C Programming under Linux

P2T Course, Martinmas 2003–4
C Lecture 10

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

r.kaiser@physics.gla.ac.uk

Summary

- Memory Allocation
- Random Numbers
- Bits and Pieces
- Famous Last Words

`http://www.physics.gla.ac.uk/~kaiser/`

Dynamic Memory Allocation

- So far we have always declared all the variables we would need during the running of the program right at the beginning. This can also be referred to as static memory allocation.
- What happens if the program finds out that it needs a bit more storage space than we allocated ? Let's say we have an array that we are going to fill as the result of a calculation. What if we don't know from the beginning how long it should be ?
- We can of course allocate just a lot of space, i.e. make the array huge from the beginning. However, this wastes a lot of space and once it's not one array but 100 arrays, we may run out of space.
- The solution lies in dynamic memory allocation. The C library contains functions that can allocate storage space at runtime.
- This allows the program to react to demands from the outside and makes it more flexible, e.g. also when it communicates with other programs while running.

Allocating Memory with `malloc`

- The most basic memory allocation function is `malloc`.
- The function prototype for `malloc` is

```
void *malloc(unsigned int);
```

- The argument of malloc is the number of bytes to allocate. If malloc runs out of memory it returns a null pointer.
- `malloc` returns a pointer to type `void`, i.e. a generic pointer that can point to anything, e.g. a string or a structure. What `malloc` doesn't give us is a normal variable with a variable name.
- The returned pointer can be cast to a specific type, but implicit conversion on assignment to a previously defined pointer is allowed in Standard C.
- In C++ the type cast is required.
- In old books you may find that `malloc` returns a pointer to character. That's outdated.

Allocating Memory with malloc - Example

- Suppose we are working on a data base program, using a structure called person:

```
struct person {
    char name[30];      /* name of the person       */
    char address[30];   /* where he/she lives       */
    int age;            /* how old he/she is        */
    float height;       /* how tall is he/she in cm */
}
```

- Instead of using an array of structures person we can now use malloc to create a person when we need one:

```
/* Pointer to a new person structure to be allocated from the heap */
struct person *new_item_ptr;
new_item_ptr = malloc(sizeof(struct person));
```

- We don't even have to know the size of a person in bytes, the sizeof function will figure this out for us.
- The heap is the total of the available memory space, which is large but not infinite. When malloc runs out of space it returns a NULL pointer - and one should check if malloc was successful.

De-allocating Memory with `free`

- If we keep allocating memory we will eventually run out. That's why there also has to be a function that de-allocates or frees the memory again. This function is called `free`.
- The function prototype is void free(void *ptr);
- In addition the pointer should be set to `NULL`, but doesn't have to. However, it does save us from trying to use freed memory.
- Here is an example using `malloc` to get memory and `free` to dispose of it:

```
cont int DATA_SIZE = (16 * 1024) ; /* Number of bytes in the  buffer */
void copy(void)
{
    char *data_ptr;   /* pointer to large data buffer */
    data_ptr = malloc(DATA_SIZE);   /* get the buffer */
    /*
     * use the buffer to do something, i.e. copy a file
     */
    free(data_ptr);
    data_ptr = NULL;
}
```

Other Memory Allocation Functions

- The memory allocation functions are, among others, declared by the header file `stdlib.h`. So if you want to use any of them you have to include it. The other allocation functions are:
  - `calloc(count, size)` allocates a region of memory large enough for an array of `count` elements, each of `size` bytes. The region of memory is declared bitwise to zero.
  - `realloc` changes the size of an already existing pointer while preserving it's contents. If necessary, the contents are copied to a new memory region; a pointer to the (possibly new) memory region is returned.
  - `mlalloc` - same as `malloc` but parameter is `unsigned long`
  - `clalloc` - same as `calloc` but parameters are `unsigned long`
  - `cfree` frees memory allocated by `calloc` or `clalloc`.
- For details please have a look at `man` pages, `info` pages, the web or a C book.

Random Numbers

- On occasion you will want to be able to generate random numbers, lots of them.
- A typical application in particle and nuclear physics are so-called Monte Carlo simulations that use large numbers of simulated physics processes with a statistical distribution of parameters. They are important to model the performance of a detector or to understand the background under the peak that signifies a new particle.
- C offers a function in stdlib.h that can be used for this purpose.
- The function prototype is simply

  ```
  int rand(void);
  ```

  It returns a pseudo-random integer number between 0 and `RAND_MAX`.
- `RAND_MAX` is defined in `stdlib.h` as 2147483647.

