Today

- Higher-level programming languages as an abstraction layer, using compiler or interpreter

  To understand security problems in software, we may have to understand how this works...

- The programming language C as an abstraction layer for code and data
  - this week: data types and their representation
  - next weeks: memory management in general

programming languages
as
abstraction layers

Programming language is an abstraction layer

- A programming language tries to provide a convenient abstraction layer over the underlying hardware

- The programmer should not have to worry about
  - machine instructions of the CPU
  - precisely where in main memory or disk data is allocated
  - how to change some pixels on the screen to show some text
  - ....

CPU


RAM


disk


I/O peripherals

abstraction

In

```
int main(int i){
  printf(”hello, world\n”);
  return 2*i/(6+i);
 }
```

we abstract from

- *how* the data is represented
- *where* in memory (in CPU, RAM or on disk) this data is stored
- *which* machine instructions are executed
- *how* data is printed to the screen

This abstraction is provided the programming language
together with the operating system (OS)

The operating system is responsible for some abstractions, esp.

- memory management
- handling I/O
  - incl. file system

For I/O the OS will provide some standard libraries to the programmer, described as part of the programming language specification.

Eg for C, this includes functions such as **`printf(), fopen(),...`**

Different levels of abstraction for data

1. In programming language we can write a string

   ”hello, world\n”

   and not care *how* this data is represented or *where* it is stored

2. At a lower level, we can think of memory as a sequence of bytes

					h	e	l	l	o	,		w	o	r	l	d	\n	\0				

3. At the level of hardware, these bytes may be spread over the CPU (in registers and caches), the main memory, and hard disk


There are still lower levels, but then we get into electronics and physics.

- **Does the programmer have to know how this works?**

- In the *ideal* situation we have representation independence for data: the programmer does *not* need to know how data is represented on lower levels of abstractions
  - *except* to understand the efficiency of programs

- However, for most programming language, the programmer does have to understand this, in to understand the behaviour of programs, esp. under unusual circumstances

  eg. when program is attacked with malicious input

Compiled vs interpreted languages

There are two ways to bridge the gap between the abstract programming language and the underlying hardware

1. a compiler that translates high-level program code to machine code that can be executed on raw hardware

   Eg: C, C++, Fortran, Ada, ...

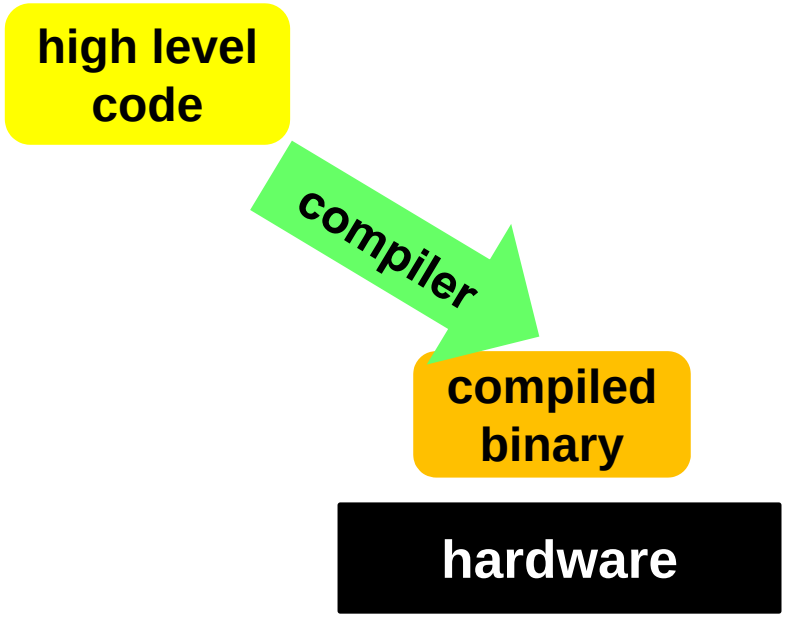
2. an interpreter that provides an execution engine aka virtual machine for the high level language

   Eg LISP, Haskell, and other functional programming languages, JavaScript, ...

