Arrays

Programação (L.EIC009)

José Proença (FCUP) & João Bispo (FEUP) – *slides by Eduardo R. B.*

Outline

- Introduction to arrays
- C-strings
- Range-based for loops
- Multi-dimensional arrays
- The (serious) issue of buffer overflows in more detail

Introduction to arrays

Introduction to arrays

What is an array?

An **array** is a sequence of contiguous elements in memory with a fixed size.

The following is a declaration of an `int` array variable named `a` and with length `10`.

```
int a[10];
```

Declaration and initialisation

```
// No initialisation.
// Array may hold arbitrary values.
int a[10];

// All elements initialized.
int b[10] { 10, 20, 30, 40, 50,
            60, 70, 80, 90, 100};

// Same as previous, but with implicit
// array size (of 10).
int c[] { 10, 20, 30, 40, 50,
          60, 70, 80, 90, 100};
```

```
// Only a few elements initialized.
// Remaining elements are
// initialised to 0.
int d[10] { 10, 20, 30 };

// All elements initialised to 0
int e[10] { 0 };
```

Array access

An array with length `n` may be accessed (read or written) using operator `[]` and array positions (also called indexes) `i` from `0` to `n-1`: `a[i]` stands for the element of `a` at position `i`.

Example:

```
int a[3] { 1, 2, 3 };
a[2] += a[1] + a[0]; // a[2] is assigned value 6
```

Memory representation

An array is stored in memory contiguously.

```cpp
int a[5] { 12, -1, 23, 0, 0 };
std::cout << "sizeof(a) " << sizeof(a) << '\n';
```

```
sizeof(a) = 20
```

	0	1	2	3	4
a	12	-1	23	0	0

If an `int` value takes 4 bytes in memory, then `a` above requires $20 = 5 \times 4$ bytes.

Invalid array access

The use of invalid (out-of-bounds) indexes **is NOT** verified at compile time or execution time, and the behavior of a program is **undefined** in these cases that we will refer to as **buffer overflows**. We discuss the matter in more detail later.

```
int a[4];
a[4] = 1;      // buffer overflow! (4 > 3)
a[3] = a[-1]; // buffer overflow! (-1 < 0)
```

Arrays and functions

An array is always **passed by reference** to a function.

```
void fill_with_zeros(int a[], int n) {
  for (int i = 0; i < n; i++)
    a[i] = 0;
}
int main(void) {
  int a[10];
  fill_with_zeros(a, 10);
  // all positions will be filled with 0
  ...
}
```

Arrays and functions (cont.)

There is no language mechanism for obtaining the length of an array supplied as argument to a call (e.g., the `length` attribute in some languages), so we often need to supply the array length as another explicit argument too, such as the parameter `n` in this code:

```c
void fill_with_zeros(int a[], int n) {
  for (int i = 0; i < n; i++)
    a[i] = 0;
}
```

Note: `sizeof(a)`, if used in the function above, would just return the size of parameter variable `a`. Since the variable in practice only holds the array memory address its size is 8 bytes (in a 64-bit architecture).

Arrays and functions (cont.)

We have the option of stating the length of an array parameter explicitly. The resulting code is not flexible however to handle arrays of any size :(

```
void fill_with_zeros(int a[10]) {
  for (int i = 0; i < 10; i++)
    a[i] = 0;
}
```

Arrays and functions (cont.)

The use of **the `const` modifier prevents modifications to an array**. In the following example, `a` can not be changed by function `reverse`:

```
// Copy n elements from a to b in
// reverse order.
void reverse(const int a[],
             int b[], int n) {
   for (int i = 0; i < n; i++) {
     b[n - i - 1] = a[i];
   }
}
```

If we make the mistake of trying to modify a `const` array like `a`, e.g., if the above loop body is changed to

```
  a[i] = b[n - i -1];
```

then a compilation error results

```
    error: read-only variable
    is not assignable
```

Arrays and functions (cont.)

```
// Copy n elements from a to b in
// reverse order.
void reverse(const int a[],
             int b[], int n) {
   for (int i = 0; i < n; i++) {
     b[n - i - 1] = a[i];
   }
}
```

Arrays and functions (cont.)

```
// Copy n elements from a to b in
// reverse order.
void reverse(const int a[],
             int b[], int n) {
   for (int i = 0; i < n; i++) {
     b[n - i - 1] = a[i];
   }
}
```

```
...
int x[5] {5,4,3,2,1};
reverse(x, x, 5);
// x now has contents:
//  [5, 4, 3, 4, 5] !
```

If we call `reverse(a,b,n)` we should guarantee that a e b are really different arrays! The call-by-reference semantics will mean that when calling `reverse(x,x,n)` for an array `x` then parameters `a` and `b` will both refer to `x` and, as a result of that, `x` will not be reversed correctly.

