Software and Embedded System Lab 2 (ELEE08022)

Data Type & Organisation in C language

Dr Jiabin Jia

Email: jiabin.jia@ed.ac.uk

Contents

- Data type in C language
- 1D and 2D array
- Structure

Python vs C

Key	Python Language	C Language
Variable declaration	No need of variable declaration for its use in Python	Variables have to be declared in C before get used in code further
Syntax	Syntax of Python is easy to learn, write and read	Syntax of C is harder than Python
Functions availability	Python has a large library of built-in functions	C has a limited number of built-in functions
Pointer	No pointers functionality available in Python	Pointers are available in C
Memory management	Python uses an automatic garbage collector for memory management	In C, the programmer has to do memory management on their own
Application	Python is a general-purpose programming language	C is generally used for hardware related applications

Common Data types in C

Type	Description	Size (bits)	Data range
`unsigned char`	Unsigned character	8	0 … 255
`char`	Signed character	8	-128 … +127
`unsigned int`	Unsigned integer	16 or 32	0 … 65,535 or 0 … 4,294,967,295
`int`	integer	16 or 32	-32,768 … 32,767 or -2,147,483,648 … +2,147,483,647
`float`	floating point	32	1.2E-38 … 3.4E+38
`double`	floating point	64	2.3E-308 … 1.7E+308

8 bits form 1 byte

sizeof (variable) can tell the size of variable in byte on your machine

Declaration Statement

- **Declaring an `int`eger named `total` :**

```
int total;
```

Statement means, "reserve storage space for an `int`eger variable called `total`".

- **Statement is terminated by a semi-colon `;`**
- **Declarations first! Straight after opening brace**

```
int main (void)
{
 char ch;                /* character */
 unsigned char reg; /* unsigned character */
 int num;                /* integer number */
 float average;          /* float number */
}
```

One-dimensional Array

A fixed number of *related* values with

- **the same data type**
- **stored using a single name for the group**

e.g. five temperature values, stored in an array declared as:

```
float temp[5];
```

Further examples of array declarations:

```
int volts[9023];

char code[42];
```

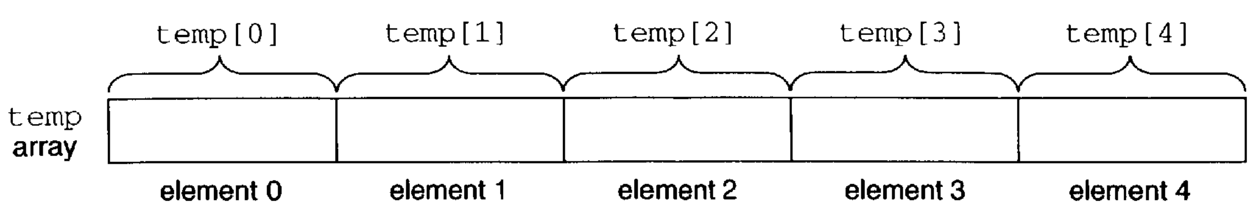
1D Array

- each element in an array is identified within the array by its **index**
- the **first** element has index `0`, the second `1`, and so on
- individual element can be referred to by **array name** and the element's **index** in brackets [ ]
- declare a float array with five element,

```
float temp[5];
```

- five single precision floating point elements:

```
temp[0] temp[1] temp[2] temp[3] temp[4]
```



1D array

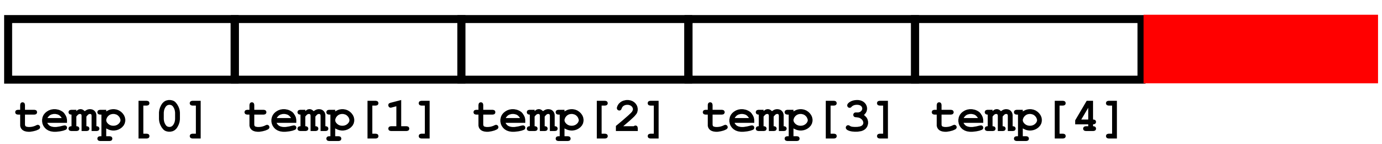
```
int i = 2;
temp[0] = 92.5;
temp[1] = temp[0] + 65.3;
temp[i] = temp[0] - temp[1];
temp[i + 1] = 79.0;
```

Program health warning

- **the size of any array in a declaration statement must be an constant. It cannot contain a variable.**

```
int amps[i];      /* WRONG */
```

- **C does not check if the index given corresponds to a valid array element, so do not touch the element you did not declare.**