The compiler and interpreter will have to be in machine code, or in a language that we have another compiler or interpreter for.

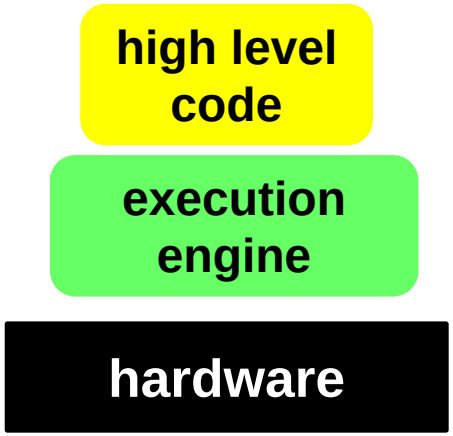
compilation vs interpretation

Compiled binary runs on the bare hardware



The compiler - and the high-level programming language - is not around at runtime

Software layer isolates code from hardware



The programming language still exists at runtime

Pros & cons of compilation vs interpretation?

- Advantage of compiler
  - compiled code is generally faster

- Advantage of interpretation
  - interpreted code is more portable
    - can be run on any hardware, given the right execution engine for that hardware
  - interpreted code can be more secure
    - more built-in security enforced by the language

Security

- A drawback of compiling to machine code:

  at runtime the programming language, with all the machinery it provides (for data types, control flow, ...) , no longer exists.

- In an interpreted language, all the information of the original (high-level) program is still available, so the execution engine can do some sanity checks at run time to control their usage

  for example for typing

  Still, a compiler could also compile in such sanity checks.

Combining compilation and interpretation

More modern programming languages such as Java or C# combine compilation and interpretation, using an intermediate language

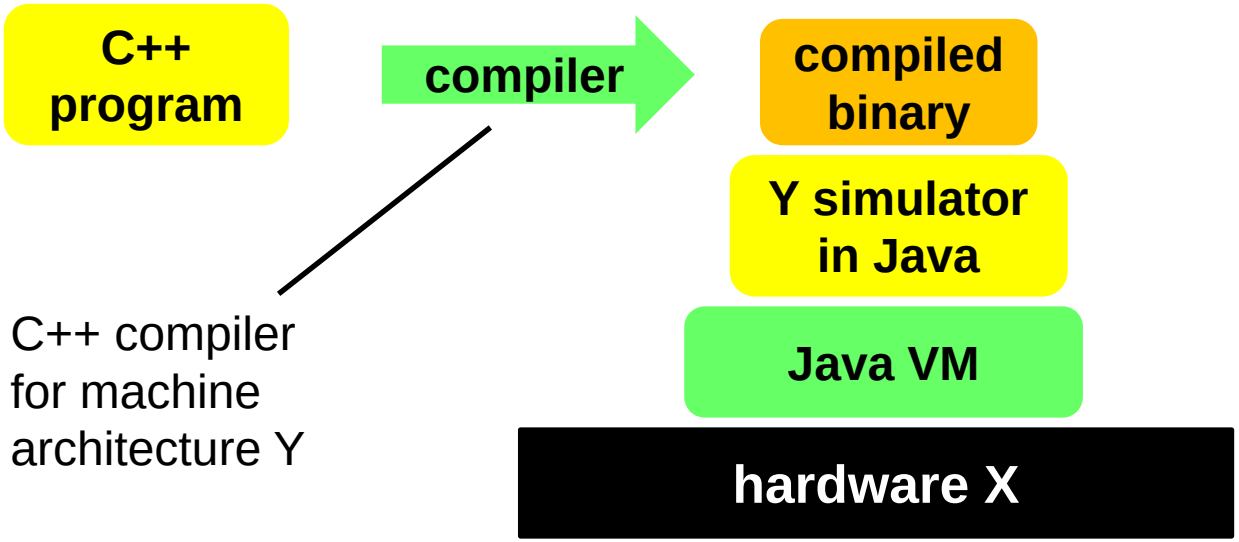
Java source code is compiled to byte code,
which can be executed (interpreted) by the Java Virtual Machine

