Introduction: C
Part II

Dr Deepayan Bhowmik, Dr Andrea Bracciali

Important concepts so far

- The C language in the overall language evolution
- Features of C …
- … and differences with Java
  (targeted compiling vs run everywhere).
- Hello World.c !
- Some development environments
  set-up your own!

if anything unclear – please go back and revise (now!).

This lecture

- Basic types
- Type definition
- Control commands

Numeric data types

type	bytes (typ.)	range
char	1	-128 ... 127
short	2	-32,768...32,767
int, long	(2), 4	-2,147,483,648 to 2,147,483,647
long long	8	2^{64}
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits)

NB: long is both a **modifier** and a **type,** so for instance you can have `long long int`
Standards prescribe "at least", as in `long int` is **at least** 32 bits.

Remarks on data types

- Range differs !

  – `int` is "native" size, e.g., 64 bits on 64-bit machines, but sometimes `int` = 32 bits, `long` = 64 bits

- Also, `unsigned` versions of integer types
  - same bits, different interpretation

- `char` = 8 bits, but only true for ASCII
  - UTF-16 – 16 bits
  - UTF-32 – 32 bits

`printf("%d”,’a’);` returns 97

(note the implicit conversion from char to integer)

A bit extra: how is float stored?

- Take a number 15.875 as an example

  15 = 8 + 4 + 2 + 1 = 1111

  .875 = 0.5 + 0.25 + 0.125 = 2^-1 + 2 ^ -2 + 2^-3
- 15.875 as 1111.111 over one byte → 01111.111
- Normalise 1111.111 → 1.111111 x 2^3 ('.' moved left 3 positions)
- Now, "1." is removed, .111111 is significand, and 11(3) is exponent)
- Now, over 32 bits:
  1 bit for sign, 8 bits for exponent, and 23 bits for significand
- The number is positive, so sign bit is 0
- The exponent is 3 , add 127 (exponent bias) we get (over 8 bits):
  130 = 10000010
- The significand over 23 bits = 11111100000000000000000
- So the 32 bit binary value of 15.875 is

  01000001 01111110 00000000 00000000 (4 bytes in memory)

Example

```
#include <stdio.h>

int main(void)
{
  int nstudents = 0; /* Initialization, required */

  printf(“How many students does Stirling have ?\n>”);
  scanf (“%d”, &nstudents);  /* Read input */
  printf(“Stirling has %d students.\n”, nstudents);

  return 0;
}
```

$ How many students does Stirling have ?: 20000 (enter)
Stirling has 20000 students.

Explicit and implicit type conversions

- *Implicit:* e.g., `s = a` (int) + `b` (char)

  what does `printf("%d\n",'a'+3);` prints?

- *Promotion:* `char` -> `short` -> `int` -> …
- If one operand is `double`, the other is made `double`
- If either is `float`, the other is made `float`, etc.

- *Explicit:* type casting – `(type)`

```
mean = (double) sum / count;
```

- Almost any conversion does something – but not necessarily what you intended

Type conversion

```
#include <stdio.h>
void main(void)
{
  int i,j = 12;          /* i not initialized, only j */
  float f1,f2 = 1.2;

  i = (int) f2;          /* explicit: i <- 1, 0.2 lost */
  f1 = i;                /* implicit: f1 <- 1.0 */

  f1 = f2 + (float) j;   /* explicit: f1 <- 1.2 + 12.0 */
  f1 = f2 + j;           /* implicit: f1 <- 1.2 + 12.0 */
}
```

Type conversion

```
int x = 100000;
short s;
```

-32,768…32,767

```
s = x;
printf("%d %d\n", x, s);
```

100000 -31072

User-defined types

- **typedef** gives names to types:

```
typedef short int smallNumber;
typedef unsigned char byte;
typedef char String[100];

smallNumber x;
byte b;
String name;
```

C – no native boolean type

- C has no booleans in C89/C90
- Emulate as int or char, with values

  0 (false), and

  1 or non-zero (true)
- Allowed by flow control statements:

```
if (n = 0){ printf("something wrong");}
if (!0)   { printf("all fine");}
```

  top: the assignment returns zero -> false

  in C commands may return a value!

Defining your own boolean

```
typedef char boolean;
#define FALSE 0
#define TRUE 1
```

- Generally works, but beware:

```
check = (x > 0);
if (check == TRUE) {…}
```

- If `x` is positive, check will be non-zero, but may not be `1`.

Defining your own boolean

```
typedef int boolean;
#define FALSE 0
#define TRUE 1
```

- Generally works, but beware:

```
check = (x > 0);
if (check == TRUE) {…}
```

- If `x` is positive, check will be non-zero, but may not be `1`.

