Hacking in C

Pointers

Radboud University, Nijmegen, The Netherlands

Spring 2019

Allocation of multiple variables

Consider the program

```
main(){
  char x;
  int i;
  short s;
  char y;
  ....
}
```

What will the layout of this data in memory be?

Assuming 4-byte ints, 2-byte shorts, and little endian architecture

Printing addresses where data is located

We can use & to see where data is located

```
char x; int i; short s; char y;

printf("x is allocated at %p \n", &x);
printf("i is allocated at %p \n", &i);
printf("s is allocated at %p \n", &s);
printf("y is allocated at %p \n", &y);
    // Here %p is used to print pointer values
```

Compiling with or without -O2 will reveal different alignment strategies

Data alignment

Memory as a sequence of bytes

...		**x**	$\mathbf{i}_0$	$\mathbf{i}_1$	$\mathbf{i}_2$	$\mathbf{i}_3$	$\mathbf{s}_0$	$\mathbf{s}_1$	**y**			...

But on a 32-bit machine, the memory is a sequence of 4-byte words

x	$\mathbf{i}_0$	$\mathbf{i}_1$	$\mathbf{i}_2$
$\mathbf{i}_3$	$\mathbf{s}_0$	$\mathbf{s}_1$	**y**
			...

Now the data elements are not nicely aligned with the words,
which will make execution slow, since CPU instructions act on words.

Data alignment

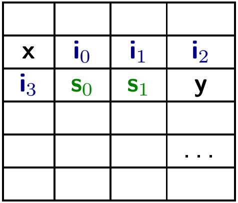
Different allocations, with better/worse alignment

x	$\mathbf{i}_0$	$\mathbf{i}_1$	$\mathbf{i}_2$
$\mathbf{i}_3$	$\mathbf{s}_0$	$\mathbf{s}_1$	**y**
			...

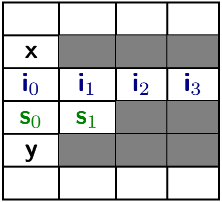
Lousy alignment, but
uses minimal memory

Data alignment

Different allocations, with better/worse alignment



Lousy alignment, but uses minimal memory



Optimal alignment, but wastes memory

Data alignment

Different allocations, with better/worse alignment

$\mathbf{x}$	$\mathbf{i}_0$	$\mathbf{i}_1$	$\mathbf{i}_2$
$\mathbf{i}_3$	$\mathbf{s}_0$	$\mathbf{s}_1$	$\mathbf{y}$
			...

Lousy alignment, but uses minimal memory

$\mathbf{x}$			
$\mathbf{i}_0$	$\mathbf{i}_1$	$\mathbf{i}_2$	$\mathbf{i}_3$
$\mathbf{s}_0$	$\mathbf{s}_1$		
$\mathbf{y}$			

Optimal alignment, but wastes memory

$\mathbf{s}_0$	$\mathbf{s}_1$	$\mathbf{x}$	$\mathbf{y}$
$\mathbf{i}_0$	$\mathbf{i}_1$	$\mathbf{i}_2$	$\mathbf{i}_3$
			...

Possible compromise

Data alignment

Compilers may introduce padding or change the order of data in memory to improve alignment.

There are trade-offs here between speed and memory usage.

Data alignment

Compilers may introduce padding or change the order of data in memory to improve alignment.

There are trade-offs here between speed and memory usage.

Most C compilers can provide many optional optimizations. E.g., use

```
man gcc
```

to check out the many optimization options of gcc.

Arrays

Arrays

An array contains a collection of data elements with the same type.
The size is constant.

```
int test_array[10];
int a[] = {30,20};
test_array[0] = a[1];

printf("oops %d \n", a[2]); //will compile & run
```

Arrays

An array contains a collection of data elements with the same type.
The size is constant.

```
int test_array[10];
int a[] = {30,20};
test_array[0] = a[1];

printf("oops %d \n", a[2]); //will compile & run
```

Array bounds are **not** checked.
Anything may happen when accessing outside array bounds. The program may crash, usually with a segmentation fault (segfault).

