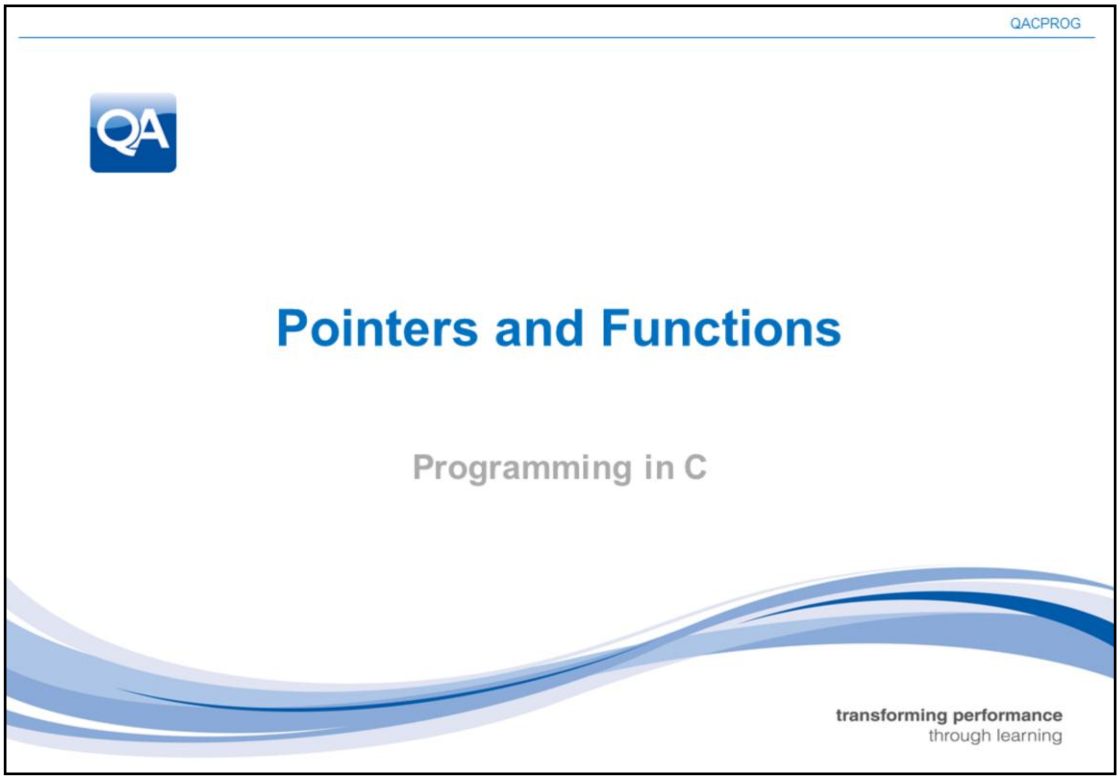
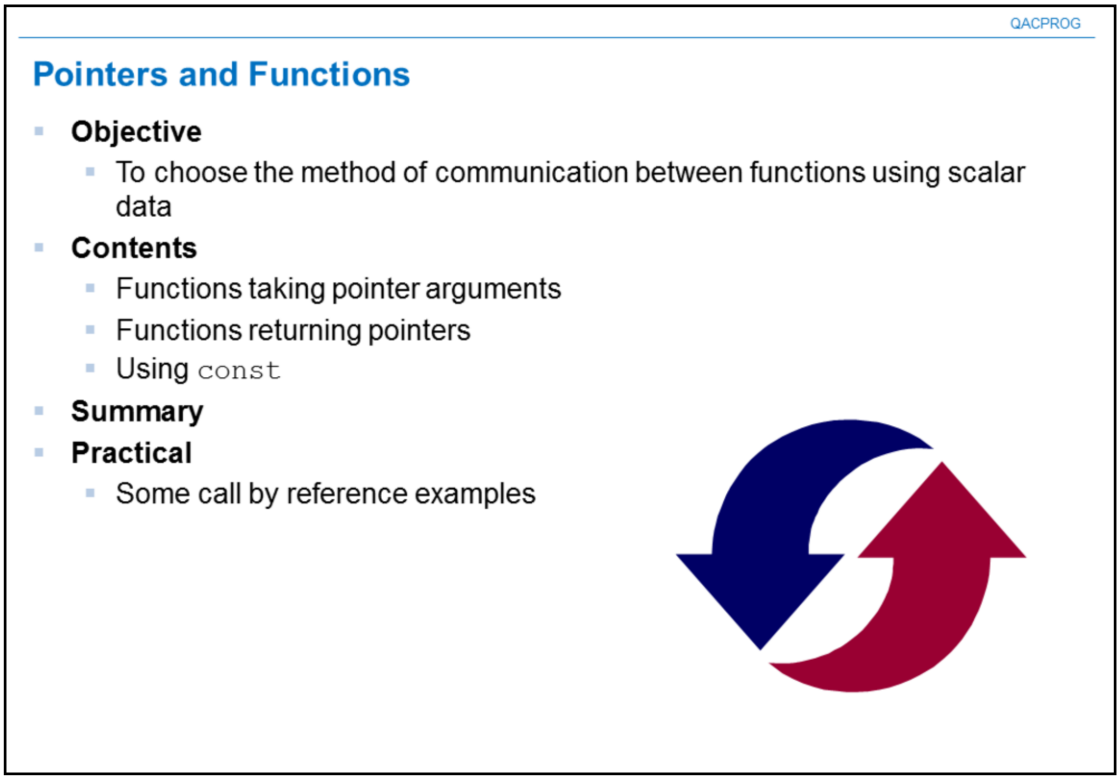


