C Programming under Linux

P2T Course, Martinmas 2003–4
C Lecture 9

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- Command-Line Arguments
- Casting
- Advanced Types

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

Command-Line Arguments

- You may have noticed that your typical Linux application may take one or several command line arguments, e.g. a filename and that it also often can be started with different flags. One example is the C-compiler itself: `gcc -o test test.c`. You also may have wondered how C is going to make this possible.
- This is the moment to reveal that the function `main` actually takes two arguments:

```
int main(int argc, char *argv[])
```

- The parameter `argc` is the number of arguments on the command line, including the name of the program. You don't enter it explicitly - C will count the arguments itself.
- The array `argv` contains the actual arguments.
- To remember the names you can use 'argument counter' (`argc`) and 'argument vector' (`argv`). These names are a convention that is almost always followed, not a part of the C specifications.

Command-Line Arguments - Example

Accessing command-line arguments (`c-line.c`).

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int count;

  printf("Program name: %s\n", argv[0]);

  if (argc > 1)
    {
      for (count = 1; count < argc; count++)
        printf("Arg %d: %s\n", count, argv[count]);
    }
  else
    printf("No command line arguments entered.");

  return(0);
}
```

- The program takes the command-line arguments and prints them out again.
- This illustrates in a simple way how the command-line arguments can be accessed and used.

Parsing Command-Line Options

- Almost all Linux commands use a standard command-line format:

  `command options file1 file2 file3...`

- Also the options (or 'flags') have a standard format (well, two): The one form starts with a dash (`-`) and is usually a single letter that maybe followed by an argument to the option. Example: `gcc -o outfile`.
- The other form starts with a double-dash (`--`) usually followed by a word. Example `gcc --help`.
- The dash or double-dash is meant to help `parse` the command-line.
- In the following we'll consider a simple parser for single dash, single letter flags with no additional space to the argument, e.g. `-ooutfile`.

Parsing Command-Line Options cont.

- To cycle through the command-line options we can use a loop like `while ((argc > 1) && (argv[1][0] == '-')){}`
- At the end of the loop is the code
  ```
  --argc;
  ++argv;
  ```
- This 'consumes' one argument: The number of arguments is decremented and the pointer to the first option is incremented, shifting the list to the left by one place.
- Character `0` of each argument is the dash (`-`), character `1` is the option character. So we can use the expression `switch (argv[1][1])` to decode the option.
- If the option has an argument, character 2 is the first character of the argument. If e.g. this is a filename we can use `out_file = &argv[1][2]` to assign the address of the begin of the output filename to a character pointer named `out_file`.

Parsing Command-Line Arguments - Example

Parsing Command-Line Arguments (`print.c`).

```
int main(int argc, char *argv[])
{
    program_name = argv[0];

    while ((argc > 1) && (argv[1][0] == '-')) {

        switch (argv[1][1]) {
            case 'v':
                verbose = 1;
                break;
            case 'o':
                out_file = &argv[1][2];
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        ++argv;
        --argc;
    }
    return (0);
}
```

Casting

- Sometimes you will have to convert one type of variable to another type. This is done with a cast or typecast operation.
- The general syntax for a cast is

  `(type) expression`

- This operation tells C to compute the value of the expression, and then convert it to the specified type.
- This is particularly useful when you work with integers and floating point numbers or for converting pointers from one type to another.
- Example:

```
int won, lost;
float ratio;
ratio = ((float) won)/((float) lost);
```

Structures

- If your are writing a program for an application in the ’real world’ you will find that often information logically belongs together that is naturally represented by different variable types in C. Some examples could be:
  - Address book: names(strings), streets(strings), street numbers(integers), phonenumbers(integers) and postal codes (integers or strings, depending on the country).
  - Particle physics event: for each track a response for each detector (floats or integers), fit results from the reconstruction (floats), timestamps (integers, probably)
  - Warehouse inventory: names of items (strings), quantity in store (integer), price (float).
- So far we have only dealt with individual variables and with arrays of variables, where each element has the same type.
- Now we’ll get to know a new data type, called a structure. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name.

Defining and Declaring Structures

- The general form of a structure definition is

