C Programming
Casting

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

C lectures

- Compiling code, program layout, printing/reading data, expressions, arithmetic, memory addresses, control flow, precedence
- Functions, pointers, file IO, arrays
- **Memory allocation, casting, masking, shifting**
- Strings, structures, dynamic space allocation, field access
- Recursive structures, 2D arrays, union types

Recap: `sizeof` and arrays

- `sizeof(`*`type`*`)` tells you how much memory a variable of that type would occupy
- `sizeof(char)` is **always** 1
  - C counts sizes in `char`s – usually 8-bit bytes

- Arrays are contiguous in memory
- Declaring array:
  `type name``[`*`length`*`];`
  allocates size *`length`* `* sizeof(`*`type`*`)`

Low-level programming in C

- We often need to care about how exactly things are laid out in memory in C...
- e.g. controlling hardware devices, using hardware registers that are mapped into memory at specific addresses
- e.g. writing code to do custom memory allocation

Declarations and space

- Amount of space for a given type depends on:
  - Type – e.g. `char` vs `float`
  - Platform – e.g. 64-bit x86, 32-bit ARM
  - Compiler and version – for structure types (primitive types are normally standardised)

- `sizeof(type)`'s return type is **`size_t`**
  - An unsigned integer type big enough to hold any size – to print with `printf`, use `%zd`

`typesize.c`

Size of standard types

Sizes of types

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("char:   %zd\n", sizeof(char));
    printf("short:  %zd\n", sizeof(short));
    printf("int:    %zd\n", sizeof(int));
    printf("long:   %zd\n", sizeof(long));
    printf("float:  %zd\n", sizeof(float));
    printf("double: %zd\n", sizeof(double));
    printf("char *: %zd\n", sizeof(char *));
    return 0;
}
```

Typical type sizes

64-bit x86 PC

```
$ ./typesize
char:    1
short:   2
int:     4
long:    8
float:   4
double:  8
char *:  8
```

32-bit ARM RPi

```
$ ./typesize
char:    1
short:   2
int:     4
long:    4
float:   4
double:  8
char *:  4
```

64-bit RISC-V

```
$ ./typesize
char:    1
short:   2
int:     4
long:    8
float:   4
double:  8
char *:  8
```

Explicitly-sized types

- What if you definitely want a 64-bit type, regardless of the platform?

- C99: `<stdint.h>` header defines types like:
  - `int64_t` – signed 64-bit integer
  - `uint32_t` – unsigned 32-bit integer
- Also macros for `printf` formats for these – see the documentation

Recap: bits, bytes, hexadecimal

- A hexadecimal constant in C:
  `0x`$h_1 h_2$`...`h_N
  where h_i == 0..9 A..F (or a..f)
- Each hex digit is 4 bits, so an 8-bit byte is 2 hex digits
- In `0xAB`, A is bits 7-4, B is bits 3-0
- e.g. `0xFF` == `1111 1111` == all bits 1
  `0x00` == `0000 0000` == all bits 0
  `0x65` == `0110 0101` == ASCII `'e'`

Big- and little-endian machines

- When a value occupies more than one byte, what order are the bytes stored in? Depends on the machine...

e.g.
```
int a;              // 32 bit type
a = 0x01234567;
```

- **Little-endian** architecture – least significant byte stored first

&a →

0x**67**	0x45	0x23	0x01
byte 0	byte 1	byte 2	byte 3

- **Big-endian** architecture – most significant byte stored first

&a →

0x01	0x23	0x45	0x**67**
byte 0	byte 1	byte 2	byte 3

Big- and little-endian machines

- When a value occupies more than one byte, what order are the bytes stored in? Depends on the machine...

e.g.
```
int a;                // 32 bit type
a = 0x01234567;
```

- **Little-endian** architecture – least significant byte stored first

&a →

0x67	0x45	0x23	0x01
byte 0	byte 1	byte 2	byte 3

Nearly all modern architectures are little-endian.
(ARM can be switched to operate in either mode,
but it's nearly always used in little-endian mode.)

Accessing bytes in C

- In C, we can access the individual bytes that make up a value by **pointer casting**
  - e.g. we can take an `int`, and reinterpret it as an array of 4 `chars`
- Casting is C's explicit **type conversion** mechanism
  - `(`*`newtype`*`)` *`expression`* – evaluate *`expression`*, then convert the result to *`newtype`*
  - e.g. `(int) sqrt(47.3)`

`cast.c`

Casting `int` to `char`

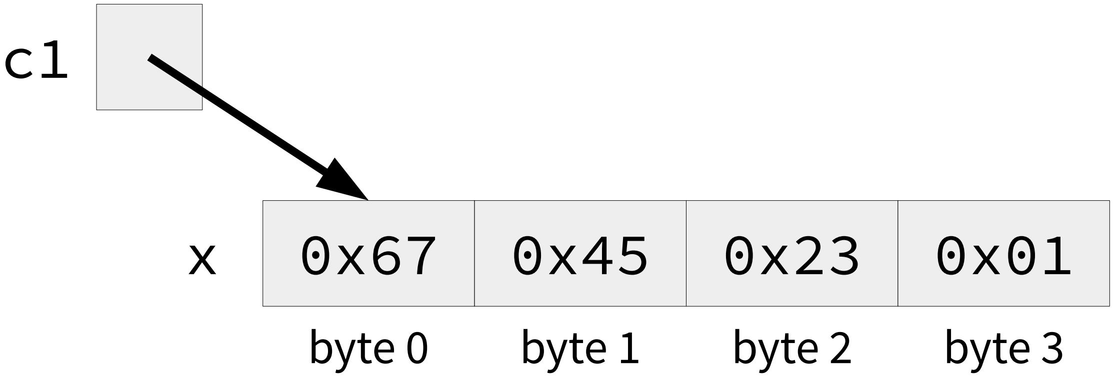
Casting integers

