Pointers and Memory Management (based on Jylhä-Ollila's lecture in 2011)

Timo Karvi

2012

Memory and C I

- In most modern computers, main memory is divided into bytes, with each byte capable of storing eight bits of information.
- Each byte has a unique address to distinguish it from other bytes in memory.
- Each variable occupies one or more bytes of memory. The address of the first byte is said to be the address of the variable.
- Below is the table showing the typical sizes of some basic data types:

char	exactly one byte
short	$\geq$ 2 bytes
int	$\geq$ typically 4 bytes
long	$\geq$ 8 bytes on 64-bit environments
long long	$\geq$ 8 bytes

Memory and C II

- In the following examples we shall assume that int is 4 bytes. However, C code should not assume that a type is of certain size. C99 only quarantees minimum sizes for arithmetic types, and that char is always a single byte.
- If you need exact bit widths, the header file stdint.h typically contains definitions for types `int8_t`, `int16_t`, `int32_t`, `int64_t`.

Although addresses are represented by numbers, their range of values may differ from that of integers, so we can't always store them in ordinary integer variables. We can, however, store the byte addresses in special pointer variables.

When we store the address of a variable i in the pointer variable p, we say that p point to i. In other word, a pointer is nothing more than a byte address.

Example 1 I

Consider the definitions:

```
int x = 5;
int* p = &x;
```

- First, the integer variable x is defined and x is given the value 5.
- Second, a pointer variable is introduced. p points to the integer type data. Here p gets the initial value `&x`, that is the address of the variable x (i.e. the address of the first byte of x).
- Now we can use the value 5 either through x or p. For example,

```
int a, b;

a = x+1;    /* a = 6 */
b = *p+2;   /* b = 7 */
```

Example 1 II

- The evaluation of the expression `*p` starts by going to the memory location stored in variable `p`. Next, the four bytes starting from that location are read as an int, because `p` is a pointer to an integer. In other words, the type of a pointer affects both how many bytes are read from memory and how those bits are interpreted.

Example 2 I

Consider the following variable declarations:

```
int x = 5;
char ch = x;
```

- On the second line, the compiled code will evaluate x and store the value 5 into ch. There is nothing surprising here, as the char type can represent the value 5. Consider now the following:

```
char* cp = (char*)&x;
```

- Here, we take the address of the int variable x and convert the typem of the address to an address of type char. All pointers are essentially unsigned integers and they all have the same size, so the address of x can be correctly stored in the variable cp. Now what happens, if we dereference cp?

Example 2 II

- The evaluation of *cp behaves according to the type of cp. The code looks at the data starting from the memory location stored in cp and interprets the single byte at that location as a char value.
- In other words, *cp evaluates the first byte of the int value x in memory. In little-endian systems, the first byte of x contains the least significant bits of x, so *cp will evaluate to 5 in that case. In big-endian systems, the first byte of x contains the most significant bits, which are all zero, so *cp will evaluate to 0.

Example 3 I

In principle, the following code is still legal, although it suggests that we are about to do something dangerous.

```
char ch = 42;
int* ip = (int*)&ch;
```

- If we try to evaluate `*ip`, the code will attempt to follow the same procedure as previously explained. It will look at the memory location stored in `ip` and read four bytes starting from that address, although we have stored a single byte to the address of `ch`. Especially, we have allocated only one byte for `ch`.
- Code evaluating `*ip` does not care that we the three bytes after `ch` are reserved for the program.
- As a result, evaluating `*ip` may result in crash or it may succeed and we may get an unexpected value, or even 42. This is an example of undefined behaviour in C.

Arrays and pointer arithmetics I

- Arrays are defined in C in the following way:
  - int arr1[10];
  - int arr2[5] = 1, 2, 3, 4, 5;
  - int arr3[20] = 1, 2, 3, 4;
  - int arr4[] = 1, 2, 3, 4, 5, 6;
- arr1 is of size 10 and of integer type. The indexing starts at zero. So the array has values in arr1[0], ...,arr1[9]. It is possible to initialize an array as is done with arr2. It is not necessary to initialize it completely as with arr3; the rest of the array will be zero (arr3[4] = 0, ..., arr3[19] = 0). If initialized, it is not necessary to give the size of an array as with arr4.
- If an array is not initialized as with arr1, then the contents of the array are whatever data previously existed in the memory allocation reserved for the array.

