CS2850 Operating System Lab

Week 2: Types, Variables, Functions

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

Variables

Functions

Types

Composite objects

Programs

C programs consist of functions and variables

A function is a set of statements that specify the operations to be performed

A variable is a location in storage with three attributes:

- an identifier or *name*,
- a storage class determining the variable *lifetime*, and
- a type determining the *meaning* of the value found in the identified storage

Declaration

Functions and variables should always be declared before they are used, i.e. you need to write

```
storage_class variable_type  variable_name;
```

The lifetime of a variable is specified by *where* it is declared (and by its *storage class*).

The declaration of a variable does not initialize it automatically.[1]

[1] But you can separate their declaration and definition.

Example

```
#include <stdio.h>
void printInt(int j);
int main() {
  int i;
  i = 1;
  printInt(i);
}
void printInt(int k) {
  printf("i=%d\n", k);
}
```

Types, names, storage classes

The variable *type* specifies how the stored bytes should be *interpreted*.

Variable names are sequences of letters and digits used for referring an object, i.e. `i` or `printInt`.

There are two important storage classes[2]:

- automatic: the variable is local to the block and reinitialized every time the function is called and
- static: the variable keeps its value across different function calls.

[2] By default, *local* variables are automatic, *global* variables are static.

Functions

A function is defined by writing

```
return_type  function_name(arguments) {statements};
```

where

- `return_type` is the *type* of the return value,
- `function_name` is the function name (used to *call* it,
- `arguments` denotes the *name and type* of the function parameters, and
- `statements` is the set of *operations* to be performed.

Function calls

A function is called by writing its name

```
f(argument_value_1, argument_value_2, ... );
```

where `argument_value_I` is the value of parameter `I` (at the right place).

In the function declaration, parameter names are *arbitrary*. Only their *type* is strictly required (see the code above for an example).

Types

There are a few basic data types:

- `char` a single character, e.g a
- `int` an integer, e.g. -1234
- `unsigned int` a positive integer, e.g. 12345
- `float` a single-precision floating point, e.g. $23,923$, and
- `void`, an empty set of values [3].

The operator `sizeof()` returns the (architecture dependent) number of bytes of any given basic type.

[3] `void` is not a regular type, e.g. you cannot declare a `void` variable.

Example

Run this program to check the size of the basic C types on your system

```
#include <stdio.h>
int main() {
  printf("sizeof(char)=%lu\n", sizeof(char));
  printf("sizeof(int)=%lu\n", sizeof(int));
  printf("sizeof(unsigned int)=%lu\n", sizeof(unsigned
      int));
  printf("sizeof(float)=%lu\n", sizeof(float));
}
```

More on `int` and `char`

Integer constants can be used to initialize a variable and given in binary (e.g. `int a = 0b01010101;`), decimal (`int a = 85;`), hexadecimal (`int a = 0x55;`) or octal (`int a = 0125;`).

Character constants are characters enclosed in single quotes, (e.g. `char a = 'x';` or `char a = '\n'`).

The numerical value of a `char` is the ASCII integer code corresponding to it, e.g. check the output of

```
printf("ASCII(r)=%d\n", 'r');
```

Printable characters, e.g. `r`, `$`, or `3`, are always positive but plain `char`'s can be signed or unsigned (depending on the machine).

Example

```c
#include <stdio.h>
int main() {
  char a = -123;
  printf("character=%c\n", (char) a);
  printf("signed int=%d\n", (int) a);
  printf("signed int(octal)=%o\n", (int) a);
  printf("unsigend int=%u\n", (unsigned int) a);
  printf("unsigend int(octal)=%o\n", (unsigned int) a);
}
```

Type casting specifiers as `(unisigned int)` force a *specified interpretation* of the following variable.

While -123 does not correspond to a *valid character*, the *stored bytes* (printed in octal) can be interpreted in different ways.

Operators

Arithmetic operators: `+`, `-`, `*`, `/`, and `%`.

Relational operators: `==`, `!=`, `>`, `>=`, `<`, and `<=`.

Logical operators: `&&` (and), and `||`(or).

The negation operator, "`!`", converts a non-zero operand into a 0, and a zero operand into 1 i.e. `if (!operand)` and `if(operand == 0)` are *equivalent*.

`%` is the modulus operator, e.g. $1\%2 = 1$, $2\%1 = 0$, $3\%2 = 1$, and $2\%3 = 2$.

Arithmetics

Operators may cause conversion of the value of an operand from one type to another.

For example, integer division may or may not truncate any fractional part, e.g. $3/2 = 1$ but $3/2. = 1.5$ and $3./2 = 1.5$.

The full list of arithmetic conversion rules is in Appendix A6.5 of The C Programming Langauge.

Non-basic types

There is a *conceptually infinite* class of derived types, which are built from the basic types in various ways.

The most important *derived types* are

- strings: null-terminated lists of `char`'s,
- arrays: lists of objects of a given type,
- functions: sets of statements returning objects of a given type,
- pointers: memory addresses of objects of a given type, and
- structures: general composite objects containing objects of different types.

The size of composite objects

The structure of composite objects, e.g. the number of entries of an array, should be declared before they are used and *cannot be changed at run time*.

The size in bytes of composite objects can be obtained using `sizeof`

```
printf("sizeof(char[10])=%lu\n", sizeof(char[10]));
printf("10 * sizeof(char)=%lu\n", 10 * sizeof(char));
```

Strings (1)

The program in the next slide,

i) *declares and initialises* a string (constant),

ii) *prints* the string using `printf` and the format specifier `%s`,

iii) prints a specific *character* of the string and non-initialised value *outside* the string.

Strings (2)

```
#include <stdio.h>
int main() {
  char *s = "hello, world\n";
  printf("s=%s", s);
  printf("s[7]=%c\n", s[7]);
  printf("s[100]=%c\n", s[100]);
}
```

Strings (3)

The program *does not crash* if you try to access uninitialised entries of `s` but the value stored there is *unpredictable*

The format specifier `%s` allows you to use `printf` for printing `s` with as a unit.

The program *knows where the string terminates* because strings are null-terminated lists of `char`'s, i.e. the last `char` of a string is *always* a `\0`.

Arrays (1)

The program in the next slide,

i) *declares* an array of 10 `int`, i.e. allocate the memory space to store 10 `int`,

ii) *loads* random integers to its entries, through component-wise assignments, and

iii) prints the vector components with `printf` and the format specifier `%d`.

Arrays (2)

```
#include <stdio.h>
#include <stdlib.h>
void loadVector(int a[], int size) {
  for (int i=0; i<size; i++) a[i] = ((float) 10 * rand
      ())/RAND_MAX;
}
void printVector(int a[], int size) {
  for (int i=0; i<size; i++) printf("a[%d]=%d\n",i,a[i
      ]);
}
int main() {
  int size = 10;
  int a[size];
  loadVector(a, size);
  printVector(a, size);
  printf("a[100]=%d\n", a[100]);
}
```

Arrays (3)

If you try to access uninitialized entries of `a` but the value stored there is *unpredictable*.

The program *let you access* uninitialized memory regions, i.e. `a[100]`.

You need a customized function to load and print `a` as a single unit.

`loadVector` and `printVector` do not know the length of the input because `a` is passed as the address of `a[0]`.