```
int x = 0x01234567;
```

x	0x67	0x45	0x23	0x01
	byte 0	byte 1	byte 2	byte 3

Casting integers

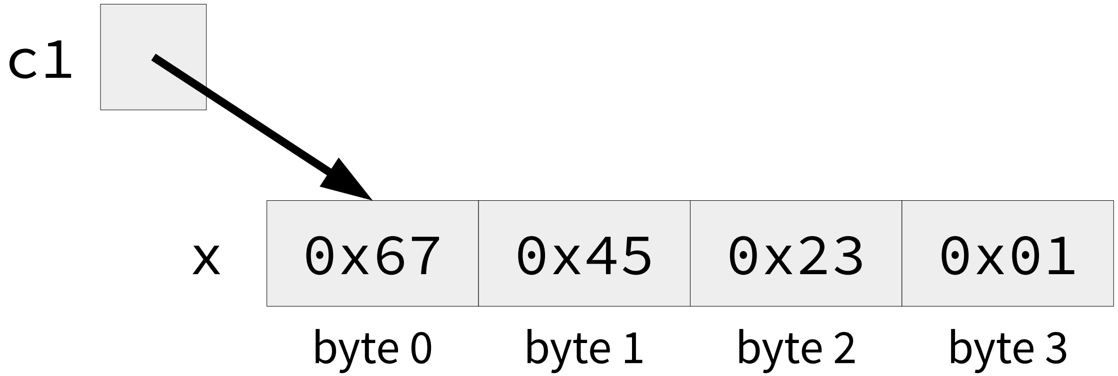
```
int x = 0x01234567;
char *c1 = (char *) &x;
```



Casting integers

```
int x = 0x01234567;
char *c1 = (char *) &x;
printf("%x\n", *c1);
```

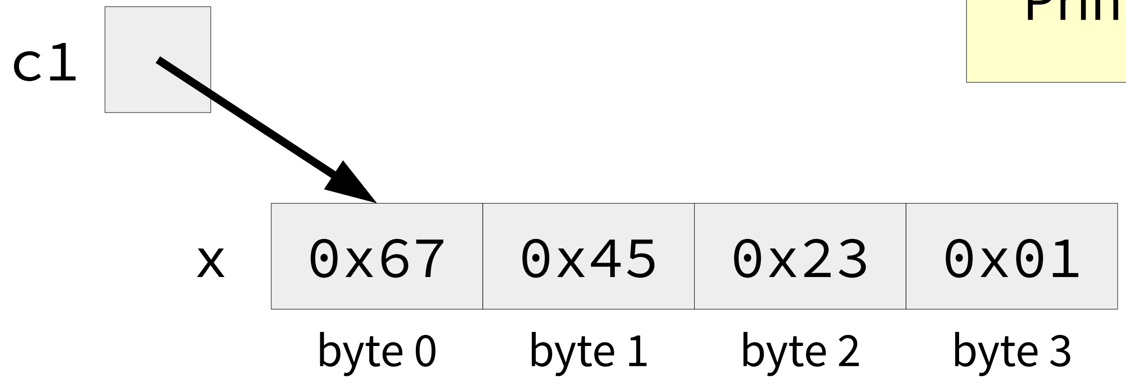
Prints: 67



Casting integers

```
int x = 0x01234567;
char *c1 = (char *) &x;
printf("%x %x %x %x\n",
       c1[0], c1[1], c1[2], c1[3]);
```

Prints: 67 45 23 1



Strict aliasing

- The previous example is legal C, but many similar pieces of code wouldn't be
  - e.g. `float *f = (float *) &a;`
- Standard C's **aliasing** rules say that you aren't allowed to refer to a value by a pointer type that doesn't match the original declaration
- ... **unless** the pointer type is `char *` – which is why the previous example's OK!

Strict aliasing

- Many C programmers are not aware of this rule, and you'll see incorrect “type punning” code in **many** real programs
- In most cases it'll work, but modern compilers are getting more aggressive about taking advantage of aliasing when optimising code...
- ... so compilers often have an option to make it legal, e.g. GCC's `-fno-strict-aliasing`