The goal is to get the best of both worlds

Virtualisation

A way to make binaries portable: implement a program on machine X that simulates the hardware of machine Y

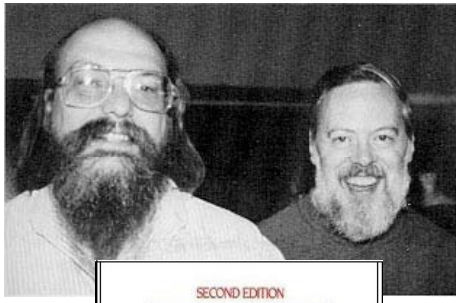
Eg, you could write an simulator for Y in Java

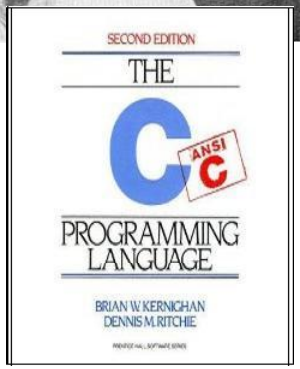


Modern CPUs offer hardware support for such virtualisation

The programming language C

The programming language C





- invented Dennis Ritchie in early 1970s
  - who used it to write the first Hello World program
  - C was used to write UNIX
- Standardised as
  - K&C (Kernighan & Ritchie) C
  - ANSI C aka C90
  - C99 newer ISO standard in 1999
  - C11 most recent ISO standard of 2011
- Basis for C++, Objective C, ... and many other languages

  *NB C++ is **not** a superset of C*
- Many other variants, eg

  MISRA C for safety-critical applications in automotive industry

The programming language C

- C is very powerful, and can be very efficient, because it gives raw access to the underlying platform (CPU and memory)

- Downside: C provides much less help to the programmer to stay out of trouble than other languages.

  C is very liberal (eg in its type system) and does not prevent the programmer from questionable practices, which can make it harder to debug programs.

  For some examples to what this can lead to, check out the obfuscated C contest!

language definitions

A programming language definitions consists of

- Syntax

  The spelling and grammar rules, which say what 'legal'

  - or syntactically correct - program texts are.

  Syntax is usually defined using a grammar, typing rules, and scoping rules

- Semantics

  The meaning of 'legal' programs.

  Much harder to define!

  The semantics of some syntactically correct programs may be left undefined (though one would rather not do this!)

C compilation in more detail

- As first step, the C preprocessor will add and remove code from your source file, eg using **`#include`** directives and expanding macros

- The compiler then translates programs into object code
  - Object code is almost machine code
  - Most compilers can also output assembly code, a human readable form of this

- The linker takes several pieces of object code (incl. some of the standard libraries) and joins them into one executable which contains machine code
  - Executables also called binaries

By default gcc will compile and link

What does a C compiler have to do?

1. represent all data types as bytes

2. translate operations on these data types to the basic instruction set of the CPU

3. translate higher-level control structures
   eg **`if then else`**, **`switch`** statements, **`for`** loops
   to jumps (**`goto`**)

4. provide some “hooks” so that at runtime the CPU and OS can handle function calls

NB function calls have to be handled at runtime, when the compiler is no longer around, so this has to be handled by CPU and OS

memory abstraction: *how* data is represented

C provides some data types, and programmer can use these without having to know *how* this is actually represented - to some degree.

eg. in C we can write

character	`'a'`
string	`"Hello World"`
floating point number	`1.345`
array of int's	`{1,2,3,4,5}`
complex number	`1.0 + 3.0 * I`

memory abstraction: *where* data is stored

We also do not need to know *where* the data is stored (aka allocated) - again to some degree.