Arrays and functions (cont.)

The following `reverse` function overload reverses the array “inline”.

(Why should the loop stop at `n/2` instead of `n`?)

```
void reverse(int a[], int n) {
  for (int i = 0; i < n / 2; i++) {
    int tmp = a[i];
    a[i] = a[n - i - 1];
    a[n - i - 1] = tmp;
  }
}
```

Arrays and struct types

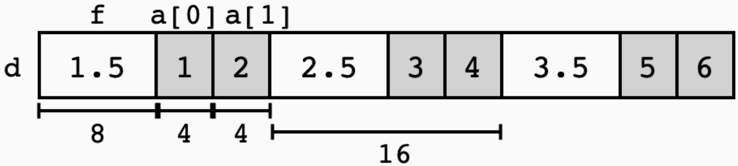
Arrays and struct types may be combined, e.g.,

```
struct data {
  double f;
  int a[2];
};
```

```
data d[3] {
  { 1.5, {1, 2} },
  { 2.5, {3, 4} },
  { 3.5, {5, 6} }
};
```

```
std::cout << "sizeof(d) = " << sizeof(d) << '\n';
```

sizeof(d) = 48



C-strings

Strings ...

A **string** is a sequence of characters.

How do strings work in C and C++ ?

- **C-strings**: C treats strings as a special case of an array - strings are array of `char` type terminated by value 0, also denoted `'\0'`;
- The C++ library supports C-strings and also provides class type `std::string` to represent strings.

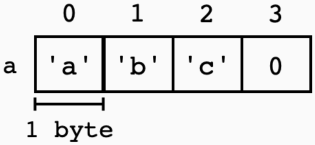
We will cover C-strings now, as they are just special types of arrays. Later, we will cover the use of `std::string`.

C-strings

A C-string `s` is an array of type `char` in which value `0`, also denoted by `'\0'`, indicates the end of the string.

String constants are normally used as in the second declaration below (`b[3] = '\0'` implicitly):

```
// Equivalent strings
char a[4] { 'a', 'b', 'c', '\0' };
char b[4] = "abc";
```

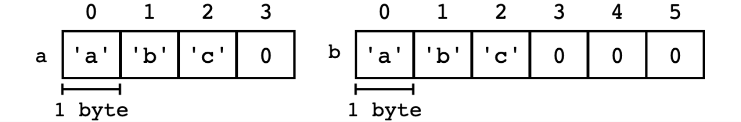


Length of a C-string

The **length** of a C-string `s` with array length `n` is `l < n` such that `a[0], ..., a[l-1] != '\0'` and `a[l] = '\0'`.

```
// string length: 3, array length: 4
char a[4] = "abc";
// Equivalent string with unused space (2 positions)
// string length: 3, array length: 6
char b[6] = "abc";
```

For a string of length `l`, we must have have `n` $\geq$ `l + 1`. An extra position at least is required for the `'\0'` terminator.



Functions over C-strings

Function to compute the length of a C-string (for illustration purposes; you should use `std::strlen` instead):

```
int length(const char str[]) {
  int l = 0;
  while (str[l] != '\0')
    l++;
  return l;
}
```

Functions over C-strings

Function to compute the length of a C-string (for illustration purposes; you should use `std::strlen` instead):

```
int length(const char str[]) {
  int l = 0;
  while (str[l] != '\0')
    l++;
  return l;
}
```

```
char x[10] = "ef";
int v =  length("abc")// 3
       + length("")   // 0
       + length("d")  // 1
       + length(x);   // 2 (not 10!)
```

Functions over C-strings (cont.)

A function that copies string contents, src to dst (again for illustration purposes, you should use std::strcpy):

```
void copy(char dst[],
          const char src[]) {
  int i = 0;
  while (src[i] != '\0') {
    dst[i] = src[i];
    i++;
  }
  dst[i] = '\0';
}
```

Functions over C-strings (cont.)

