Numeric data types

type	bytes (typ.)	range
char	1	-128 … 127
short	2	-32,767…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)

1

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

2

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
- Represent 15.875 as 1111.111 → 01111.111
- Move the '.' left 3 positions: 1111.111 = 1.111111 x 2^3
- As per the standard, remove "1." (111111 is significand, and 11(3) is exponent)
- Now construct a 32 bit binary number: 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 130 = 10000010
- The significand = 11111100000000000000000
- So the 32 bit binary value of 15.875 is
  0 10000010 11111100000000000000000 (4 bytes in memory)

Example

```
#include <stdio.h>

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

  printf("How many students does Stirling have ?:");
  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 conversions

- Implicit: e.g., `s = a` (int) `+ b` (char)
- 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)`*
- Almost any conversion does something – but not necessarily what you intended

5

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 */
}
```

6

Type conversion

```
int x = 100000;
short s;

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

100000 -31072

7

C – no booleans

- 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");
}
```

- Assignment returns zero -> false

8

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;
```

9

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`.

10

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`
  - can't print as symbol (unlike Pascal)
  - but debugger generally will

11

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
```

12

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

13

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);
```

14

Data objects

- Every data object in C has
  - 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)

15

Data objects

- Unlike scripting languages and Java, all C data objects have a fixed size over their lifetime
  - except dynamically created objects
- size of object is determined when object is created:
  - global data objects at compile time (data)
  - local data objects at run-time (stack)
  - dynamic data objects by programmer (heap)

16

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);
}
```

17

Control structures

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

18

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

19

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

20

The `if` statement

- Can omit {}, but careful

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

21

The `switch` statement

- Allows choice based on a single value

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

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

22

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");
}
```

23

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

24

The `break` statement

- break allows early exit from one loop level

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

25

The `continue` statement

- `continue` skips to next iteration, ignoring rest of loop body
- does execute `next` statement

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

- often better written as `if` with block

26

Structured data objects

- 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)

27