```
struct structure-name {
    field-type field-name; /* comment */
    field-type field-name; /* comment */
    ......
} variable-name;
```

- For example, we want to define a bin to hold printer cables. The structure definition is

```
struct bin {
    char name[30];    /* name of the part */
    int quantity;     /* how many are in the bin */
    float price;      /* cost of a single part */
} printer_cable_bin;
```

- This defines a structure `bin` and also declares the variable `printer_cable_bin` as type `bin`.
- We can now also declare other variables of type `bin`:

```
struct bin terminal_cable_bin; /* Place to put terminal cables */
```

Defining and Declaring Structures cont.

- It is possible to declare a structure variable, without at the same time defining a data type. This is known as an anonymous structure:

```
struct {
    char name[30];   /* name of the part */
    int quantity;    /* how many are in the bin */
    float price;     /* cost of a single part */
} printer_cable_bin;
```

- It is also possible to define a structure data type without immediately declaring a structure variable:

```
struct coord{
    int x;  /* x coordinate */
    int y;  /* y coordinate */
    int z;  /* z coordinate */
};
```

- While it is also possible (and the code will compile) to create an anonymous structure without declaring a variable, this would be a completely pointless exercise...

Accessing Structure Members

- Structure members are accessed using the structure member operator '.'. We use the syntax `variable.field`.
- If we have a variable `printer_cable_bin` defined as above, we can assign a value to the field `cost` using

  `printer_cable_bin.cost = 12.95; /* Price in $*/`.

- And we can calculate the total value of the cables in the bin:

```
total_cost = printer_cable_bin.cost * printer_cable_bin.quantity;
```

- Structures may be initialised at declaration time by putting the list of elements in curly braces:

```
struct bin {
    char name[30];   /* name of the part */
    int quantity;    /* how many are in the bin */
    float price;     /* cost of a single part */
} printer_cable_bin = {
    "Printer Cables",
    100,
    12.95
};
```

Structures Containing Structures

- A C structure can contain any of C's data types. Naturally, it can also contain structures.
- Let's assume we have structure `coord`

```
struct coord {
    int x;
    int y;
};
```

- We can define a structure `box` and create an instance `mybox` of this structure by

```
struct box {
   struct coord topleft;
   struct coord bottomright;
} mybox;
```

- To access the actual data, we must now apply the membership operator twice:

```
mybox.topleft.x = 10;
```

Structures Containing Structures - Example

box structure containing coord structures (structs.c).

```
#include <stdio.h>

int width, height, area;

struct coord {
  int x;
  int y;
} topleft, bottomright;

struct box {
  struct coord topleft;
  struct coord bottomright;
} mybox;
```

```
int main()
{
  printf("Top left x coordinate:");
  scanf("%d", &mybox.topleft.x);
  printf("Top left y coordinate:");
  scanf("%d", &mybox.topleft.y);
  printf("Bottom right x coordinate:");
  scanf("%d", &mybox.bottomright.x);
  printf("Bottom right y coordinate:");
  scanf("%d", &mybox.bottomright.y);

  width = mybox.bottomright.x - mybox.topleft.x;
  height = mybox.bottomright.y - mybox.topleft.y;
  area = width * height;

  printf("\nThe area is %d units.\n\n", area);

  return(0);
}
```

Output:

```
Top left x coordinate:22
Top left y coordinate:33
Bottom right x coordinate:444
Bottom right y coordinate:555

The area is 220284 units.
```

Arrays of Structures

- We have already seen, or implicitly assumed that structures can have arrays as members (when we put `char name[30]` into a structure). Can we also have an array with elements of type structure ? Yes. We can.
- Let's assume that we want to maintain a list of phone numbers. We might define a structure like

```
struct entry {
   char fname[20];   /* First name   */
   char lname[20];   /* Last  name   */
   char phone[20];   /* Phone number */
};
```

- Now we can define our phone list as an array of structures:

```
struct entry list[1000]; /* phone list */
```

- Now `list[1]` refers to a structure, `list[1].fname` to a member of that structure (that itself is an array) and `list[1].fname[2]` to the third letter of the first name of the second entry in our list. (C starts to count at zero !).

Arrays of Structures - Example

Enter names and numbers and print them again (a_`structs.c`).

