German Jordanian University الجامعة الألمانية الأردنية

# Pointers

Most of the slides in this lecture are either from or adapted from the slides provided by Dr. Christina Class

A pointer is a variable that stores an address.

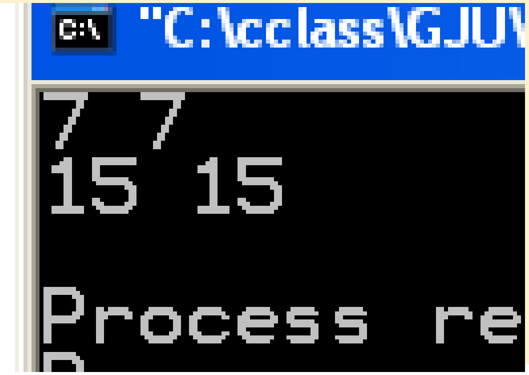
It can store the address of another variable.

A pointer in C is bound to a data type, that it stores the address of values of a specific type.

```
type *name;
int *myPointer;
```

```
#include <stdio.h>

int main()
{
    // declare variable
    int x = 7;
    // declare pointer
    int *pointer = &x;

    printf("%d %d\n", x, *pointer);

    *pointer = 15;

    printf("%d %d\n", x, *pointer);


    return 0;
}
```

| Var | Addr | Value |
|---|---|---|
| x | 100 | ~~7~~ 15 |
| | 101 | |
| | 102 | |
| p | 103 | *adr100* |
| | 104 | |
| | 105 | |



```
7 7
15 15

Process re
P
```

The address operator & returns the address of a variable.

The indirection operator * returns the content at the address stored in a pointer variable.

```
int x;
```

| Var | Addr | Value |
|---|---|---|
| x | 100 | ??? |
| | 101 | |
| | 102 | |
| | 103 | |
| | 104 | |
| | 105 | |
| | 106 | |
| | 107 | |
| | 108 | |

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

| Var | Addr | Value |
|---|---|---|
| x | 100 | ??? |
| p | 101 | ??? |
| | 102 | |
| | 103 | |
| | 104 | |
| | 105 | |
| | 106 | |
| | 107 | |
| | 108 | |

```
int x;
int *p;
x = 7;
```

| Var | Addr | Value |
|---|---|---|
| x | 100 | 7 |
| p | 101 | ??? |
| | 102 | |
| | 103 | |
| | 104 | |
| | 105 | |
| | 106 | |
| | 107 | |
| | 108 | |

```
int x;
int *p;
x = 7;
p = &x;
```

| Var | Addr | Value |
|---|---|---|
| x | 100 | 7 |
| p | 101 | *adr100* |
| | 102 | |
| | 103 | |
| | 104 | |
| | 105 | |
| | 106 | |
| | 107 | |
| | 108 | |

```
int x;
int *p;
x = 7;
p = &x;
*p = 15;
```

| Var | Addr | Value |
|---|---|---|
| x | 100 | ~~7~~ 15 |
| p | 101 | *adr100* |
|  | 102 |  |
|  | 103 |  |
|  | 104 |  |
|  | 105 |  |
|  | 106 |  |
|  | 107 |  |
|  | 108 |  |

Given are the following definitions:

- int a = 6, *p = &a;

instead of a we can use *p:

- a = 17; ➔ *p = 17;
- printf(“a=%d”,a); ➔ printf(“a=%d”,*p);
- if (a ==8) ➔ if (*p==8)

instead of &a we can use p:

- scanf(“%d”,&a); ➔ scanf(“%d”,p);

**A pointer variable itself has also an address.**

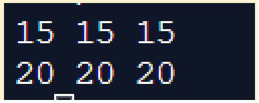
**We can define a pointer that has as value the address of another pointer.**

```
int a = 13;
int *p = &a;
int **pp = &p;
```

| Var | Addr | Value |
|---|---|---|
| a | 100 | 13 |
| p | 101 | *adr100* |
| pp | 102 | *adr101* |
| | 103 | |
| | 104 | |
| | 105 | |
| | 106 | |
| | 107 | |
| | 108 | |

# Example – pointer to a pointer

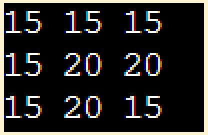
```c
#include <stdio.h>

int main(void) {
   int a = 13;

   int *p = &a;
   int **pp = &p;

   **pp = 15;
   printf("%d %d %d\n", a, *p, **pp);

   *p = 20;
   printf("%d %d %d\n", a, *p, **pp);
   return 0;
}
```



```
15 15 15
20 20 20
```

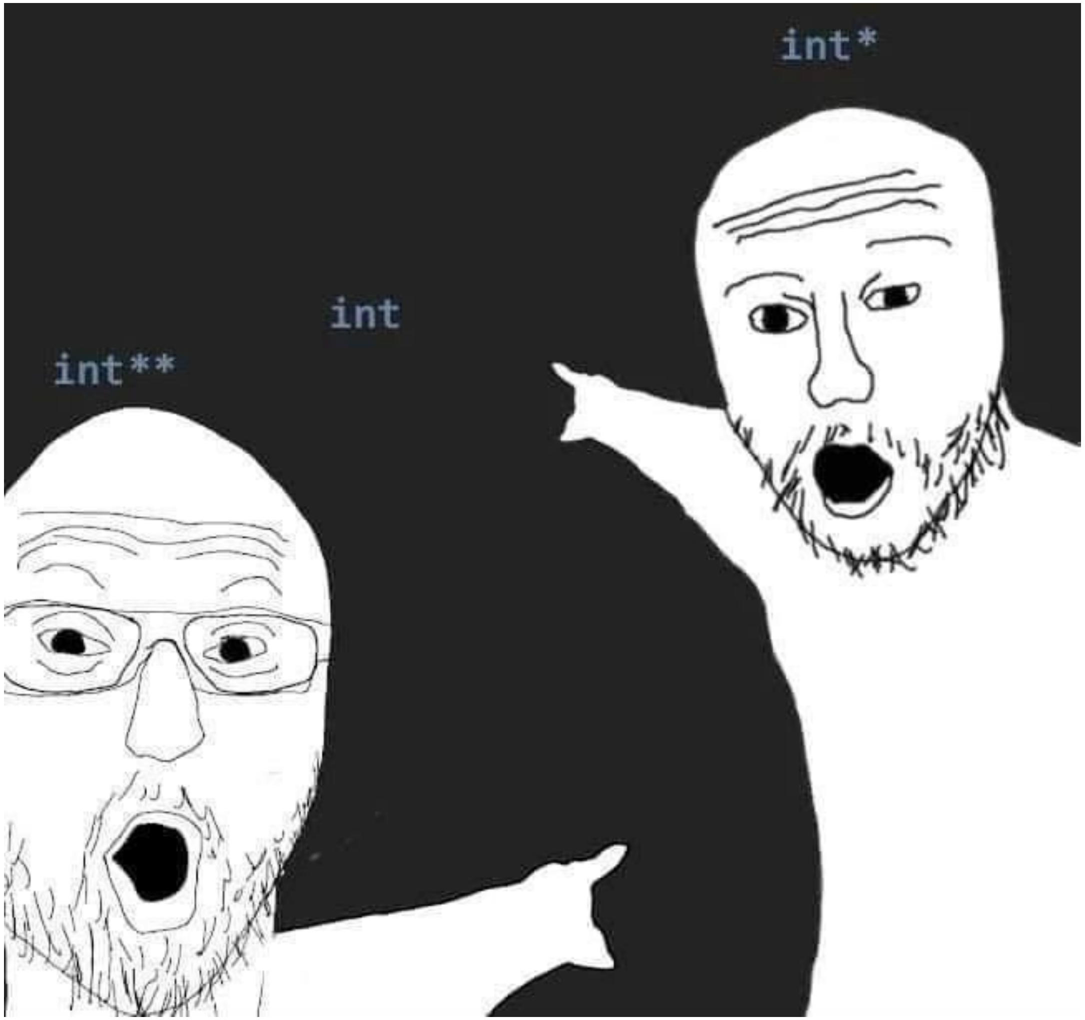
# Example – pointer to a pointer

```
#include <stdio.h>

int main(void) {
    int a = 13, b = 3;
    int *p1 = &a, *p2;
    int **pp = &p1;
    **pp = 15;
    printf("%d %d %d\n", a, *p1, **pp);
    *pp = &b;
    *p1 = 20;
    printf("%d %d %d\n", a, *p1, **pp);
    pp = &p2;
    p2 = &a;
    printf("%d %d %d\n", a, *p1, **pp);

    return 0;
}
```



```
15 15 15
15 20 20
15 20 15
```

# Pointers!



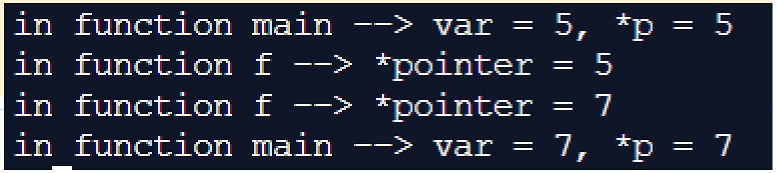
# using call-by-reference

```c
#include <stdio.h>
void read(int *a)
{
    printf("Please enter a value: ");
    scanf("%d",a);
}
int main()
{
    int value;
    read(&value);
    printf("The entered value is %d.\n", value);
    return 0;
}
```

# call-by-reference

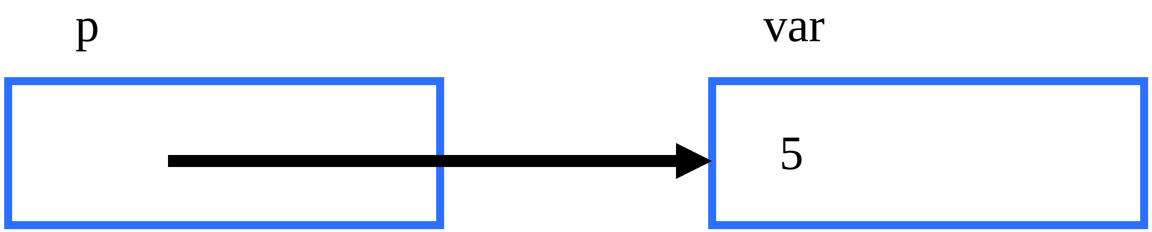
- in C is a “simulation”, as always a value is copied and passed to the function
- the question is simply: what is copied?

```c
#include <stdio.h>

void f (int *pointer){
  printf("in function f --> *pointer = %d\n", *pointer);
  *pointer = 7;
  printf("in function f --> *pointer = %d\n", *pointer);
}

int main(void) {
  int var = 5;
  int *p = &var;
  printf("in function main --> var = %d, *p = %d\n", var, *p);
  f(p);
  printf("in function main --> var = %d, *p = %d\n", var, *p);
  return 0;
}
```

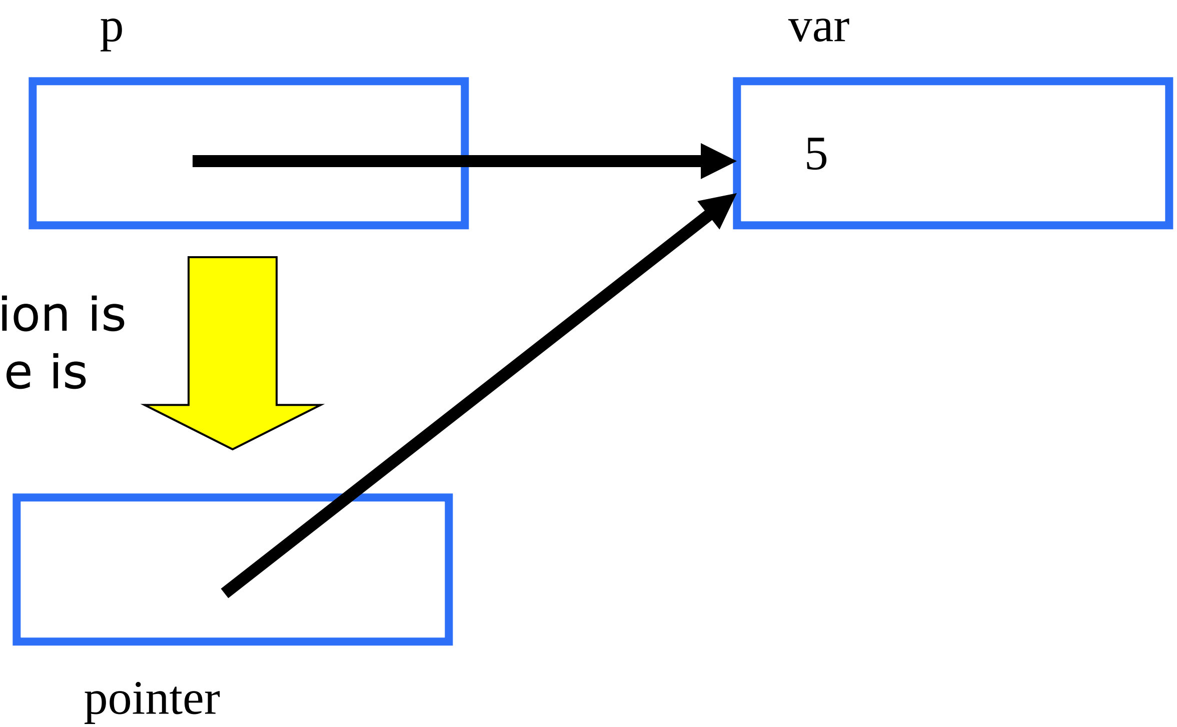


```
in function main --> var = 5, *p = 5
in function f --> *pointer = 5
in function f --> *pointer = 7
in function main --> var = 7, *p = 7
```

```
void function (int *pointer);
int var = 5, *p = &var;
```

