C Programming under Linux

P2T Course, Martinmas 2003–4
C Lecture 7

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

r.kaiser@physics.gla.ac.uk

Summary

- Pointers, Pointers, Pointers

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

People and Addresses

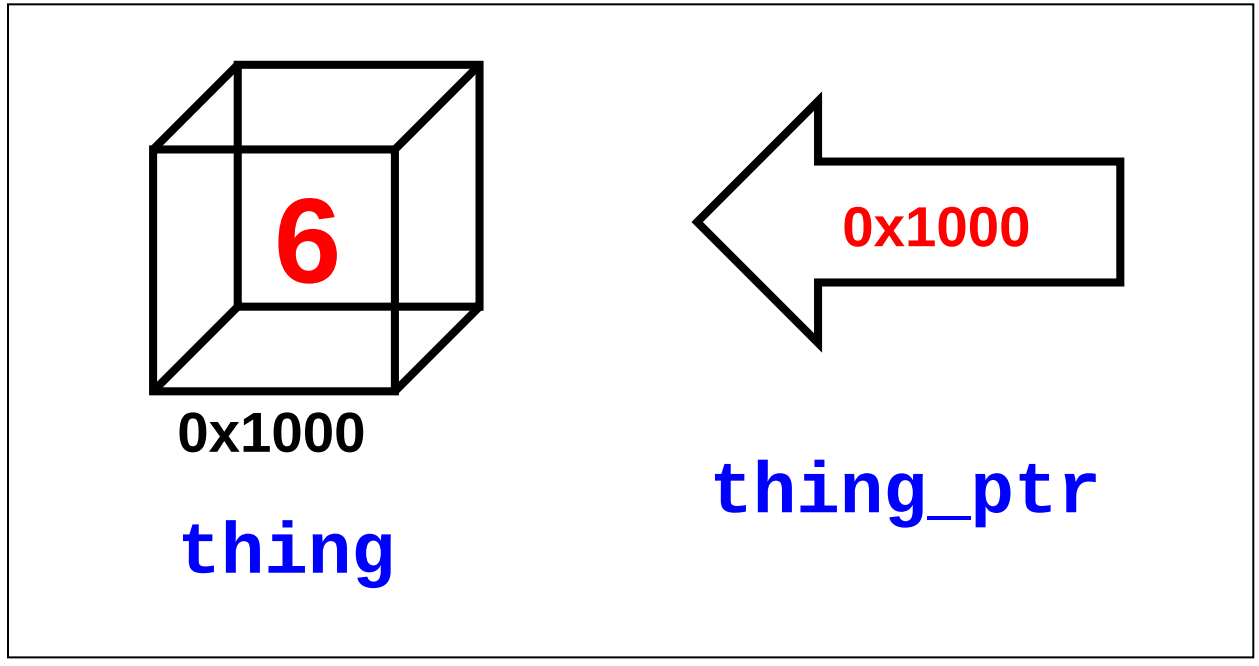
- You may remember receiving a letter addressed to 'the occupier' of your flat. This may e.g. have been a reminder to register for the next election or it may have been a letter from the electricity supplier.
- They clearly meant you, without knowing you name. When you answered them you would have included your name, e.g. by signing the letter.
- Another example would be that instead of 'Tony Blair', i.e. the name of the prime minister, the news often just refer to '10 Downing Street' and everybody know that this means Tony Blair.
- More precisely, it means Tony Blair at the moment - in the future there will likely be a different name at the same address while Tony Blair will still be around but at a different address.
- Now comes the relevance to C programming: Any variable has (at a given time) a particular value that is stored at a specific address.

Variables, Values and Addresses

- Each variable has several attributes that belong to it and that define it:
  - name
  - type
  - value
  - address

  Usually the variable address is something that the system uses implicitly and that we don't have to worry about.
- However, we can also have a variables where the value is a memory address.
- These variables known as address variables and in C they are called pointers, because they point to the location (address) of something (the value of the variable).
- Pointers are typical for C, not very intuitive, but very powerful. And they are a bit harder to grasp. That's why we will spend a bit more time on them.

Things and Pointers



- Assume we have a variable called `thing`. The value of the variable is 6. The address of `thing` is `0x1000`.
- Our pointer `thing_ptr` contains the address `0x1000`. Because this is the address of `thing`, we say that `thing_ptr` points to `thing`.