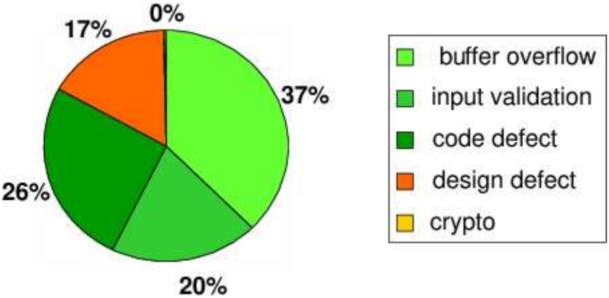
Array bounds checking

The historic decision **not** to check array bounds is responsible for in the order of 50% of all the security vulnerabilities in software, in the form of so-called buffer overflow attacks.

Other languages took a different (more sensible?) choice here. E.g. ALGOL60, defined in 1960, already included array bound checks.

Typical software security vulnerabilities

Security bugs found in Microsoft's first security bug fix month (2002)



Here *buffer overflows* are platform-specific.
Some of the *code defects* and *input validation* problems might also be.
Crypto problems are much more rare, but can be of very high impact.

Array bounds checking

Tony Hoare in Turing Award
speech on the design principles of ALGOL 60

"The first principle was *security*: . . . A consequence of this principle is that every subscript was checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency. Unanimously, they urged us not to – they knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law."

[C.A.R.Hoare, The Emperor's Old Clothes, Communications of the ACM, 1980]

Overrunning arrays

Consider the program

```
int y = 7;
char a[2];
int x = 6;
printf("oops %d \n", a[2]);
```

What would you expect this program to print?

Overrunning arrays

Consider the program

```
int y = 7;
char a[2];
int x = 6;
printf("oops %d \n", a[2]);
```

What would you expect this program to print?

If the compiler allocates `x` directly after `a`, then (on a little-endian machine) it will print 6.
There are no guarantees! The program could simply crash, or return any other number, re-format the hard drive, explode, . . .

Overrunning arrays

Consider the program

```
int y = 7;
char a[2];
int x = 6;
printf("oops %d \n", a[2]);
```

What would you expect this program to print?

If the compiler allocates `x` directly after `a`, then (on a little-endian machine) it will print 6.
There are no guarantees! The program could simply crash, or return any other number, re-format the hard drive, explode, . . .

By overrunning an array we can try to reverse-engineer the memory layout

Arrays and alignment

The memory space allocated for an array is guaranteed to be contiguous, i.e. `a[1]` is allocated right after `a[0]`.

Arrays and alignment

The memory space allocated for an array is guaranteed to be contiguous, i.e. `a[1]` is allocated right after `a[0]`.

For good alignment, a compiler could again add padding at the end of arrays.

Arrays and alignment

The memory space allocated for an array is guaranteed to be contiguous, i.e. `a[1]` is allocated right after `a[0]`.

For good alignment, a compiler could again add padding at the end of arrays.

E.g. a compiler might allocate 16 bytes rather than 15 bytes for

```
char text[15];
```

Arrays are passed by reference

Arrays are always passed by reference.

For example, given the function

```
void increase_elt(int x[]) {
  x[1] = x[1]+23;
}
```

What is the value of a[1] after executing the following code?

```
int a[2] = {1, 2};
increase_elt(a);
```

Arrays are passed by reference

Arrays are always passed by reference.

For example, given the function

```
void increase_elt(int x[]) {
  x[1] = x[1]+23;
}
```

What is the value of `a[1]` after executing the following code?

```
int a[2] = {1, 2};
increase_elt(a);
```

25

Recall call by reference from Imperative Programming

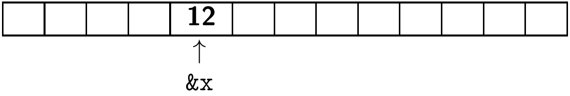
Pointers

Retrieving addresses of *pointers* using &

We can find out *where* some data is allocated using the & operation.
If

```
int x = 12;
```

then `&x` is the memory address where the value of `x` is stored,
aka a pointer to `x`.

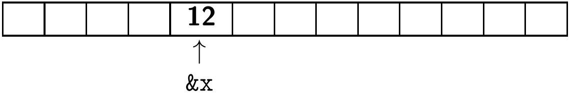


Retrieving addresses of *pointers* using &

We can find out *where* some data is allocated using the & operation.
If

```
int x = 12;
```

then `&x` is the memory address where the value of `x` is stored, aka a pointer to `x`.



It depends on the underlying architecture how many bytes are needed to represent addresses: 4 on 32-bit machines, 8 on a 64-bit machine.

