A visual representation to better understand C pointers

Manuel Selva
selva.manuel@gmail.com

This document tries to give a visual representation of what are C pointers and how to use them by looking how the content of the stack evolves during the execution of a program. It is neither precise, nor complete, so please refer to other resources for learning pointers in depth. The author of this document has been inspired by a practical lab proposed at the University of Strasbourg. The subject of this practical lab has been written by someone, whose name is unknown, thanks her/him. The reader is supposed to know:

- what a C function is and what a C type is;
- what is `printf`;
- that pointers are something weird.

To illustrate the concept of C pointers, we'll study the three swapping functions which source code is displayed in Figure 1.

```
void swap1(int i1, int i2) {
  int tmp = i1;                    // s11
  i1 = i2;                         // s12
  i2 = tmp;                        // s13
}

void swap2(int* pi1, int* pi2) {
  int tmp = *pi1;                  // s21
  *pi1 = *pi2;                     // s22
  *pi2 = tmp;                      // s23
}

void swap3(int** ppi1, int** ppi2) {
  int* tmp = *ppi1;                // s31
  *ppi1 = *ppi2;                   // s32
  *ppi2 = tmp;                     // s33
}
```

Figure 1: Swapping functions

These three functions are called by a `main` function which source code is displayed in Figure 2.

```c
int main(void) {
  int i1;                                                    // m1
  int i2;                                                    // m2
  int* pi1;                                                  // m3
  int* pi2;                                                  // m4

  i1 = 17; i2 = 19; pi1 = &i1; pi2 = &i2;                    // m5
  printf("Before swap1: i1=%d, i2=%d, *pi1=%d, *pi2=%d\n",
         i1, i2, *pi1, *pi2);                                // m6
  swap1(i1, i2);                                             // m7
  printf("After swap1: i1=%d, i2=%d, *pi1=%d, *pi2=%d\n",
         i1, i2, *pi1, *pi2);                                // m8

  i1 = 17; i2 = 19; pi1 = &i1; pi2 = &i2;                    // m9
  printf("Before swap2: i1=%d, i2=%d, *pi1=%d, *pi2=%d\n",
         i1, i2, *pi1, *pi2);                                // m10
  swap2(pi1, pi2);                                           // m11
  printf("After swap2: i1=%d, i2=%d, *pi1=%d, *pi2=%d\n",
         i1, i2, *pi1, *pi2);                                // m12

  i1 = 17; i2 = 19; pi1 = &i1; pi2 = &i2;                    // m13
  printf("Before swap3: i1=%d, i2=%d, *pi1=%d, *pi2=%d\n",
         i1, i2, *pi1, *pi2);                                // m14
  swap3(&pi1, &pi2);                                         // m15
  printf("After swap3: i1=%d, i2=%d, *pi1=%d, *pi2=%d\n",
         i1, i2, *pi1, *pi2);                                // m16
}
```

Figure 2: `main` function

Before explaining pointers, we need to know that each execution of a program, i.e, a process, has a corresponding "memory map", evolving over time. There are several different types of memory that can be mapped by a process. Among these memory types, because of the simplicity of the code we look at, in this document we are only interested in the memory that store local vairables and parameters. These variables are the ones defined **inside** functions. The positions in the local memory of these variables are decided at compilation time, hence by the compiler. The concepts presented in this document are valid for other types of memory.

For pedagogical purpose, we make the following assumptions in the rest of the document (even if they are not all always true in practice depending on a lot of parameters):

- the local memory is implemented by a stack;

- this stack grows down;
- the size of `int` is four bytes, i.e, 32 bits.
- the size of `int*` is eight bytes, i.e, 64 bits.
- when a function returns, its part of the stack is deleted.