QACPROG

Pointers and Functions

Programming in C

transforming performance
through learning

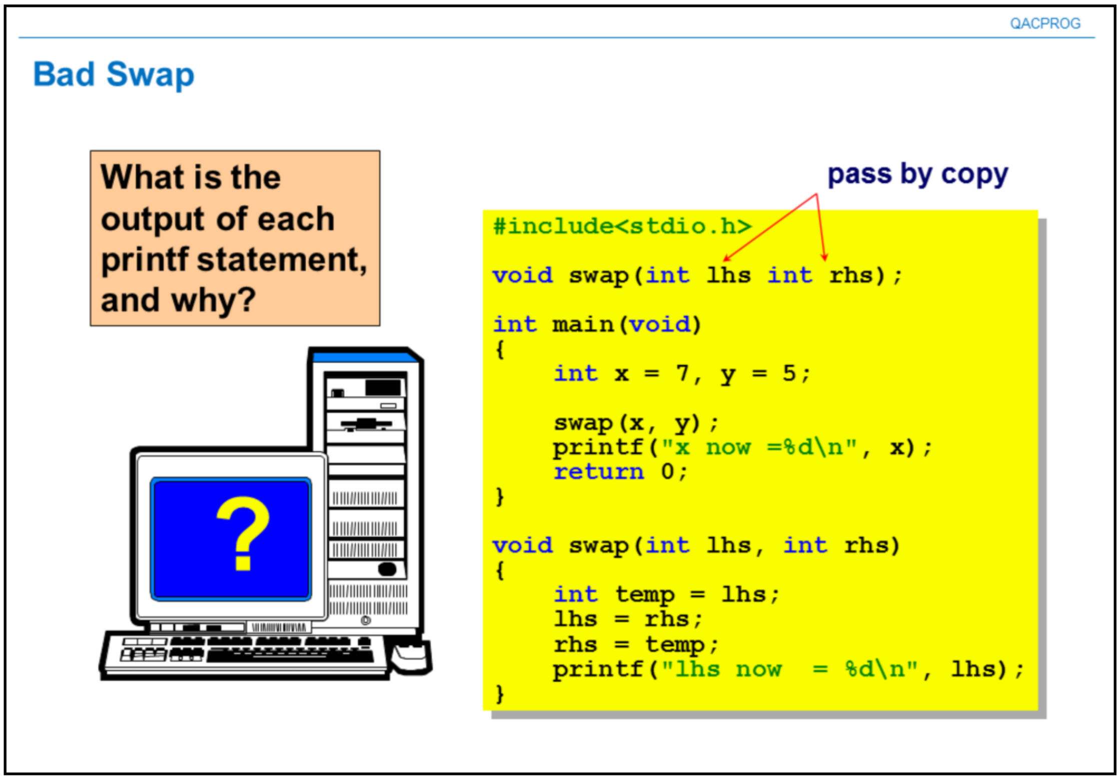


QACPROG

Pointers and Functions

- **Objective**
  - To choose the method of communication between functions using scalar data
- **Contents**
  - Functions taking pointer arguments
  - Functions returning pointers
  - Using `const`
- **Summary**
- **Practical**
  - Some call by reference examples

The objective of this chapter is to describe the concept of passing data by reference. It covers the passing of pointers into and out of functions and the use of const within the passing and returning mechanisms.



