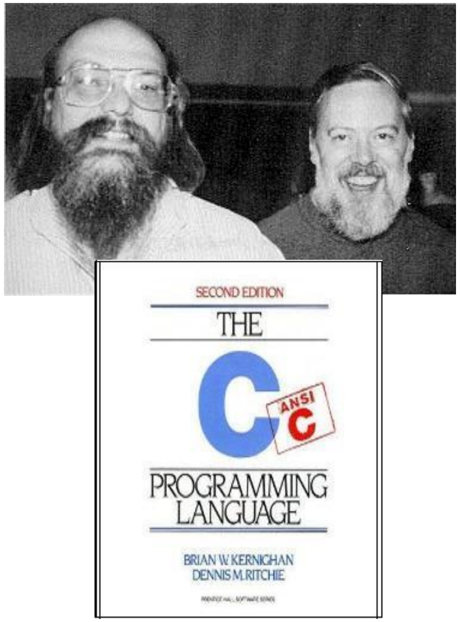
The programming language C

The programming language C



- invented by Dennis Ritchie in early 1970s
  - who used it to write the first Hello World program
  - C was used to write UNIX
- Standardised as
  - K&R (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 car 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 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 example, see the program silly_bool_argument.c]

syntax & semantics

A programming language definition 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 (but it is better 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. decide where pieces of data are stored (memory management)

3. translate all operations into the basic instruction set of the CPU

   - this includes translating higher-level control structures, such as **`if then else`**, **`switch`** statements, **`for`** loops, into 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 (1): *how* data is represented

C provides some data types, and programmer can use these without having to know *how* they are 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 (2): *where* data is stored

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

This is called memory management.

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

Where is data allocated? &

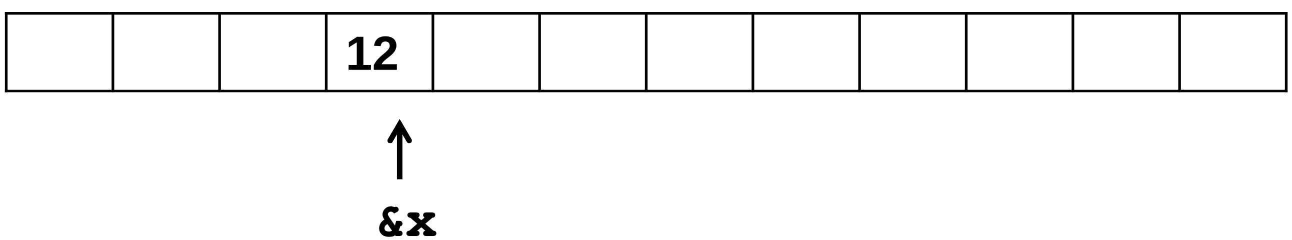
We can find out *where* some data is allocated using the & operation.

Suppose we have a variable

```
int x = 12;
```

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

aka a pointer to x



Much more on pointers later!

Where is data allocated? pointers

```
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
```

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, respectively

- 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)

Printing values in different notations

The procedure **`printf`** can print numeric values in different notations.

```
int j = 15;
printf(″The value of j is %i \n″, j);
printf(″In octal notation: %o \n″, j);
printf(″In hexadecimal notation: %x \n″, j);
printf(″Idem with capitals: %X \n″, j);
```

Check websites such as cppreference.com for details.

See course webpage for more links.

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 by default 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.

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

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 cannot only loose precision, but also change the meaning

some *implicit* conversion can also be dangerous

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

the compiler might (should?) complain that we loose bits

what if c is negative?

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 $–(2^n)$

-128	64	32	16	8	4	2	1

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