1 Content of the stack before calling swapping functions

Before looking into the swapping functions, let's see what the stack looks like when the main function is about to call one of them, i.e, just before executing line `m7`, `m11`, `m15`. Because local variable are always reset to the same values before calling a swapping function, the stack content is always the same in the three execution points defined just before `m7`, `m11`, `m15`.

variable name	*type*	*stack value*	*effective address*	*symbolic address*	*owning function*
i1	int	17	0xFF28	&i1	main
i2	int	19	0xFF24	&i2	main
pi1	int*	0xFF28	0xFF20	&pi1	main
pi2	int*	0xFF24	0xFF18	&pi2	main

Figure 3: Stack content before calling any swap function

Figure 3 shows the stack at these three particular times along a lot of other information, that will, hopefully, help us to understand what is happening behind the scene. This figure shows that each variable definition in the source code of the main function correspond to a value in the stack, identified by the column labeled *stack* in the figure. Each stack variable has the following properties:

- a *type* defining the size required in the stack to store the value and used by the compiler to ensure some forms of correctness in the program, e.g, we assign values to variables with correct types;
- a value in the *stack* as already said;
- an *effective address* defining where in the stack the value is stored;

- a *symbolic address* which is simply the name of the variable preceded by the & symbol. In other words, this & operator allows to retrieve the *effective address* of a `variable` from its name;
- an owning function.

From Figure 3, it is clear that four variables have been defined in the `main` function. We have to integers, `i1` and `i2`, and two pointers to these integers `pi1` and `pi2`. Because `pi1` is initialized with the address of `i1` through the `pi1=&i1` statement, its `value`, i.e the content of the stack at its location, is equal to the *effective address* of `i1` which is 0xFF28. The same happens for `pi2`. Figure 3 also shows that the space required to store a pointer is twice as bis as the space to store an integer as stated in our assumptions.

2 Understanding the dereference operator *

Because the stack content is always the same before calling any swapping function, the `printf` calls located before calls to swapping functions will all output the same results which is:

```
Before swapX: i1 = 17, i2 = 19, *pi1 = 17, *pi2 = 19
```

Figure 4: Result of all the `printf` calls located before the calls to swapping functions

To understand this output, we need to understand the dereference operator which is the * symbol. In our `printf` call, the two last parameters are expected to have the integer type because the two last symbolic characters in the first argument of `printf` are `%d`. So let's consider what will happen if we write:

```
printf("Before swap1: i1=%d, i2=%d, *pi1=%d, *pi2=%d\n",
        i1, i2, pi1, pi2);
```

Figure 5: Wrong printf because types are incompatible

In this code, the two last parameters have type pointer to integer which is `different` from integer. As a consequence this code is incorrect.

But wait, what is the purpose of our printf ? It is to print `i1` and `i2` and the values of the stack locations whose addresses are the values of `pi1` and `pi2`. That is exactly what the dereference operator * does. **It takes the value of the memory location which address is the value of the variable being dereferenced.**

Let's now read again the bold sentence above and try to understand it by taking `*pi1` as an example. What is its value of the variable being dereferenced,

i.e, `p1`, when calling `printf` ? It is 0xFF28 according to Figure 3. Then what is the value of the stack location which address is 0xFF28 ? Again looking at Figure 3, we can see that it is 17. So `*p1` is 17.

Figure 6 below is a modified version of Figure 3 where we added the meaning of the `*` operator. This clearly shows where our pointers are pointing and what are the values `*pi1` and `*pi2` by following the arrows.

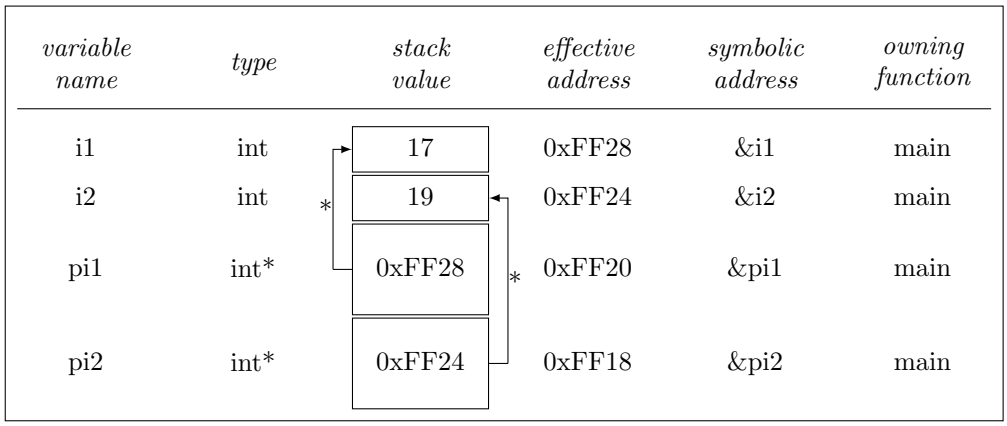

Figure 6: Stack content before calling any swap function showing what the dereference operator `*` does

It is **crucial** to understand that the `*` symbol has two different meanings. It is the dereference operator when used in the front of a variable name which has been already defined. It is a type information when used in the definition of a variable such as `int* p1 = ...`.