Array Size & Initialisation

- Initialising an array to reserve storage for five elements `gallons[0]` to `gallons[4]`. Values are included in the declaration statement.

```
int gallons[5] = {16, 12, 10, 14, 11};
```

- The size of an array may be omitted. The following two declarations are equivalent:

```
char codes [6] = {'s', 'a', 'm', 'p', 'l', 'e'};
char codes []  = {'s', 'a', 'm', 'p', 'l', 'e'};
```

- Character arrays are used so often in C to store strings of characters that a special syntax exists to initialise them:

```
char codes[] = "sample";
```

- The last character (\0) is called the *nul* **character** and is automatically appended to the end of array.

codes[0]	codes[1]	codes[2]	codes[3]	codes[4]	codes[5]	codes[6]
s	a	m	p	l	e	\0

Two-Dimensional Array

- Declare a 2D array with 3 rows and 4 columns

```
int val[3][4];
```

- Access an element in a 2D array in the same way as for 1D arrays, e.g. `val[1][3]` is the element in row `1`, column `3` of the array `val` - as shown below:

	Col 0	Col 1	Col 2	Col 3	
Row 0	8	16	9	52	
Row 1	3	15	27	6	<---val[1][3]
Row 2	14	25	2	10	

- 2D arrays can be initialised from within their declaration statements. Braces can be used to separate individual rows:

```
int val[3][4] = { { 8, 16, 9, 52},
                  { 3, 15, 27, 6},
                  {14, 25, 2, 10} };
```

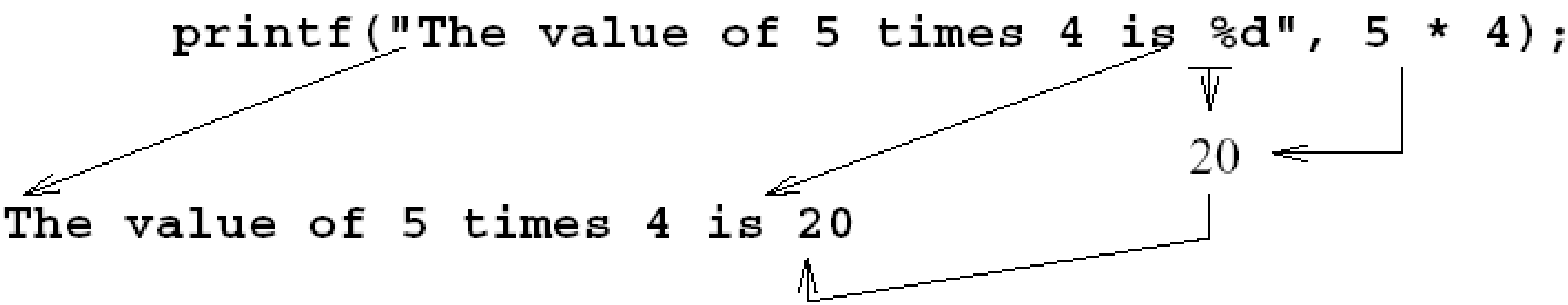
- Inner braces can be omitted:

```
int val[3][4] = {8,16,9,52,3,15,27,6,14,25,2,10};
```

Emitting Results - `printf()`

`printf("The value of 5 times 4 is %d", 5 * 4);`

- two arguments are passed to `printf()`
- the *first* argument, i.e. up to the comma, is a *string* (inside `""`) of characters which we wish to be printed
- `%d` is a *conversion control sequence* which controls printing of second argument (`5 * 4`) as a decimal number



Conversion Control Sequences: `printf()`

Control Sequence	*Prints Argument as:*
`%d`	signed decimal integer
`%ld`	long signed decimal integer
`%f`	floating point number
`%lf`	double precision number
`%e`	scientific notation (float)
`%le`	scientific notation (double)
`%s`	string (character array)
`%c`	character

run

```
printf("%f plus %f equals %f", 5.0, 7.0, 5.0 + 7.0);
```

display

```
5.000000 plus 7.000000 equals 12.000000
```

More printing: `printf()`

To help the compiler with arcane details of I/O, we have to include a file of standard property definitions

```
#include <stdio.h>
```

`#` must be at very beginning of line

Directive `#include` tells the compiler to literally include the contents of the file `stdio.h` at this point in the program

`<` and `>` characters tell the compiler to look in a special place for the file

`\n` (*backslash* and n) prints out a newline:

```
printf("\n");
```

Reading Data - `scanf()`

- **formatted input -** `scanf()` is the *input* equivalent of `printf()`
- `scanf()` reads from the keyboard and changes the value stored in a variable

  e.g. `scanf("%f", &num);`
- variable's ***address*** obtained by putting the symbol **&** immediately before the variable's name
- the ampersand symbol **&** is the ***address operator***
  - operation is ***"take the address of variable"***

Reading in 1D array values

If it is desired to *read-in* the initial values, then a `for` loops can be used. e.g.

```
int val[3], i;    /* declare array val and interger i */
for (i = 0; i < 3; ++i) /* index i starts from 0 */
   {                 /* and increases by 1 until 3 */
    scanf("%d", &val[i]);  /* read in integer value */
   }
```

This reads values and stores them in the array *`val`* in the same order as the elements' locations are organised internally.