QACPROG

Bad Swap

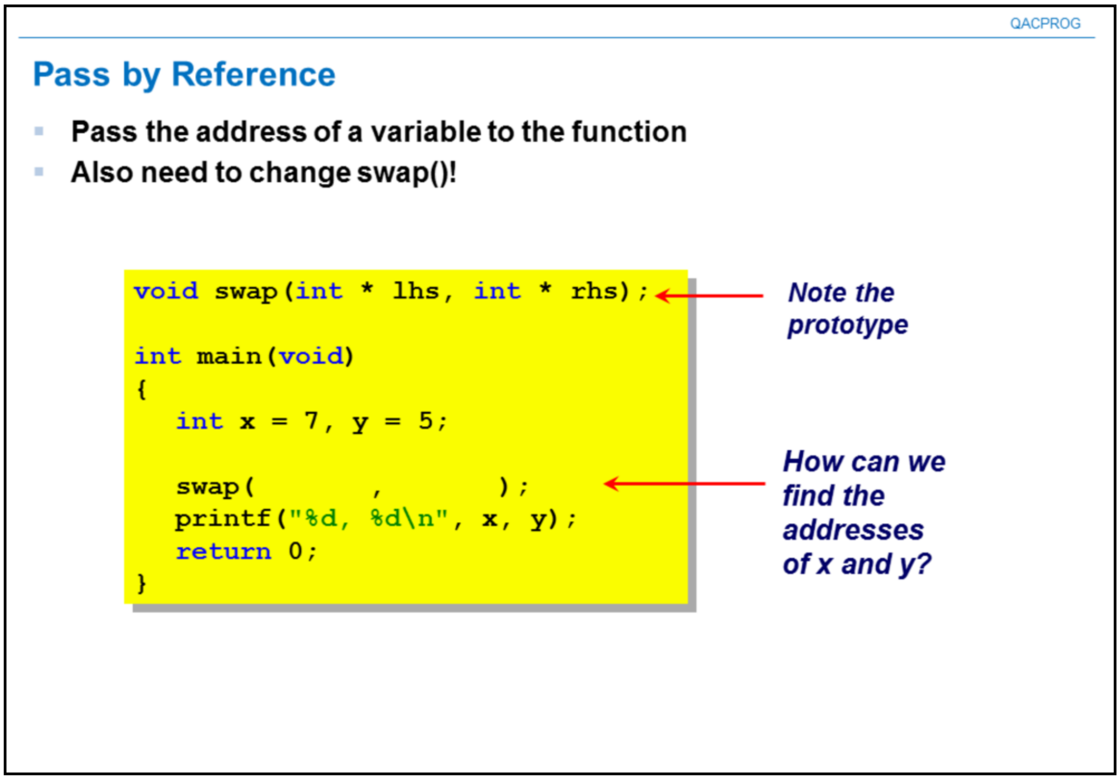
What is the output of each printf statement, and why?

pass by copy

```
#include<stdio.h>

void swap(int lhs int rhs);

int main(void)
{
    int x = 7, y = 5;

    swap(x, y);
    printf("x now =%d\n", x);
    return 0;
}

void swap(int lhs, int rhs)
{
    int temp = lhs;
    lhs = rhs;
    rhs = temp;
    printf("lhs now  = %d\n", lhs);
}
```

This example shows is an attempt to swap the data items `x` and `y` by calling a function. It does not work because the `swap` function works on *copies* of the arguments. The function was only able to swap the temporary copies holding the data.

We shall devise a method to achieve our aim.



QACPROG

Pass by Reference

- Pass the address of a variable to the function
- Also need to change swap()!

```
void swap(int * lhs, int * rhs);

int main(void)
{
   int x = 7, y = 5;

   swap(        ,        );
   printf("%d, %d\n", x, y);
   return 0;
}
```

Note the prototype

How can we find the addresses of x and y?