3 Understanding `swap1`

We now look at the content of the stack inside the `swap1` function at the execution point located just before `s12` as shown in Figure 7 below. In the following, all the stack diagram that we'll show are chronological in program execution order. In a diagram we'll show in red the parts that have changed compared to the previous diagram.

Below (because our stack grows down) the local variables of the `main` function we now have three new local variables belonging to `swap1`. The first two are the two integer parameters `i1` and `i2` of `swap1`. Their **values** have been filled by the `main` function just before calling `swap1` at `m7` (parameters are passed by value in the C programming language). It is **crucial** here to understand, and hopefully clear from the figure, that we have two different variables named `i1` and two different variables named `i2`. Inside the `swap1` function, every usage of the `i1` and of the `i2` variables refer logically to the stack location of the parameters `i1` and `i2` at the bottom of our stack. In other words, it is **impossible** to

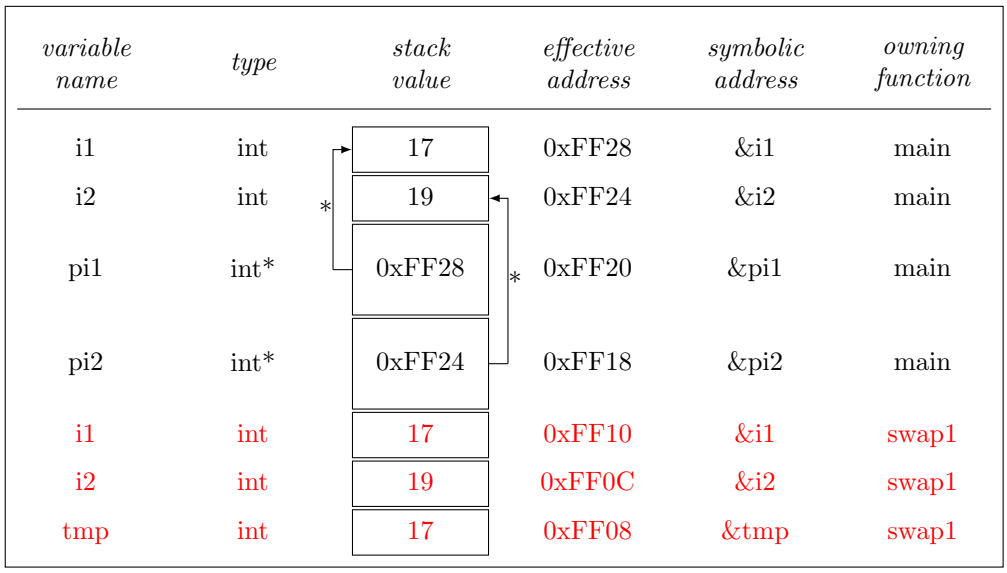

Figure 7: Stack content in `swap1` just before `s12`

access the `i1` and `i2` local variables of the `main` function inside the `swap1` one. We have deliberately chosen to give the same name to main local variables and to swapping functions parameters to illustrate this point. Nevertheless, changing the name of the two parameters of `swap1` to `a` and `b` will have no effect to the result of our program. It will just change the content of the first column in our drawings. In our stack, we also have now the `tmp` variable which **value** is 17 because it has been assigned to `i1` with the `tmp = i1` statement.

The two following lines in `swap1`, `s12` and `s13`, exchange the content of `i1` and `i2`. Because we are inside the `swap1` function, these `i1` and `i2` variables refer to the parameters of `swap1`. Figure 8 shows the content of the stack just after `s13`. We see that `i1` and `i2`, the ones belonging to the `swap1` function, have been swapped.