Random Numbers cont.

- The numbers returned by `rand` are pseudo-random, meaning that the distribution is flat, i.e. the probability for any given interval of the same size is about equal.
- However, the sequence is deterministic, i.e. repeated calls to `rand` in one program give different numbers, but if the program writes out 1000 'random' numbers they will always be the same in the same order.
- The function `srand` can be used to change the so-called seed, i.e. the input value of the algorithm that works behind `rand`.
- The function prototype here is

```
void srand(unsigned seed);
```

- The same argument in `srand` will produce the same chain of values coming from `rand`. So for 'really' random numbers one can use `rand` to produce a seed for `srand` and then produce a large set of numbers.

Integer and Float Random Numbers - Example

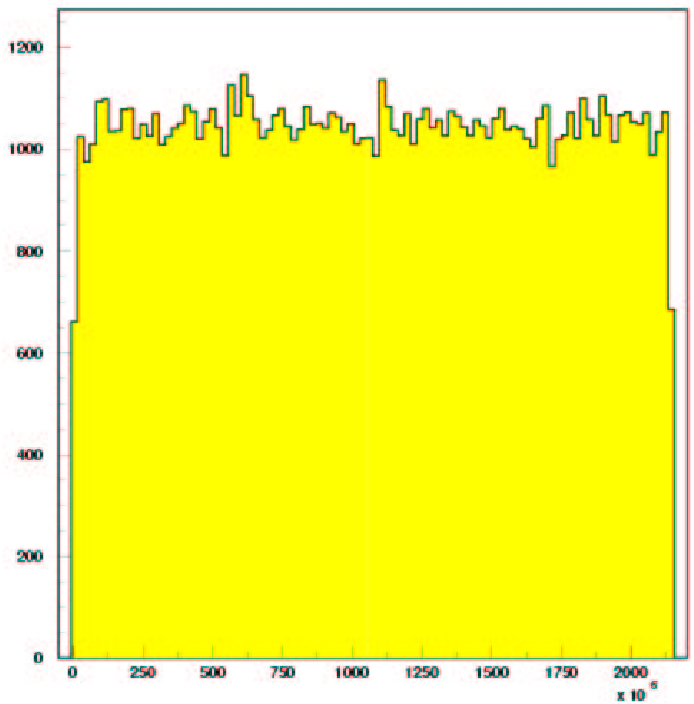
Integer and floating point random numbers (`random.c`).

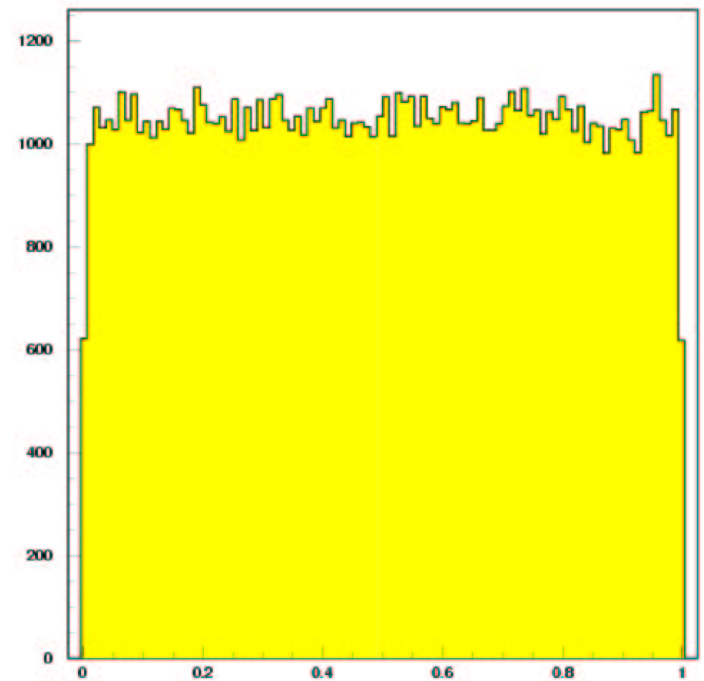
```
/* integer random numbers */

f_ptr = fopen("random1.dat", "w");

for (i=0; i<100000; i++) {
  fprintf(f_ptr, "%d\n", rand());
}
fclose(f_ptr);  /* close file */



/* random numbers between 0 and 1 */
/* flat distribution              */

f_ptr = fopen("random2.dat", "w");

for (i=0; i<100000; i++) {
  number = (float) rand()/RAND_MAX;
  fprintf(f_ptr, "%f\n", number);
}
fclose(f_ptr);  /* close file */
```

- The standard output of `rand` consists of integers. However, you may want to have floating point values between 0 and 1 instead.
- This can be constructed in a simple fashion (see example).
- `stdlib.h` also contains additional functions that do this for you, but `rand` is Standard C.

