Assignment 3

COMP9021, Session 2, 2014

1 Aims

The main purpose of the assignment is to let you practice the following programming techniques:

- perform operations on pointers to basic and more complex types;
- pass pointers as arguments to functions;
- perform operations on strings;
- parse some text.

2 General presentation

You will design and implement a program that will

- take as command line arguments descriptions of types, possibly associated with variable names, some of those variable names possibly referring to other variable names,
- check that those descriptions are correct according to some well defined set of rules, and
- output appropriate C representations of those descriptions.

Four descriptions of types of increasing complexity will be considered. The ultimate objective is to be able to parse descriptions of the fourth and last, most general kind, and you are advised to have those descriptions in mind when you develop your code for dealing with the simpler descriptions. Still having four levels of complexity will help you gradually design the solution to the general version of the problem, and you should be prepared to adopt the solution to descriptions of a given level when tackling descriptions of the next level.

We refer to each type description as a *phrase*. So the command line arguments are expected to make up a number of phrases. The appropriate C representation of any description is assumed to require no more than 100 characters, spaces included.

When the command line arguments are of the expected form, the program should output a number of lines equal to the number of phrases, the first line corresponding to the first phrase, the second line to the second phrase, etc.

3 Descriptions of the first level

3.1 Syntax

At this level, a phrase represents a basic type, that is, a type whose *default representation* is one of:

- char
- signed char
- unsigned char
- short
- int
- long
- long long
- unsigned short
- unsigned
- unsigned long
- unsigned long long
- float
- double
- long double

Such a phrase will be of the form:

`A`/`An` *`basic_type_description`* [*`variable_name`*]`.`

Of course, a phrase begins with `A` or `An` depending on whether the second word starts with a consonant or a vowel. The square brackets around *`variable_name`* indicate that the latter is optional; *`variable_name`* can be any valid C identifier, except for any of the words that can appear in the input, which includes `int`, `A`, `array`, `to`, etc. Note that a phrase ends in a full stop. As for *`basic_type_description`*, it follows the usual rules (see relevant lecture notes):

- `signed`, when not used alone, is optional, and when used alone, has `int` as default representation;
- `int`, when not used alone, is optional;
- the order of the words is irrelevant, but words cannot be duplicated (expect for the word `long` in the type `long long` of course).

The default representation should be output.

3.2 Example

Here is an example of how your program should behave, in which the command line arguments following `a.out` span the first two lines below:

```
$ a.out A signed. An int short x. A long unsigned long int. A double long x. A float.
A long long _007.
int
short x
unsigned long long
long double x
float
long long _007
```

4 Descriptions of the second level

4.1 Syntax

At this level, a phrase represents a simple array, pointer or function type, and is of one of the following forms:

- An array [variable_name] of *N* datum/data of type *basic_type_description.*
- A pointer [*variable_name*] to void.
- A pointer [*variable_name*] to a datum of type *basic_type_description.*
- A function [*variable_name*] returning void.
- A function [*variable_name*] returning a datum of type *basic_type_description.*

For the first form, *N* is a strictly positive number, and is of course followed by datum if equal to 1, and by data otherwise.

4.2 Example

Here are three examples of how your program should behave, in which the command line arguments following `a.out` span three, two and two lines, respectively:

```
$ a.out A long int long john. A pointer to a datum of type char. An array lisa2 of 10
data of type double. A pointer peter to a datum of type unsigned int. An array hanna
of 1 datum of type char signed.
long long john
char *
double lisa2[10]
unsigned *peter
signed char hanna[1]
$ a.out  An array of 290 data of type int. A function returning a datum of type
char. A function that_is_right returning void. A pointer may_be_2 to void.
int [290]
char ()
void that_is_right()
void *may_be_2
$ a.out A function really returning a datum of type float. A float x. A function x
returning a datum of type unsigned int.
float really()
float x
unsigned x()
```

5 Descriptions of the third level

5.1 Syntax

At this level, a phrase represents a complex array, pointer or function type, and is of one of the following forms:

- An array [*variable_name*] of *N* array/arrays of ...
- An array [*variable_name*] of *N* pointer/pointers to ...
- A pointer [*variable_name*] to a pointer to...
- A pointer [*variable_name*] to an array of ...
- A pointer [*variable_name*] to a function returning ...
- A function [*variable_name*] returning a pointer to ...

where ... allows for an arbitrary number of applications of legitimate type descriptions that follow the given format, and ends with the representation of a basic type. Be careful with the use of the singular versus plural in expressions involving pointers: we can have a pointer to a pointer, an array, a function or a datum, and we can have pointers to pointers, arrays, functions or data. Also note that only the leading type can be associated with a variable.

5.2 Example

Here are two examples of how your program should behave, in which the command line arguments following `a.out` span six and two lines, respectively:

```
$ a.out A pointer to a pointer to a pointer to a pointer to a datum of type int. An
array gee of 10 arrays of 20 arrays of 30 pointers to arrays of 40 arrays of 50
data of type int. An array of 20 pointers to data of type int. An array
that_is_an_array of 2000 pointers to arrays of 1000 data of type int. An array of
1 pointer to an array of 25 pointers to functions returning a pointer to a datum
of type int.
int ****
int (*gee[10][20][30])[40][50]
int *[20]
int (*that_is_an_array[2000])[1000]
int *(*(*[1])[25])()
$ a.out A function returning a pointer to an array of 3 pointers to pointers to arrays
of 60 pointers to functions returning a pointer to a datum of type int.
int *(*(**(*())[3])[60])()
```

6 Descriptions of the fourth level

6.1 Syntax

At this level, a phrase represents a complex array, pointer or function type with some representations possibly referring to others, by ending in:

a datum of type the type of *variable_name*.