Printing out 1D array values

```
for (i = 0; i < 3; ++i)
   {
    printf("%d ", val[i]);/* print ONE value at one time*/
   }
```

Reading in 2D array values

If it is desired to *read-in* the initial values, then a pair of nested `for` loops can be used. e.g.

```
int val[3][4], i, j;
for (i = 0; i < 3; ++i) {      /* outer - ROW loop */
  for (j = 0; j < 4; ++j) {    /* inner - COLUMN loop *
    scanf("%d", &val[i][j]);   /* read in integer value */
  }
}
```

This reads values and stores them in the array *`val`* in the same order as the elements' locations are organised internally.

Printing out 2D array values

```
for (i = 0; i < 3; ++i) {        /* ROW loop */
  for (j = 0; j < 4; ++j)        /* COLUMN loop */
    printf("%d ", val[i][j]);    /* print ONE value */
  printf("\n");                  /* \n at end of ROW */
}
```

Structure

- **An** **array** stores ***related items***:
  - but each **element** has an **identical data type**
- **A** **structure** stores ***related items*** of **any data type**
  - individual parts of a structure are called **members**
- A **structure** is a complex variable customised to the needs of a particular program
- Each **member** can be **any type** of *variable*
  - a structure can hold as many members as we wish
  - *arrays*, or *other structures*, can be members
- **Arrays of structures** can be created (c.f. 2D arrays)

Declaring and Defining Structures

- A structure must be **declared** and **defined** before use
- Keyword `struct` introduces declaration
- For example, consider a structure intended to hold a **date**:

```
struct {
            int day;
            int month;
            int year;
           } a_date;
```

- **Structure *defined* (storage allocated) with name `a_date`. Structure `a_date` has three members:**

  `day`, `month` and `year`

  which can each store an `int`eger.

Initialising structure members

```
struct {
          int day;
          int month;
          int year;
         } a_date;
```

- Access to **individual members** requires both **structure name** and **member name,** joined by a dot
- Individual members of `a_date` are:

  `a_date.day`, `a_date.month` and `a_date.year`
- We may assign values to each member by:

```
a_date.day    =   22;
a_date.month  =    1;
a_date.year   = 2021;
```

- Each of these members is an `int`eger and can be treated just like any other integer variable

```c
#include <stdio.h> /* input structure members */
int main(void)/* then print structure members */
{
   struct {
            int day;
            int month;
            int year;
            } a_date;

   scanf("%d%d%d", &a_date.day, &a_date.month,
                   &a_date.year);

   printf("Date is: %02d:%02d:%4d\n",
          a_date.day, a_date.month, a_date.year);
   return 0;
}
```

Running this program by:

```
Enter the date as DD MM YYYY: 22 01 2021
Output: Date is: 22:01:2021
```

Structure Templates – tag names

- We can **declare** a structure ***separately from*** the **definition** of an actual structure of this type by including a **tag name** ***before*** the list of structure members

  e.g. for *date* structure we have used before, declare:

```
struct date {          /* tag name date */
            int day;
            int month;
            int year;
            };
```

- Now use *tag name* **date** to **define** four structures:

```
struct date a_date, today, tomorrow, birthday;
```

- structure ***template declaration*** is often done ***globally***
  - — outside any function usually at top of program
  - — so new structure type available everywhere in program file
- structure instances of this type then ***defined*** inside functions

```
#include <stdio.h>  /* Example program */
struct date {  /* declare template for 'date' */
           int day;
           int month;
           int year;
          }; /* no variable name given after closing brace } */

int main(void)
{      /* declare and define a real structure */
  struct date a_date; /* structure instances */
  scanf("%d%d%d", &a_date.day,
                  &a_date.month,
                  &a_date.year);

  printf("Date is: %04d-%02d-%02d\n",
          a_date.year, a_date.month, a_date.day);
  return 0;
}
```