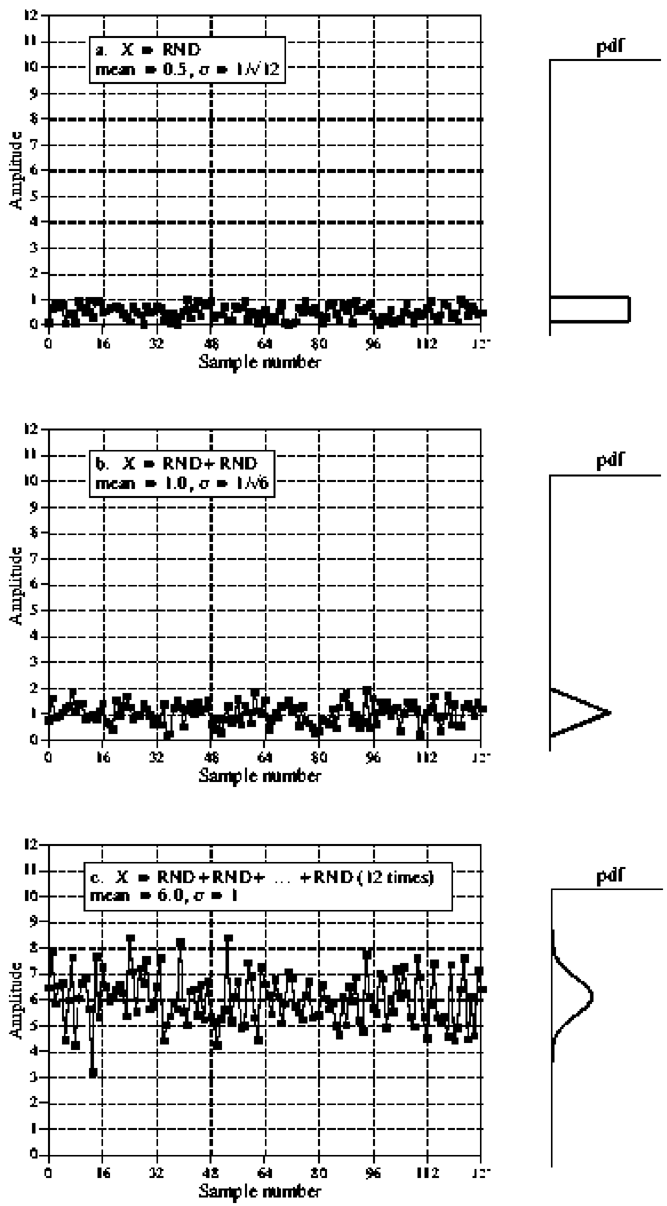
Integer and Float Random Numbers - Example

Integer and floating point random numbers (`random.c`), histograms of random number output.





Gaussian Random Numbers

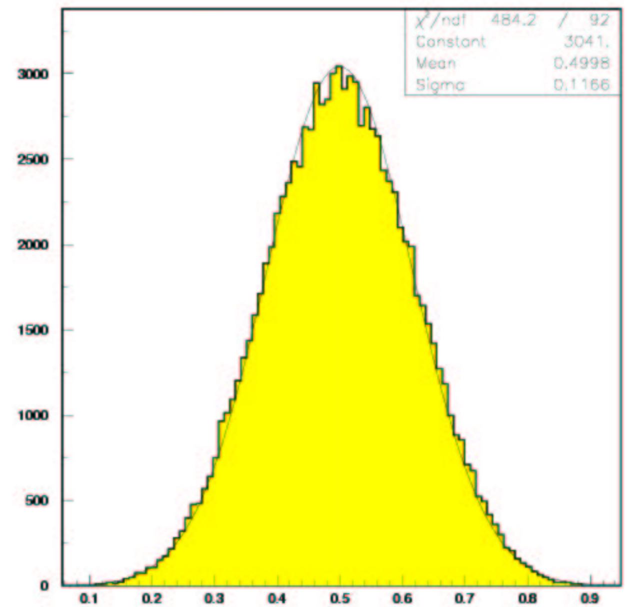


- The key to a Gaussian distribution is that it can be created from a flat distribution by sampling.
- Already the sum of 6 random numbers from a flat distribution is very close to a Gaussian.