Then, when returning from the `swap` function, the associated part of the stack is removed as stated in our assumptions. So just before `m8`, our stack content is the one shown in Figure 9. Because the `swap1` function has just exchanged the values of its local parameters that have been removed from the stack, the content of the stack is exactly the same than the one before calling `swap1` shown previously in Figure 3.

As a consequence, the output of the `printf` call just after the `swap1` call at line `m8` will be the same as the one before the call to `swap1` as shown on Figure 10 below.

variable name	*type*	*stack value*	*effective address*	*symbolic address*	*owning function*
i1	int	17	0xFF28	&i1	main
i2	int	19	0xFF24	&i2	main
pi1	int*	0xFF28	0xFF20	&pi1	main
pi2	int*	0xFF24	0xFF18	&pi2	main
i1	int	19	0xFF10	&i1	swap1
i2	int	17	0xFF0C	&i2	swap1
tmp	int	17	0xFF08	&tmp	swap1

Figure 8: Stack content in `swap1` just after `s13`

4 Understanding `swap2`

In the previous section we have seen that the `swap1` function has no effect. In other words, it is useless because it does not swap anything excepted its local parameters. The `swap2` function is the correct way to swap two integers in C (if we want a function for that). Compared to `swap1`, the `swap2` parameters are of type pointer to integers and not integers. Let's see why it makes the difference by looking at the stack evolution again.

Figure 11 shows the state of the stack just before `s22`. As for the `swap1` function, but with pointers to integer instead of integers, we have three new entries in the stack compared to the stack before the call to `swap2`. These three lines correspond to the two parameters of `swap2` having the type pointers to integer and to the `tmp` local variable also having the type pointer to integer. The **values** of the `pi1` and `pi2` parameters have been filled by the `main` function just before calling `swap2` at `m11` (parameters are passed by value in the C programming language, this is true for pointers parameters also). Again we used the same name for the variables local to `main` and the parameters of `swap2` to illustrate that there are really two different versions of `pi1` and two different versions of `pi2` in the stack as shown by Figure 11. Nevertheless, the values of the two different versions of `pi1` and the values of the two different versions of `pi2` are equal because the effective parameters of the `swap2` call `are` the `main` local variables `pi1` and `pi2`. Also, the value of the `tmp` `swap2` local variable has been initialized by dereferencing the pointer `pi1` with `*pi1`. Recalling that in our stack diagrams, dereferencing consists in following the arrows starting from

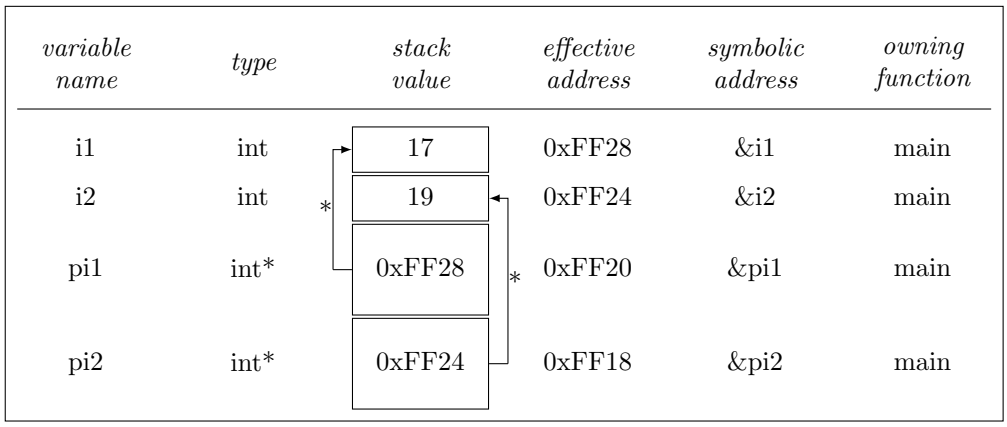

Figure 9: Stack content after the call to `swap1` has returned, i.e, just before `m8`

```
After swap1: i1 = 17, i2 = 19, *pi1 = 17, *pi2 = 19
```

Figure 10: Result of the call to `printf` calls located after the call to `swap1`

the variable being dereferenced, we know that `*pi1` is equal to 17.