Declaring pointers

Pointers are typed:
the compiler keeps track of what data type a pointer points to

```
int *p;   // p is a pointer that points to an int
float *f; // f is a pointer that points to a float
```

Creating and dereferencing pointers

Suppose

```
int y, z; int *p; // i.e. p points to an int
```

How can we create a pointer to some variable?

Creating and dereferencing pointers

Suppose

```
int y, z; int *p; // i.e. p points to an int
```

How can we create a pointer to some variable? Using **&**

```
   y = 7
p = &y; // assign the address of y to p
```

How can we get the value that a pointer points to?

Creating and dereferencing pointers

Suppose

```
int y, z; int *p; // i.e. p points to an int
```

How can we create a pointer to some variable? Using **&**

```
   y = 7
p = &y; // assign the address of y to p
```

How can we get the value that a pointer points to? Using *

```
   y = 7
p = &y; // pointer p now points to y
z = *p; // give z the value of what p points to
```

Creating and dereferencing pointers

Suppose

```
int y, z; int *p; // i.e. p points to an int
```

How can we create a pointer to some variable? Using **&**

```
   y = 7
p = &y; // assign the address of y to p
```

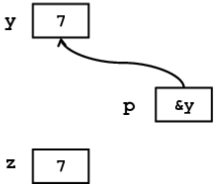
How can we get the value that a pointer points to? Using *

```
   y = 7
p = &y; // pointer p now points to y
z = *p; // give z the value of what p points to
```

Looking up what a pointer points to, with *, is called dereferencing.

Confused? draw pictures!

```
int y = 7;
int *p = &y; // pointer p now points to cell y
int z = *p;  // give z the value of what p points to
```



Pointer quiz

What is the value of `y`?

```
int y = 2;
int x = y;
y++;
x++;
```

Pointer quiz

What is the value of `y`?

```
int y = 2;
int x = y;
y++;
x++;
```

3

Pointer quiz

What is the value of `y`?

```
int y = 2;
int x = y;
y++;
x++;
```

3

What is the value of `y`?

```
int y = 2;
int *x = &y;
y++;
(*x)++;
```

Pointer quiz

What is the value of `y`?

```
int y = 2;
int x = y;
y++;
x++;
```

3

What is the value of `y`?

```
int y = 2;
int *x = &y;
y++;
(*x)++;
```

4

Note that $*$ is used for 3 different purposes, with 3 different meanings

Note that * is used for 3 different purposes, with 3 different meanings

1. In declarations, to declare pointer types

```
int *p;  // p is a pointer to an int
         // i.e.  *p is an int
```

Note that * is used for 3 different purposes, with 3 different meanings

1. In declarations, to declare pointer types

```
int *p;  // p is a pointer to an int
         // i.e.  *p is an int
```

2. As a prefix operator on pointers

```
int z = *p;
```

Note that * is used for 3 different purposes, with 3 different meanings

1. In declarations, to declare pointer types

```
int *p;  // p is a pointer to an int
         // i.e.  *p is an int
```

2. As a prefix operator on pointers

```
int z = *p;
```

3. Multiplication of numeric values

Note that * is used for 3 different purposes, with 3 different meanings

1. In declarations, to declare pointer types

```
int *p;  // p is a pointer to an int
         // i.e.  *p is an int
```

2. As a prefix operator on pointers

```
int z = *p;
```

3. Multiplication of numeric values

   Some legal C code can get confusing, e.g.

```
z = 3 * *p
```

Style debate: `int* p` or `int *p`?

What can be confusing in

```
int *p = &y;
```

is that this is an assignment to `p`, not `*p`

Style debate: `int* p` or `int *p`?

What can be confusing in

```
int *p = &y;
```

is that this is an assignment to `p`, not `*p`

Some people prefer to write

```
int* p = &y;
```

Style debate: `int* p` or `int *p`?

What can be confusing in

```
int *p = &y;
```

is that this is an assignment to `p`, not `*p`

Some people prefer to write

```
int* p = &y;
```

but C purists will argue this is C++ style.

Style debate: `int* p` or `int *p`?

What can be confusing in

```
int *p = &y;
```

is that this is an assignment to `p`, not `*p`

Some people prefer to write

```
int* p = &y;
```

but C purists will argue this is C++ style.

Downside of writing `int*`

```
int* x, y, z;
```

declares `x` as a pointer to an `int` and `y` and `z` as `int`...

Still not confused?

```
x = 3;
p1 = &x;
p2 = &p1;
z = **p2 + 1;
```

What will the value of z be?