Arrays and pointer arithmetics II

- The name arr1 is a synonym for the address of the first element in arr1, so the following is always true:

$$\text{arr1} \equiv \&\text{arr1}[0]$$

arr1 is the address of the first byte in the block of at least 40 bytes (int size is 4). The type of `arr1[0]` is int, and therefore the type of `&arr1[0]` is `int*`. As a result, we can do following:

```
int* ap = arr1;
```

In other words, we can set the pointer to point to the firt element of the array. Also note that the type of ap does not indicate in any way that there is more than one `int` value stored at the memory area starting at ap.

- So what happens when we do the following:

```
arr1[5] = 20;
```

Arrays and pointer arithmetics III

IThe code determines the correct destination in memory for number 20 by starting at the base address of the array `arr1`. From that address, it moves forward 5 times the size of int and stores the number 23 there.

- C does not check bounds for arrays, so it is possible to compile the following (maybe recent versions of gcc may give a warning in this simple case):

```
arr1[10] = 99;
```

The compilid code is not concerned with the fact that the index 10 is not a valid index of `arr1`. It will still move forward a space of 10 times the size of int and attempt to store 99 in that memory location outside the array. Thus it may overwrite other important parts of program memory.

Arrays and pointer arithmetics IV

- The memory location `arr1[k]` can be expressed in the form $arr + k$, i.e. the following is always true:

$$arr1 + k \equiv \&arr1[k]$$

- When one of the operands of an addition has a pointer or array type and the other is integer, the integer operand is scaled based on the type of the pointer or array. Thus the memory location $arr1 + k$ is not k bytes from the base address of *arr1*, but $k \times \mathrm{sizeof(int)}$ bytes, as `arr1` is an array of integers.
- Based on the correspondence between $arr1 + k$ and `&arr1[k]`, the following are also true:

$$\begin{aligned} *(arr + k) &\equiv arr[k], \\ *arr &\equiv arr[0] \end{aligned}$$

Arrays and pointer arithmetics V

- An interesting consequence of this property is that, due addition being commutative, it is possible to write both `arr[k]` and `k[arr]`. The latter is not recommended.

Example 4 I

```
short arr[5] = {0, 11, 22, 33, 44};
char* cp = (short*)arr;
cp[4] = 1;
```

- In this example, the contents of the short array are modified through the char pointer cp. The memory location of `cp[4]` is calculated based on the type information of cp. Therefore, the code will step $4 \times \mathrm{sizeof}(\mathrm{char})$ or 4 bytes from cp.
- This address is also $2 \times \mathrm{sizeof}(\mathrm{short})$ bytes from the base address of arr, i.e. the address of `arr[2]`. Therefore, the assigment `cp[4] = 1` replaces the first byte of the short value 22 stored in `arr[2]` with the char representation of 1.
- Furthermore, an assignment `cp[5] = 42` would replace the second byte of `arr[2]` with the char representation of 42.

Example 5 I

Consider the following type definition and array:

```
typedef struct
{
    int num;
    int den;
} Fraction;

Fraction fa[3];
```

- We assume that Fraction is stored in memory as two consecutive integers (int). However, a C implementation may insert padding bytes between struct members and after the last member (but never before the first element). Thus sizeof(Fraction) may be greater than $2 \times \text{sizeof}(\text{int})$. However, the address of a struct is always the same as the address of its first member.

Example 5 II

- We can modify the contents of fractions in arrays in the familiar way. Here, we store the fraction 1/2 in the first elemntof the array:

```
fa[0].num = 1;
fa[0].den = 2;
```

- What happens if we do the following:

```
Fraction* fa2 = (Fraction*)&fa[0].den;
fa2[0].num = 5;
fa2[0].den = 6;
```

Example 5 III

- As a result of the pointer cast in the first row, the address of the den-field of the first fraction in the array is treated like an address of Fraction. Therefore, the address of `fa2[0].num` is the same as the address of `fa[0].den`. The fraction in `fa[0]` will be 1/5, and the fraction in `fa[1]` will have 6 as the numerator (denominator is uninitialized).