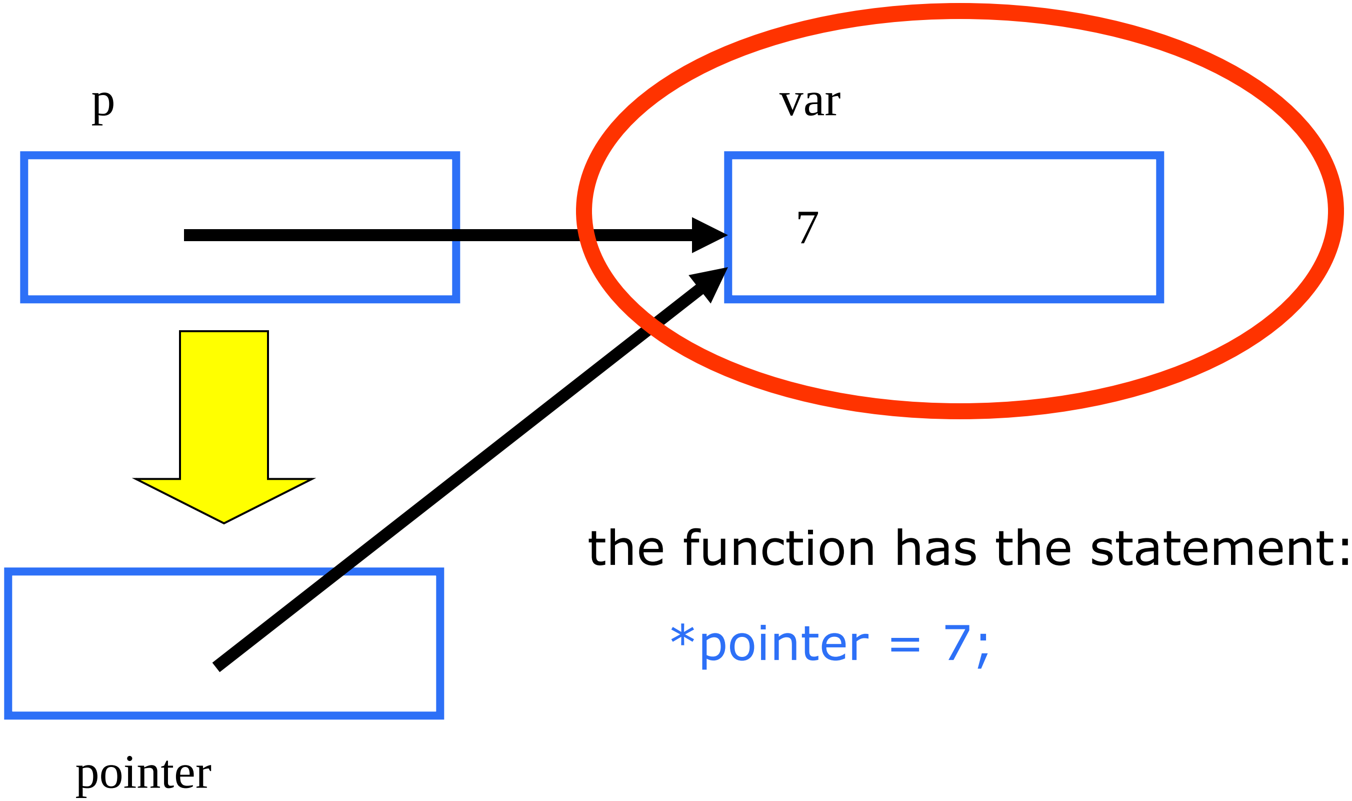


```
void function (int *pointer);
int var = 5, *p = &var;
function (p);
```



when the function is called, the value is copied

```
void function (int *pointer);
int var = 5, *p = &var;
function (p);
```

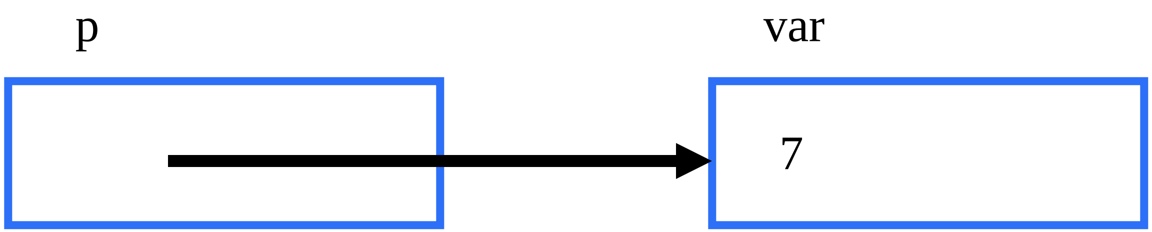


the function has the statement:

```
*pointer = 7;
```

we see the changes in the original variable

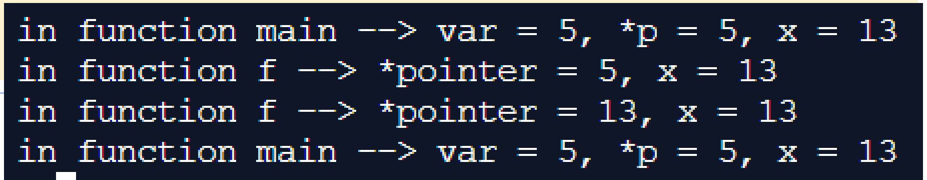
```
void function (int *pointer);
int var = 5, *p = &var;
function (p);
```



after the function call is finished

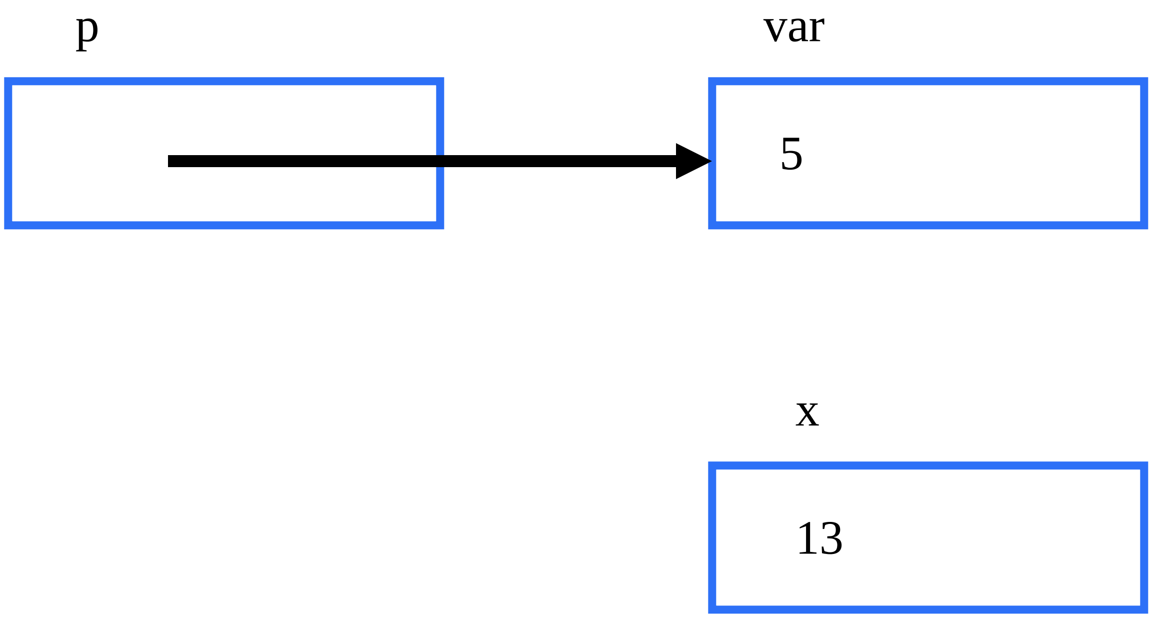
the changes are still there

```c
#include <stdio.h>

int x = 13;//global variable

void f (int *pointer){
 printf("in function f --> *pointer = %d, x = %d\n", *pointer, x);
 pointer = &x;
 printf("in function f --> *pointer = %d, x = %d\n", *pointer, x);
}

int main(void) {
 int var = 5;
 int *p = &var;
 printf("in function main --> var = %d, *p = %d, x = %d\n", var, *p, x);
 f(p);
 printf("in function main --> var = %d, *p = %d, x = %d\n", var, *p, x);
 return 0;
}
```

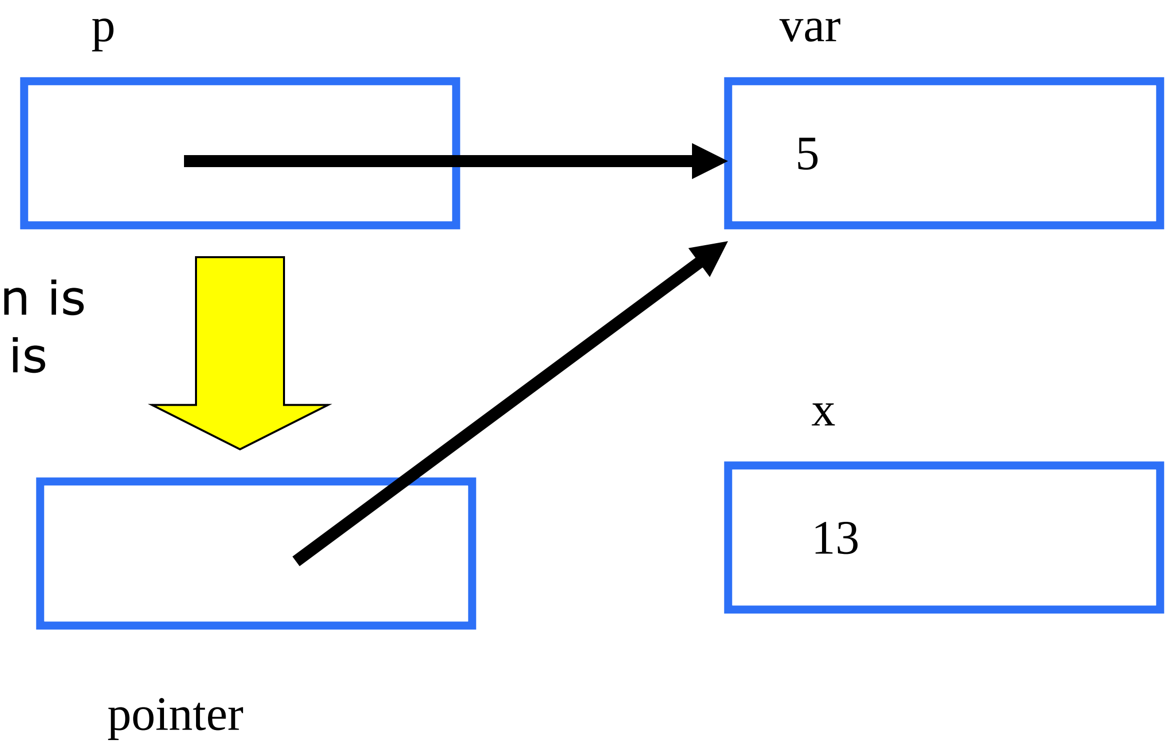


```
in function main --> var = 5, *p = 5, x = 13
in function f --> *pointer = 5, x = 13
in function f --> *pointer = 13, x = 13
in function main --> var = 5, *p = 5, x = 13
```

```
void function (int *pointer);
int var= 5, *p = &var;
int x = 13; // this is a global variable
```