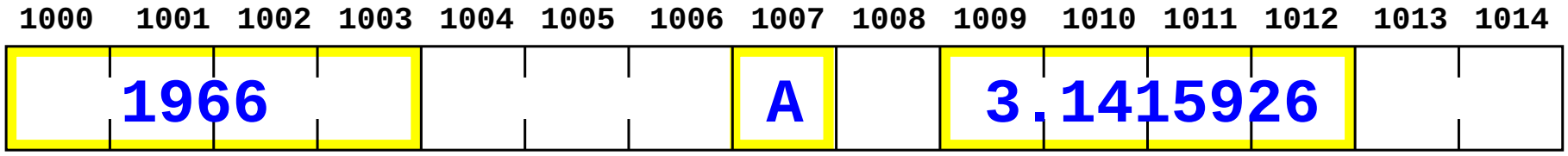
Pointers

- Pointers are linked to a specific type of variable. A pointer is declared by putting an asterisk (*) in front of the variable name in the declaration statement:

```
int thing;           /* define a thing */
int *thing_ptr;      /* define a pointer to an integer */
float *result_ptr;   /* define a pointer to a float */
```

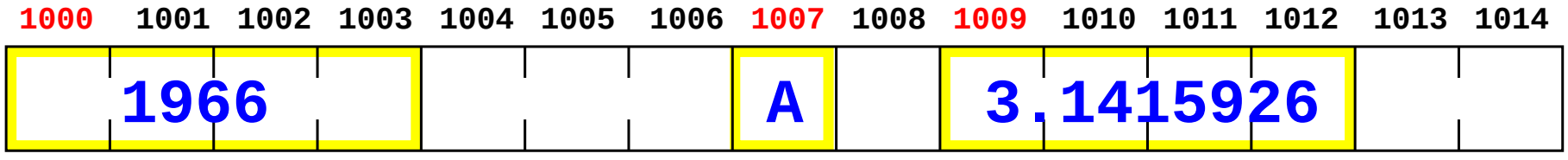
- Different types of variables occupy different amounts of storage space in memory (e.g. 1 byte for `char` and 4 byte for `float`)
- The compiler "remembers" the size of the variable that the pointer points to.
- It is a practical convention (but not required) to give pointer variables names with the extension `_ptr`. This helps with keeping pointers and variables apart. (You may find different conventions elsewhere, e.g. using `p_` as a prefix.)

Visualising Pointers




```
int a = 1966;    char b = 'A'; float c = 3.1415926;
```

```
int *a_ptr;    char *b_ptr;    float *c_ptr;
a_ptr = &a;    b_ptr = &b;     c_ptr = &c;
```



Pointer Operators

- Two unary operators are used in conjunction with pointers:
- The operator ampersand (&) returns the address of a thing - which is a pointer.
- The operator asterisk (*) returns the object to which a pointer points - i.e. what is found at the address that is the value of the pointer.

C Code	Description
`thing`	the integer variable named "thing"
`&thing`	address of the variable "thing" (a pointer)
`*thing`	is illegal, the operation is invalid
`thing_ptr`	pointer to an integer (may or may not be the specific integer `thing`)
`*thing_ptr`	integer variable at the address `thing_ptr` points to
`&thing_ptr`	is legal, but odd, because it's a pointer to a pointer

Pointer Operators cont.

```
int thing;  /* declare an integer (a thing) */
thing = 4;
```

The variable `thing` is a thing. The declaration `int thing` does not contain an *, so `thing` is not a pointer.

```
int *thing_ptr;  /* declare a pointer to a thing */
```

The variable `thing_ptr` is a pointer, indicated by the * in the declaration (and by the extension `_ptr`).

```
thing_ptr = &thing;  /* point to the thing */
```

The expression `&thing` is a pointer (the address of the variable `thing`). This is now assigned to `thing_ptr`.

```
*thing_ptr = 5;  /* set "thing" to 5 */
```

The expression `*thing_ptr` indicates a thing, because the * tells C to look at the data pointed to (an integer in this case), not the pointer itself. We now have set `thing` to 5. Note that `thing_ptr` points at any integer, it may or may not point to the specific variable `thing`.