A function that copies string contents, `src` to `dst` (again for illustration purposes, you should use `std::strcpy`):

```
void copy(char dst[],
          const char src[]) {
  int i = 0;
  while (src[i] != '\0') {
    dst[i] = src[i];
    i++;
  }
  dst[i] = '\0';
}
```

```
char src[] = "LEIC";
char dst[4 + 1]; // must have
                 // enough space
copy(dst, src);
```

Functions over C-strings (cont.)

A function that concatenates strings (similar in spirit to std::strcat)

```
void concat(char dst[],
            const char src[]) {
  int i = length(dst);
  int j = 0;
  while (src[j] != '\0') {
    dst[i] = src[j];
    i++; j++;
  }
  dst[i] = '\0';
}
```

Functions over C-strings (cont.)

A function that concatenates strings (similar in spirit to `std::strcat`)

```
void concat(char dst[],
            const char src[]) {
  int i = length(dst);
  int j = 0;
  while (src[j] != '\0') {
    dst[i] = src[j];
    i++; j++;
  }
  dst[i] = '\0';
}
```

```
char dst[7 + 1] = "LEIC";
char src[3 + 1] = "@UP";
concat(dst, src);
// dst now holds "LEIC@UP"
```

Range-based for loops

Range-based for loop

A range-based `for` loop (also known as “for-each” loop) can be used to iterate elements of an array `a` of type `T` and size `N`.

```
T a[N];
...
for (T v : a)
  body
```

`v` in the loop body is assigned to `a[i]`, for `i = 0,1, ..., N - 1`. The loop is equivalent to

```
T a[N];
. . .
for (int i = 0; i < N; i++) {
  T v = a[i]; // get value of a[i]
  body
}
```

Range-based for loop - example

Example:

```
int a[10];
. . .
// Compute the sum of all
// array elements
int sum = 0;
for (int v : a) sum += v;
```

is equivalent to

```
int a[10];
. . .
// Compute the sum of all
// array elements
int sum = 0;
for (int i = 0; i < 10; i++) {
  int v = a[i]; // get value
                // of a[i]
  sum += v;
}
```

Range-based for loop - restrictions

Invalid example - **the array length must be known in the loop’s scope**:

```
int f(int a[]) { // array length is not known!
  int sum = 0;
  for (int v : a) sum += v; // INVALID!
  return sum;
}
```

```
error: cannot build range expression with
array function parameter 'a'
```

Range-based for loop - variant

Alternatively, we can iterate **references** to array positions, and use the references to modify the array:

```
T a[N];
. . .
for (T& v : a)
  body
```

is equivalent to

```
T a[N];
. . .
for (int i = 0; i < N; i++) {
  T& v = a[i]; // get reference to a[i]
  body
```

Range-based for loop - variant (cont.)

Example:

```
int a[10];
for (int& v : a) v = 1;
```

is equivalent to

```
for (int i = 0; i < 10; i++) {
  T& v = a[i]; // get reference to a[i]
  v = 1;       // changes a[i]
}
```

Range-based for loop - another example

```
int a[4] { 1, 2, 3, 4};

for (int& v : a) v *= v;
// a  now stores {1,4,9,16}

int n = 0;
for (int v: a) n += v; // --> n = 30
```

Multi-dimensional arrays

Multi-dimensional arrays

We can declare arrays with more than one dimension, e.g.:

```
int a[3][4];
int b[3][5] = {
  { 11, 12, 13, 14, 15},
  { 21, 22, 23, 24, 25},
  { 31, 32, 33, 34, 35}
};
```

The array indexing scheme generalises for multiple dimensions:

```
a[1][3] = b[0][4] + b[1][2] + b[2][0]; // 15 + 23 + 31
```

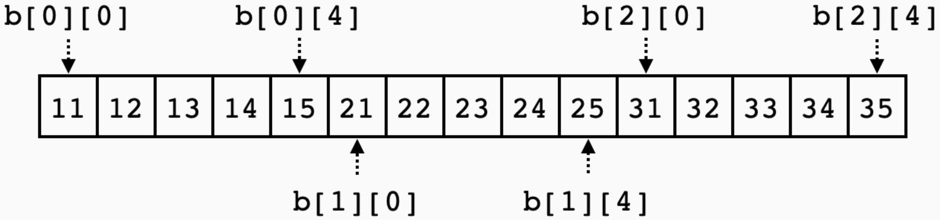
Multi-dimensional arrays (cont.)

```
int b[3][5] = { { 11, 12, 13, 14, 15},
                { 21, 22, 23, 24, 25},
                { 31, 32, 33, 34, 35}
};
```

```
std::cout << "sizeof(b) = "
          << sizeof(b)
          << '\n';
```

```
sizeof(b) = 60
```

A multi-dimensional array is stored in contiguous memory just like a 1-d array; b above occupies $60 = 3 \times 5 \times 4$ bytes in memory.



b[0][0] b[0][4] b[2][0] b[2][4]

11	12	13	14	15	21	22	23	24	25	31	32	33	34	35

b[1][0] b[1][4]

Multi-dimensional arrays (cont.)