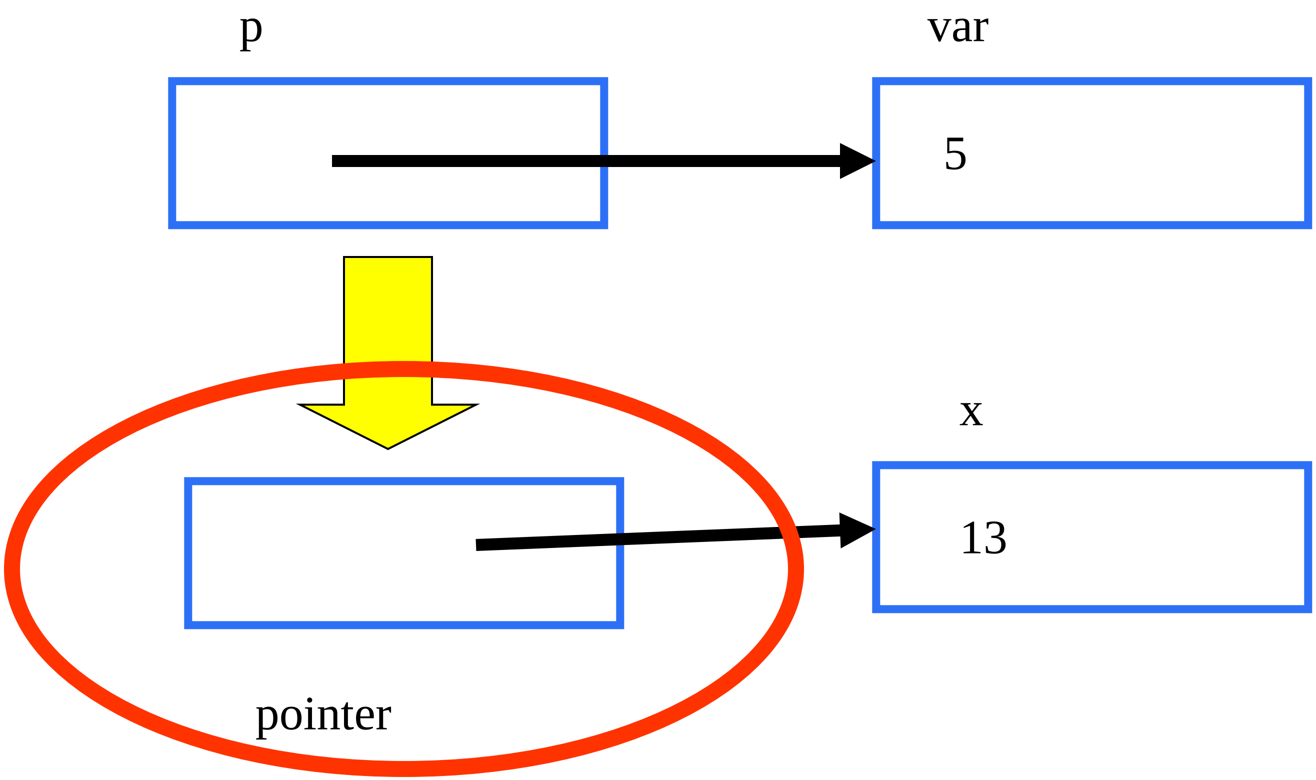


```
void function (int *pointer);
int var= 5, *p = &var;
int x = 13; // this is a global variable
function (p);
```



when the function is called, the value is copied

```
void function (int *pointer);
int var= 5, *p = &var;
int x = 13; // this is a global variable
function (p);
```

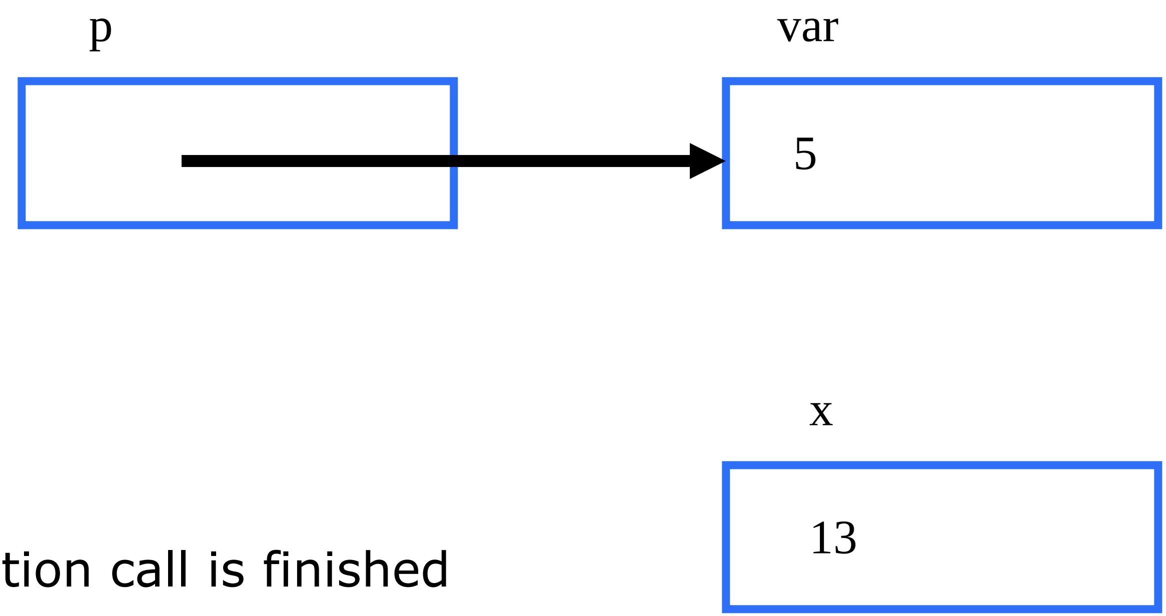


the function has the statement:

```
pointer= &x;
```

we see the changes in the copy of the function

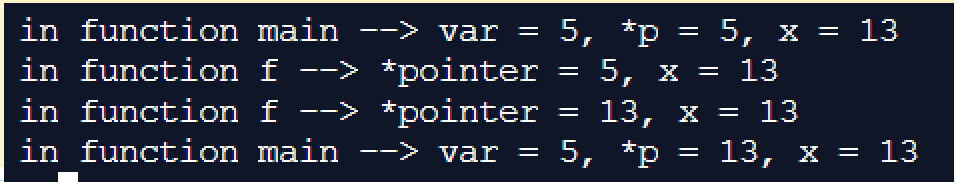
```
void function (int *pointer);
int var= 5, *p = &var;
int x = 13; // this is a global variable
function (p);
```



after the function call is finished

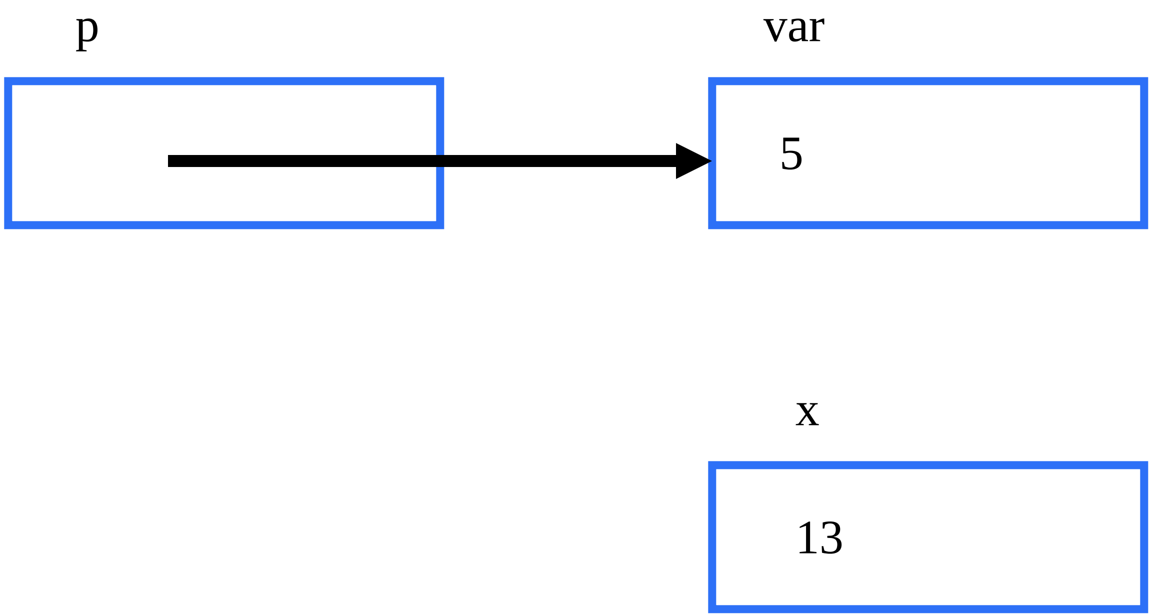
the changes are lost

```
#include <stdio.h>
int x = 13;//global variable

void f (int **pointer){
  printf("in function f --> *pointer = %d, x = %d\n", **pointer, x);
  *pointer = &x;
  printf("in function f --> *pointer = %d, x = %d\n", **pointer, x);
}

int main(void) {
 int var = 5;
 int *p = &var;
 printf("in function main --> var = %d, *p = %d, x = %d\n", var, *p, x);
 f(&p);
 printf("in function main --> var = %d, *p = %d, x = %d\n", var, *p, x);
 return 0;
}
```

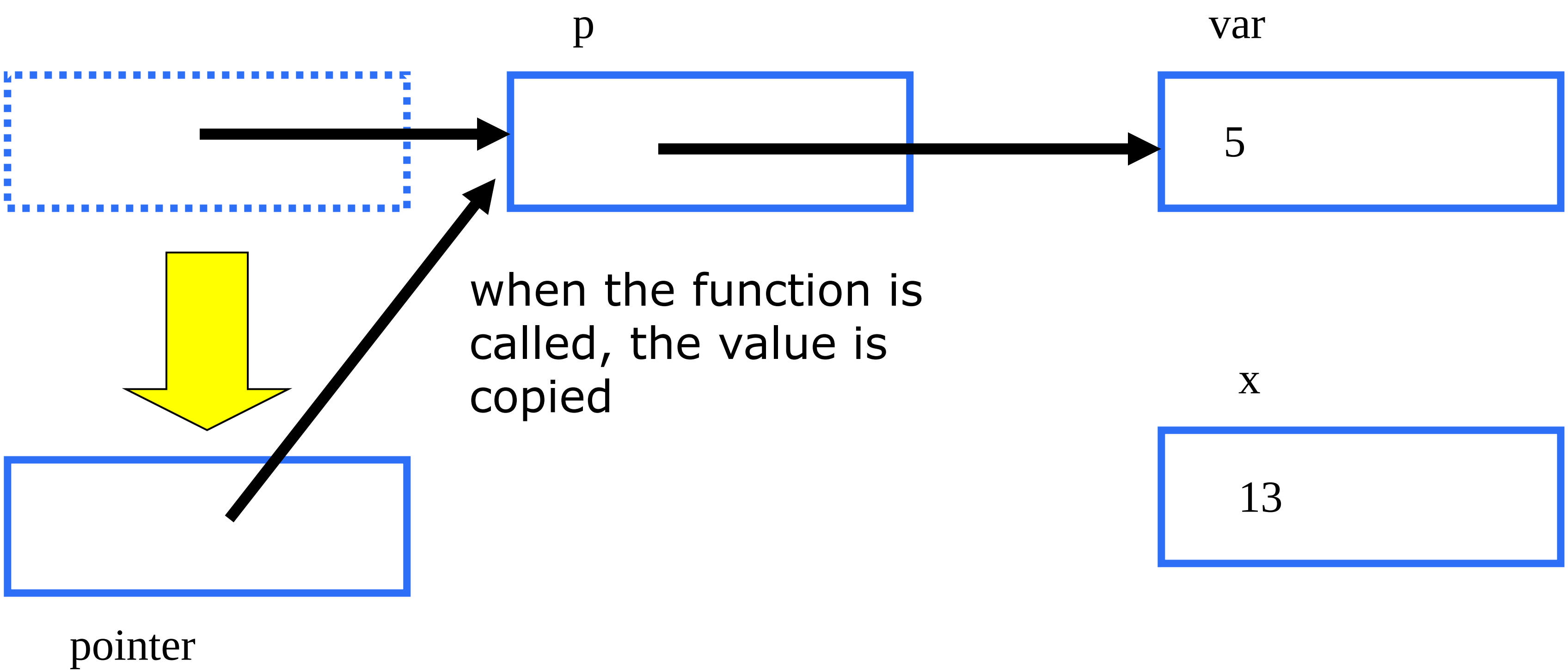


```
in function main --> var = 5, *p = 5, x = 13
in function f --> *pointer = 5, x = 13
in function f --> *pointer = 13, x = 13
in function main --> var = 5, *p = 13, x = 13
```

```
void function (int **pointer);
int var= 5, *p = &var;
int x = 13; // this is a global variable
```

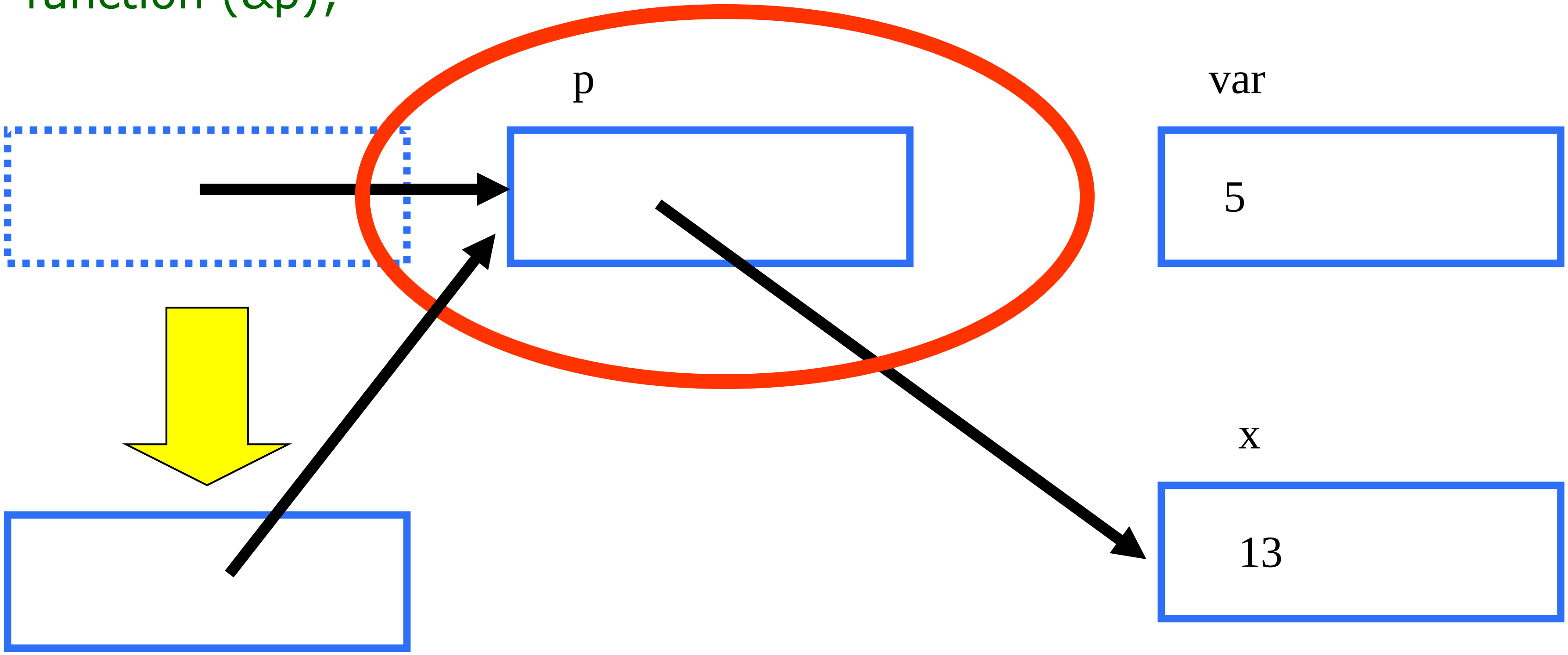


```
void function (int **pointer);
int var= 5, *p = &var;
int x = 13; // this is a global variable
function (&p);
```



when the function is called, the value is copied

```
void function (int **pointer);
int var= 5, *p = &var;
int x = 13; // this is a global variable
function (&p);
```