Two rules have to be abided by:

- There can be no circular references, so a variable cannot refer to itself, or to a variable that refers to the former, etc.
- Variables referred to are unique. More precisely, a phrase can define a type with no associated variable, and many phrases can define types associated with a given variable as long as that variable is not referred to by any other phrase, but only one phrase can define a type associated with a given variable in case that variable is referred to by other phrases.

6.2 Example

Here are two examples of how your program should behave, in which the command line arguments following `a.out` span three and five lines, respectively:

```
$ a.out An array U of 50 pointers to pointers to data of type the type of X. A pointer
Z to a datum of type the type of U. An array Z of 70 pointers to data of type the
type of U. An int Z. An unsigned V. A char. A pointer X to a datum of type double.
double ***U[50]
double ***(*Z)[50]
double ***(*Z[70])[50]
int Z
unsigned V
char
double *X
$ a.out An array of 8 data of type the type of A1. A pointer to a function returning a
datum of type the type of P. An array A1 of 20 pointers to data of type the type of
C. A double long R. A pointer C to a function returning a pointer to a datum of type
the type of D. A pointer P to an array of 102 data of type the type of R. A function D
returning a datum of type the type of P.
long double (*(*(**[8][20])())())[102]
long double (*(*)())[102]
long double (*(*(**A1[20])())())[102]
long double R
long double (*(*(*C)())())[102]
long double (*P)[102]
long double (*D())[102]
```

7 Incorrect input

Whenever any part of the command line arguments is not of the expected form, the program should output `Incorrect input` and stop. Here are examples of tests with invalid command line arguments.

```
$ a.out A long float.
Incorrect input
$ a.out A int.
Incorrect input
$ a.out An int a.
Incorrect input
$ a.out A double
Incorrect input
$ a.out a char.
Incorrect input
$ a.out An int long int.
Incorrect input
$ a.out A signed float.
Incorrect input
$ a.out An int 2bad.
Incorrect input
$ a.out An int do-it.
Incorrect input
$ a.out An array of 20 datum of type int.
Incorrect input
$ a.out An array of +3 data of type int.
Incorrect input
$ a.out An array of data of type int.
Incorrect input
$ a.out An array of 1 int.
Incorrect input
$ a.out A pointer to an int.
Incorrect input
$ a.out A pointer to data of type int.
Incorrect input
$ a.out A function returning an array of 20 data of type int.
Incorrect input
$ a.out A pointer x to an array y of 10 data of type int.
Incorrect input
$ a.out An int x. A pointer to a datum of type the type of x. A double x.
Incorrect input
$ a.out A pointer x to a datum of type the type of y. A pointer y to a datum of type the
type of x.
Incorrect input
$ a.out An array of 10 data of type the type of x. An int.
Incorrect input
```

8 Format of the output for correct input

A single space is displayed after a word that is part of the description of a basic type when that word is followed by another word; no other space is output. In particular, there is no space before and after the first and last nonspace characters, respectively.

9 Assessment and submission

9.1 Assessment

Up to eight marks will reward correctness of solutions by automatically testing your program on some tests, all different to the provided examples. Each of the four levels of descriptions will be associated with a batch of tests worth 2 marks. Every batch will include tests with incorrect command line arguments, for which the expected output is `Incorrect input`.

Up to one mark will reward good formatting of the source code and reasonable complexity of the underlying logic as measured by the level of indentation of statements. For that purpose, the `mycstyle` script will be used, together with your customised `style_sheet.txt`, that you have to submit, and in which **Maximum level of indentation has to be set to 4** for this assignment. If the script identifies problems different to excessive indentation levels then you will score 0 out of 1. If the script identifies excessive indentation levels (which means that at least one line exhibits at least 5 indentation levels), then you will score 0.5 out of 1. If the script identifies no problem then you will score 1 out of 1. If your program attempts too little and contains too little code then the `mycstyle` script won't be used and you will score 0 out of 1.

Up to one mark will reward good comments, good choice of names for identifiers and functions, readability of code, simplicity of statements, compactness of functions. This will be determined manually.

Late assignments will be penalised: the mark for a late submission will be the minimum of the awarded mark and 10 minus the number of full and partial days that have elapsed from the due date.

9.2 Submission

Your program will be stored in one or more files with the main function stored in a file named `types.c`.

Go through `assignment_checklist.pdf` and make sure that you can tick all boxes (or at the least, are aware that you should tick them all). Then upload your files (the source code of your program and your customised `style_sheet.txt`) using WebCMS. Assignments can be submitted more than once: the last version is marked. Your assignment is due by November 2, 11:59pm.

9.3 Reminder on plagiarism policy

You are permitted, indeed encouraged, to discuss ways to solve the assignment with other people. Such discussions must be in terms of algorithms, not C code. But you must implement the solution on your own. Submissions are routinely scanned for similarities that occur when students copy and modify other people's work, or work very closely together on a single implementation. Severe penalties apply to a submission that is not the original work of the person submitting it.