Pointer Operators - Example

`printf` using variable and pointer to variable. (`thing.c`)

```
#include <stdio.h>
int main()
{
    int   thing_var;                    /* define a variable for thing */
    int  *thing_ptr;                    /* define a pointer to thing */

    thing_var = 2;                      /* assigning a value to thing */
    printf("Thing %d\n", thing_var);
    thing_ptr = &thing_var;             /* make the pointer point to thing */
    *thing_ptr = 3;                     /* thing_ptr points to thing_var so */
                                        /* thing_var changes to 3 */
    printf("Thing %d\n", thing_var);

    printf("Thing %d\n", *thing_ptr); /* another way of doing the printf */
    return (0);
}
```

Output:

```
Thing 2
Thing 3
Thing 3
```

Pointers as Function Arguments

- C passes parameters to a function using call by value, i.e. the parameters go only one way into a function.
- Not the parameter itself, only it's value is handed to the function. The only result of a function is a single return value.
- Pointers can be used to get around this restriction.
- Instead of passing a variable to a function (which would only pass the value of the variable) we pass a pointer to the function (which passes the value of the pointer).
- The value of the pointer is an address, the address of the variable that we can change now.
- The parameter handed to the function (the address) is not changed, but what it points to (the value of the variable at the address) is changed.

Pointers as Function Arguments - Example

Function that demonstrates the use of pointers to pass parameters that can be changed (`call.c`).

```
#include <stdio.h>
void inc_count(int *count_ptr)
{
    ++(*count_ptr);
}

int main()
{
    /* number of times through */
    int  count = 0;

    while (count < 10)
        {
        inc_count(&count);
        printf("%d\n", count);
        }

    return (0);
}
```

- `main` calls the function `inc_count` to increment the variable `count`.
- Passing `count` would only pass it's value (0).
- So the address `&count` is passed instead, as a parameter specified as a pointer to an integer `(int *count_ptr)`.
- Note that the parameter (the address) is not changed, but what it points to (the value at the address) is changed.

`const` Pointers

- Pointers can be constant, but this is a little tricky, because either we can have a constant pointer or a pointer to a constant.
- `const char *answer_ptr = "Forty-Two";`
  does not mean that the variable `answer_ptr` is a constant, but that the data pointed to by `answer_ptr` is a constant. The data cannot change, but the pointer can.
- If we put `const` after the `*` we tell C that the pointer is constant, e.g. `char *const name_ptr = "Test";`.
  In this case the pointer cannot be changed, but the data it points to can.
- Finally, if we really want to, we can create a `const` pointer to a `const` variable:
  `const char *const title_ptr = "Title";`.

Pointers and Arrays

- C allows pointer arithmetic (addition and subtraction). Because the elements of an array are assigned to consecutive addresses, this allows to navigate an array.
- C automatically scales pointer arithmetic so that it works correctly, by incrementing/decrementing by the correct number of bytes.
- If we create an array and a pointer to it's first element
  `char array[5];`
  `char *array_ptr = &array[0];`
  we can then refer to the nth element of the array `array[n]` as *(array_ptr+n).
- The brackets () are important; (*array_ptr)+n is the same as `array[0]+n`, not `array[n]`.
- C provides a shorthand for dealing with arrays: `array_ptr = array;` instead of `array_ptr = &array[0];`

Pointers and Arrays - Example

Scanning an array using array index and pointer increment in comparison (`ptr2.c, ptr3.c`).

```
#include <stdio.h>

int array[] = {9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
    index = 0;
    while (array[index] != 0)
        ++index;

    printf("%d elements before zero\n",
                  index);
    return (0);
}
```

```
#include <stdio.h>

int array[] = {9, 8, 1, 0, 1, 9, 3};
int *array_ptr;

int main()
{
    array_ptr = array;

    while ((*array_ptr) != 0)
        ++array_ptr;

    printf("%d elements before zero\n",
                  array_ptr - array);
    return (0);
}
```

The index operation required for the `while` loop on the left side takes longer than the pointer dereference on the right side.