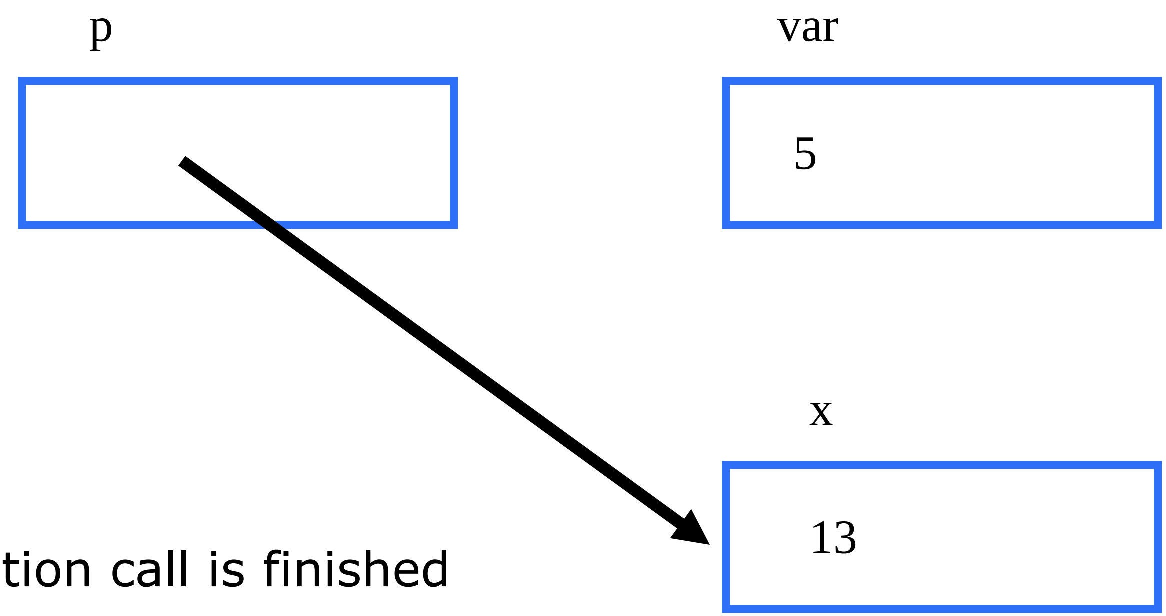


the function has the statement:

```
*pointer= &x;
```

we see the changes in the original variable

```
void function (int **pointer);
int var= 5, *p = &var;
int x = 13; // this is a global variable
function (&p);
```



after the function call is finished

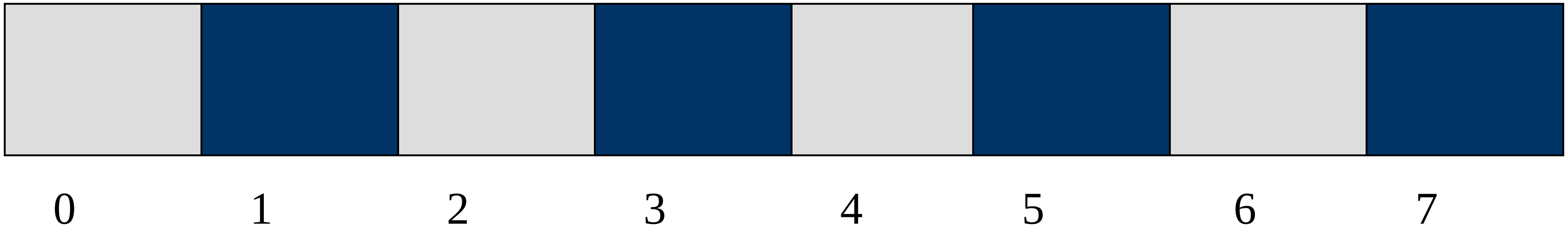
the changes are still there

- □ Pointers and arrays have many things in common.
- □ The array name, e.g., is the address of the memory space where the array is stored.
- □ That is why arrays are call-by-reference parameters.
- □ This also explains why a locally created array variable cannot be returned by a function.

- □ The address of an array is constant, after the array is created it never is changed.
- □ Therefore, the array variable cannot be on the left hand of an assignment operator.

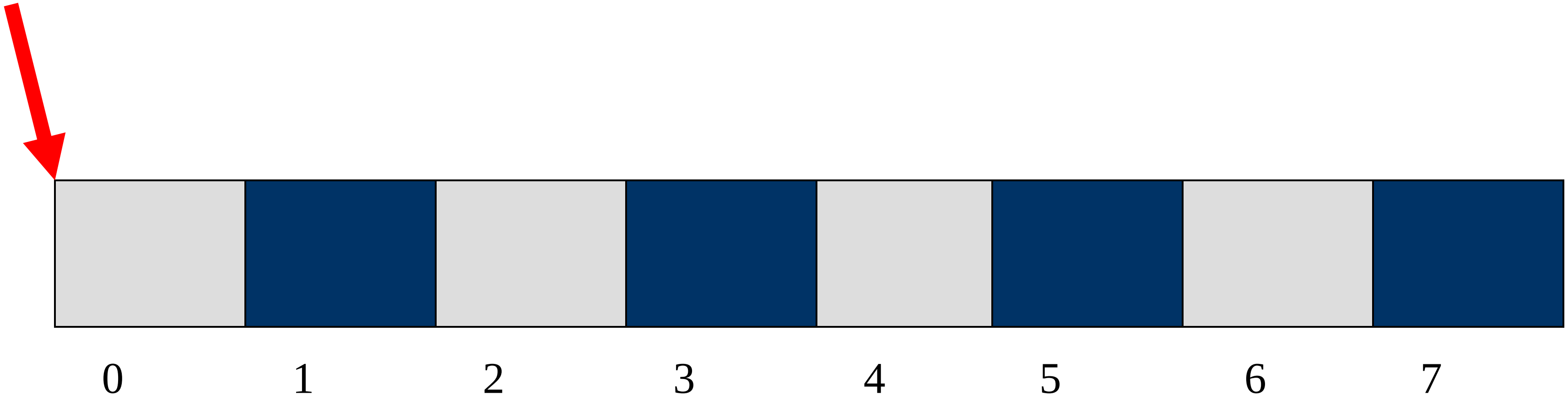
- Using arrays (and pointers in general, provided they point to adequate memory) we can apply the arithmetic operators +, -, ++ and --.

# int array[8];



0 1 2 3 4 5 6 7

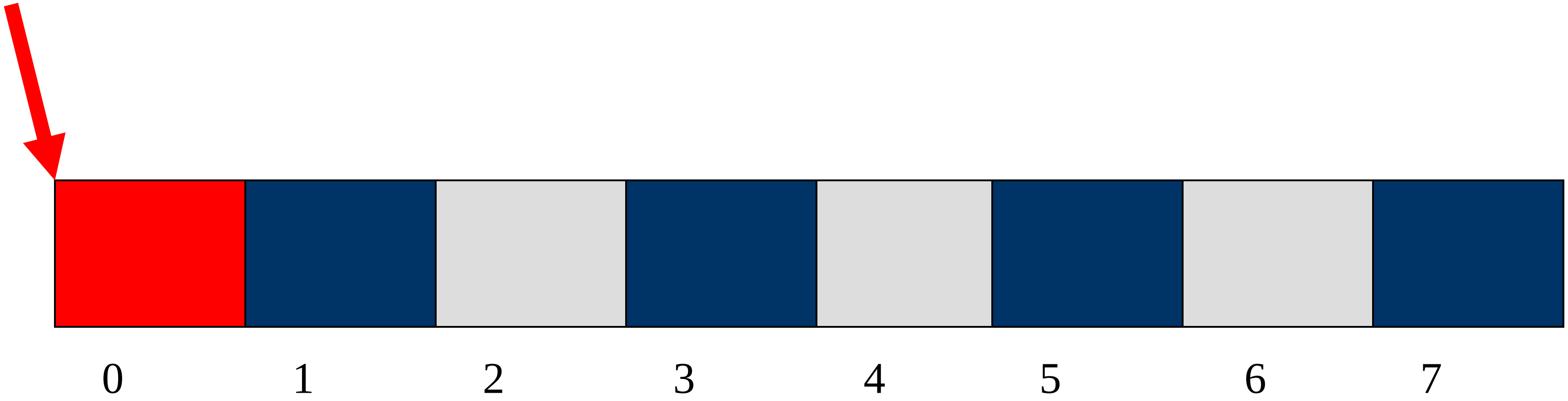
```
int array[8];
```



0 1 2 3 4 5 6 7

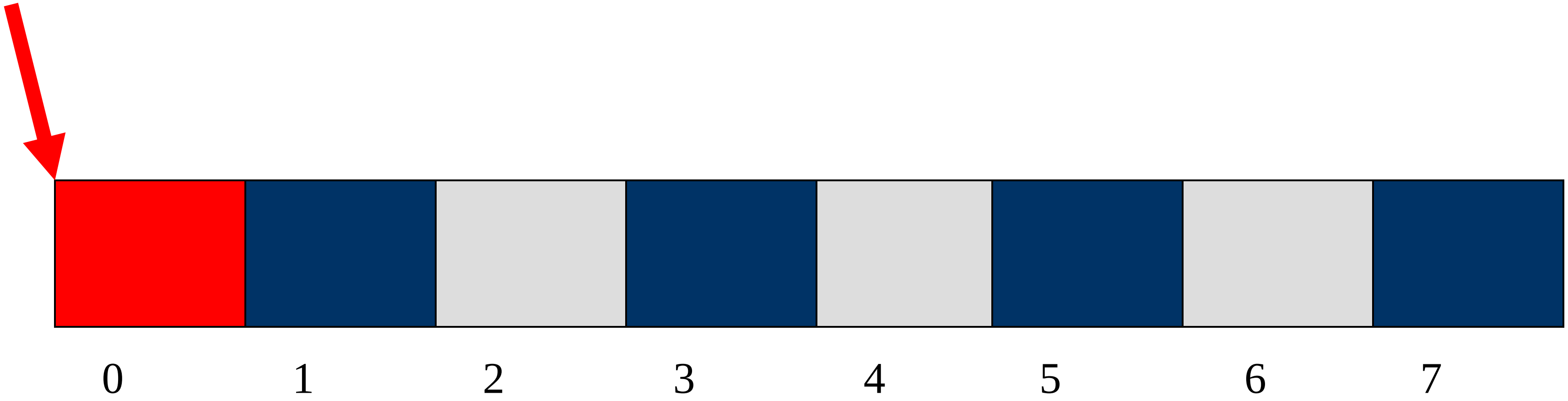
array

int array[8];



