More on I/O and types

`getchar, putchar, typedefs`

Dr Deepayan Bhowmik Dr Andrea Bracciali

In this lecture …

- A quick overview on other I/O primitives
- On type definition - important!

`getchar` and `putchar` for Character I/O

- `putchar` writes a single character:

```
putchar(ch);
```

- `getchar` reads and returns one character, including white-spaces (its integer value – typically ASCII - `ch` will often have type `int`):

```
ch = getchar();
```

`getchar` and `putchar` for Character I/O

- Using `getchar` and `putchar`, rather than `scanf` and `printf`, saves execution time.
- **Disadvantage** of `getchar` is that the input keeps buffering till enter is pressed.
  - In this process you have to hit enter first to send anything to your program (try it!).

`getchar` and `putchar` for Character I/O

Examples:

- scan (flush) all chars till the end of the line:

```
while (getchar() != '\n')
  ;
```

- skip all the next blank characters:

```
while (getchar() == ' ')
    ;
```

`getchar` and `putchar` for Character I/O

Examples:

- careful when mixing `scanf()` and `getchar()`:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

`scanf()` will leave behind any characters that weren't consumed during the reading of `i`, including (but not limited to) the new-line character. `getchar` will fetch the first leftover character.

Program: Determining the Length of a Message

- The `length.c` program displays the length of a message entered by the user:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

- The length includes spaces and punctuation, but not the new-line character at the end of the message.
- We could use either `scanf` or `getchar` to read characters; most C programmers would choose `getchar`.

length.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  int len = 0;

  printf("Enter a message: ");
  while (getchar() != '\n'){
    len++;
  }
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

Type Definitions

- The `#define` directive can be used to create a "Boolean type" macro:

```
#define BOOL int
```

- There's a better way: a ***type definition:***

```
typedef int Bool;
```

- `Bool` can now be used in the same way as the built-in type names. Example:

```
Bool flag;   /* same as int flag; */
```

Advantages of Type Definitions

- Type definitions can make a program more understandable.

- If the variables `cash_in` and `cash_out` will be used to store dollar amounts, declaring `Dollars` as

```
typedef float Dollars;
```

and then writing

```
Dollars cash_in, cash_out;
```

is more informative than just writing

```
float cash_in, cash_out;
```

Advantages of Type Definitions

- Type definitions can also make a program easier to modify.

- To redefine `Dollars` as `double`, only the type definition need be changed:

```
typedef double Dollars;
```

- Without the type definition, we would need to locate all `float` variables that store dollar amounts and change their declarations.

Advantages of Type Definitions

- Important:

Type definitions allow us to define complex types as sets of data and associated operations (in terms of functions in the case of C). See also *abstract data types*.

This is clearly connected to that path of the evolution of programming languages towards Object Oriented Languages.

Type Definitions and Portability

- Example: the "quantity" type:

  ```
  typedef int Quantity;
  ```
  and use this type to declare variables:

  ```
  Quantity q;
  ```

- When we transport the program to a machine with shorter integers, i.e. less bytes, we'll change the type definition:

  ```
  typedef long Quantity;
  ```
- Note that changing the definition of `Quantity` may affect the way `Quantity` variables are used.
- Important this does not happen in Java – Why?

Type Definitions and Portability

- The C library itself uses `typedef` to create names for types that can vary from one C implementation to another; these types often have names that end with `_t`.
- Typical definitions of these types:

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

- In C99, the `<stdint.h>` header uses `typedef` to define names for integer types with a particular number of bits.

The `sizeof` Operator

- The value of the expression

  `sizeof ( `***type-name***` )`

  is an unsigned integer representing the number of bytes required to store a value belonging to *type-name,* e.g.

  - `sizeof(char)` is always 1
  - on a 32-bit machine, `sizeof(int)` is normally 4.

- The `sizeof` operator can also be applied to **constants**, **variables**, and **expressions** in general.
  - If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`.

The `sizeof` Operator

- Printing a `sizeof` value requires care, because the type of a `sizeof` expression is an implementation-defined type named `size_t`.
- In C89, it's best to convert the value of the expression to a known type before printing it:

```
printf("Size of int: %lu\n",
       (unsigned long) sizeof(int));
```

- The `printf` function in C99 can display a `size_t` value directly if the letter `z` is included in the conversion specification:

```
printf("Size of int: %zu\n", sizeof(int));
```