The history of C

What is C?
- Systems programming language.
- Dennis Ritchie at Bell labs, early 70's.
- Developed for coding UNIX.
- A language in the Algol 60 family (as Pascal and Ada).

C versions:
- Kernighan & Ritchie (K&R) C.
- ANSI C.

C++:
- Object oriented language.
- More or less an extension of C.

C: pros and cons

Why learn and use C?
- C is one of the most used languages.
- High quality tools available.
- Flexible.
- Portable (with care).

Disadvantages?
- Low level language.
- (Unnecessarily) complicated and difficult to learn in some cases.
- Intrinsically poor support for large scale programming.

The traditional first C program

```c
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Compare the following Pascal program:
```pascal
program hello;
begin
    writeln('Hello world!')
end.
```

Or the following in C++:
```c++
#include <iostream.h>
int main()
{
    cout << "Hello world!";
    return 0;
}
```

Interest calculation

Write a program that reads the capital sum paid into a bank account and then tabulates the balance the following 10 years assuming a fixed rate of interest of 8.5 % and disregarding any taxes.

<table>
<thead>
<tr>
<th>Year</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1085.00</td>
</tr>
<tr>
<td>2</td>
<td>1177.22</td>
</tr>
<tr>
<td>3</td>
<td>1277.29</td>
</tr>
<tr>
<td>4</td>
<td>1385.86</td>
</tr>
<tr>
<td>5</td>
<td>1503.66</td>
</tr>
<tr>
<td>6</td>
<td>1631.47</td>
</tr>
<tr>
<td>7</td>
<td>1770.14</td>
</tr>
<tr>
<td>8</td>
<td>1920.60</td>
</tr>
<tr>
<td>9</td>
<td>2083.86</td>
</tr>
<tr>
<td>10</td>
<td>2260.98</td>
</tr>
</tbody>
</table>
Interest calculation: C and C++ versions

/* Fixed interest calc. */
#include <stdio.h>
#define YEARS 10
#define RATE 8.5

int main(void)
{
    float c;
    int y;
    printf("Capital sum? ");
    scanf("%f", &c);
    printf("Year   ");
    printf("Balance
");  
    printf("====  ");
    printf("==========
");
    for (y=1; y<=YEARS; y++)
    {
        c = c*(1+RATE/100);
        printf("%4d%12.2f
", y, c);
    }
    return 0;
}

Interest calculation: Pascal version

(* Fixed interest calc. *)
program interest;
const
    years = 10;
    rate = 8.5;
var
    c : real;
    y : integer;
begin
    write('Capital sum? '); readln(c);
    writeln;
    write('Year   '); writeln('Balance');
    write('====  '); writeln('==========');
    for y := 1 to years do
    begin
        c := c*(1+rate/100);
        writeln(y:4, c:12:2);
    end.

Compilation and the preprocessor

foo.c -> pp -> cc -> a.out

#include <stdio.h>  
#include "stdio.h"
#define M "Hello!
"  
main()
{  
    printf(M);
}
The function printf

Flexible and powerful function for formatted printing. Basic idea:
- First argument is the format string:
  - plain text
  - conversion specifications.
- Then follow 0 or more arguments to be printed according to format specifications.

Given the following:
```c
int n = 7; float x = 3.14;
printf("n=%d, x=%f\n", n, x);
```
the output would look like:
```
n=7, x=3.140000
```
NOTE! Make sure the format string is consistent with the arguments that follow!

The function scanf

Flexible and powerful function for formatted input. Similar in spirit to printf:
- First argument is the format string consisting of conversion specifications.
- Then follow 0 or more arguments which are addresses of variables in which to put what is read.
- scanf returns the number of successful conversions or EOF.

Typical example:
```c
int i, j; float x;
scanf("%d%d%f", &i, &j, &x);
```
Note: the return value is not used here.
Finding maximum and mean

```c
#include <stdio.h>

float max(float x, float y) {
    if (x > y)
        return x;
    else
        return y;
}