In order to have a multi-dimensional array as function parameter, we must specify the size of all array dimensions except the first one.

Otherwise, the compiler cannot infer the memory layout for the array.

The following definition of `fill_with_zeros` is valid:

```
void fill_with_zeros(int m[][100], int rows) {
  for (int r = 0; r < rows; r++)
    for (int c = 0; c < 100; c++)
      m[r][c] = 0;
}
```

Multi-dimensional arrays (cont.)

In contrast, the following definition of `fill_with_zeros` is invalid:

```
void fill_with_zeros(int m[][], int rows, int cols) {
  for (int r = 0; r < rows; r++)
    for (int c = 0; c < cols; c++)
      m[r][c] = 0;
}
```

```
error: array type has incomplete element type ‘int[]’

note: declaration of ‘m’ as multidimensional array
must have bounds for all dimensions except the first
```

This lack of flexibility makes the use of multi-dimensional arrays relatively rare. Other mechanisms are employed for representing general-purpose matrixes, 3d-arrays, etc.

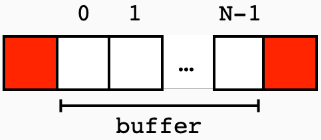
Buffer overflows in more detail

Buffer overflows

A **buffer overflow** is a program anomaly that occurs when a memory buffer is accessed (read or written) beyond its boundary in adjacent memory positions.

Some use the term buffer overflow only in the case of a write access, and buffer over-read for an overflowing read access. We use the same term for both cases.

```
int a[4];
a[4] = 1;      // buffer overflow! (4 > 3)
a[3] = a[-1]; // buffer overflow! (-1 < 0)
```



Why do they happen?

Buffer overflows are a particularity of C (and inherently also of C++ or Objective-C) programs:

- Array accesses are not checked at compile (it is undecidable) time or execution time (no bounds check).
- C/C++ allows pointer variables (to be discussed in future classes) that can read/write memory arbitrarily.

Buffer overflows and other anomalies in the use of memory imply that C/C++ programs are not guaranteed to be **memory-safe**.

Buffer overflows - what can happen as a result?

The program semantics are **undefined** in a buffer overflow. *Anything can happen* - ranging from buggy, non-deterministic behavior to whole-system vulnerabilities:

- Nothing in functional terms. Any side effect of the buffer overflow does not affect the functional behavior of the program.
- The program behavior becomes buggy with a noticeable deviation of correct behavior.
- The program crashes due to an invalid memory address (`segmentation fault`).
- Depending on numerous aspects such as compiler and compiler options used, host architecture, inputs and runtime conditions, ... sometimes the program runs normally, sometimes it is buggy, sometimes it crashes.
- ...
- Fed with malicious data, the program may give access to a system shell and allow an attacker to compromise security of the entire system!

Buffer overflows - example

```cpp
// bo.cpp
#include <iostream>

int main() {
  char s[5]="LEIC";
  char inp[4] { 0 };
  std::cout << "inp? "; cin >> inp;
  std::cout << inp << '@' << s << '\n';
  return 0;
}
```

Let us compile the program with no special options apart from −g (to generate debug symbols):

```
g++ -g bo.cpp -o bo
```

Buffer overflows - example (cont.)

Safe execution with no buffer overflow - string `"C++"` fits in `inp`:

```
inp? C++
C++@LEIC
```

Possible execution with buffer overflow - `"C/C++"` does not fit in `inp`, implying that `cin >> inp` writes data beyond the end of `inp` (note that the behavior may be compiler-dependent).

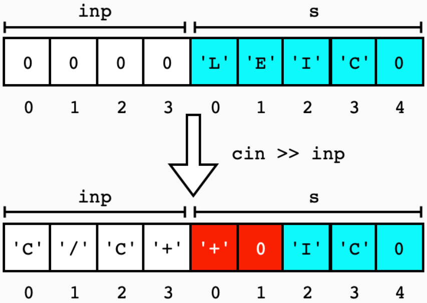
```
inp? C/C++
C/C++@+
```

What happened?