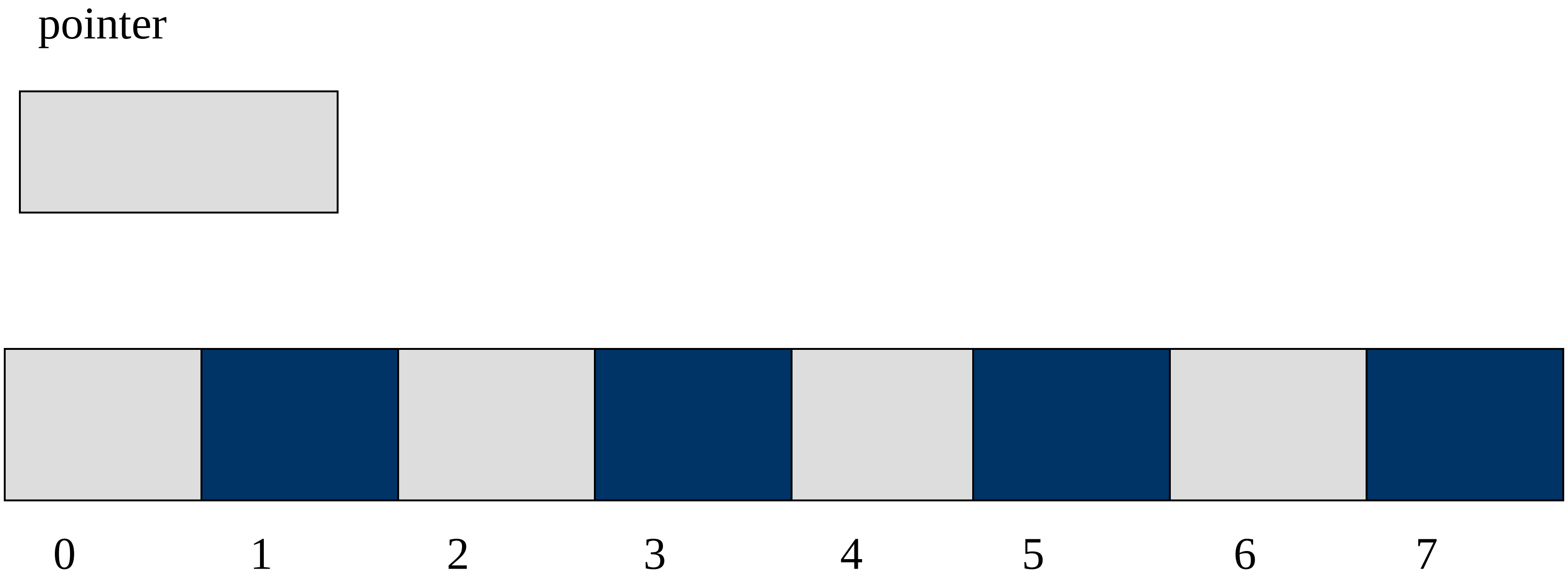
*array

int array[8];

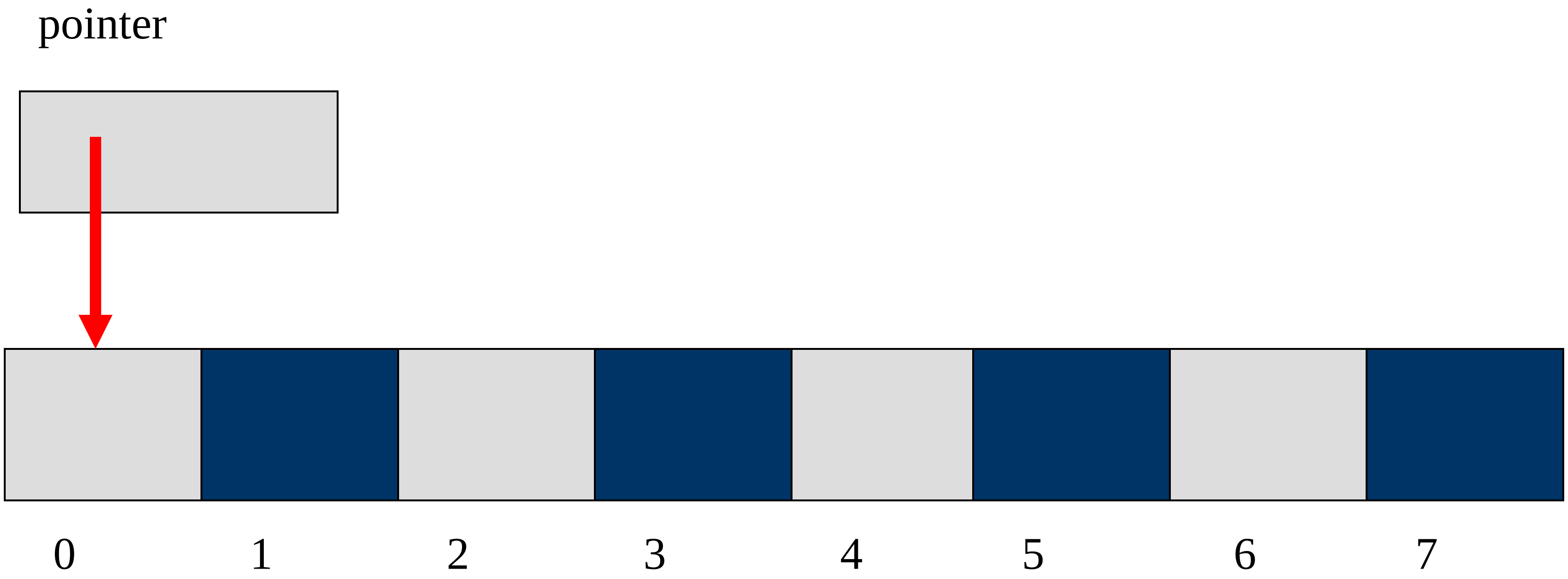


*array ≡ array[0]

```
int array[8];
int *pointer;
```

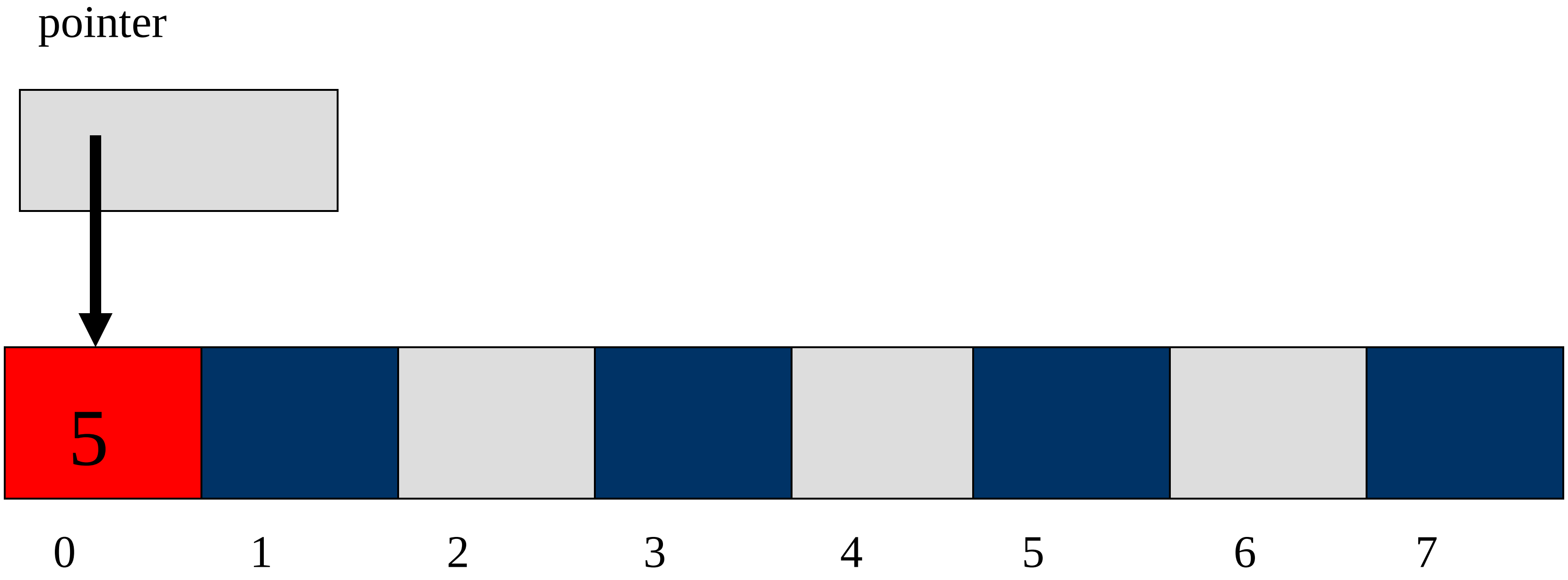


```
int array[8];
int *pointer;
```



```
pointer = array;
```

```
int array[8];
int *pointer;
```

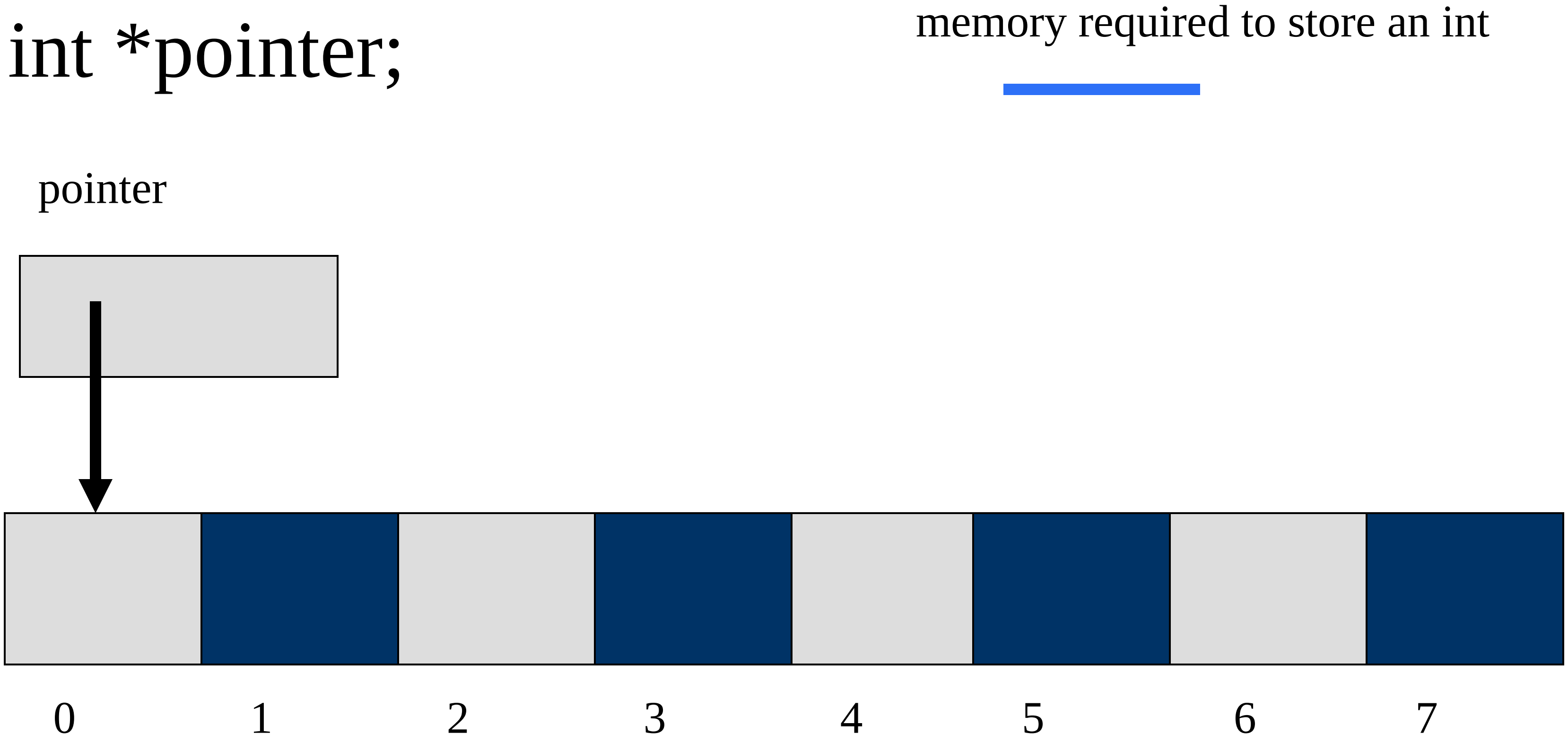


```
*pointer = 5;
*array = 5;
 array[0] = 5;
```