```
#include <stdio.h>

struct entry {
  char fname[20];
  char lname[20];
  char phone[20];
};

struct entry list[4];

int main()
{
  int i;
  for (i = 0; i < 4; i++)
    {
      printf("First Name: ");
      scanf("%s", list[i].fname);
      printf("Last Name: ");
      scanf("%s", list[i].lname);
      printf("Phone Number: ");
      scanf("%s", list[i].phone);
    }
  for (i = 0; i < 4; i++)
    {
      printf("Name: %s %s",
             list[i].fname,
             list[i].lname);
      printf("\t\tPhone: %s\n",
             list[i].phone);
    }
  return(0);
}
```

- Please remember: there should be comments here everywhere and there only left out so that the code fist onto the page.

Pointers as Structure Members

- We have already said that structures can have members of any type, so it is only logical that they can also have members that are pointers to any type.
- For example

```
struct data {
   int *value1;
   int *value2;
} first;
```

  creates an instance `first` of the structure `data` with two pointers to int as members.

- Assuming that the integer variables `adc1` and `adc2` have been defined we can initialize the pointers as

```
first.value1 = &adc1;
first.value2 = &adc2;
```

- Now we can use the indirection operator (`*`) in the same way as for regular pointers, i.e. `*first.value1` evaluates to the value of `value1`.

Pointers to Structures

- As you probably already suspected, we can also declare pointers to structures.
- This is in particular practical if we want to pass a structure to a function. (And we remember that this implies that the values in the structure can be changed by the function.)
- Let's assume we have the structure `coord` again:

```
struct coord {
   int x;
   int y;
} point1;
```

- Then we can declare a pointer to the structure `coord` by

```
struct coord *point_ptr;
```

- To initialize the pointer to `point1` we use

```
point_ptr = &point1;
```

Pointers to Structures cont.

- There are now three ways to access a structure member:
- `point1.x` - the structure name.
- `(*point_ptr).x` - a pointer to the structure with the indirection operator (*). The brackets () are necessary because the precedence of the membership operator '.' is higher than that of the dereference operator `*`. (Precedence is something we have mostly ignored so far, and we'll use brackets whenever necessary and keep it that way.)
- `point_ptr->x` - a pointer with the indirect membership operator `->`.
- The structure pointer operator `->` is also known as the indirect membership operator.
- Now think about a structure with a member that is a pointer to the same structure. Thankfully that's beyond the scope of this course.

Unions

- While a structure is used to define a data type with several fields where each field takes up a separate storage location, a union defines a single location that can be given many different field names.
- Example of a `union`:

```
union value {
   long int i_value;  /* integer version of value */
   float f_value;     /* float version of value   */
} data;
```

- Because all fields occupy the same space, assigning a value to `f_value` wipes out a possibly existing value of `i_value`.
- Using a `union`:

```
data.f_value = 5.0;
data.i_value = 3;   /* data.f_value overwritten */
i = data.i_value;   /* legal, */
f = data.f_value;   /* illegal, will generate unexpected results */
data.f_value = 5.5; /* put something into f_value / clobber i_value */
i = data.i_value;   /* illegal, will generate unexpected results */
```

The `typedef` Statement

- C allows the programmer to define his/her own variable types through the `typedef` command. The general form is

  `typedef type-declaration`

- For example, the declaration typedef int count; defines a new type `count` that is identical with `int`. Therefore the declaration `count flag;` is identical to `int flag;`.
- One frequent use of `typedef` is the definition of a new structure as a variable type.
- Example:

```
struct complex_struct {
   double real;
   double imag;
};
typedef struct complex struct complex;

complex voltage1 = {3.5, 1.2};
```