At runtime, an **`int x`** could be stored

- in a register on the CPU
- in the CPU cache
- in RAM
- on hard disk

Compiler will make some decisions here, but it's up to the operating system and CPU to do most of this work at runtime

C data types
and their representation

Computer memory

- The memory can be seen as a sequence of bytes
- Actually, it is a sequence are n-bytes words
  - where n=1, 2,4,8 on 8,16, 32, 64 bit architecture
- All data is in the end just bytes
  - *everything* is represented as bytes; not just data, also code
  - *different data* can have the *same representation* as bytes
    - hence the *same* byte can have *different* interpretations, depending on the context
  - the *same* piece of data may even have *different* representations

char

The simplest data type in C is **`char`**.
A **`char`** is always a byte.

The type **`char`** was traditionally used for ASCII characters,
so **`char`** values can be written as numbers or as characters, e.g.

```
char c = '2';
char d = 2;
char e = 50;
```

QUIZ: which of the variables above will be equal?

`c` *and* ***`e`*** *, as they both have value 50:*

the character ***`'2'`*** *is represented as its ASCII code 50*

other integral types

C provides several other integral types, of different sizes

- **`short`** or **`short int`** usually 2 bytes
- **`int`** usually 2 or 4 bytes
- **`long`** or **`long int`** 4 or 8 bytes
- **`long long`** 8 bytes or longer

The exact sizes can vary depending on the platform!

You can use **`sizeof()`** to find out the sizes of types,

eg **`sizeof(long)`** or **`sizeof(x)`**

Integral values can be written in decimal, hexadecimal (using 0x) or octal notation (using 0) , where 0 is zero, not O

eg **`255`** is **`0xFF`** (hexadecimal) or **`0177777`** (octal)

stdint.h

Because the bit-size (or width) of standard types such as **`int`** and **`long`** can vary, there are standard libraries that define types with guaranteed sizes.

Eg **`stdint.h`** defines

`uint16_t` for unsigned 16 bit integers

floating point types

C also provides several floating point types, of different sizes

- `float`
- `double`
- `long double`

Again, sizes vary depending on the platform.

The floating point types will probably not be used in this course.

signed vs unsigned

Numeric types have **`signed`** and **`unsigned`** versions

The default is **`signed`** - except possibly for **`char`**

For example

`signed char`	can have values	-128 ... 127
`unsigned char`	can have values	0 ... 255

In these slides, I will assume that **`char`** is a **`signed char`**

register

- Originally, C had a keyword **`register`**

```
register int i;
```

  This would tell the compiler to store this value in a CPU register rather than in main memory. The motivation for this would be that this variable it is used frequently.

- NB you should *never ever* use this! Compilers are *much* better than you are at figuring out which data is best stored in CPU registers.

implicit type conversions

Values of numeric type will automatically be converted to wider types when necessary.

Eg **`char`** converts to **`int`**, **`int`** to **`float`**, **`float`** to **`double`**

```
char c = 1;
int i = 2;
float f = 3.1415;
double d = i * f; // i converted to float, then
                  // i * f converted to double
long g = (c*i)+i; // c converted to an int
                  // then result to a long
```

What happens if **`c*i`** overflows as 32-bit int, but not as 64-bit long?
My guess is that it's platform-specific, but maybe the C spec says otherwise?

explicit type casts

You can cast a value to another type

```
int i = 23456;
char c = (char) i;  // drops the higher order bits
float f = 12.345;
i = (int) f;        // drops the fractional part
```