Still not confused?

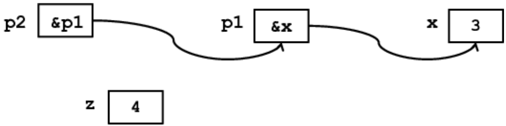
```
x = 3;
p1 = &x;
p2 = &p1;
z = **p2 + 1;
```

What will the value of `z` be?

What should the types of `p1` and `p2` be?

Still not confused? pointers to pointers

```
int x = 3;
int *p1 = &x; // p1 points to an int
int **p2 = &p1; // p2 points to a pointer to an int
int z = **p2 + 1;
```



Pointer test (Hint: example exam question)

```
int y = 2;
int z = 3;
int* p = &y;
int* q = &z;
(*q)++;
*p = *p + *q;
q = q + 1;
printf("y is %d\n", y);
```

What is the value of `y` at the end?

Pointer test (Hint: example exam question)

```c
int y = 2;
int z = 3;
int* p = &y;
int* q = &z;
(*q)++;
*p = *p + *q;
q = q + 1;
printf("y is %d\n", y);
```

What is the value of `y` at the end?
6

Pointer test (Hint: example exam question)

```
int y = 2;
int z = 3;
int* p = &y;
int* q = &z;
(*q)++;
*p = *p + *q;
q = q + 1;
printf("y is %d\n", y);
```

What is the value of `y` at the end?
6
What is the value of `*p` at the end?

Pointer test (Hint: example exam question)

```
int y = 2;
int z = 3;
int* p = &y;
int* q = &z;
(*q)++;
*p = *p + *q;
q = q + 1;
printf("y is %d\n", y);
```

What is the value of `y` at the end?
6
What is the value of `*p` at the end?
6

Pointer test (Hint: example exam question)

```
int y = 2;
int z = 3;
int* p = &y;
int* q = &z;
(*q)++;
*p = *p + *q;
q = q + 1;
printf("y is %d\n", y);
```

What is the value of `y` at the end?
6
What is the value of `*p` at the end?
6
What is the value of `*q` at the end?

Pointer test (Hint: example exam question)

```
int y = 2;
int z = 3;
int* p = &y;
int* q = &z;
(*q)++;
*p = *p + *q;
q = q + 1;
printf("y is %d\n", y);
```

What is the value of `y` at the end?
6
What is the value of `*p` at the end?
6
What is the value of `*q` at the end?
We don't know! `q` points to some memory cell after `z` in the memory

Pointer arithmetic

You can use + and − with pointers.
The semantics depends on the *type of the pointer*:

- adding 1 to a pointer will go to the "next" location, given the size and the data type that it points to.

Pointer arithmetic

You can use + and - with pointers.
The semantics depends on the *type of the pointer*:

adding 1 to a pointer will go to the "next" location, given the size and the data type that it points to.

For example, if

```
int *ptr;    char *str;
```

then
`ptr + 2` means "Add `2 * sizeof(int)` to the address in `ptr`"
`str + 2` means "Add 2 to the address in `str`"
(because `sizeof(char)` is 1)

Using pointers as arrays

The way pointer arithmetic works means that a pointer to the head of an array behaves like an array.

Suppose

```
int a[10] = {1,2,3,4,5,6,7,8,9,19};
int *p = (int *) &a; // the address of the head of a
                     // treated as pointer to an int
```

Using pointers as arrays

The way pointer arithmetic works means that a pointer to the head of an array behaves like an array.

Suppose

```
int a[10] = {1,2,3,4,5,6,7,8,9,19};
int *p = (int *) &a; // the address of the head of a
                     // treated as pointer to an int
```

Now

```
p + 3
```

points to

```
a[3]
```

So we use addition to pointer `p` to move through the array

Pointer arithmetic for strings

What is the output of

```
char *msg = "hello world";
char *t = msg + 6;
printf("t points to the string %s.", t);
```

Pointer arithmetic for strings

What is the output of

```
char *msg = "hello world";
char *t = msg + 6;
printf("t points to the string %s.", t);
```

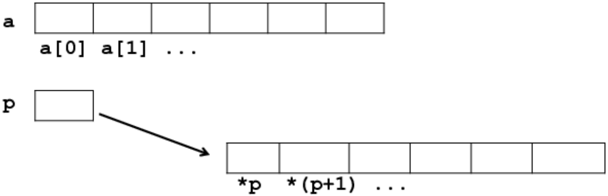
This will print

```
t points to the string world.
```