Buffer overflows - example (cont.)

```
char s[5]="LEIC";
char inp[4] { 0 };
... cin >> inp;
```

- `inp` and `s` are allocated on the program stack, contiguously, and `s` appears *after* `inp`.
- `cin >> inp` writes beyond the end of `inp` (`'+'` and the null terminator) affecting the contents of `s`.



inp				s				
0	0	0	0	'L'	'E'	'I'	'C'	0
0	1	2	3	0	1	2	3	4

cin >> inp

inp				s				
'C'	'/'	'C'	'+'	'+'	0	'I'	'C'	0
0	1	2	3	0	1	2	3	4

Buffer overflow - another example

```
struct mydata {
  int a[4];
  int n;
};
```

```
int main() {
  mydata d = { { 1, 2, 3, 4 }, 0 };
  for (int i = 0; i <= 4; i++) d.n += d.a[i];
  std::cout << "d.n = " << d.n << '\n';
  return 0;
}
```

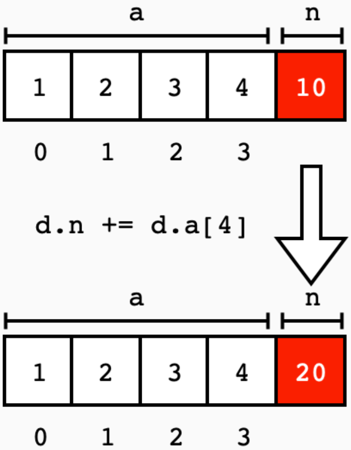
```
d.n = 20
```

In this case the overflow happens when reading `d.a[4]` for `i=4`, which corresponds to the location of `n` ...

This means `n` is summed with itself (10+10) when `i=4`.

Buffer overflow - another example (cont.)

```
struct mydata {
  int a[4];
  int n;
};
...
mydata d {
  { 1, 2, 3, 4 },
  0
};
for (int i = 0; i <= 4; i++)
  d.n += d.a[i];
```



Using runtime sanitizers

AddressSanitizer is a runtime memory error detector for C/C++ programs.

It is enabled using the `-fsanitize=address` compiler option in GCC:

```
g++ -fsanitize=address -g bo.cpp -o bo
```

The resulting binary is instrumented to detect memory errors, including several types of buffer overflows.

The use of AddressSanitizer became quite popular (e.g., among developers of Chrome, Firefox, and the Linux kernel), given it is quite effective at finding memory errors with a speed/memory overhead significantly lower than other alternatives (e.g., valgrind).

Using runtime sanitizers (cont.)

First example:

```
char inp[4];
...
cin >> inp;
```

```
  ... AddressSanitizer: stack-buffer-overflow
  on address 0x7ff7bb095054 ...
  . . .
  #0 0x104e6d93d in main bo.cpp:5
```

Program crashes in a deterministic and controlled manner. We have a precise indication of the source code location causing the buffer overflow error.

Using runtime sanitizers (cont.)

Second example:

```
mydata d { { 1, 2, 3, 4 }, 0 };
for (int i = 0; i <= 4; i++) d.n += d.a[i];
```

In the second example AddressSanitizer cannot detect the buffer overflow, but the Undefined Behavior Sanitizer can. We just need to compile programs using `-fsanitize=undefined` as well:

```
$ g++ -g -fsanitize=undefined -fsanitize=address \
  -g bo2.cpp -o bo2
$ ./bo2

  bo2.cpp:13:44: runtime error: index 4
  out of bounds for type 'int [4]'
```

Buffer overflows and security

Extra-curricular (for this course at least) but "general culture" you should be aware of - you may start by reading the "Buffer overflows" Wikipedia entry:

- Buffer overflows are the source of many significant software security vulnerabilities.
- Attacks are based on feeding input, "shellcode" in particular, that rely on buffer overflow vulnerabilities to "carefully" overwrite sensitive data (such as the program execution stack)
- Although compiler and OS mechanisms offer different types of protections for buffer overflows (e.g. stack canaries), **the C/C++ programmer has a key role in avoiding buffer overflows** through:
  - robust validation of program input
  - use of safe software libraries - for example `strcpy` and many other string-related functions are inherently unsafe, see for instance "Common vulnerabilities guide for C programmers - CERN Computer Security""

Mini-test MT1

18 Mar, 14:30-16:30 & 17:00-19:00

== Study content until here ==