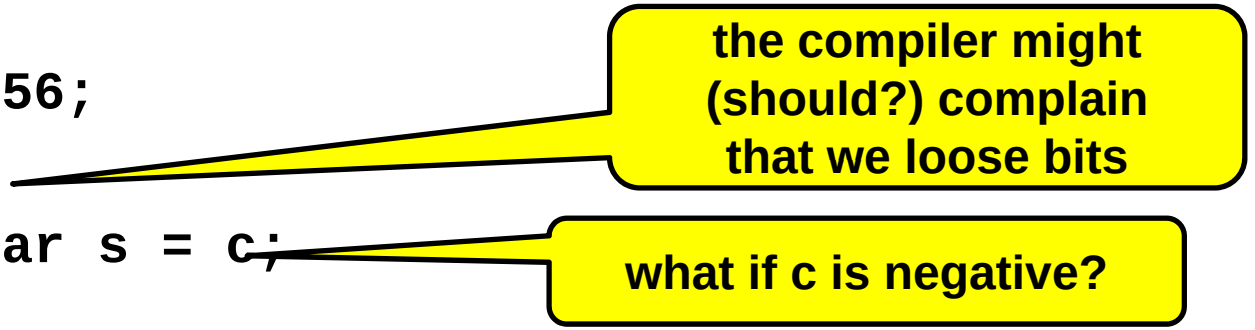
Such casts can loose precision, but the cast make this explicit.

*Question: can **`c`** have a negative value after the cast above?*
*It may have, if the lower 8 bits of 23456 happen to represent a negative number, for the representations of **`int`** and **`char`** (incl. negative **`char`**s) used.*

So casts can not just loose precision, but also change the meaning

some *implicit* conversion can also be dangerous

```
int i = 23456;
char c = i;
unsigned char s = c;
```



Is this legal C code? Is the semantics clear?

C compilers do not always warn about dangerous implicit conversions which may loose bits or change values!

Conversions between signed and unsigned types do not always give intuitive results.

Of course, a good programmer will steer clear of such implicit conversions.

Quiz: signed vs unsigned

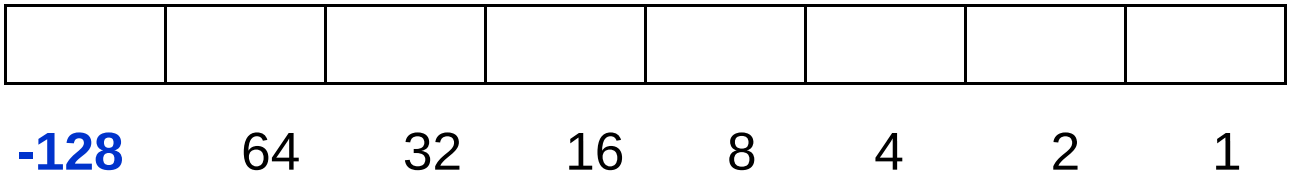
Conversions between signed and unsigned data types do not always behave intuitively

```
unsigned char x = 128;
  signed char y = x; // what will value of y be?
```

Moral of the story: mixing signed and unsigned data types in one program is asking for trouble

Representation: two's complement

- Most platforms represent negative numbers using the two's complement method. Here the most signification bit represents a large negative number $-(2^n)$



So -128 is represented as 1000 0000

-120 as 1000 0100

and the largest possible signed byte value, 127,

as 0111 1111

Representation: big endian vs little endian

Integral values that span multiple bytes can be represented in two ways

- big endian : most significant byte first
- little endian : least significant byte first (ie backwards)

For example, a **`long long x = 1`** is represented as

`00 00 00 01` big endian

`01 00 00 00` litte endian

Some operations are easier to implement for a big endian representation, others for little endian.

Little endian may seems strange, but has the advantage that types of different lengths can be handled more uniformly:

eg litte endian, an **`int`** 1 would be represented as **`01 00`**

data alignment

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

data alignment

Memory as a sequence of bytes

…		x	i_4	i_3	i_2	i_1	s_2	s_1	y			…

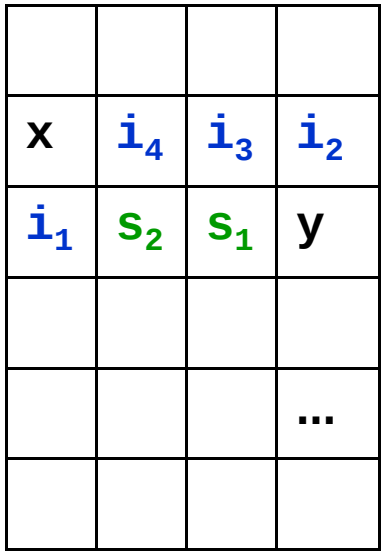
But on 32-bit machine, the memory be a sequence of 4-byte words