Example 6: Stack I

In this example we present a simple stack implemetations for the operations push and printStack.

```
#include <stdio.h>

struct stackElement {
 int value;
 struct stackElement* next;
};
```

Example 6: Stack II

```
void printStack(struct stackElement* S){

    struct stackElement* sp = S;

    while (sp != NULL){
        printf("%d  ", sp->value);
        sp = sp->next;
    }

    printf("\n");
}
```

Example 6: Stack III

```
struct stackElement* push(struct stackElement* S, int n)
{
    struct stackElement* newNode;
    newNode =
    (struct stackElement*) malloc(sizeof(struct stackElement)
    newNode->value = n;

    if (S == NULL) {
       newNode->next = NULL;
       S = newNode;
    }
    else {
       newNode->next = S;
       S = newNode;
    }
    return S;   }
```

Example 6: Stack IV

```
int main(void)
{

    struct stackElement* S;

    S = push(S,1);
    S = push(S,2);
    S = push(S,3);
    printStack(S);

    return 0;
}
```

Example 7: Swap I

The following function swaps two int values by acceptin their memory locations as arguments:

```
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

The function can be called as follows:

```
int x = 5;
int y = 19;
swap(&x, &y);
```

Example 7: Swap II

The only part of the code specific to the int type is the amount of bytes swapped. A generic version of the swap function accepts a third argument that specifies the size of the elements being swapped:

```
void swap(void* a, void* b, size_t elem_size)
{
    char* ca = a;
    char* cb = b;
    for (size_t i = 0; i < elem_size; i++)
    {
        char temp = ca[i];
        ca[i] = cb[i];
        cb8i] = temp;
    }
    return;
}
```

Example 7: Swap III

There are some new things here:

- A value of type `void*` is a generic pointer, which does not have type information associated with it. Any pointer type can be converted to a generic pointer implicitly, so it is easy to pass pointers of any type as arguments a and b.
- As generic pointers do not have size information associated with them, we need to tell the swap function the amount of bytes to be swapped. The `size_t` type is an unsigned integer type used by the `sizeof()` function and some library functions in the context of element sizes.
- In the following rows, we set up a per-byte access to the memory areas to be swapped:

```
char* ca = a;
char* cb = b;
```

Example 7: Swap IV

Thus ca points to the first byte of the data a, and cb to the first byte of the data b.

- Dereferencing generic pointers is not allowed in C, so we need another pointer type for accessing the data pointed to by a and b (or ca and cb). The `char*` type suits well, as the char type is always one byte in size.
- After we have access to the individual bytes through what are essentially char arrays, the rest of the code is swapping the corresponding bytes of the two memory areas one at a time.
- Example call of the generic swap function:

Example 7: Swap V

```
int x = 5;
int y = 32;
swap(&x, &y, sizeof(int));

double a = 3.14159;
double b = 2.71828;
swap(&a, &b, sizeof(double);

char* n1 = "Alice";
char* n2 = "Bob";
swap(&n1, &n2, sizeof(char*));
```

Note that the last row only swaps the contents of the pointers n1 and n2; the contents of the strings do not move.

Example 7: Swap VI

- We have lost some safety features with the generic swap function. As any pointer type may be converted to a generic pointer, the compiler can not check our code for mistakes related to pointer type. For example, the compiler will accept the following calls without giving any warnings:

```
short x = 12;
long  y = 500;
swap(&x, &y, sizeof(short)) //parts of y are not moved
char* n1 = "Alice";
char* n2 = "Bob";
swap(&n1, &n2, sizeof(char)); // only swaps
                              //  the first byte
swap(n1, n2, sizeof(char)); // attempts to swap
                 //the bytes with address n1 and n2.
```

Example 7: Swap VII

The last function call will most likely crash, since it is not allowed to modify string literals.

Dynamic memory allocation I

Consider the following simple structure that stores a collection of int values and how many values are stored:

```
typedef struct Elements
{
    int* numbers;
    int length;
} Elements;
```

We may not know beforehand how many numbers the user of the structure needs. Therefore, we do not want to specify the numbers field as a fixed-size array. Instead, we allocate the memory for the numbers at run time by using dynamic memory allocation. Also, we may not want to restrict the lifetime of an Elements structure to a single block of code. Therefore, we also allocate memory for the struct itself dynamically.

Dynamic memory allocation II

The next function creates a new Elements structure and allocates memory for as many int values as requested.

```
Elements* Elements_create(int length)
{
    Elements* e = malloc(sizeof(Elements));
    if (e == NULL)
        return NULL;
    e->numbers = malloc(sizeof(int) * length);
    if (e->numbers == NULL)
    {
        free(e);
        return NULL;
    }
    e->length = length;
    return e;}
```

Dynamic memory allocation III

The malloc function reserves memory blocks from a structure called heap (not related to the priority queue structure). The parameter passed to malloc is the number of bytes required. Memory blocks allocated with malloc stay reserved until explicitly freed with the free function. In other words, dynamically allocated memory blocks are not constrained to code blocks where they are created. It is important to note that malloc may fail to allocate memory, in which case it returns NULL.

The `Elements_create` function calls malloc twice. If the second call fails, it has only allocated memory for the Elements struct, but not the dynamic int array inside. In this case, it needs to free the Elements struct so that it can fail cleanly.

The operator `->` is used to access struct members through a pointer to a struct. In other words, `a->b` and `(*a).b` are equivalent.

Dynamic memory allocation IV

It is worth noting that `Elements_create` is very similar to constructors in object-oriented languages. The most important difference is that the memory allocation is explicit.

As dynamically allocated memory must be explicitly freed in C, we also need a destructor to clean things up. The next function frees memory allocated to an Elements object by the `Elements_create` function:

```
void Elements_destroy(Elements* e)
{
    if (e == NULL)
        return;
    free(e->numbers);
    free(e);
    return;
}
```

Dynamic memory allocation V

Note that the order of the free calls is important. If we freed the memory allocated to e before `e->numbers`, evaluation of `e->numbers` would dereference e, which would no longer be a legal memory location. Also, it is important to not call free more than once for a reserved memory block.

It is good programming practice to write the deallocation code right after the allocation code, if it is practical to do so. This way, it is less likely that we forget to write proper deallocation code.

Undefined behaviour I

According to C99, undefined behaviour is:

> *"behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements."*

A C programmer should take the "no requirements" part very literally, as undefined behaviour need not make sense in any way. For instance, it may result in corruption of data that is apparently unrelated to the code executed. Therefore, it is important to avoid code that causes undefined behaviour.

Example 1

Dereferencing a NULL pointer causes undefined behaviour. NULL is a special pointer value that never points to a valid memory location. A related case is passing a NULL pointer associated with a %s format to the printf function, which also causes undefined behaviour:

Undefined behaviour II

```
char* str = NULL;
printf("%s\n", str);
```

Running the code may have the following result, which indicates that the program attempted to access a memory location that is not reserved for the program:

```
Segmentation fault
```

Example 2

```
short x = 0;
scanf("%hd", &x);
short y = (x * 2) / 2;
```

Undefined behaviour III

The expression `x * 2` may cause an integer overflow if the user enters a large number like 30000. Thus, the expression `(x * 2) / 2` may yield a result different from x. Nevertheless, an optimising compiler is allowed to remove the multiplication and division by 2, as signed integer overflow causes undefined behaviour. Therefore, the exact result of the computation is not considered important in the case of overflow. (Note that unsigned integer overflow is allowed and well-defined in C.)

Buffer overflows I

Example 3

The following function contains a common bug in reading user input:

```
void parrot(void)
{
    printf("What should I say?\n");
    char str[20] = "";
    scanf("%s", str); // user input may cause buffer overflow
    printf("I say: %s\n", str);
    return;
}
```

Buffer overflows II

Writing data to memory locations not allocated for the code leads to undefined behaviour. Buffer overflow bugs are a common source of security vulnerabilities in C programs.

Typical C implementations use a stack as storage for local variables (including arrays) and some bookkeeping information. Specifically, one piece of bookkeeping data is a return address which is used to find the point of execution after a function call returns. If user input is allowed to overwrite the return address, the user may be able to change the execution path of the program.

The parrot function can be fixed by specifying the maximum field width in the scanf format string:

```
scanf("%19s", str);
```

Buffer overflows III

Strings in C are essentially char arrays that end in a terminating zero byte (written in code as '\0'). The fixed scanf call reads at most 19 (non-whitespace) characters from the standard input and stores them, along with the terminating '\0', to the memory block starting at str. When dealing with functions that process strings (such as strlen, str(n)cpy, snprintf and fgets), it is important to check how they handle the '\0' byte.

Other memory errors I

Example 4

Another common memory error is to access an object outside of its lifetime. The following code is a simple example:

```
int* foo(void)
{
    int x = 5;
    return &x;
}
```

Memory for the variable x is allocated by using automatic memory allocation. As a result, the memory reserved for x is automatically freed at the end of the code block where x was declared, i.e. before the caller of foo receives the address of x.

However, accidentally accessing an object outside of its lifetime is much more common with dynamic memory allocation. Consider the Elements

Other memory errors II

structure from the previous section and what would happen if we made the following mistake in the deallocation code:

```
Elements* Elements_destroy(Elements* e)
{
    if (e == NULL)
        return;
    free(e);
    free(e->numbers); // *e is no longer valid here!
    return;
}
```

If we freed the memory allocated to e before `e->numbers`, evaluation of `e->numbers` would dereference e, which would no longer be a legal memory location. Also, it is important to not call free more than once for a reserved memory block.

Sequence points I

The C99 standard grants compilers a lot of freedom in evaluation order of expressions. In the following statement, the functions f, g and h may be called in any order:

