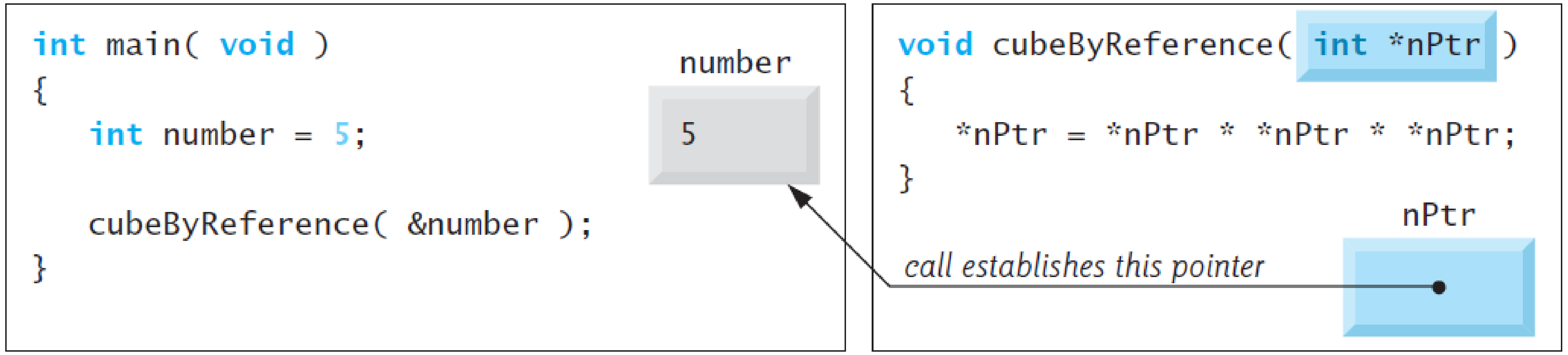
Step 1: Before `main` calls `cubeByReference`:



```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

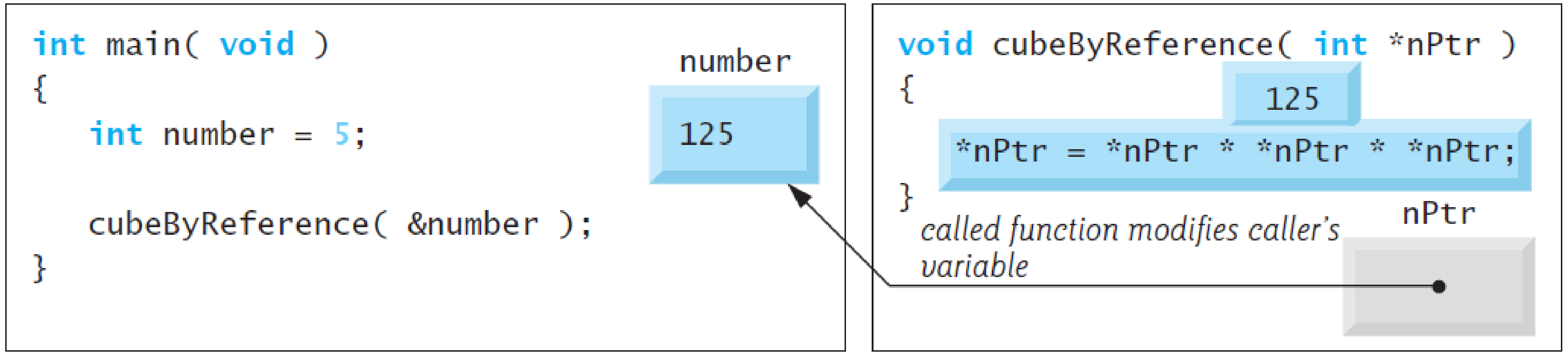
Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Step 3: After `*nPtr` is cubed and before program control returns to `main`:



```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

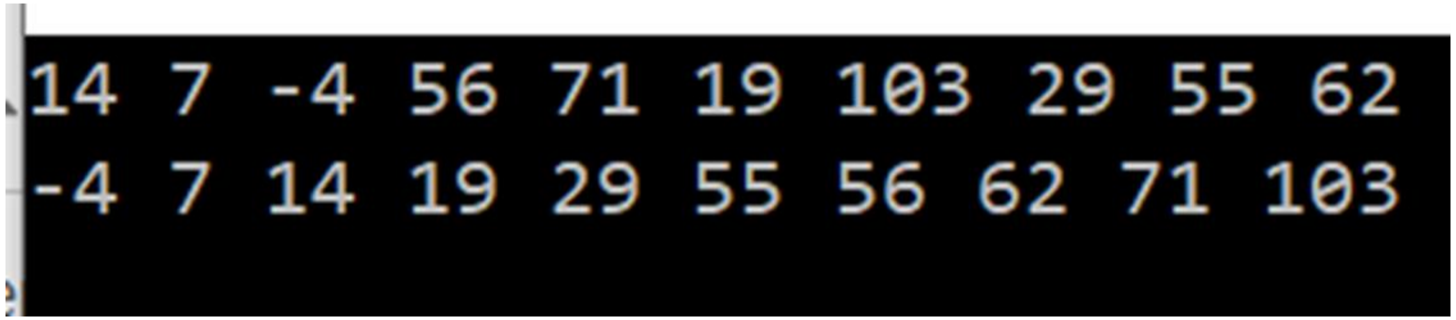
```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Swap example