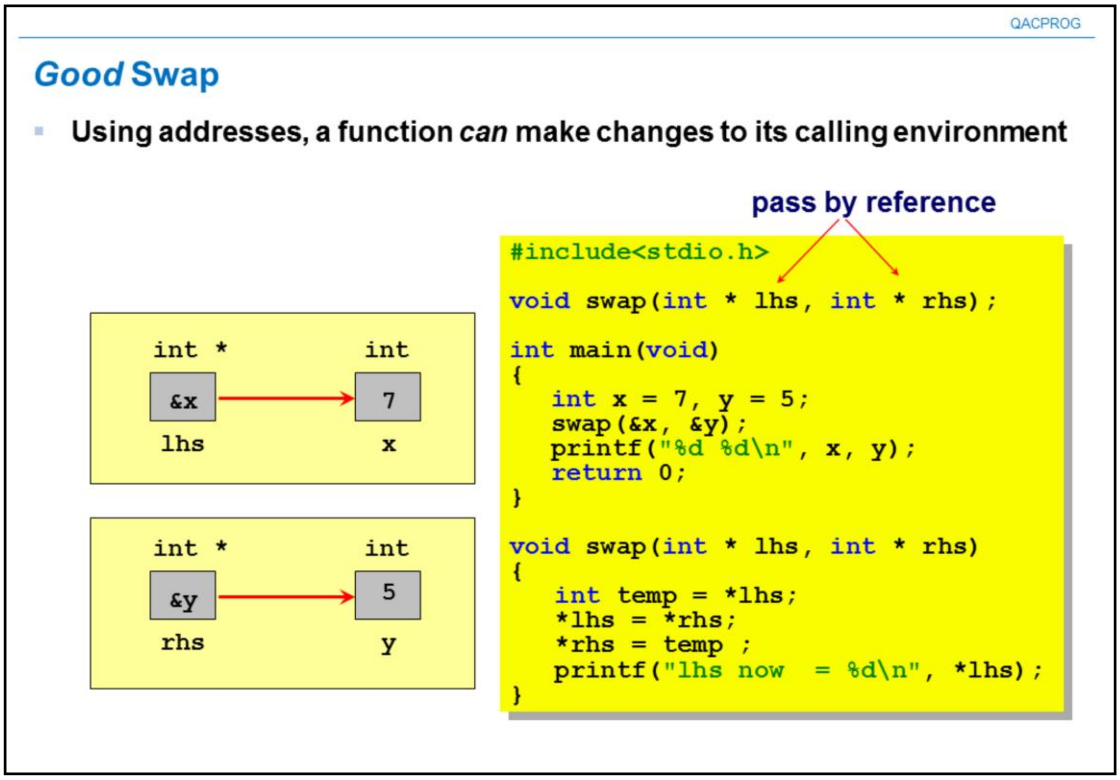
The initial reaction is to pass the address of the two data items into the function. This will mean that the new swap function will receive machine locations, which it can then use to access the actual data. Firstly, we call swap with arguments `&x` and `&y`, which are the addresses of `x` and `y`. The prototype indicates that the swap function will receive the addresses of two integers, i.e. two 'pointers to int'. The type `int *` is now introduced formally as a 'pointer to int'. Others data types are as follows:

`char *`	pointer to `char`
`float *`	pointer to `float`
`double *`	pointer to `double`
`long int *`	pointer to `long int`
...	
`struct date *`	pointer to `struct date` (more later)

The parameter names on the prototype are optional.

```
void swap(int *, int *);
```

Most programmers find it easier to read a prototype that includes parameter names. They help to clarify the interface. It is also the preferred form suggested in many C coding standards.