```
arr[f(x)] = g(y) * (h(z) + 1);
```

Consider the implications of this freedom if we have a statement like the following:

```
arr[i] = i++;
```

What value of i is used as an index of arr? C99 does not specify that; in fact, the behaviour of the statement is undefined.

A sequence point is a point of execution where previous side effects in the code have finished and the following ones have not yet taken place. The following code points (among some others) are sequence points in C:

Sequence points II

- ; (end of statement)
- && and || operators — these allow the following construct: `if (p != NULL && *p ... )`
- After the ? in the short-hand if-else operator ?:
- Before/after loop condition tests and for loop index initialisation/increment
- Before a function call (after evaluation of the arguments) and after function return

However, it is notable what points in code are not sequence points. For instance, the expression

```
arr[i] = i++
```

does not contain any sequence points. Also, evaluation order of function arguments is not limited by sequence points. If a memory location is updated between adjacent sequence points, it is not permitted to read it (or update again) between the same sequence points.

Defensive Programming I

In order to write safe C code, it is important to adopt clear and consistent style, and prepare for mistakes while programming.

Example 5: assert

Whenever you make a non-trivial assumption about the program state in your C code, test it with the assert macro (defined in assert.h). Good places for assert tests include checking of function arguments:

```
#include <assert.h>

int Elements_get(Elements* e, int index)
{
    assert(e != NULL);
    assert(index >= 0);
    assert(index < e->length);
    // ...
```

Defensive Programming II

If an assertion test is false, the execution of the program terminates with an error message describing the source file and location of the test. It is also possible to disable the assert statements at compile time by passing the flag -DNDEBUG to gcc. Therefore, the programmer does not need to worry about performance loss caused by assert checks when building a release version of the program.

Note that you should not test user input with assert. The assert macro is designed to help with finding programming errors. Also, it is important to avoid assert code that modifies data outside the assert check. Such code could make the debug version of the program produce different results from the release version.

Other techniques and habits I

Remember the gcc options -std=c99 -pedantic -Wall -Wextra -Werror that enable the compiler detect many potential errors at compile time.

Initialise all variables with an explicit value, especially pointers. An uninitialised variable in C contains a value represented by a bit pattern that happens to reside in the memory location reserved for the variable. This may be essentially random data, or worse, data injected by a malicious user.

Whenever you call malloc, write the corresponding free call as soon as possible. This way you consider the lifetime of the dynamically allocated object and are less likely to forget the free call.

Read the man pages of the library functions you use. You can find information about a standard C library function by writing `man 3 function_name` in the terminal, or by going to the address: `http://linux.die.net/man/3/function_name` (Ubuntu users can

Other techniques and habits II

install the package “manpages-dev” to get the man pages). Note: The man pages typically refer to many different C standards. It is important to check the “CONFORMING TO” section to see that the features you plan to use are specified in the C99 standard.

When good habits are not enough I

Even with good programming habits, memory-related bugs are common in C programming. Valgrind is a very useful tool for debugging memory use in C programs. It can detect problems like use of uninitialised variables, buffer overflows (with dynamically allocated memory blocks) and memory leaks (errors where dynamically allocated memory blocks are not freed).

If you want to run your program with Valgrind, compile your C program with the -g switch.

Example 6
The following code contains a buffer overflow on a dynamically allocated block:

When good habits are not enough II

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int* arr = malloc(sizeof(int) * 500);
    if (arr == NULL)
    {
        printf("No memory :-(\n");
        exit(EXIT_FAILURE);
    }
    // oops: the loop writes to arr[500]
    for (int i = 0; i <= 500; ++i)
    {
        arr[i] = 10 * i;
    }
```

When good habits are not enough III

```
    printf("%d\n", arr[45]);
    free(arr);
    exit(EXIT_SUCCESS);
}
```

The code accidentally writes one int value beyond the end of the allocated memory block. Although the program is likely to run "correctly", compiling the program with the -g switch and running with Valgrind reveals the following issue (emphasis added):

```
$ valgrind --leak-check=full --show-reachable=yes
         ./error_overflow
==25700== Memcheck, a memory error detector
==25700== Copyright (C) 2002-2010, and GNU GPL'd,
          by Julian Seward et al.
==25700== Using Valgrind-3.6.1 and LibVEX; rerun with -h f
```