Arrays vs pointers

Arrays and pointers behave similarly, but are very different in memory
Consider

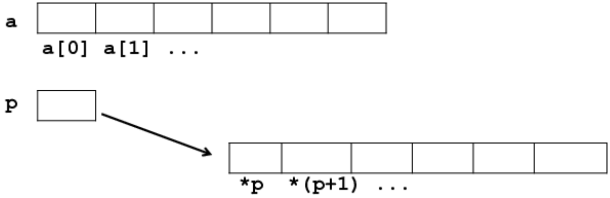
```
int a[]; int *p
```



Arrays vs pointers

Arrays and pointers behave similarly, but are very different in memory
Consider

```
int a[]; int *p
```



A difference: `a` will always refer to the same array,
whereas `p` can point to different arrays over time

Using pointers as arrays

Suppose

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

Then

```
int sum = 0;
for (int i = 0; i != 10; i++) {
  sum = sum + a[i];
}
```

can also be implemented using pointer arithmetic

Using pointers as arrays

Suppose

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

Then

```
int sum = 0;
for (int i = 0; i != 10; i++) {
  sum = sum + a[i];
}
```

can also be implemented using pointer arithmetic

```
int sum = 0;
for (int *p = (int *)&a; p != &(a[10]); p++) {
  sum = sum + *p;
}
```

This cast is needed because a is an integer array, so `&a` is a pointer to `int[]`, not pointer to an `int`.
An alternative would be to write `*p = &(a[0])`

Instead of `p != &(a[10])` we could also write `p != ((int *)&a)+10`

Using pointers as arrays

Suppose

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

Then

```
int sum = 0;
for (int i = 0; i != 10; i++) {
  sum = sum + a[i];
}
```

can also be implemented using pointer arithmetic

```
int sum = 0;
for (int *p = (int *)&a; p != &(a[10]); p++) {
  sum = sum + *p;
}
```

This cast is needed because a is an integer array, so **&a** is a pointer to `int[]`, not pointer to an `int`. An alternative would be to write **`*p = &(a[0])`**

Instead of **`p != &(a[10])`** we could also write **`p != ((int *)&a)+10`**

But nobody in their right mind would ☺

A problem with pointers: . . .

```
int i; int j; int *x;
...
// lots of code omitted
i = 5;
j++
// what is the value of i here?
(*x)++;
// what is the value of i here?
```

A problem with pointers: ...

```
int i; int j; int *x;
...
// lots of code omitted
i = 5;
j++
// what is the value of i here?
(*x)++;
// what is the value of i here?
```

5

5 or 6, depending on whether `*x` points to `i`

Two pointers are called aliases if they point to the same location

```
int i = 5;
int *x = &i;
int *y = &i;
// x and y are aliases now
(*x)++;
// now i and *y have also changed to 6
```

Two pointers are called aliases if they point to the same location

```
int i = 5;
int *x = &i;
int *y = &i;
// x and y are aliases now
(*x)++;
// now i and *y have also changed to 6
```

Keeping track of pointers, in the presence of potential aliasing, can be really confusing, and really hard to debug. . .

Recap – so far

We have seen pointers, e.g. of type `char *p`
with the operations `*` and `&`
These are tricky to understand, unless you draw pictures

Recap – so far

We have seen pointers, e.g. of type `char *p`
with the operations `*` and `&`
These are tricky to understand, unless you draw pictures

We can have aliasing, where two names, say `*p` and `c`, can refer to the same variable (location in memory)

Recap – so far

We have seen pointers, e.g. of type `char *p`
with the operations `*` and `&`
These are tricky to understand, unless you draw pictures

We can have aliasing, where two names, say `*p` and `c`, can refer to the same variable (location in memory)

We can use pointer arithmetic, and e.g. write `*(p+1)`, and use this to access arrays

Recap – so far

We have seen pointers, e.g. of type `char *p`
with the operations `*` and `&`
These are tricky to understand, unless you draw pictures

We can have aliasing, where two names, say `*p` and `c`, can refer to the same variable (location in memory)

We can use pointer arithmetic, and e.g. write `*(p+1)`, and use this to access arrays

Confusingly, the meaning of addition for pointers depends on their type, as +1 for pointers of type `int *` really means `+sizeof(int)`