Gaussian Random Numbers - Example

Gaussian random numbers (`random.c`), histogram of random number output with Gaussian fit.

```
/* random numbers between 0 and 1   */
/* Gaussian distribution around 0.5 */

f_ptr = fopen("random3.dat", "w");

for (i=0; i<100000; i++) {
  number = 0.0;
  for (j=0; j<6; j++) {
    number += (float) rand()/RAND_MAX;
  }
  fprintf(f_ptr, "%f\n", number/6.0);
}
fclose(f_ptr);  /* close file */
```



(The histograms on this slide and the previous one were produced using PAW, a data analysis package from CERN which is available for Linux, but unfortunately based on FORTRAN. The successor is called root and based on C++.)

The Ternary Operator '?:'

- We have seen unary operators (like '++') that act on one variable and binary operators (like '+') that require two.
- There is also a single ternary operator, the operator '?:'. It's use is similar to an `if/then/else` construction, but in contrast to `if/then/else` '?:' can be used inside of an expression.
- The general form is

  (expression) ? value1 : value2

- For example, the following construct assigns to `amount_owed` the value of the balance or zero, depending on the amount of the balance:

  amount_owed = (balance < 0) ? 0 : balance;

- The following macro returns the minimum of it's two arguments:

  #define min(x,y) ((x) < (y) ? (x) : (y))

The `do/while` Statement

- The `do/while` statement has the following syntax:

```
do {
      statement;
      statement;
} while (expression);
```

- The program will loop, test the expression, and stop if the expression is false (0).
- This construct will always execute at least once.
- It's not frequently used in C programming. Most programmers prefer to use a `while/break` combination.

The Comma (,) Operator

- The comma operator (,) can be used to group statements, for example instead of

```
if (total < 0) {
   printf("You owe nothing\n");
   total = 0;
}
```

  we cam also write

```
if (total < 0)
   printf("You owe nothing\n"), total = 0;
```

- In most cases curly braces {} should be used instead. The only places where the comma operator is useful is for the declaration of a couple of simple variables in one line

```
int i, j, k;  /* a couple of counters */
```

  and in `for` statements:

```
for (two = 0, three = 0;
     two < 10;
     two += 2, three +=3)
         printf("%d %d\n", two, three);
```

Finally: The `goto` Statement

- You thought perhaps that `goto` has been banished to forever live only in damp dungeons of BASIC. You were wrong. It exists, but you don't every have to use it.
- For those infrequent times when you actually want to use a `goto`, the correct syntax is

  `goto label;`

  where `label` is a statement label.
- Example:

```
for (x = 0; x < X_LIMIT; x++) {
    for (y = 0; y < Y_LIMIT; y++) {
       if (data[x][y] == 0)
          goto found;
    }
}
printf("Not found\n");
exit(8);

found:
   printf("Found at (%d,%d)\n", x, y);
```

Things not to do

- '=' is not the same as '==' ! Don't mix them up !
- C starts counting at '0' not at '1' ! Stick to this, don't start at 1 !
- `matrix[10,12]` is not the correct C syntax !
- Don't forget the `\0` at the end of a string !
- Don't use 8 spaces at the begin of a line in a Makefile instead of a tab !
- Never put an assignment inside another statement !
- Don't make a block of code inside or a function longer than a few (about 3) pages !

Things to do

- Use `++` and `--` on lines by themselves.
- One variable declaration per line.
- Use comments. Lots of them. Useful ones.
- Make variable names explicit. 'total' is better than 't'.
- Always put a `default:` case into a `switch` statement.
- Use `const` instead of #define wherever possible.
- Put () around each argument of a preprocessor macro.
- Generally use enough () if you don't know the precedence rules. (And you haven't learned them here...).
- Use a Makefile.
- Make a backup of your work. Then make another one. If it's important make one more. For my thesis I made one on another disk, one on another machine, one on another continent and one on tape - every evening.