Enumerated types

- Define new integer-like types as enumerated types:

```
typedef enum {
  Red, Orange, Yellow, Green, Blue, Violet
} Color;
enum weather {rain, snow=2, sun=4};
```

- look like C identifiers (names)
- are listed (enumerated) in definition
- treated like integers
  - can add, subtract – even `color + weather`

Enumerated types

- Just syntactic sugar for ordered collection of integer constants:

```
typedef enum {
   Red, Orange, Yellow
} Color;
```

is like

```
#define Red 0
#define Orange 1
#define Yellow 2
```

… but the top one defines `Color` !

Structured data objects (FYI – more later)

- Structured data objects are available as

object	property
`array []`	enumerated, numbered from 0
`struct`	names and types of fields
`union`	occupy same space (one of)

Objects (or lack thereof)

- C does not have objects (C++ does)

- Variables for C's primitive types are defined very similarly:

```
short int x;
char ch;
float pi = 3.1415;
float f, g;
```

- Variables defined in {} block are active only in block, notion of *(execution) environment*

- Variables defined outside a block are global (persist during program execution), but may not be globally visible (static)

Data objects

- Variable = container that can hold a value
- default value is (mostly) undefined – treat as random
  - compiler may warn you about uninitialized variables if `-Wall` enabled.
- `ch = 'a'; x = x + 4;`
- Always pass by value, but can pass address to function:

  `scanf("%d%f", &x, &f);`

- & - unary operator – returns the (memory) address of a variable, e.g. &x (more to come – different from java).

Data objects

- Every data object in C has (IMPORTANT!):
  - a name and data type (specified in definition)
  - an address (its relative location in memory)
  - a size (number of bytes of memory it occupies)
  - visibility (which parts of program can refer to it)
  - lifetime (period during which it exists)

Data object creation

```
int x;
int arr[20];

int main(int argc, char *argv[]) {
  int i = 20;
  {int x; x = i + 7;}
}

int f(int n)
{
  int a, *p;
  a = 1;
  p = (int *)malloc(sizeof int);
}
```

Control structures

Same concept as in Java

- sequencing: `;`
- grouping: `{...}`
- selection: `if`, `switch`
- iteration: `for`, `while`

Sequencing and grouping

- statement1; statement2; statement n;
  - executes each of the statements in turn
  - a semicolon after every statement
  - not required after a {...} block

- { statements} {declarations statements}
  - treat the sequence of statements as a single operation (block)
  - data objects may be defined at beginning of block

The `if` statement

- Same as Java

```
if (condition1) {statements1}
else if (condition 2) {statements2}
else if (condition n-1) {statements n-1}
else {statementsn}
```

- evaluates statements until find one with non-zero result
- executes corresponding statements

The `if` statement

- Can omit {}, but careful (don't initially)

```
if (x > 0)
   printf("x > 0!");
   if (y > 0)
     printf("x and y > 0!");
```

The `switch` statement

- Allows choice based on a single value

```
switch(expression) {
  case const1:
    statements1;
    break;
  case const2:
    statements2;
    break;
  default:
    statements;
}
```

- Effect: evaluates integer expression
- looks for case with matching value
- executes corresponding statements (or defaults)

The `switch` statement

```
Weather w;
switch(w) {
   case rain:
     printf(“bring umbrella”);
   case snow:
     printf(“wear jacket”);
     break;
   case sun:
     printf(“wear sunscreen”);
     break;
   default:
     printf(“strange weather”);
}
```

Repetition

- C has several control structures for repetition

Statement	**repeats an action…**
while(c) {}	zero or more times, while condition is ≠ 0
do {…} while(c)	one or more times, while condition is ≠ 0
for (start; cond; upd)	zero or more times, with initialization and update

The `break` statement

- break allows early exit from one loop level (exit from the enclosing environment)

```
for (init; condition; next) {
  statements1;
  if (condition2)
    break;
  statements2;
}
```

The `continue` statement

- `continue` skips to next iteration, ignoring rest of loop body
- does execute `next` statement, e.g. the case `x++`

```
for (init; condition1; next) {
    statement1;
    if (condition2)
      continue;
    statement2;
}
```

- if true, does not execute `statement2;` and jumps to the next iteration.

```
printf ("starting loop:\n");
for (int n = 0; n < 7; ++n) {
  printf ("in loop: %d\n", n);
  if (n == 2) continue;
  printf ("    survived first guard\n");
  if (n == 4) break;
  printf ("    survived second guard\n");
  }
printf ("end of loop or exit via break\n");
```

This will lead to following output:

```
starting loop:
in loop: 0
    survived first guard
    survived second guard
in loop: 1
    survived first guard
    survived second guard
in loop: 2
in loop: 3
    survived first guard
    survived second guard
in loop: 4
    survived first guard
end of loop or exit via break
```