main() {
    float x, sum = 0, xmax = 0;
    int n = 0;
    printf("Enter numbers, EOF to stop.\n");
    while (scanf("%f", &x) == 1) {
        n++;
        sum += x;
        xmax = max(xmax, x);
    }
    printf("Maximum: %f\n", xmax);
    printf("Mean:    %f\n", sum / n);
}
```

Exercise 1

Write a program to produce a table like the following for $1 \leq i \leq n$:

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i*i$</th>
<th>$i<em>i</em>i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Read $n$ interactively by means of a suitable `scanf`-statement (use "%d" as conversion specification).

Integer types

8 integer types in ANSI C, their sizes are implementation dependent:

- `signed char` (≥ 8 bits)
- `unsigned char` (≥ 8 bits)
- `(signed) short (int)` (≥ 16 bits)
- `unsigned short (int)` (≥ 16 bits)
- `(signed) int` (≥ 16 bits)
- `unsigned int` (≥ 16 bits)
- `(signed) long (int)` (≥ 32 bits)
- `unsigned long (int)` (≥ 32 bits)

The type `char` is either signed or unsigned (implementation dependent).

Literal integers: 1000, 01750, 0x3E8.

Have type `int` unless too large to fit.

Note! Literal character constants have type `int`! E.g. (Sun): ‘A’ = 65, ‘Å’ = -59.

Integer types (cont.)

As an example, for Sun workstations we have:

- `char` = [-128, 127] (8)
- `unsigned char` = [0, 255] (8)
- `short` = [-32768, 32767] (16)
- `unsigned short` = [0, 65535] (16)
- `int` = [-2147483648, 2147483647] (32)
- `unsigned int` = [0, 4294967295] (32)
- `long` = [-2147483648, 2147483647] (32)
- `unsigned long` = [0, 4294967295] (32)

This information is available in `<limits.h>`:

```c
#define CHAR_BIT     8
#define SCHAR_MIN    (-128)
#define SCHAR_MAX    127
#define UCHAR_MAX    255
#define CHAR_MIN     SCHAR_MIN
#define CHAR_MAX     SCHAR_MAX
...```

```c
#define CHAR_MIN (char)(-128)
#define CHAR_MAX (char)(127)
#define UCHAR_MIN (unsigned char)(0)
#define UCHAR_MAX (unsigned char)(255)
```
### Integer usage

**When to use which type?**

- **Char** is normally used to represent ASCII characters.
- **Int** is used for integers unless
  - large range needed, or
  - space is paramount.
- **Int** is also used for *truth values* (boolean in Pascal):
  - 0 represents *false*
  - 1 represents *true*.
- **Unsigned integer types** when e.g.:
  - counting modulus $2^{size}$
  - doing bitwise operations.

### Floating point types

ANSI C defines three floating point types:

- **float**
- **double**
- **long double**

The type float has at least a precision of 6 decimal digits. The smallest float (apart from 0) is $\leq 10^{-37}$ and the largest is $\geq 10^{37}$. The others are ‘at least as good’.

Literal floating point constants are written as in most other languages, e.g.:

- $0.17, 104.7, 0.98765e15, 1.28E-7$

### Arithmetic type conversions

The arithmetic types may be ordered in C:

- long double (‘highest’ type)
- double
- float
- unsigned long int
- long int
- unsigned int
- int
- unsigned short int
- short int
- unsigned char
- signed char (‘lowest’ type)

### Arithmetic type conversions (cont.)

Automatic type conversions are sometimes performed, e.g.:

- arithmetic expressions with operands of mixed types
- assignments with mixed types.

Ground rule for arithmetic expressions:

- conversion to the ‘highest’ type involved takes place.
- `char c; int n; float x;`
  - `c = 'a';`
  - `x = c + 3;`
  - `x = x * (n + c);`

The exact rules are somewhat complicated.
Explicit type casts

Explicit type casts are possible in C:

$(typename) \ expression$

Casts between arithmetic types are OK and behave sensibly.

But exact rules again somewhat complicated.

Some examples:

$(float) 123 = 123.000$
$(int) 188.7 = 188$
$(float) 1999999999 = 2.00000e9$

Exercise 2

getchar reads a single character (or EOF):

```c
int getchar(void);
```

Supposing that char is unsigned and 8 bits, int is 32 bits and EOF is defined to be -1, try to explain why the following program loops:

```c
#include <stdio.h>
main()
{
    char c;
    int n = 0;

    while ((c = getchar()) != EOF)
        n++;
}
```

(The value of an assignment is the value on the right hand side, e.g. $x = (y = 0.17)$ will assign 0.17 to both x and y.)

Arithmetic and mathematical functions

The usual arithmetic operators are available:

$+, -, *, /, \%$ (modulus)

Works on both integers and floating point numbers (except $\%$):

- Both arguments converted to the same type if necessary.
- The result type is the same as the (resulting) argument type.

Thus, $\text{int/int}$ yields integer division!

Other mathematical functions ($\sin, \exp, \sqrt{\text{...}}$) available in $\text{libm}$:

```c
#include <math.h>
cc prog.c -lm
```

Increment and decrement operators

Operators $++$ and $--$ available for adding/subtracting 1 from a variable:

```c
++i
--i
i++
i--
```

When used as statements, just short for:

```c
i = i + 1;
i = i - 1;
```

However:

- Prefix forms return the value of the variable after the update.
- Postfix forms return the value of the variable before the update.
Increment and decrement... (cont.)