Running this program by:

```
  Enter the date as DD MM YYYY: 22 01 2021
  Output: Date is: 2021-01-22
```

Structure Initialisation

Like array initialisation, a structure can be initialised by following the definition with a list of initialisers:

```
struct date a_date = { 22, 01, 2021 };
```

More Complex Structures

Structures members usually have different data types

e.g. part of a *student record*

```
Name:
Matriculation Number:
Course Code:
Year of Course:
Mode of Study:
```

A possible declaration for the template of a student structure:

```
struct student  /* student structure template */
{
  char name[40+1];  /* student name - string  */
  long matric_no;   /* matriculation - long   */
  int  course_code; /* course code - integer  */
  int  course_year; /* course year e.g. 1 - 5 */
  char study_mode;  /* full/part time 'F'/'P' */
};
```

actual structure using this template for a student could then be declared and initialised as:

```
 struct student a_student =
     {"A. N. Other", 1606521, 8413, 2, 'F' };
```

- *arrays* can be members of a structure (`char name[]` above)
- members can even be ***other structures***, say next example

To incorporate a **date** member in a **student** structure:

```
struct date {/* template for 'date' structure */
  int day;
  int month;
  int year;
};

struct student {/* student structure template */
  char name[40+1];    /* student name - string */
  long matric_no;     /* matriculation - long  */
  int  course_code;   /* course code - integer */
  int  course_year;   /* course year eg. 1 - 5 */
  char study_mode;    /* full/part time 'F'/'P'*/
  struct date birth_date;     /* date of birth */
};

struct student a_student =
{"A.N.Other", 1606521, 8413, 2, 'F', {1, 4, 1988}};
```

Arrays of Structures

- structures are useful for lists of data - often called ***records***
- each structure holds the information for one record
- For example, to store the records of, say 5 students

— we use an ***array of 5 structures***:

```
struct student {/* student structure template */
   char name[40+1];   /* student name - string */
   long matric_no;    /* matriculation - long  */
   int  course_code;  /* course code - integer */
   int  course_year;  /* course year eg. 1 - 5 */
   char study_mode;   /* full/part time 'F'/'P'*/
};
/* declare array of 5 student structures */
struct student students[5];
```

- declares ***array*** of 5 elements: `students[0]` to `students[4]`
- each ***element*** is a **structure** of type `student`
- each ***structure*** has `5` ***members*** - ***array of structures*** stores 25 items
- access individual items as:

```
    students[3].matric_no = 1402316;
```

Storing values in structures

- Data can be stored by reading it in at run time, or by initialisation at load time
- Initialising an array of structures is similar to the initialisation of a 2D array

```
#include <stdio.h>   /* Example program */
#define NSTUD  5     /* number of students in group */
#define NAMLN 40     /* max name length */

struct student {            /* declare template */
  char name[NAMLN+1];       /* student name - string */
  long matric_no;           /* matriculation - long  */
  int  course_code;         /* course code - integer */
  int  course_year;         /* course year eg. 1 - 5 */
  char study_mode;          /* full/part time 'F'/'P'*/
};

int main(void)
{
  int i;
  /* define & initialise actual array of structures */
  struct student students[NSTUD] =
  { { "A Student",     1601023, 8413, 2, 'F' },
    { "A N O Student",  901429, 8413, 2, 'F' },
    { "N X T Student", 1614945, 8402, 2, 'F' },
    { "A P T Student", 1623467, 9300, 2, 'P' },
    { "T H E Last",    1621732, 8413, 2, 'F' }
  };
```

```c
. . .
int main(void)
{
   int i;
               /* define & initialise actual array of structures */
   struct student students[NSTUD] =
   {
      { "A Student",      1601023, 8413, 2, 'F' },
      { "A N O Student",   901429, 8413, 2, 'F' },
      { "N X T Student", 1614945, 8402, 2, 'F' },
      { "A P T Student", 1623467, 9300, 2, 'P' },
      { "T H E Last",     1621732, 8413, 2, 'F' }
   };
   printf("Name                          MatricNo Course Year F/PT\n");
   for (i = 0; i < NSTUD; ++i) {
     printf("%-20s %07ld  %4d  %2d    %c\n",
          students[i].name,
          students[i].matric_no,
          students[i].course_code,
          students[i].course_year,
          students[i].study_mode);
   }
   return 0;
}
```

```
Name                 MatricNo Course Year F/PT
A Student             1601023  8413   2    F
A N O Student         0901429  8413   2    F
N X T Student         1614945  8402   2    F
A P T Student         1623467  9300   2    P
T H E Last            1621732  8413   2    F
```