Passing Arrays to Functions

- You may want to pass an array to a function. If you do so, C will automatically change the array into a pointer - because you can only hand single values over to a function. In this case the address of the first element of the array.
- If you want your function to know how many elements the array has, you may want to pass on the number of elements as a second (integer) variable.

```
int some_function(int *data_ptr, int nelements);
```

- If you pass a single variable to a function you only pass a copy of the variable's value - so the original variable cannot be changed. This is different if you pass an array. Because you are passing it as a pointer, the code in the function is working with the actual array elements.

Passing Arrays to Functions - Example

Passing an array to a function and initialising it (`init-a.c`).

```
#define MAX 10

void init_array_1(int data[])
{
    int  index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

void init_array_2(int *data_ptr)
{
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}
```

```
int main()
{
    int  array[MAX];

    void init_array_1();
    void init_array_2();

    /* 4 ways of initializing
                the array */

    init_array_1(array);
    init_array_1(&array[0]);
    init_array_1(&array);
    init_array_2(array);

    return (0);
}
```

When passing an array to a function, C will automatically change the array into a pointer. In fact, C will issue a warning if you put a & before the array, i.e. as in version 3.

Pointers-to-Pointers

- Pointers are variables whose value is an address.
- According to the type of variable at the address, we can have e.g. a pointer-to-integer or a pointer-to-float.
- Like any other variable, also pointers are stored at a specific place in the computer's memory - they also have an address.
- We can therefore define a variable that has as it's value the address of a pointer - a pointer-to-pointer.

```
int x = 12;                 /* x is an integer variable with value 12 */
int *x_ptr = &x;            /* x_ptr is a pointer to the integer x    */
int **ptr_to_ptr = &x_ptr; /* ptr_to_ptr is a pointer to a pointer   */
                            /* to type int     */
```

- Note the double indirection operator (**) when declaring the pointer-to-pointer.
- There is in principle no limit to the level of multiple indirection - you could point and point and point and point. But there is no real advantage to anything with more than two levels of pointing.

Arrays of Pointers

- Because pointers are one of C's data types you can declare and use arrays of pointers.
- One possible use for this is an array of pointers to type `char`, aka an array of strings.
- It's much easier to pass an array of pointers to a function than to pass a series of strings.
- In fact, you are of course passing a pointer to the function - this pointer is a pointer-to-pointer.
- This actually is one of the main applications of multiple indirection.

Arrays of Pointers - Example

Passing an array of pointers to a function (p2p.c).

```
#include <stdio.h>

void print_message(char *ptr_array[], int n) {
  int count;
  for (count = 0; count < n; count++)
    {
      printf("%s ", ptr_array[count]);
    }
  printf("\n");
}

int main() {
  char *message[9] = {"Dennis", "Ritchie", "designed", "the", "C",
                      "language", "in", "the", "1970s"};
  print_message(message, 9);
  return (0);
}
```

Output:

```
kaiser@npl03:~/linuxc/oreilly/pracc/p2p> p2p
Dennis Ritchie designed the C language in the 1970s
```

Pointers to Functions

- A pointer holds an address. This may be the address of a variable or the address of the first element of an array - the cases we looked at so far.
- However, a pointer may also hold the starting address of a function, i.e. the address where the function is stored in memory.
- Pointers to functions provide another way to call functions.
- The general form (and some examples) for the declaration of a pointer to a function:

```
type (*ptr_to_func)(parameter_list);
int (*func1)(int x);
char (*func2)(char *p[]);
```

- The parentheses around the function name are necessary because of the relatively low precedence of the indirection operator (*).

How not to Use Pointers

- Pointers in themselves are already confusing enough. However, the combination of pointers with increment and decrement operators (++/–) can easily make matters worse.
- Have a look at these lines of code as examples of how not to use pointers:

```
data_ptr = &array[0]; /* Point to the first element of the array.      */
value = *data_ptr++;  /* Get element #0, data_ptr points to element #1.*/
value = *++data_ptr;  /* Get element #2, data_ptr points to element #2.*/
value = ++*data_ptr;  /* Increment element #2, return it's value       */
                      /* Leave data_ptr alone.                          */
```

- While it's not impossible to figure out what is going on, it is not the point of programming to provide challenging little puzzles to the programmers that have to look at your code in the future.
- Here is another bad example:

```
void copy_string(char *p, char *q)
{
   while (*p++ = *q++);
}
```