Good Swap

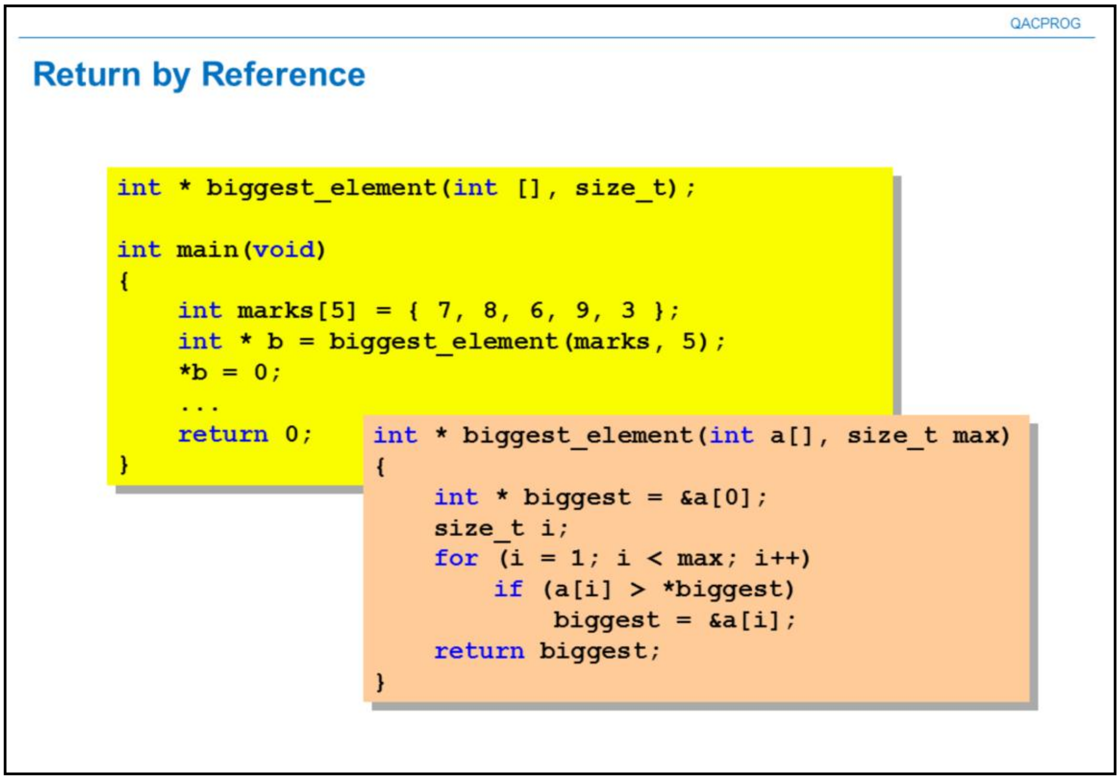
- Using addresses, a function *can* make changes to its calling environment

```
#include<stdio.h>

void swap(int * lhs, int * rhs);

int main(void)
{
   int x = 7, y = 5;
   swap(&x, &y);
   printf("%d %d\n", x, y);
   return 0;
}

void swap(int * lhs, int * rhs)
{
   int temp = *lhs;
   *lhs = *rhs;
   *rhs = temp ;
   printf("lhs now  = %d\n", *lhs);
}
```

The definition of the swap function completes the picture. The two parameters `lhs` and `rhs` are now declared as 'pointers to `int`', and as with all parameters, they are automatically initialised with the values passed by the call, i.e. `&x` and `&y`.

The function body makes use of the `*` operator to get hold of the data pointed to by `lhs` and `rhs`.

This call, using addresses `&x` and `&y`, is said to have passed `x` and `y` 'by reference'. In C, we perform referencing and dereferencing explicitly with the appropriate operators `&` and `*`. In some languages, call by reference is the default! These languages do not need the `&` operator, the `*` pointer qualifier in the parameter nor the `*` dereferencing in the function body, since the referencing is implicit.


QACPROG

Return by Reference

```
int * biggest_element(int [], size_t);

int main(void)
{
    int marks[5] = { 7, 8, 6, 9, 3 };
    int * b = biggest_element(marks, 5);
    *b = 0;
    ...
    return 0;
}
```

```
int * biggest_element(int a[], size_t max)
{
    int * biggest = &a[0];
    size_t i;
    for (i = 1; i < max; i++)
        if (a[i] > *biggest)
            biggest = &a[i];
    return biggest;
}
```

Returning a pointer is like returning an `int`, `char`, etc. The return type must be properly specified in the prototype and definition, i.e. `int *`. The return value must be anticipated and used properly in the call, i.e. assigned to `b` (itself an `int` *). Finally, the expression returned must be the correct type, i.e. `cb` (again, an `int` *).

