A practical guide to computer simulation II

Alexander K. Hartmann, University of Göttingen

May 9, 2003

supplement:
Strings are arrays of char. End of string is indicated by a 0.

```
char name[100];            /* up to 99 characters */
```

Functions to handle strings are defined in `string.h`.
Example

```
strcpy(name, "Robert Smith");                /* copies text into string */
printf("length(%s)=%d\n", name, strlen(name));         /* prints length */
```

Useful: `sprintf(string, <format string>, ....)` (defined in `stdlib.h`) works like `printf` but prints to string instead of printing to standard output.

2.7 Structures, self-defined data types

Used to group several elements into one data type.
Example for definition

```
struct particle
{
  double       mass;         /* in kg                        */
  int         charge;        /* in units of e                */
  double[3] position;        /* position in space. in meters */
}
```

Variable declaration:

```
struct particle particle1;
```

Access

```
particle1.mass = 9.109e-31;
particle1.charge = 1;
particle1.position[0] = -2.3e-3;
```

For easier use, define own datatypes. Write `typedef` followed by a "normal" declaration, e.g.

```
typedef double vector_t[3];                     /* new type 'vector_t' */
typedef struct particle particle_t;          /* new type 'particle_t' */
...

vector_t velocity;                    /* velocity is of type 'vector_t` */
particle_t electron;  /* variable 'electron' is of type 'particle_t' */
```

Convention: collect all types in extra header (.h) file.

2.8 Pointers

Pointer = Address in memory of a variable.

Declartion: <type> *ptr makes ptr an address of variables of type <type>.

&-Operator gives adress of a variable: & <variable>.

*ptr = content of the variable where ptr points to. i.e. one can set the content by *prt= <expression>. Example:

```
int number, *address;

number = 50;
address = & number;
*address = 100;
printf("%d\n", number);
```

will print: 100.
Arrays = pointers, int value[10] ⇒ value= address of the beginning of the array, i.e. of variable[0]. Both int value[0] and int *value2 define an pointer to int variables, but for value an array of length 10 is reserved in memory and value points to the beginning of the array. value2 is NOT assigned any value initially.

Access: value[5] is equivalent to *(value+5).

If a pointer points to a structure, access to elements by -> operator.

```
struct particle *atom;
...
atom->mass = 2.0;
```

Pointers can be used to generate connections between different variables, e.g. to construct complex datatypes (lists or trees, see below).

Pointers can be used to return value from a function without using the return statment. (Useful in case of many return values)

```
void add_numbers(int n1, int n2, int *result_p)
{
  *result_p = n1+n2;
}
```

Note: the pointer result_p itself cannot be changed in add_number, only the content of the memory where result_p points to.

2.9 File handling

Useful: write results of simulations, configuration files etc directly on disk.
General recipe:

- Open file using fopen obtaining a FILE pointer.

- Write data using `fprinf` (equivalent to `printf`) but to file instead to standard output.
- close file using `fclose`

Example: write configuration file

```
struct particle atom[100];                              /* simulation data */
int t, cfg_id;          /* auxiliray counter, counter for filenames */
FILE *file_p;
char filename[100], command[200];                    /* auxilary strings */

...

sprintf(filename, "run%04d.cfg", cfg_id);
file_p = fopen(filename, "w");                 /* open file for WRITING */
fprintf(file_p, "# id     x       y       z\n");     /* write header */
for(t=0; t<100; t++)                                      /* write data */
  fprintf(file_p, "%d   %f %f %f\n", t, atom[t].position[0],
          atom[t].position[1], atom[t].position[2]);
fclose(file_p);
sprintf(command, "gzip -f %s", filename);              /* compress file */
system(command);
```

At the end: file automatically compressed ⇒ saves disk space.
To read a configuration file:

```
char *pos, line[200];    /* auxilary pointer, line to read from file */
double x,y,z;                              /* for reading atom positions */
int id;                                       /* for reading id of atoms */

...

sprintf(filename, "run%04d.cfg.gz", cfg_id);
sprintf(command, "gzip -df %s", filename);          /* decompress file */
system(command);
pos = strstr(filename, ".gz");                    /* strip .gz appendix */
*pos = 0;
file_p = fopen(filename, "r");                        /* open for READING */
while(!feof(file_p))            /* read while not end of file reached */
{
  fgets(line, 100, file_p);                             /* read one line */
  if(feof(file_p))
    continue;
  if(line[0] == '#')                 /* ingnore lines starting with '#' */
    continue;
  sscanf(line, "%d %lf %lf %lf", &id, &x, &y, &z);     /* obtain data */
  atom[id].position[0] = x;
  atom[id].position[1] = y;
  atom[id].position[2] = z;
}
fclose(file_p);
```

```
sprintf(command, "gzip -f %s", filename);      /* compress file again */
system(command);
```

Remark: First reading using fgets, then obtaining the data by sscanf is safer than using fscanf.

2.10 Dynamic memory allocation

Often one does not know the size of an array at compile time.
⇒ Allocate arrays dynamically with malloc(<number of bytes>) (defined in stdlib.h). Use sizeof(<data type>) to determine array size.
Example:

```
struct particle *atom2;
int num_atoms;
...
atom2 = (struct particle *) malloc(num_atoms*sizeof(struct particle));
```

Now atom2 can be used like a normal array.
When the array is not used any more, it can be given back to the memory management:

```
free(atom2);
```

One should never forget to free memory, otherwise the program might grow to an enormous size.
Allocating matrices of variable size is done in two steps, example:

```
int num_rows, num_columns, row;
double **matrix;
...
matrix = (double **) malloc(num_rows*sizeof(double *));
for(row=0; row<num_rows; row++)
  matrix[row] = (double **) malloc(num_columns*sizeof(double));
```

Freeing:

```
for(row=0; row<num_rows; row++)
  free(matrix[row]);
free(matrix);
```