# pointer arithmetic

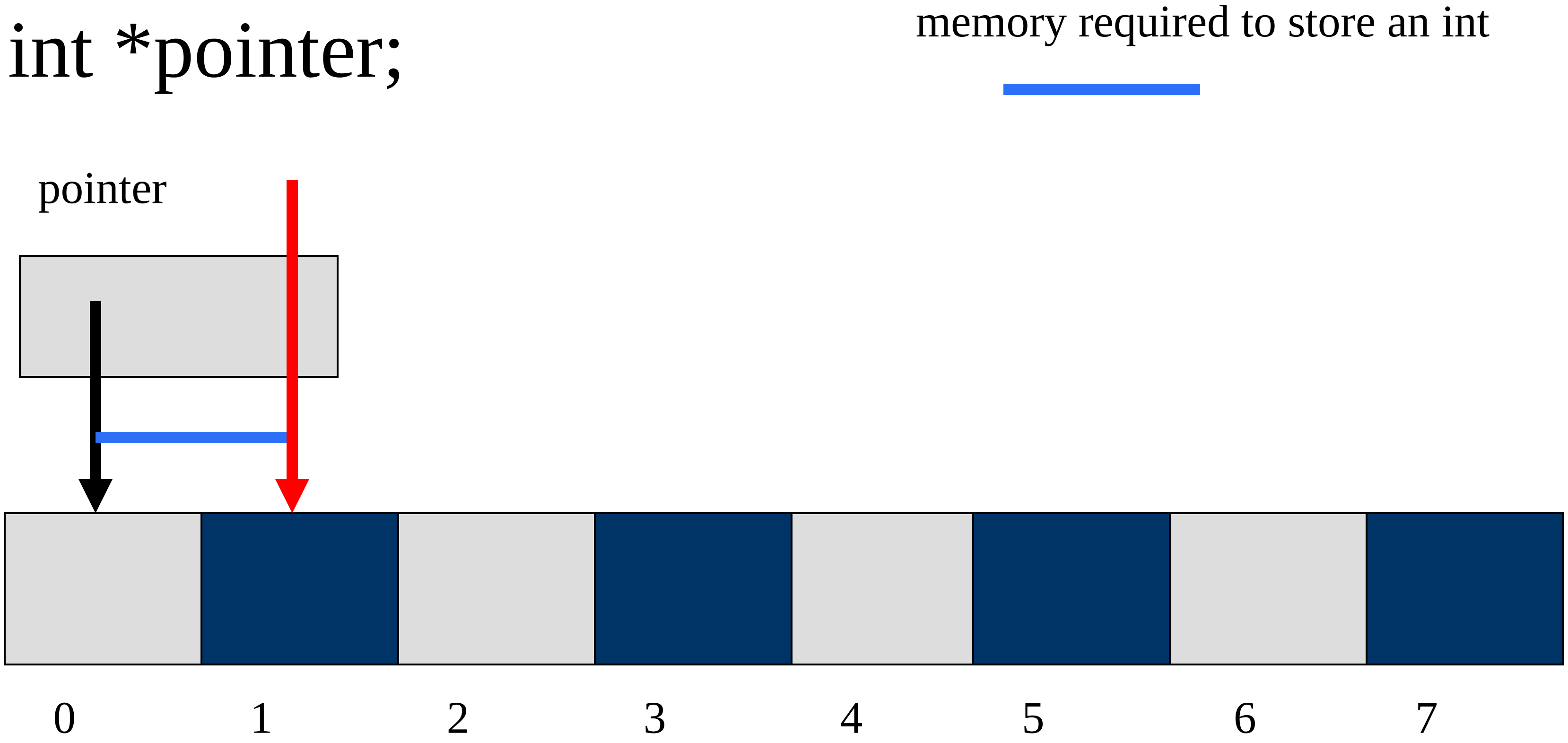
- A pointer describes an address.
- A pointer is bound to a data type. This defines also how many bytes are required to store a value of the respective data type.
- Adding a scalar x value to a pointer, adds x times the number of bytes required to store an element of the respective data type to the address.

```
int array[8];
int *pointer;
```



pointer + 1 ???

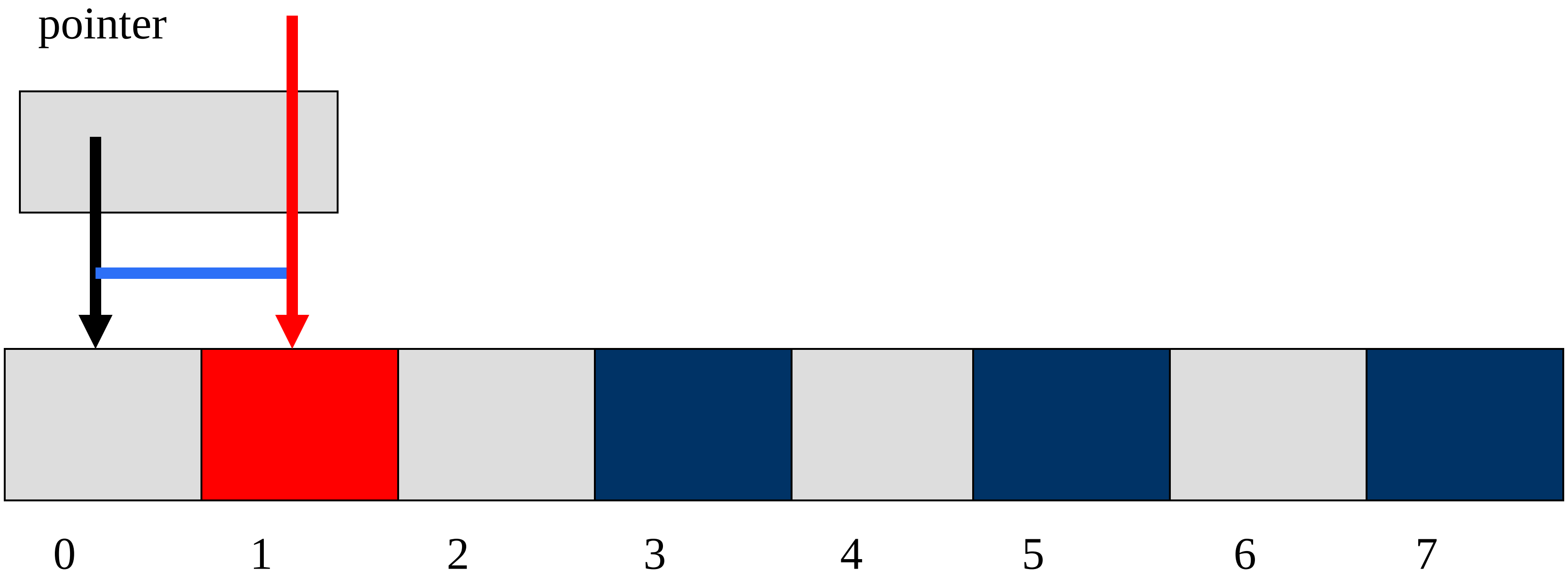
```
int array[8];
int *pointer;
```



pointer + 1

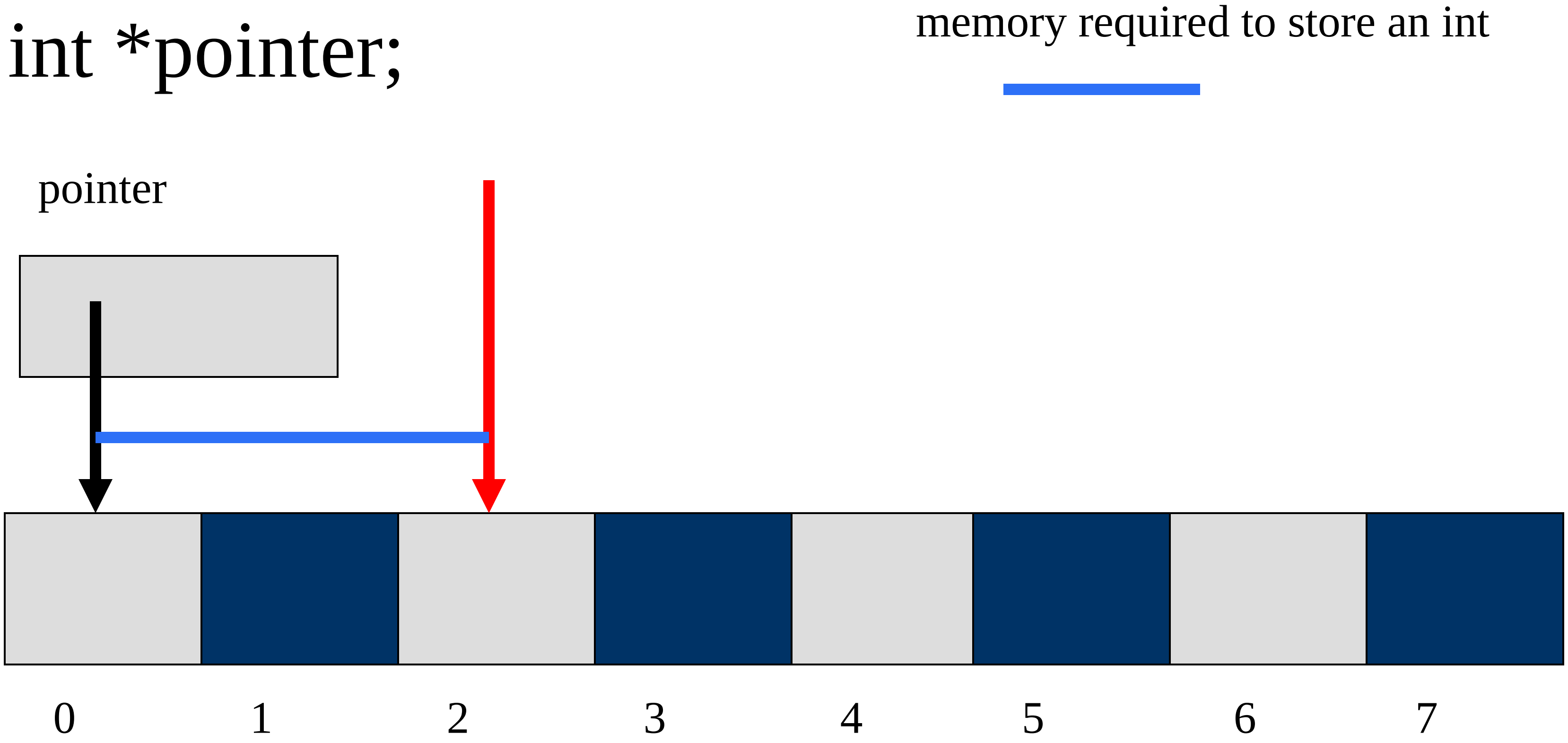
```
int array[8];
int *pointer;
```

memory required to store an int



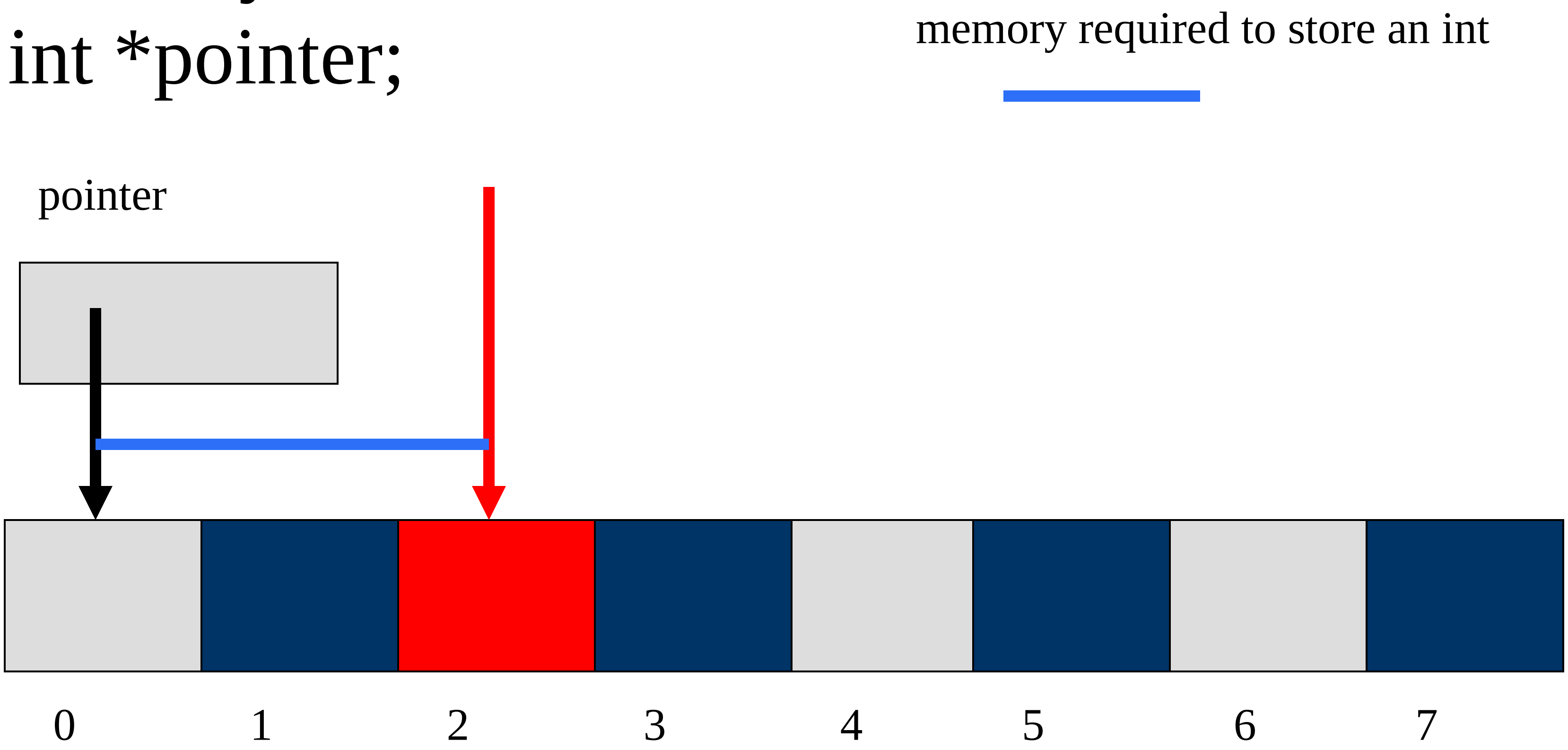
*(pointer + 1) ≡ array[1] ≡ *(array+1)

```
int array[8];
int *pointer;
```