```
#include <stdio.h>

void swap( int *aPtr, int *bPtr )
{
        int tz = *aPtr;
        *aPtr = *bPtr;
        *bPtr = tz;
}

int main( void )
{
        int a = 5, b = 7;
        printf( "Original values: %d %d", a, b );

        swap( &a, &b );

        printf( "\nNew values:  %d %d", a, b );
}
```

```c
#include <stdio.h>

void swap( int *aPtr, int *bPtr )
{
    int tz = *aPtr;
    *aPtr = *bPtr;
    *bPtr = tz;
}

void bubble(int *a, int n)
{
int i, j;
for (i = 0; i < n - 1; i++)
 for(j = 0; j< n - 1; j++)
  if (a[j] > a[j+1])  swap(&a[j], &a[j+1]);
}
```

```c
void print(int *a, int n)
{
    int i;
    for (i = 0; i < n; i++)
       printf("%d ", a[i]);
    printf("\n");
}

int main( void )
{
    int a[100] = { 14, 7, -4, 56, 71,
                   19, 103, 29, 55, 62},
                   n = 10;
    print(a, n);
    bubble(a,n);
    print(a, n);
}
```



```
14 7 -4 56 71 19 103 29 55 62
-4 7 14 19 29 55 56 62 71 103
```

sizeof()

`size_t`.

Alias of one of the fundamental unsigned integer types. It is a type able to represent the size of any object in bytes: `size_t` is the type returned by the `sizeof` operator and is widely used in the standard library to represent sizes and counts.

sizeof example 1

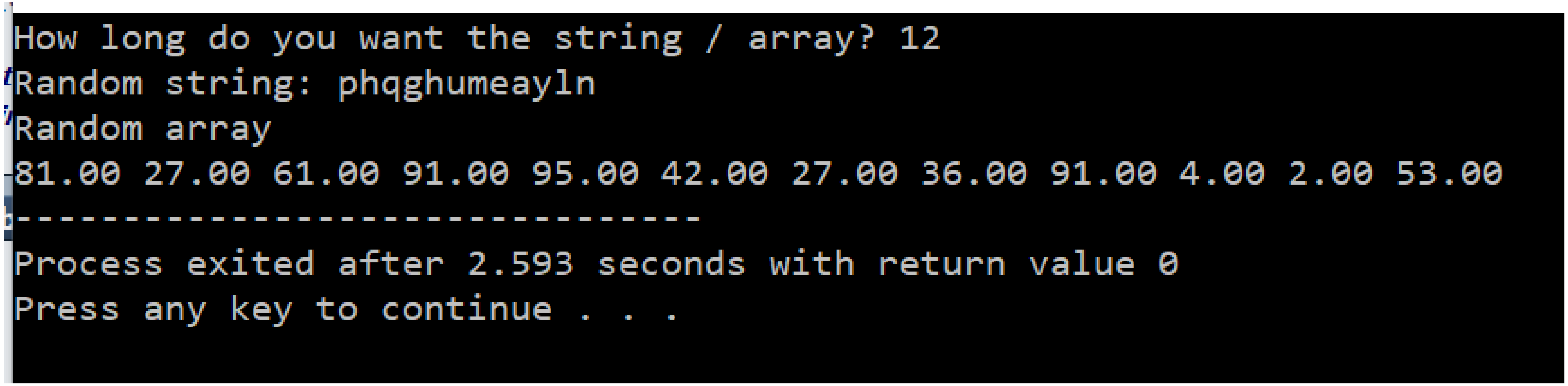
static array

```
#include <stdio.h>
#define SIZE 20

size_t getSize(float *ptr ); // prototype

int main( void )
{
float array[ SIZE ]; // create array
printf( "The number of bytes in the array is %u \nThe
number of bytes returned by getSize is %u\n",
sizeof( array ), getSize( array ) );

} // end main

size_t getSize(float *ptr )
 {
       return sizeof( ptr );
 }
```

sizeof example 2

dynamic array

```c
#include <stdio.h>      /* printf, scanf, NULL */
#include <stdlib.h>     /* malloc, free, rand */

int main ()
{
  int i,n;  char  *buffer;  float *buff;

  printf ("How long the string / array? ");  scanf ("%d", &n);

  buffer = (char*) malloc (n+1); if (buffer==NULL) exit(1);
  buff = (float*) calloc (n, sizeof(float)); if (buff==NULL) exit(1);

  for (i = 0; i < n; i++) buffer[i] = rand() % 26 + 'a';
  buffer[n] = '\0';

  for (i = 0; i < n; i++) buff[i] = rand() % 100;

  printf ("Random string: %s\n",buffer);
  free (buffer);
  printf ("Random array \n");
  for (i = 0; i < n; i++) printf ("%.2f ", buff[i]);
  free (buff);

  return 0;
}
```



```
How long do you want the string / array? 12
Random string: phqghumeayln
Random array
81.00 27.00 61.00 91.00 95.00 42.00 27.00 36.00 91.00 4.00 2.00 53.00
--------------------------------
Process exited after 2.593 seconds with return value 0
Press any key to continue . . .
```