The potential of pointers: inspecting raw memory

To inspect a piece of raw memory, we can cast it to a

```
unsigned char *
```

and then inspect the bytes

```
   float f = 3.14;
unsigned char *p = (unsigned char *) &f;
printf("The representation of float %f is", f);
for (int i = 0; i < sizeof(float); i++, p++);) {
printf("%d", *p);
}
printf("\n");
```

Turning pointers into numbers

`intptr_t` defined in `stdint.h` is an integral type that is guaranteed to be wide enough to hold pointers.

```
int *p; // p points to an int
intptr_t i = (intptr_t) p; // the address as a number
p++;
i++;
// Will i and p be the same?
```

Turning pointers into numbers

`intptr_t` defined in `stdint.h` is an integral type that is guaranteed to be wide enough to hold pointers.

```
int *p; // p points to an int
intptr_t i = (intptr_t) p; // the address as a number
p++;
i++;
// Will i and p be the same?
// No!  i++ increases by 1, p++ with sizeof(int)
```

Turning pointers into numbers

`intptr_t` defined in `stdint.h` is an integral type that is guaranteed to be wide enough to hold pointers.

```
int *p; // p points to an int
intptr_t i = (intptr_t) p; // the address as a number
p++;
i++;
// Will i and p be the same?
// No!  i++ increases by 1, p++ with sizeof(int)
```

There is also an unsigned version of `intptr_t`: `uintptr_t`

Strings

Strings

Having seen arrays and pointers, we can now understand C strings

```
char *s = "hello world\n";
```

Strings

Having seen arrays and pointers, we can now understand C strings

```
char *s = "hello world\n";
```

C strings are `char` arrays, which are terminated by a special null character, aka a null terminator, which is written as $\backslash 0$

Strings

Having seen arrays and pointers, we can now understand C strings

```
char *s = "hello world\n";
```

C strings are `char` arrays, which are terminated by a special null character, aka a null terminator, which is written as \0

There is a special notation for string literals, between double quotes, where the null terminator is implicit.

Strings

Having seen arrays and pointers, we can now understand C strings

```
char *s = "hello world\n";
```

C strings are `char` arrays, which are terminated by a special null character, aka a null terminator, which is written as \0

There is a special notation for string literals, between double quotes, where the null terminator is implicit.

As other arrays, we can use both the array type `char[]` and the pointer type `char *` for them.

String problems

Working with C strings is highly error prone!
There are two problems

String problems

Working with C strings is highly error prone!
There are two problems

1. As for any array, there are no array bounds checks
   so it's the programmer's responsibility not to go outside the array bounds

String problems

Working with C strings is highly error prone!
There are two problems

1. As for any array, there are no array bounds checks
   so it's the programmer's responsibility not to go outside the array bounds
2. It is also the programmer's responsibility to make sure that the string is properly terminated with a null character.
   If a string lacks its null terminator, e.g. due to problem 1, then standard functions to manipulate strings will go off the rails.

Safer strings and array?

There is no reason why programming language should not provide safe versions of strings (or indeed arrays).
Other languages offer strings and arrays which are safer in that:

Safer strings and array?

There is no reason why programming language should not provide safe versions of strings (or indeed arrays).

Other languages offer strings and arrays which are safer in that:

1. Going outside the array bounds will be detected at runtime (e.g. Java)

Safer strings and array?

There is no reason why programming language should not provide safe versions of strings (or indeed arrays).
Other languages offer strings and arrays which are safer in that:

1. Going outside the array bounds will be detected at runtime (e.g. Java)
2. Which will be resized automatically if they do not fit (e.g. Python)

Safer strings and array?

There is no reason why programming language should not provide safe versions of strings (or indeed arrays).
Other languages offer strings and arrays which are safer in that:

1. Going outside the array bounds will be detected at runtime (e.g. Java)
2. Which will be resized automatically if they do not fit (e.g. Python)
3. The language will ensure that all strings are null-terminated (e.g. C++, Java and Python)

Safer strings and array?

There is no reason why programming language should not provide safe versions of strings (or indeed arrays).
Other languages offer strings and arrays which are safer in that:

1. Going outside the array bounds will be detected at runtime (e.g. Java)
2. Which will be resized automatically if they do not fit (e.g. Python)
3. The language will ensure that all strings are null-terminated (e.g. C++, Java and Python)

More precisely, the programmer does not even have to know how strings are represented, and whether null-terminator exists and what they look like: the representation of strings is completely transparant/invisible to the programmer.