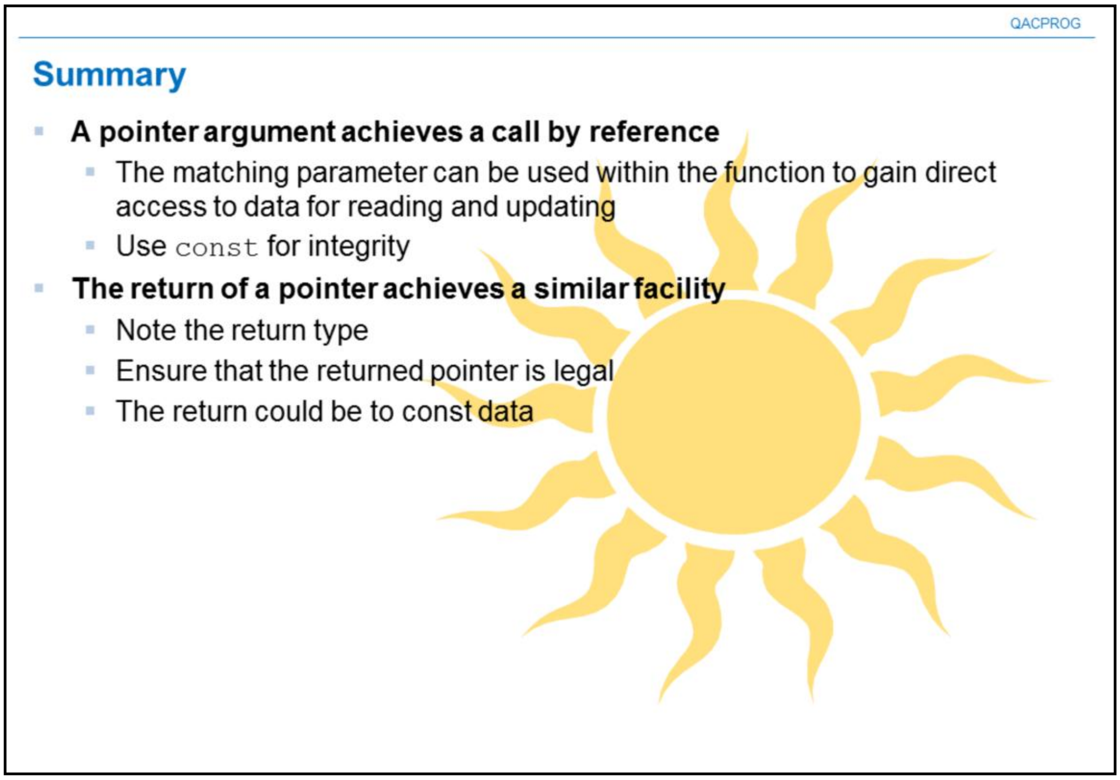
The only problem in returning pointer types is that the value of the returned address has to be legal. The address must be accessible to the calling function's environment and not, for example, the address of a local automatic data item. In our example shown above, the address is part of the local array, marks. This is perfectly acceptable.

There is a very small amount of duplication in this code fragment. The size of the marks array (5) occurs in the code twice. We can remove the first occurrence completely by letting the compiler count the number of elements in the initialiser list. We can also use the sizeof operator to write an expression that evaluates to the length of the array:

```
int marks[] = { 7, 8, 6, 9, 3 };
const size_t no_of_marks
       = sizeof(marks) / sizeof(marks[0]);
int * b = biggest_element(marks, no_of_marks);
*b = 0;
```

It is also possible to collapse the last two statements into one and avoid the need to declare b although this is perhaps a step too far!

```
*biggest_element(marks, no_of_marks) = 0;
```



QACPROG

Summary

- **A pointer argument achieves a call by reference**
  - The matching parameter can be used within the function to gain direct access to data for reading and updating
  - Use `const` for integrity
- **The return of a pointer achieves a similar facility**
  - Note the return type
  - Ensure that the returned pointer is legal
  - The return could be to const data

Pointer arguments are extremely common. They are used to allow functions to update the data passed. They are also used for efficiency, especially for large data. The latter use becomes more apparent for structures. Remember that, by default, arrays are passed by reference. Using const within the argument declaration adds integrity to the function process.

Returning pointers are not so common. They usually return the address of an element within an array that has been passed into the function. Functions sometimes return a pointer that has been passed in. This is used to ease the task of testing or for calling functions using the value just returned from a function, i.e. calling a function whose argument is a function call. The example shown below uses the biggest_element function from the previous page.

```
    scanf("%d", biggest_element(marks, 5));
```

The code replaces the largest element in the marks array with an integer entered from the keyboard. This is dangerous, since no checking is performed! Note the absence of the `&` operator. The argument is already an address.

Return a pointer to const data is even rarer but possible. Again, it preserves data integrity, this time after the function has returned.

Intentionally left blank