From Figure 11 it is clear to see (the red arrows) where the three integer pointers `pi1`, `pi2` and `tmp` of the `swap2` function are pointing to. `pi1` and `tmp` point to the memory location holding the `main` local variable `i1` while `pi2` points to the memory location holding the `main` local variable `i2`.

Let's now perform the swapping by executing the lines `s22` (`*pi1 = *pi2`) and `s23` (`*pi2 = tmp`). Here, on both sides of the `s22` line we are dereferencing a pointer. Again, by following the arrows starting from the variable being dereferenced, it is easy to see that `*pi1` is the stack location where the `main` local variable `i1` is and to see that `*pi2` is the stack location where the `main` local variable `i2`. Figure 12 shows the stack content after the execution of `s22` and `s23`.

Then when the `swap2` function returns, the part of the stack associated to it is removed, and we see that the values of the `main` local variable `i1` and `i2` have been really swapped. As a consequence, the output of the `printf` call just after the `swap2` at line `m12` one will be the one shown on Figure 13 below.

5 Understanding `swap3`

Now that we have understood how pointer to integers behave, let's look at pointers to pointer to integer. In this case, to access an integer value we'll have to dereference the pointer to pointer to integer two times.

Figure 14 shows the state of the stack just before `s32`. As for the `swap2`

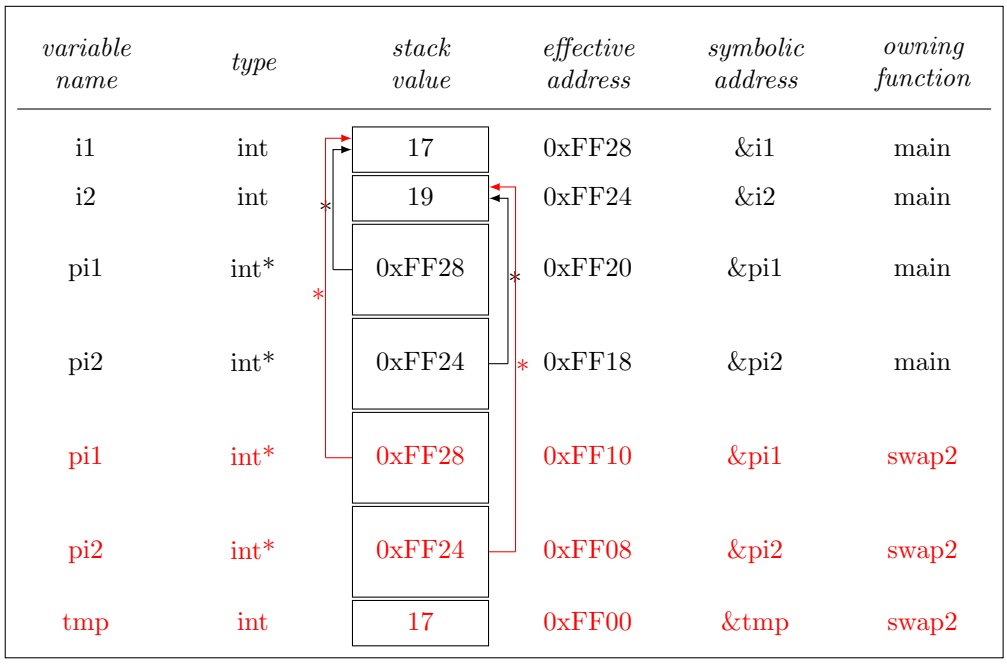

Figure 11: Stack content in `swap2` just before `s22`

function, but with pointers to pointer to integer instead of pointers to integer, we have three new entries in the stack compared to the stack before the call to `swap3`. These three lines correspond to the two parameters of `swap3` having the type pointers to pointer to integer and to the `tmp` local variable also having the type pointer to pointer to integer. The **values** of the `ppi1` and `ppi2` parameters have been filled by the `main` function just before calling `swap3` at `m15` (parameters are passed by value in the C programming language, this is true for pointers to pointer parameters also). These values are the effective addresses of the `main` local variables `pi1` and `pi2` that are retrieved by `&pi1` and `&pi2` in line `m15`.