pointer + 2

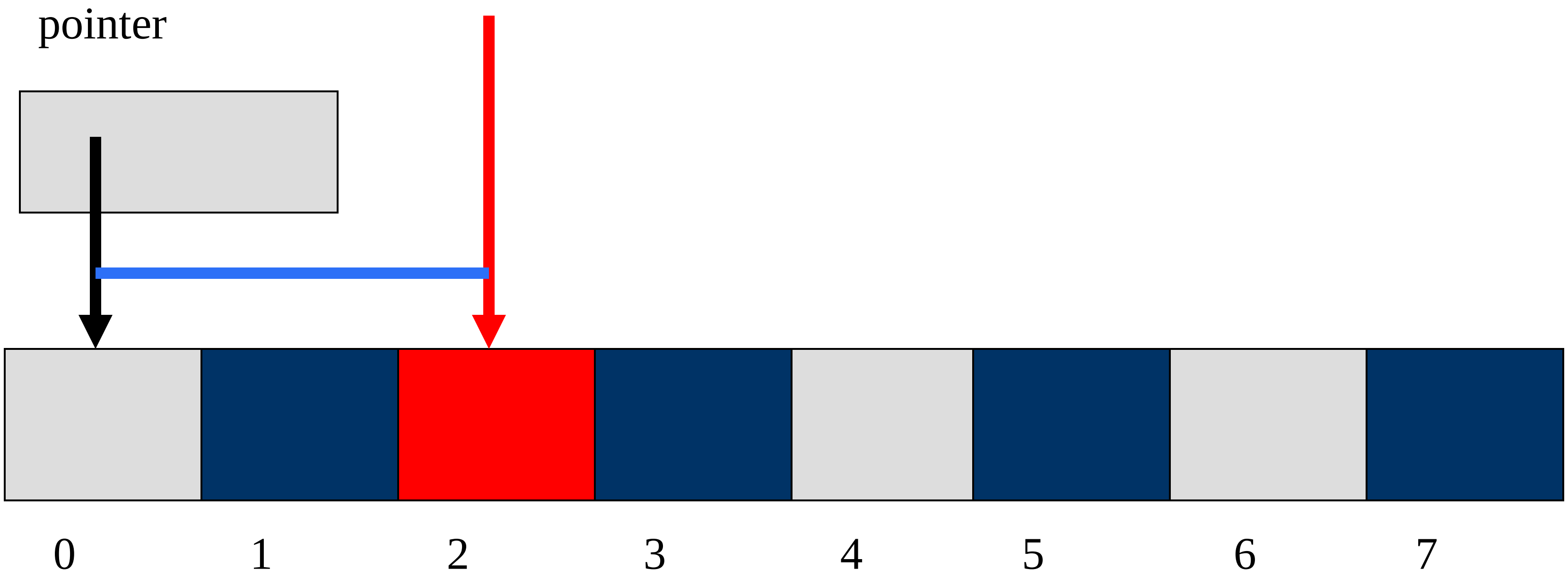
```
int array[8];
int *pointer;
```



*(pointer + 2)

```
int array[8];
int *pointer;
```

memory required to store an int



*(pointer + 2) ≡ array[2] ≡ *(array+2)

# Example

```
int array[] = {1,4,9,16,25};

int main (void)
{
   int *pointer = array;
   int z;
   for (z = 0; z < 5; z ++) {
      printf (“%d\t“, *(pointer+z));
   }
   return 0;
}
```

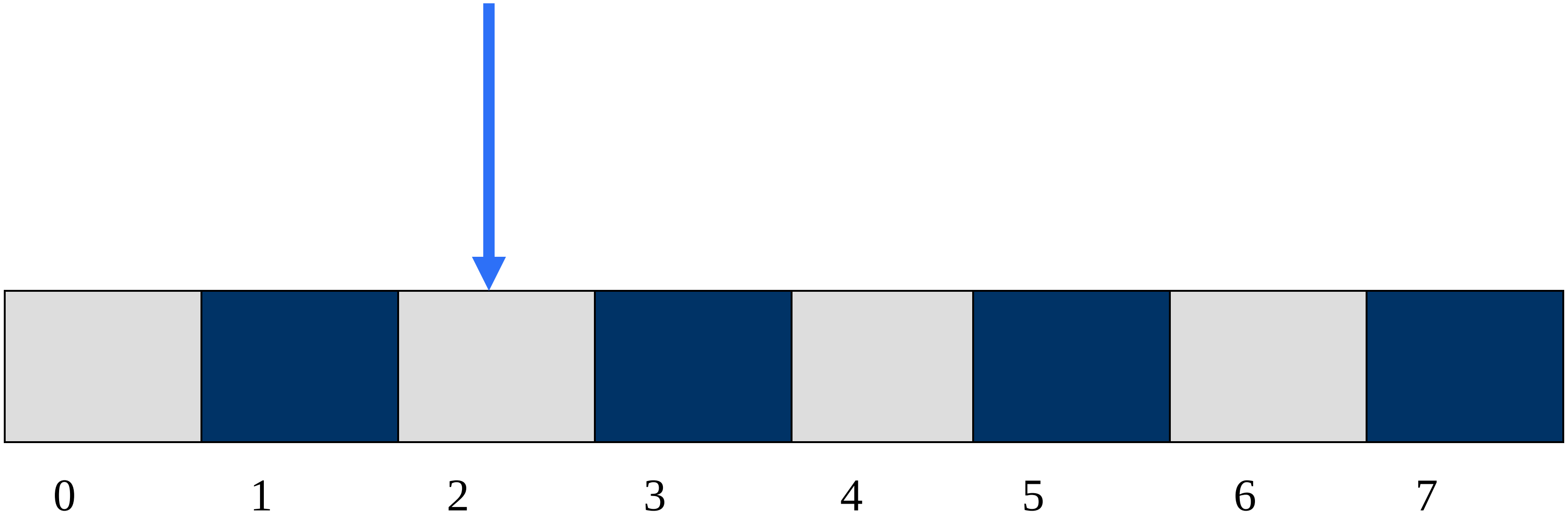
# Example

```
int array[] = {1,4,9,16,25};

int main (void)
{
    int *pointer = array;
    int z;
    for (z = 0; z < 5; z ++) {
        printf (“%d\t“, *pointer);
        pointer = pointer + 1;
    }
    return 0;
}
```

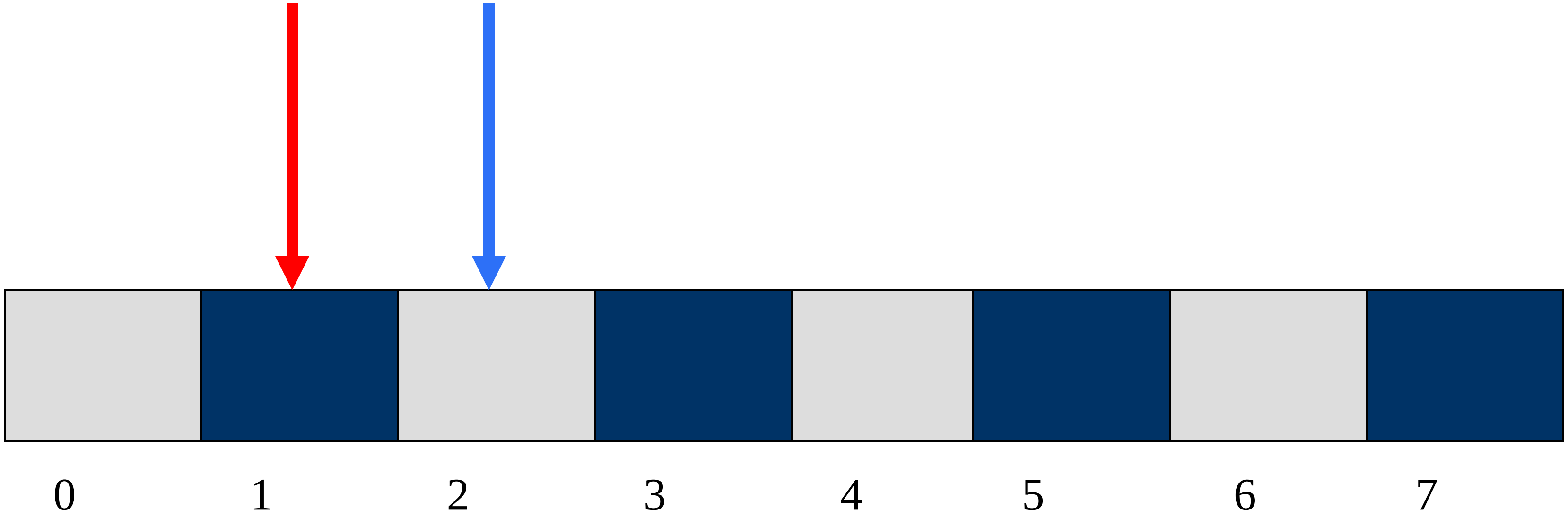
# Example

```
int array[] = {1,4,9,16,25};

int main (void)
{
   int *pointer = array;
   int z;
   for (z = 0; z < 5; z ++) {
      printf (“%d\t“, *(pointer++));
   }
   return 0;
}
```

```
int *pointer;
```

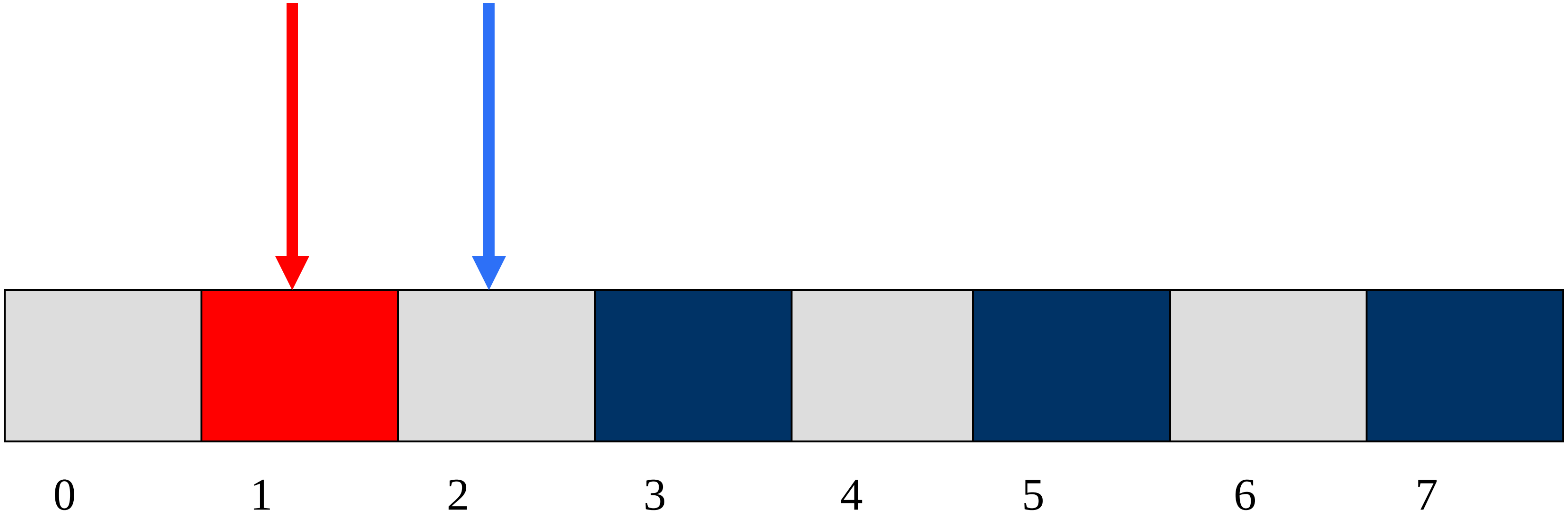


int *pointer;



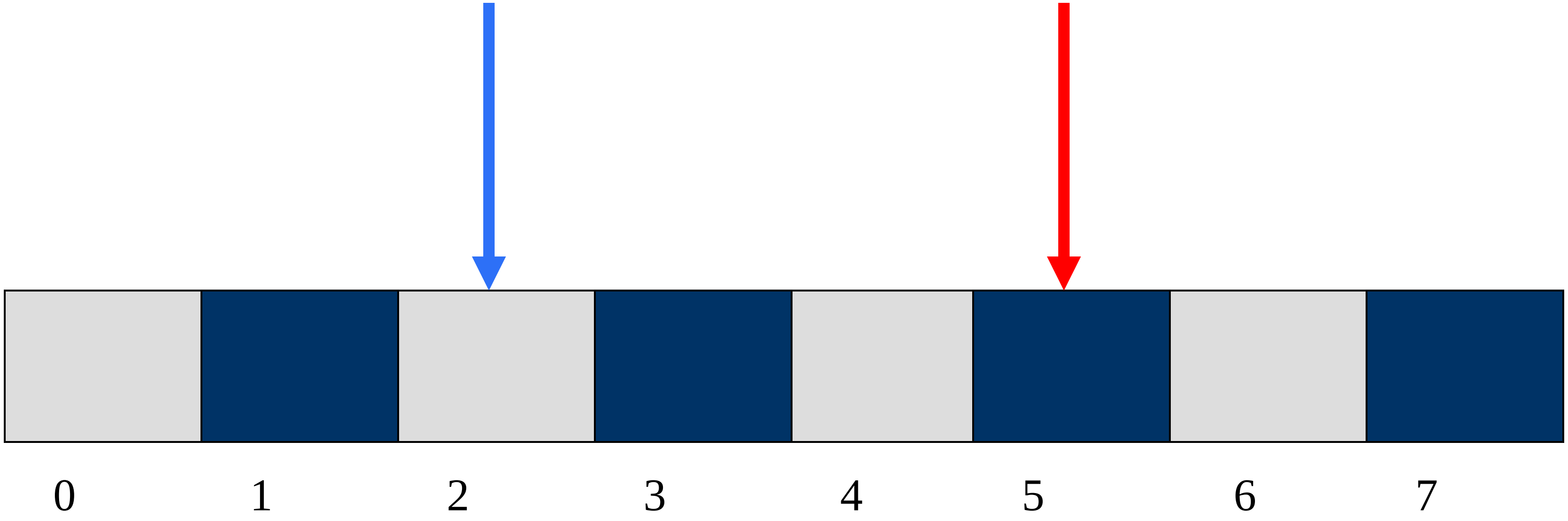
pointer - 1

int *pointer;