x	i_4	i_3	i_2
i_1	s_2	s_1	y
			…

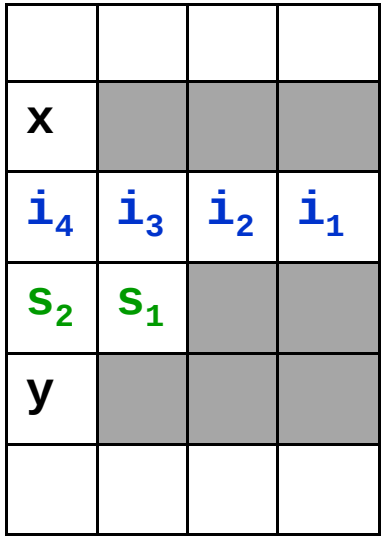
Now the data elements are not nicely aligned with the words,
which will make execution slow, since CPU instructions act on words.

data alignment

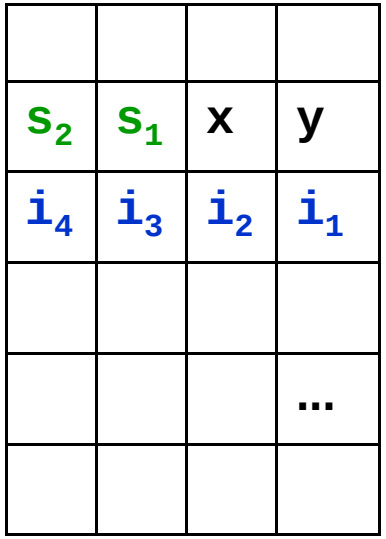
Different allocations, with better/worse alignment



lousy alignment, but using minimal memory



optimal alignment, but wasting memory



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.

Most C compilers can provide many optional optimisations.

Eg use

```
man gcc
```

to check out the many optimisation 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 %i \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.

Eg ALGOL60, defined in 1960, already included array bound checks.

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 a[10];
int x = 6;
printf("oops %i \n", a[10]);
```

What would you expect this program to print?

If the compiler allocates **y** directly after **a**, then 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 a array is guaranteed to be contiguous

ie **`a[1]`** is allocated right after **`a[0]`**

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

eg a compiler might allocate16 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 **b[1]** after executing the following code?

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

25

Recall call by reference from Imperatief Programmeren!

pointers

retrieving addresses or *pointers* using &

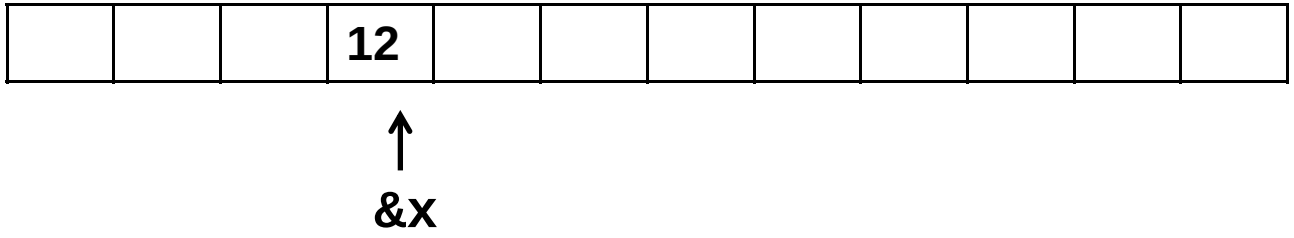
We can find out *where* some data is allocated using the & operation.

If

```
int x = 12;
```

then **&x** is the memory address where the value of x is stored,

aka a pointer to where x is stored



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

pointers

Eg for the alignment example discussed earlier
we can use **&** to see if the compiler aligned data