Let's now perform the swapping by executing the lines `s32` (`*ppi1 = *ppi2`) and `s33` (`*ppi2 = tmp`). Here, on both sides of the `s32` line we are dereferencing a pointer. Again, by following the arrows starting from the variable being dereferenced, it is easy to see that `*ppi1` is the stack location where the `main` local variable `pi1` is and to see that `*ppi2` is the stack location where the `main` local variable `pi2`. Figure 15 shows the stack content after the execution of `s32` and `s33`.

Then when the `swap3` function returns, the part of the stack associated to it is removed, and we see that the values of the `main` local variable `pi1` and `pi2` have been really swapped. As a consequence, the output of the `printf` call just after the `swap3` at line `m16` one will be the one shown on Figure 16 below.

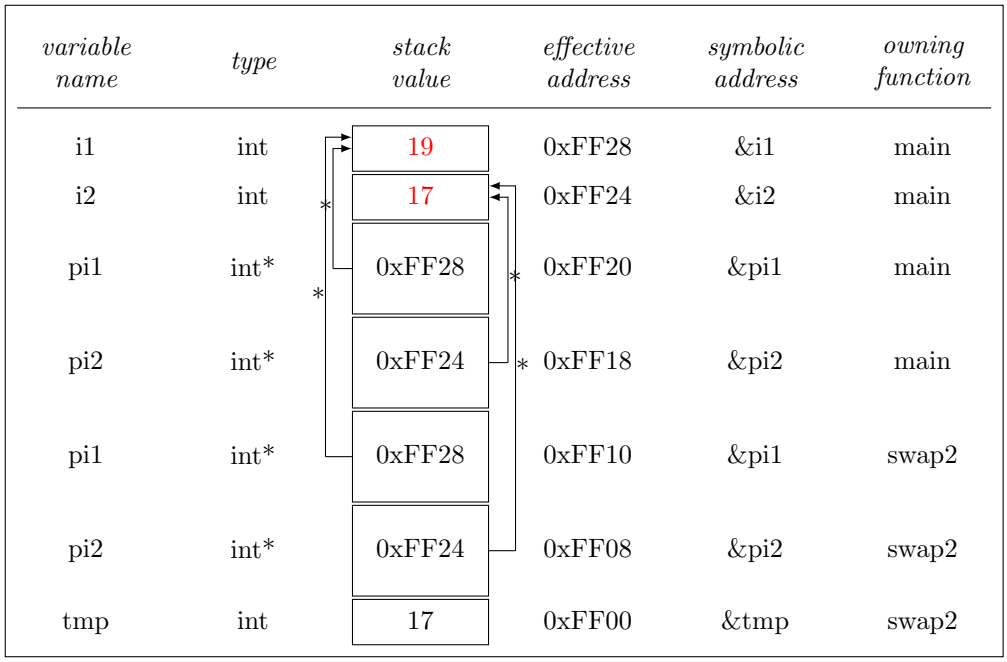

Figure 12: Stack content in `swap2` just after `s23`

```
After swap2: i1 = 19, i2 = 17, *pi1 = 19, *pi2 = 17
```

Figure 13: Result of the call to `printf` calls located after the call to `swap2`