Example:
```c
int i = 10, j = 10;
printf("%d, %d\n", ++i, j++);
```
will print 11, 10.

Never write expressions where the side affected variable occurs more than once, e.g:
```c
n = i++ + i * j;
```
The order of evaluation as well as exactly when `i` is updated is undefined.

Relational operators

The usual relational operators are available:
```c
<, >, <=, >=, ==, !=
```
Arguments should have same type and be arithmetic or pointers.
Type conversions are performed on arithmetic types if necessary.
Returns `true` (`≠ 0`, typically `1`) if relation holds, `false` (`0`) otherwise.

Examples:
```c
3 < 3.14159 yields 1.
3 != 7 % 4 yields 0.
```
Beware of:
- Comparing signed and unsigned ints.
- Comparing floats for equality.

Logical operators

Three logical operators available:
```c
& & – logical AND.
| | – logical OR.
! – logical NOT.
```
An argument = 0 represents `false`, any other argument `true`.

Evaluation order is left to right and ‘short-circuit’:
- Second argument to `& &` only evaluated if first ≠ 0.
- Second argument to `| |` only evaluated if first = 0.

Example:
```c
(x ! = 0) & & (1.0 / x < 0.1)
```

Bitwise operators and shifts

Bitwise operations, logical operations on integers considered as strings of binary digits, are also available:
```c
& – bitwise AND
^ – bitwise exclusive OR
| – bitwise OR
~ – bitwise NOT (1’s complement).
```
Example:
```c
fd = open("./foo",
  O_WRONLY | O_APPEND);
```
Left and right shifts: `<<, >>`. Right shift is arithmetic when applied to a signed type.
```c
7 << 2 = 28
(unsafe char 255) >> 2 = 63
(signed char -1) >> 2 = -1
```
Assignment operators

A plethora of assignment operators available:

\[ =, +=, -=, \ast=, /=, \%=(\text{and more}). \]

E.g., \( a += 7; \) short for \( a = a + 7; \).

The value of an assignment is the value assigned. Typical usage:

\[
a = b = c = d = 0;
\]

\[
\text{if } ((x = \text{foo}(...) ) != 13) ... 
\]

Some special operators

The conditional operator:

\[ \text{cond} \ ? \ \exp_1 : \exp_2 \]

First \( \text{cond} \) is evaluated. Then, if \( \text{true} \), \( \exp_1 \) is evaluated and its value is the value of the entire expression, otherwise \( \exp_2 \) is evaluated.

\[
\max = x > y ? x : y;
\]

The comma operator:

\[ \exp_1, \exp_2 \]

First \( \exp_1 \) is evaluated, then \( \exp_2 \). The value is the value of \( \exp_2 \).

\[
a = 0, \ b = x \ast y, \ i = 1
\]

The sizeof-operator:

\[ \text{sizeof}(\text{type}) \text{ or } \text{sizeof}(\text{object}) \]

Operator precedence and associativity

\[
\begin{array}{c}
[][], \rightarrow, \ & - \ & + \ & \ast \ & \% \ & \text{sizeof} \\
\& \ & \& \ & ^\ & | \ & \|
\end{array}
\]

Left to right

Exercise 3

Write a single expression that returns true (\( \neq 0 \)) if a year kept in the integer variable \( y \) is a leap year, false (0) otherwise.

A year is a leap year if it is divisible by 4 but not by 100, unless it happens to be divisible by 400, in which case it is a leap year.

Then write a suitable statement to print the result in 'human readable' form, e.g.:

\[
\begin{align*}
1900 & \text{ is not a leap year.} \\
1995 & \text{ is not a leap year.} \\
2000 & \text{ is a leap year.}
\end{align*}
\]

Can you do this using a single printf-statement? (Hint: ?)
Solution exercise 1

A solution using a for-loop:

```c
#include <stdio.h>
int i, n;
main()
{
    printf("n = "); scanf("%d", &n);
    printf(" i    i*i    i*i*i
" == " ===  =====  ======
" = " i, i*i, i*i*i);
    for (i = 1; i <= n; i++) {
        printf("%3d  %5d  %7d
", i, i*i, i*i*i);
    }
}
```

The loop could also have been written as a while-loop:

```c
i = 1;
while (i <= n) {
    printf("%3d  %5d  %7d
", i, i*i, i*i*i);
    i++;
}
```

Solution exercise 3

Logical expression that is true if \( y \) is a leap year, false otherwise.

\[
((y \mod 4 == 0 \&\& y \mod 100 != 0) \mid \\
(y \mod 400 == 0))
\]

Single printf-statement to print result:

```c
printf("%d is%s a leap year.\n", y, EXP ? "" : " not");
```

where EXP is the expression above.