When good habits are not enough IV

```
          or copyright info
==25700== Command: ./error_overflow
==25700==
==25700== Invalid write of size 4
==25700==    at 0x40066A: main (error_overflow.c:17)
==25700==  Address 0x51c3810 is 0 bytes after a block of
           size 2,000 alloc'd
==25700==    at 0x4C28FAC: malloc (vg_replace_malloc.c:236)
==25700==    by 0x400625: main (error_overflow.c:9)
==25700==
450
==25700==
==25700== HEAP SUMMARY:
==25700==     in use at exit: 0 bytes in 0 blocks
==25700==   total heap usage: 1 allocs, 1 frees,
            2,000 bytes allocated
```

When good habits are not enough V

```
==25700==
==25700== All heap blocks were freed --
==25700==  no leaks are possible
==25700==
==25700== For counts of detected and suppressed errors,
==25700== rerun with: -v
==25700== ERROR SUMMARY: 1 errors from 1 contexts
==25700== (suppressed: 4 from 4)
```

Valgrind detects the buffer overflow and reports the location in the code where the error occurred (line 17 in `error_overflow.c`) and where the related memory block was allocated (line 9). If Valgrind does not display the source files or the line numbers, the executable was likely compiled without the -g switch.

Even after the code is fixed, Valgrind outputs quite a bit of information (the line with 450 is the output of our program):

When good habits are not enough VI

```
$ valgrind --leak-check=full --show-reachable=yes
                  ./error_overflow
==26387== Memcheck, a memory error detector
==26387== Copyright (C) 2002-2010, and GNU GPL'd,
          by Julian Seward et al.
==26387== Using Valgrind-3.6.1 and LibVEX; rerun
           with -h for copyright info
==26387== Command: ./error_overflow
==26387==
450
==26387==
==26387== HEAP SUMMARY:
==26387==     in use at exit: 0 bytes in 0 blocks
==26387==   total heap usage: 1 allocs, 1 frees,
            2,000 bytes allocated
==26387==
```

When good habits are not enough VII

```
==26387== All heap blocks were freed -- no leaks are possible
==26387==
==26387== For counts of detected and suppressed errors,
          rerun with: -v
==26387== ERROR SUMMARY: 0 errors from 0 contexts
          (suppressed: 4 from 4)
```

Example 7

The following code does not free the dynamically allocated memory block:

When good habits are not enough VIII

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int* arr = malloc(sizeof(int) * 500);
    if (arr == NULL)
    {
        printf("No memory :-(\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < 500; ++i)
    {
        arr[i] = 10 * i;
    }
    printf("%d\n", arr[45]);
```

When good habits are not enough IX

```
    // we forget to free arr
    exit(EXIT_SUCCESS);
}
```

Valgrind outputs the following diagnostic (note the command line arguments –leak-check=full –show-reachable=yes passed to Valgrind):

```
$ valgrind --leak-check=full --show-reachable=yes
                    ./error_mem_leak
==26299== Memcheck, a memory error detector
==26299== Copyright (C) 2002-2010, and GNU GPL'd,
            by Julian Seward et al.
==26299== Using Valgrind-3.6.1 and LibVEX; rerun
            with -h for copyright info
==26299== Command: ./error_mem_leak
==26299==
```

When good habits are not enough X

```
450
==26299==
==26299== HEAP SUMMARY:
==26299==     in use at exit: 2,000 bytes in 1 blocks
==26299==   total heap usage: 1 allocs, 0 frees,
             2,000 bytes allocated
==26299==
==26299== 2,000 bytes in 1 blocks are still reachable
           in loss record 1 of 1
==26299==    at 0x4C28FAC: malloc (vg_replace_malloc.c:236)
==26299==    by 0x4005E5: main (error_mem_leak.c:9)
==26299==
==26299== LEAK SUMMARY:
==26299==    definitely lost: 0 bytes in 0 blocks
==26299==    indirectly lost: 0 bytes in 0 blocks
==26299==      possibly lost: 0 bytes in 0 blocks
```

When good habits are not enough XI

```
==26299==    still reachable: 2,000 bytes in 1 blocks
==26299==         suppressed: 0 bytes in 0 blocks
==26299==
==26299== For counts of detected and suppressed errors,
          rerun with: -v
==26299== ERROR SUMMARY: 0 errors from 0 contexts
          (suppressed: 4 from 4)
```

Valgrind tells where the leaked memory block was allocated (line 9 in `error_mem_leak.c`).