Safer strings and array?

There is no reason why programming language should not provide safe versions of strings (or indeed arrays).
Other languages offer strings and arrays which are safer in that:

1. Going outside the array bounds will be detected at runtime (e.g. Java)
2. Which will be resized automatically if they do not fit (e.g. Python)
3. The language will ensure that all strings are null-terminated (e.g. C++, Java and Python)

More precisely, the programmer does not even have to know how strings are represented, and whether null-terminator exists and what they look like: the representation of strings is completely transparant/invisible to the programmer.

Moral of the story: if you can, avoid using standard C strings.
E.g. in C++, use C++ type strings; in C, use safer string libraries.

A final string pecularity

Strig literals, as in

```
char *msg = "hello, world";
```

are meant to be constant or read-only: you are not supposed to change the character that made up a string literal.

A final string pecularity

Strig literals, as in

```
char *msg = "hello, world";
```

are meant to be constant or read-only: you are not supposed to change the character that made up a string literal.

Unfortunately, this does not mean that C will *prevent* this. It only means that the C standard defines changing a character in a string literal as having undefined behaviour ☹

A final string pecularity

Strig literals, as in

```
char *msg = "hello, world";
```

are meant to be constant or read-only: you are not supposed to change the character that made up a string literal.

Unfortunately, this does not mean that C will *prevent* this. It only means that the C standard defines changing a character in a string literal as having undefined behaviour ☹
E.g.

```
char *t = msg + 6;
*t = ';';
```

Has undefined behaviour, i.e. anything may happen.

A final string pecularity

Strig literals, as in

```
char *msg = "hello, world";
```

are meant to be constant or read-only: you are not supposed to change the character that made up a string literal.

Unfortunately, this does not mean that C will *prevent* this. It only means that the C standard defines changing a character in a string literal as having undefined behaviour ☹

E.g.

```
char *t = msg + 6;
*t = ';';
```

Has undefined behaviour, i.e. anything may happen.

Compilers can emit warnings if you change string literals, e.g.

```
gcc -Wwrite-strings
```

Recap

We have seen

Recap

We have seen

- The different C types

Recap

We have seen

- The different C types
  - primitive types
    `(unsigned) char, short, int, long, long long, float ...`

Recap

We have seen

- The different C types
  - primitive types
    `(unsigned) char, short, int, long, long long, float ...`
  - implicit conversions and explicit conversions (casts) between them

Recap

We have seen

- The different C types
  - primitive types
    `(unsigned) char, short, int, long, long long, float ...`
  - implicit conversions and explicit conversions (casts) between them
  - arrays `int[]`

Recap

We have seen

- The different C types
  - primitive types
    `(unsigned) char, short, int, long, long long, float ...`
  - implicit conversions and explicit conversions (casts) between them
  - arrays `int[]`
  - pointers `int *` with the operations `*` and `&`

Recap

We have seen

- The different C types
  - primitive types
    `(unsigned) char, short, int, long, long long, float ...`
  - implicit conversions and explicit conversions (casts) between them
  - arrays `int[]`
  - pointers `int *` with the operations `*` and `&`
  - C strings, as special `char` arrays

Recap

We have seen

- The different C types
  - primitive types
    `(unsigned) char, short, int, long, long long, float ...`
  - implicit conversions and explicit conversions (casts) between them
  - arrays `int[]`
  - pointers `int *` with the operations `*` and `&`
  - C strings, as special `char` arrays
- Their representation

Recap

We have seen

- The different C types
  - primitive types
    `(unsigned) char, short, int, long, long long, float ...`
  - implicit conversions and explicit conversions (casts) between them
  - arrays `int[]`
  - pointers `int *` with the operations `*` and `&`
  - C strings, as special `char` arrays
- Their representation
- How these representations can be 'broken', i.e. how we can inspect and manipulate the underlying representation (e.g. with casts)

Recap

We have seen

- The different C types
  - primitive types
    `(unsigned) char, short, int, long, long long, float ...`
  - implicit conversions and explicit conversions (casts) between them
  - arrays `int[]`
  - pointers `int *` with the operations `*` and `&`
  - C strings, as special `char` arrays
- Their representation
- How these representations can be 'broken', i.e. how we can inspect and manipulate the underlying representation (e.g. with casts)
- Some things that can go wrong
  e.g. due to access outside array bounds or integer under/overflow