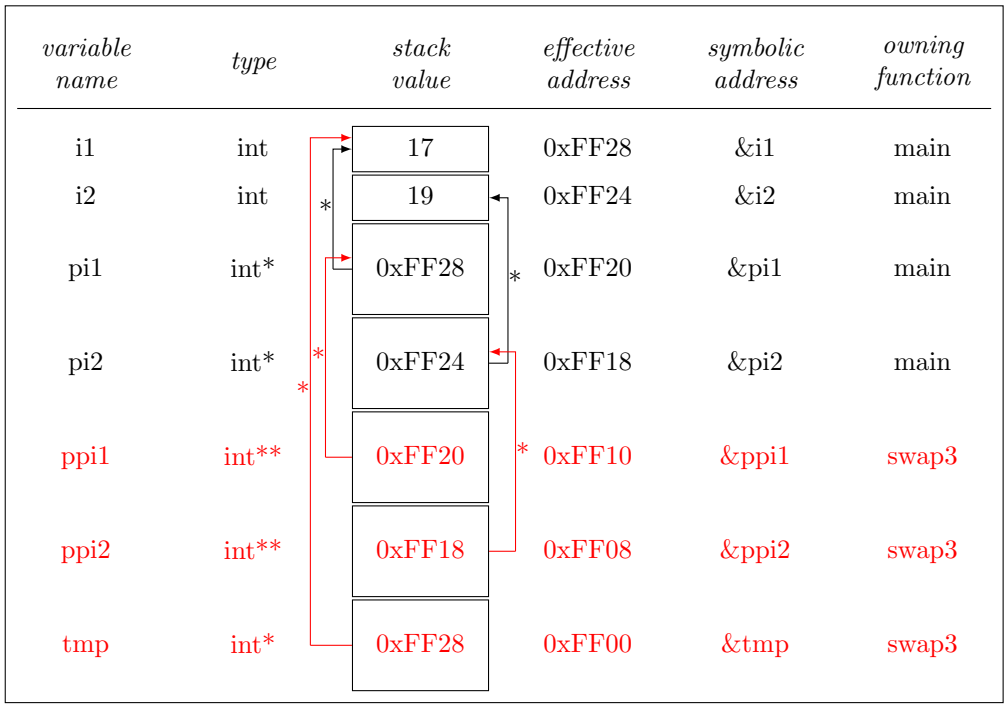

Figure 14: Stack content in `swap3` just before `s32`

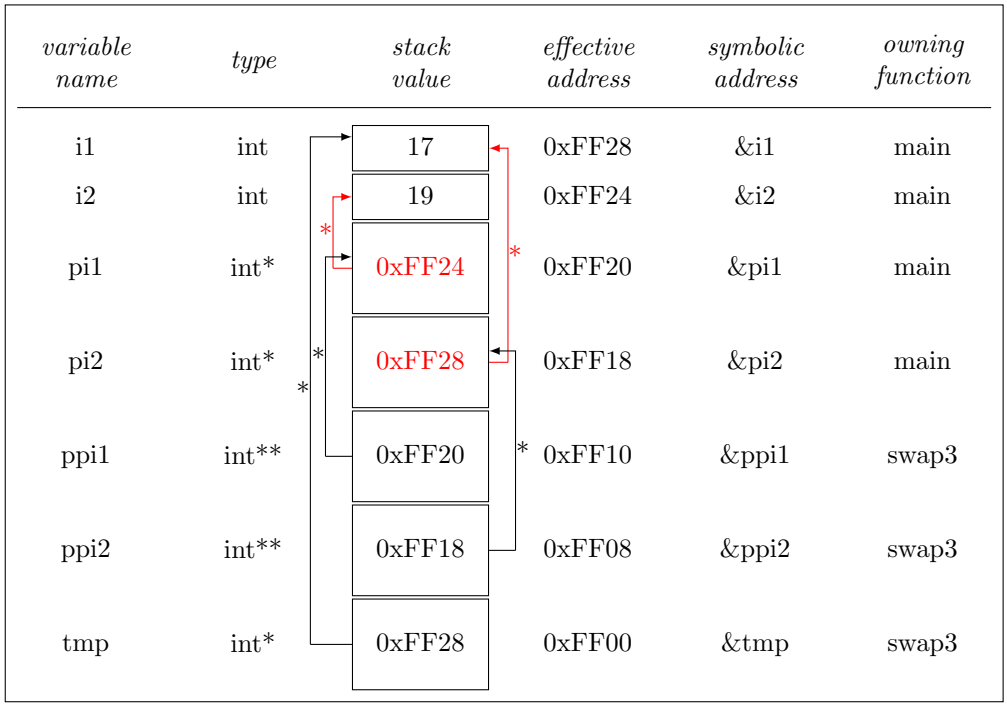

Figure 15: Stack content in `swap2` just after `s23`

```
After swap3: i1 = 17, i2 = 19, *pi1 = 19, *pi2 = 17
```

Figure 16: Result of the call to `printf` calls located after the call to `swap3`