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 *lifetime* of the identified storage
- a type determining the *meaning* of the value found in the identified storage

declaration

`int main()` is a *special* function because the program starts executing from its first line

try to compile with `gcc -Wall -Werror -Wpedantic` a program with *more than one* or *without* any `main` to see what happens

in C functions and variables should always be declared before they are used (but you can separate declaration and definition)

example

```
1 #include <stdio.h>
2 void printInt(int j);
3 int main() {
4   int i;
5   i = 1;
6   printInt(i);
7 }
8 void printInt(int k) {
9   printf("i=%d\n", k);
10 }
```

note that, in C, the declaration of a variable does not initialise it automatically

comment out Line 5 to see what happens

variables

a variable is a *named* storage location with a specified *lifetime* and *meaning*

the variable properties are fixed in its declaration

```
storage_class variable_type  variable_name;
```

the lifetime of a variable is specified by *where* it is declared and by its *storage class*

types and names

the meaning of a variable is determined by its *type* and specifies how the stored bytes should be *interpreted*

this is why the type of a variable should be *always* be specified

names (or *identifiers*) are sequences of letters and digits used in the program for referring to objects, i.e. names like `i` or `printInt` in the example above

storage classes

two important storage classes are

- automatic: the variable is local to the block and reinitialised every time the function is called

- static: the variable keeps its value across different function calls

by default, *local* variables are automatic and *global* variables are static

uninitialised automatic variables have an *undefined value*

you can add the *qualifier* `const` to the declaration of any variable to specify that its value will not change

Example

```
#include <stdio.h>
static int j = 0;
void printInt(int i) {
  i++;
  printf("i=%d\n", i);
  j++;
  printf("j=%d\n", j);
}
int main() {
  int i = 1;
  printInt(i);
  printInt(i);
}
```

try to predict the output of the program above and what happens if you add the qualifier `const` to the two variables?*

*always compile your programs with `gcc -Wall -Werror -Wpedantic`

functions

a function is defined by writing

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

where

- **`return_type`** is the *type* of the value that the function returns
- **`function_name`** is the name of the function used to *call* it from other functions
- **`arguments`** is the name and type of the *parameters* that a calling function passes to the called function
- **`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, \dots );
```

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

parameter names are not important in the function declaration and only their *type* is required (see the code above for an example)

example

this is the first code in Kernighan & Ritchie's The C Programming Langauge

```
                ]
#include <stdio.h>
main() {
        printf("hello, world\n");
}
```

if you compile the program with `gcc -Wall -Werror -Wpedantic` you get the *error* message

```
gcc -Wall -Werror -Wpedantic kernighan.c
kernighan.c:3:1: error: return type defaults to 'int' [-Werror=implicit-int]
    3  main()  ^~~~
cc1: all warnings being treated as errors
```

did Kernighan and Ritchie make a mistake?

notes

what is missing is the return type of the `main` function but the book was written before ANSI C conventions changed

`printf` can be just called because it is declared and defined in `stdio.h`

the argument list in `main` is empty

the argument value in `printf("hello, world\n")` is the string constant `"hello, world\n"`

types of C (1)

the type of a variable specifies the *meaning of the bytes* stored at the associated location

there are a few basic data types in C
- `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. $23,923$

for *more* types available in C, check the full list in appendix A4 of The C Programming Langauge

`void` refers to an empty set of values but it is not a regular type, e.g. you cannot declare a `void` variable by writing `void n;`

types of C (2)

the number of bytes associated with each type may be architecture/compiler specific

you can use the operator sizeof() to get the number of bytes of any given basic type

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) conversions

the conversion of a ‘true’ character to a positive integer is called integral promotion in Appendix A6.1 of The C Programming Langauge

the conversion of an integer to an unsigned int (by left-truncation of its two’s complement representation) is called integral conversion in Appendix A6.2

floating points are converted to integral types by discarding their fractional part example

```
#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

the output of this program shows that -123 does not correspond to a *valid character* and that the *stored bytes* (printed in octal) can be interpreted in different ways

operators in C

arithmetic operators: `+`, `-`, `*`, `/`, `%`

relational operators: `==`, `!=`, `>`, `>=`, `<`, `<=`

logical operators: `&&` (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 $2\%3 = 2$

bitwise operators (1)

C has an *"and"*, an *"or"*, two *"shift"*, and a *"complement"* bitwise operators: `&`, `|`, `>>`, `<<`, `~`

bitwise operators can be used to manipulate *directly* the binary representation of variables

compile and run the program in the next slide to understand how they work

bitwise operators (2)

```
#include <stdio.h>
int main() {
  int i = 077;
  printf("i=%o\n", i);
  printf("~i=%o\n", ~i);
  printf("(i&i)=%o\n", i & i);
  printf("(i&~i)=%o\n", i & ~i);
  printf("(i|i)=%o\n", i | i);
  printf("(i|~i)=%o\n", i | ~i);
  printf("i<<1=%o\n", i<<1);
  printf("i<<8=%o\n", i<<8);
  printf("i>>1=%o\n", i>>1);
  printf("i>>8=%o\n", i>>8);
}
```

arithmetics

depending on the operands, 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 $3./2 = 1.5$

have a look at the full list of arithmetic conversion rules in Appendix A6.5 of The C Programming Langauge

there are no explicit or implicit conversion rules between a `void` and non-`void` types

non-basic types

besides the 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* of C 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
- structures: general composite objects containing objects of different types

a few notes on composite objects

composition of objects can be run recursively e.g. you can have an *array of arrays*, include an array in a `struct`, or let a pointer point to an *array of pointers*

in C, 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`

```
#include <stdio.h>
int main() {
  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

iv) prints the string *starting at position* 7

you will see more about strings and understand all details of the program in the next weeks

strings (2)

```
1 #include <stdio.h>
2 int main() {
3   char *s = "hello, world\n";
4   printf("s=%s", s);
5   printf("s[7]=%c\n", s[7]);
6   printf("s[100]=%c\n", s[100]);
7   printf("&s[7]=%s", &s[7]);
8   //s[7] = 'x'; //uncomment this line if you declare s as a char array
9 }
```

the *ampersand* operator, `&` returns the address of a variable and here is used to *read the string starting at* `s[7]`

what happens if you replace the statement in Line 3 with

```
char s[] = "hello, world"
```

strings (3)

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

you *cannot* change a specific entry of `s` because `s` is a string (and not an array of `char`'s)

the format specifier `%s` allows you to use `printf` for printing `s` with as a whole

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

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

C does not have built-in functions to manipulate arrays *as a unit*, e.g. you cannot print an array with a single call of `printf`

arrays are handled by referring to *the address or their first entry*

arrays (2)

```c
#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]);
  a[7] = 100;
}
```

note that the changes made in `a` by `loadVector` are *not discarded* when the function returns

arrays (3)

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

the program *does not block* the access to unallocated memory regions, i.e. `a[100]`

you can change a specific entry of `a` by adding a statement like `a[7] = '100';`

you need to write a customised function to load and print `a` as a single unit

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