```
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

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; // ie. 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
```

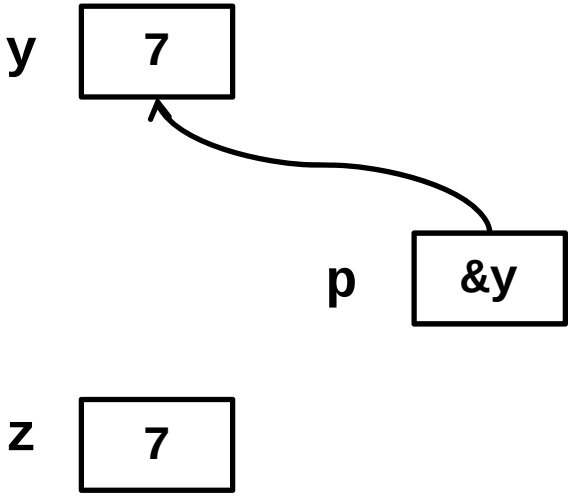
- How can we get the value that a pointer points to? Using *****

```
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
```



Read Section 9.1 of “Problem Solving with C++” for another explanation.

pointer quiz

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

What is the value of **y**?

3

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

What is the value of **y**?

4

Note that ***** is used for 3 different purposes

1. in declarations, to declare pointer types

```
int *p;  //  p is a pointer to an int
         //  ie. *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, eg **`z = 3 * *p;`**

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

What can be confusing in

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

is that this an assignment to **`p`**, not to **`*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 pointer to an **`int`** and **`y`** and **`z`** as **`int...`**

still not confused?

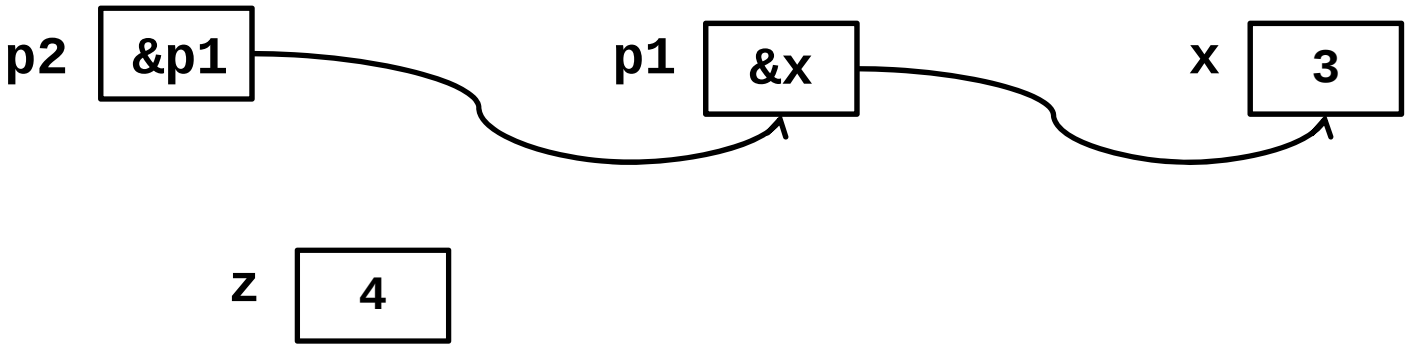
```
x = 3;
p1 = &x;
p2 = &p1;
z = **p2 + 1;
```

What will the value of **z** be?

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 refresher (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 %i\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!!!!! It is the address where **`z`** *is allocated plus* **`sizeof(int)`**, *ie* **`&z + sizeof(int)`**

pointer arithmetic

Pointers can be added to and subtracted from.

The semantics depends on the *type of the pointer:*

adding 1 to a pointer will go to the “next” location,

given the size of the data type that it points to.