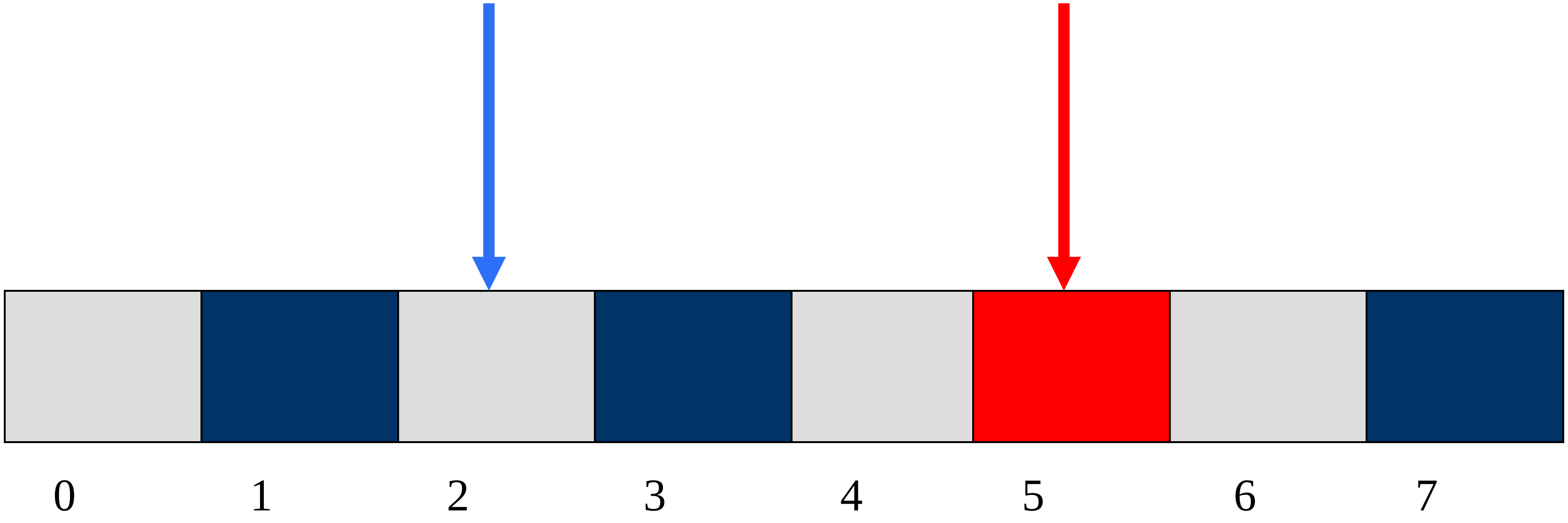
*(pointer - 1)

int *pointer;



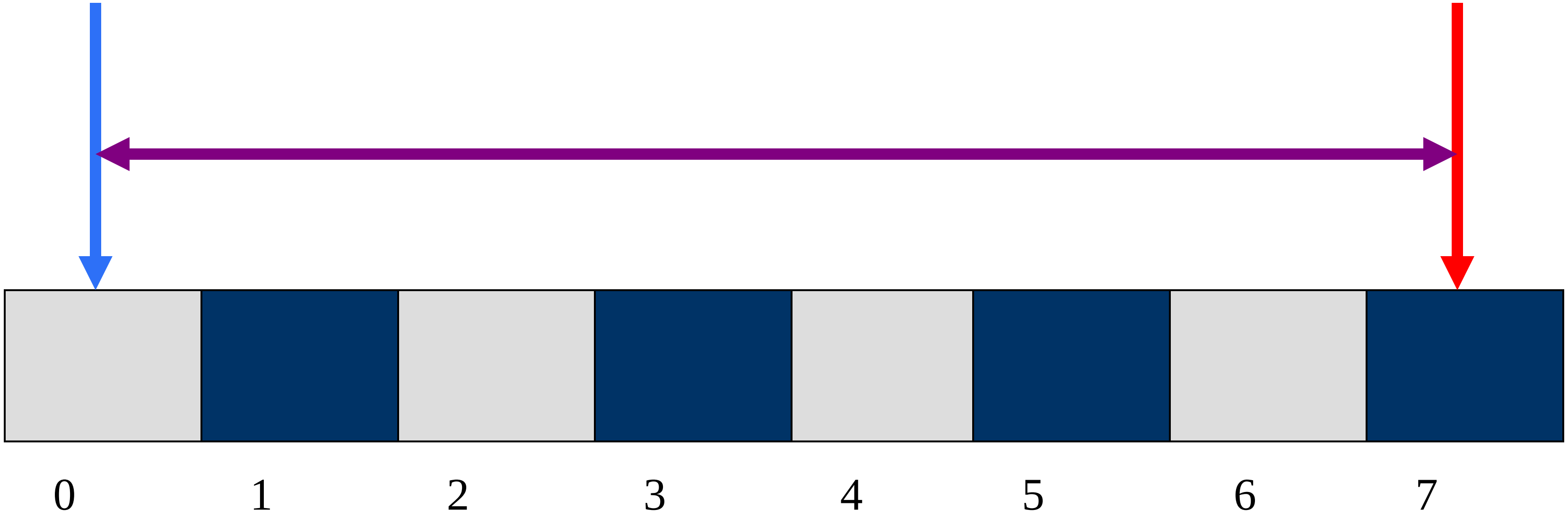
pointer + 3

```
int *pointer;
```



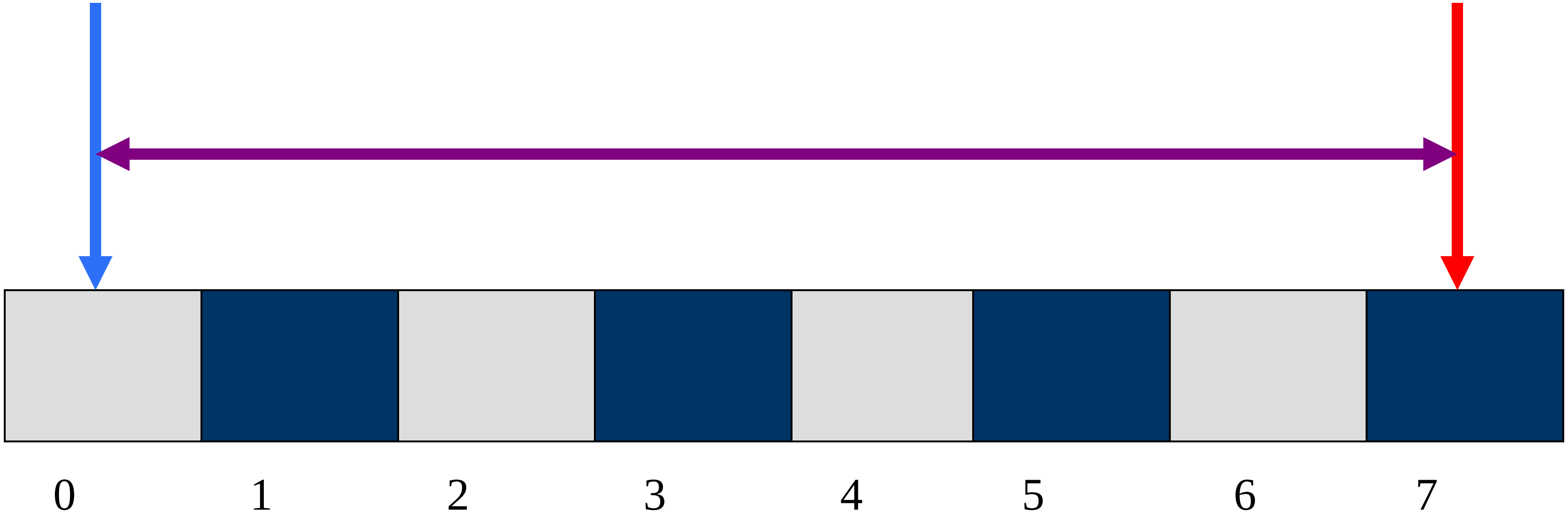
```
*(pointer + 3)
```

```
int *pointer1;
int *pointer2;
```



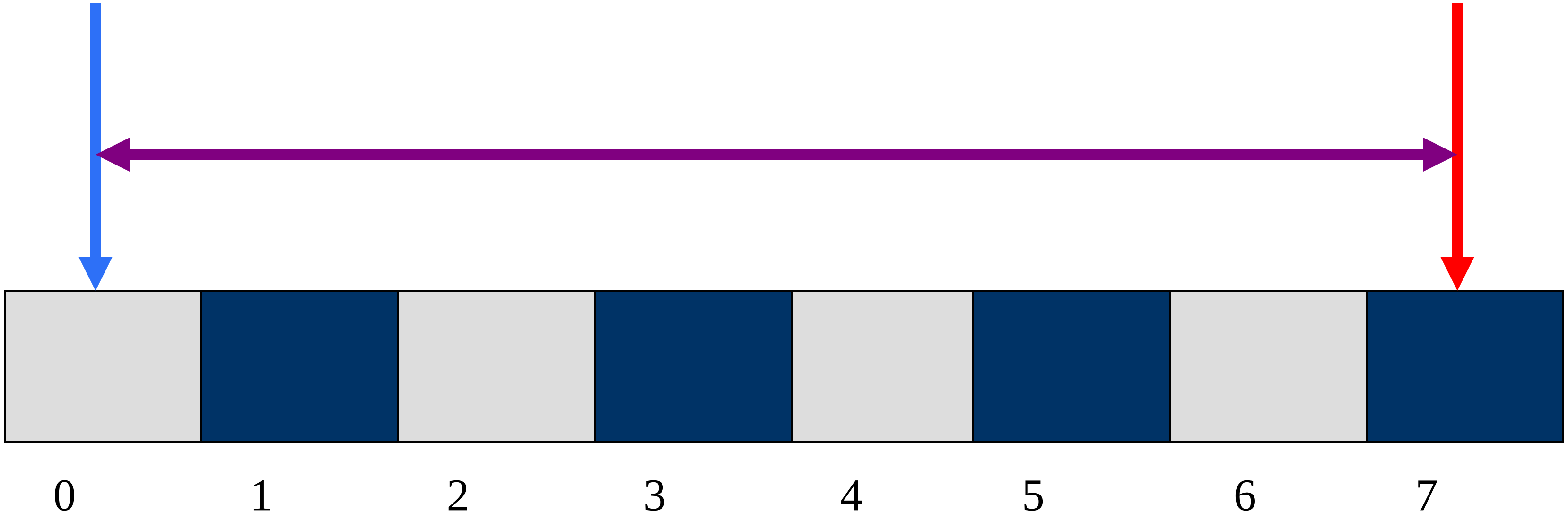
```
pointer2 - pointer1
```

int *pointer1;
int *pointer2;



pointer2 - pointer1 ➔ results in an integer value

```
int *pointer1;
int *pointer2;
```



pointer2 - pointer1 + 1 ➔ length of the array

Assume that n is the length of the array

```
int arraySum1(int array[], const int n)
{
    int sum = 0;
    int *ptr;
    int * const arrayEnd = array + n;

    for (ptr = array; ptr < arrayEnd; ++ptr) {
        sum += *ptr;
    }

    return sum;
}
```

Assume that n is the length of the array

```
int arraySum2(int *array, const int n)
{
    int sum = 0;
    int *ptr;
    int * const arrayEnd = array + n;

    for (ptr = array; ptr < arrayEnd; ++ptr) {
        sum += *ptr;
    }

    return sum;
}
```

```
int main()
{
    int a[] = {1,4,-3,4,6,2,4,90,12,27};

    printf("result of arraySum1: %d\n", arraySum1(a,10));
    printf("result of arraySum2: %d\n", arraySum2(a,10));

    return 0;
}
```

```
int stringLength(char *string)
{
    int l = 0;
    while (*string != '\0') {
        l++;
        string++;
    }
    return l;
}
```

```
void copyString(char *to, char *from)
{
    for (;*from != '\0'; from++, to++){
        *to = *from;
    }
    *to = '\0';
}
```

# easiest

```
void copyString(char *to, char *from)
{
    for (;*from != '\0'; from++){
        *to = *from;
        to++;
    }
    *to = '\0';
}
```

```
int main()
{
    char s1[]="Guten Morgen";
    char s2[80];

    printf("length of s1: %d\n",stringLength(s1));

    copyString(s2,s1);
    printf("s1: %s\ns2: %s\n", s1, s2);

    return 0;
}
```

- **Pointers** can point to **any data type**. This includes also structures (note: and even functions*).

```
struct person {
    char name[30];
    char address[30];
    int age;
};

int main()
{
    struct person maria = {"Maria", "Berlin", 27};
    struct person *pToMaria = &maria;
    printf("%s lives in %s and is %d years old.\n",
   maria.name, maria.address, maria.age);
```

```
    printf("%s lives in %s and is %d years old.\n",
   (*pToMaria).name,(*pToMaria).address,
   (*pToMaria).age);
    printf("%s lives in %s and is %d years old.\n",
   pToMaria->name,
           pToMaria->address, pToMaria->age);
    return 0;
}
```