For example, if

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

then

`ptr + 2` means **`ptr + 2 * sizeof(int)`**

`str + 2` means **`ptr + 2`**

because **`sizeof(char)`** is 1

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.
```

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

Now

p+3

points to

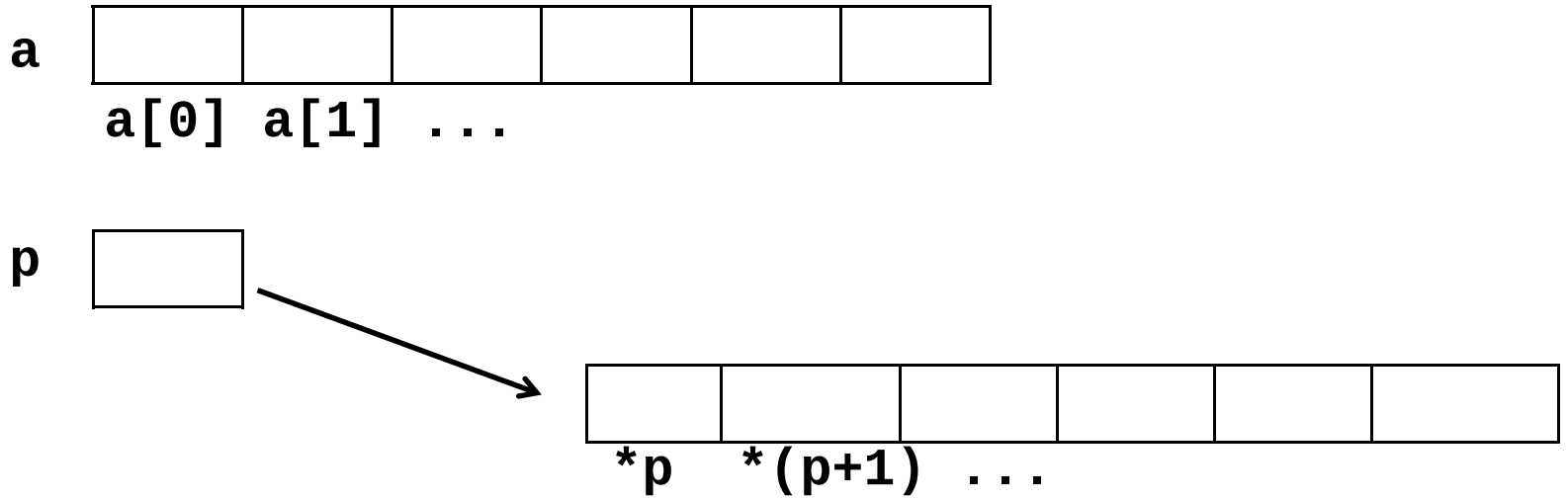
a[3]

so we use addition to pointer **p** to acces the array

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

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

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?
```

5

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

A problem with pointers: ***aliasing***

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...

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; i <sizeof(float); p++;) {
    printf("%i", *p); i++;
  }
 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 number
p++;
i++;
// Will i and p be the 'same'?
// No! i++ increases by 1, p++ with sizeof(int)!
```

There is also an unsiged version of **intptr_t: uintptr_t**

strings

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 null terminator, which is written as \0

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

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

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 programmers responsibility not to go outside the array bounds
2. moreover, 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, eg due to problem 1, then standard functions to manipulate strings will go off the rails.

safer strings and arrays?

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:

- going outside the array bounds will be detected at runtime (eg Java)
- which will be resized automatically if they do not fit (eg Python)
- the language will ensure that all strings are null-terminated (eg 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.

Eg in C++, use C++ type strings; in C, use safer string libraries.

a final string peculiarity

String literals, as in

```
char *msg = “hello, world”;
```

are meant to be constant or read-only: you are not supposed to change the characters that make 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 an string literal as having undefined behaviour ☹

Eg

```
char *t = msg + 6; *t = ’;’;
```

has undefined behaviour, ie. anything may happen

compilers can emit warnings if you change string literals, eg

```
gcc -Wwrite-strings
```

Recap

We have seen

- the different C types
  - primitive types
    `(unsigned) char, short, int, 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 representations
- how these representations can be `broken', ie. how we can inspect and manipulate the underlying representation (eg. with casts)
- some things that can go wrong
  eg due to access outside array bounds or integer under/overflow