introduction_to_c

October 5, 2016

1 A crashcourse to C

Simon Funke
Date: **Oct 5, 2015**

1.1 C programming language

- Strongly typed, compiled language
- Low level memory management, no garbage collector
- If used correctly, it results in computationally very efficient programs.
- Initially developed 1969, Bell Labs
- Different standards: C89/C90, C99, C11
- Different compilers: GNU C (gcc), Intel C++, ...
- Lots of references available online, for example Coding united C tutorial

1.2 Hello world in C

```
// File helloworld.c
#include <stdio.h>
int main()
{
    // A comment
    int x;

    printf( "Hello world\n" );

    return 0;
}
```

Compile with

```
gcc helloworld.c -o helloworld
```

Run with

```
./helloworld
```

1.3 Some compile options

Compile with optimisations and show warnings

```
gcc -Wall -O3 helloworld.c -o helloworld
```

- `-Wall`: print out compiler warnings (e.g. unused variables, ...)
- `-O3`: switch on compiler optimisation to produce faster code
- `-o filename`: name of outfile file (executable)

1.4 Variables

- C is strongly typed
- That is, every variable must be declared before its usage:

```
int i;                      // declare integer, no defined value
int j = 10;                 // integer with value 10
float y = 10.0;             // float
const float pi = 3.14;      // constant float
double z = 10.0;            // double
char c = 'A';               // (one) character,
                            // represented as numbers between 0..255

i = 2;                      // OK
c = 'B';                    // OK
y = 2;                      // OK, implict conversion to float
pi = 3.1415;                // compile error: assignment of read-only variable
i = 1.4;                    // compile error: i is of type int
c = 1;                      // works. why?
```

Note: The end of each command is indicated with a semicolon.

1.5 Conditions

```
if ( c == 'A')
{
    printf("In first case\n"); //  '\n' prints a new line
}
else if ( (i > 10 && j > 20) || c == 'B' )
{
    printf("In second case\n");
}
else
    printf("In third case\n");
```

- Condition is wrapped in ()
- Conditional statements are wrapped {}
- For one line statements, the {} can be dropped.

1.6 Loops

- `for` loop

```
int cnt;
for (cnt = 0; cnt < 10; cnt++)
{
    printf ("%i\n", cnt);
}
```

- `do while` loop

```
int cnt = 10;
do
{
    cnt -= 1;
    printf ("%i\n", cnt);
}
while(cnt > 0);
```

1.7 Functions

```
void PrintMe()
{
    printf("Printing in a function.\n");
}

int Add(int in1, int in2 )
{
    return in1 + in2;
}

void main()
{
    int sum;

    PrintMe();
    sum = Add(1, 2);
    sum = Add(1, 2.0);  // compilee rror: type mismatch
}
```

- C functions have no or more input arguments and exactly one return value.
- Return value can be `void` (nothing).
- Type of parguments and return values must be specified.

1.8 Arrays

```
int a[3];   // Array with integers of length 3
a[0] = 1; a[1] = 2; a[2] = 3;
```

```c
int i;
for ( i = 0; i < 3; i=i+1 )
    a[i] = 0;
a[4] = 6;    // Might work, or you get a Segmentation fault
```

- The last line tries to access an element outside the array.
- The C compiler does not check if the index is within the bounds.
- There is no C-equivalent to `len(a)` in Python. You need to keep track of the array length yourself.
- This means that, when passing an array as an argument to a function, one also needs to pass the length of the array as an integer argument:

1.9 Pointers

- Every variable is stored in memory. The location of the variable in memory is called the *memory address*.
- A pointer is a data type that can store a memory address. *A pointer points to another variable.*
- There are two basic operations that we can do with pointers:

1. Get the memory address of a variable and store it in a pointer.
2. Retrieve the variable a pointer points to.

Use cases for pointers: * To store and access information stored in arrays. * Create dynamic data structures. * Pass variables to a function without copying. * To create functions that change many variables.

1.10 Getting the memory address of a variable

The `&` operator in C returns the memory adress of a variable.

```c
int i=8;
char c='a';

&i;                     // Get the memory address of i
printf("%i\n", &c);   // Print the memory adress of c as an integer
```

Running this program on my computer yields

```
>> ./a.out
6295624
```

The memory address might be different next time the program is executed.

1.11 Creating pointers

- A pointer stores one memory location
- Pointers are typed
- Adding a `*` to the type in a declaration creates a pointer

- When initialising a pointer, it points to an undefined memory address

```
int* ptr_i;     // Declare a pointer to an integer
float* ptr_f;   // Declare a pointer to a float
char* ptr_c;    // Declare a pointer to a character
```

1.12 Using pointers

```
int* ptr_i;     // Declare a pointer to an integer
int i=8;
ptr_i = &i;     // Store the memory address of i in ptr_i
```

1.13 Using pointers

```
int* ptr_i;        // Declare a pointer to an integer
float* ptr_f;      // Declare a pointer to a float
int i=8;
```

Pointers are typed variables.

```
ptr_i = &i;        // Store the memory address of i in the ptr_i
ptr_f = &i;        // Dangerous: ptr_f is a point to a float, not int
                   // No error, just a compiler warning
```

Applying the * operator to a pointer yields the variable it points to.

```
int ii = *ptr_i;     // Get the variable the pointer point to (a int)
float ff = *ptr_i;   // error: ptr_i is a int pointer
float ff = *ptr_f;   // undefined behaviour: ptr_f was never initialised
```

1.14 A full example of pointer usage

```
#include <stdio.h>

int main() {
    int i;
    int *ptr_i;

    i = 3;
    ptr_i = &i;
    *ptr_i = 10;
    printf("%d\n", i);
    return 0;
}
```

1.15 Pointer as function arguments

- Task: Write a function that swaps two integer variables.
- Problem: C only allows one return value for a function.

- Solution: Pass pointers to the variables that need to be swapped.

```
void swapping(int *ptr_c, int *ptr_d) {
    int tmp;

    tmp = *ptr_c;
    *ptr_c = *ptr_d;
    *ptr_d = tmp;
}

int main(void) {
    int a=1;
    int b=3;

    printf("before swap: %d %d\n", a, b);
    swapping(&a, &b);
    printf("after swap: %d %d\n", a, b);
    return 0;
}
```

1.16 Pointers and arrays

- An array name is a constant pointer to the first element of the array:

```
double array[3];   // array is equivalent to &array[0]
double *ptr_array;

ptr_array = array;  // ptr_array points to first element of array
```

- Pointer arithmetic can be used as an alternative to array indexing

```
array[2]         // Access 3rd element in array
*(ptr_array+2)   // Access 3rd element in array
                 // (Increment the memory adress twice, and get its value)
```

1.17 Pointers and arrays

- An array can be used in a function by passing a pointer to its first element.
- We need to keep know the length of the array. Hence the length is typically passed as an int parameter.

```
// Compute a*fac (elementwise) without copying a
void multiply(int n, double *a, float fac)
// n: the length of the array
// a: an array (pointer to the first entry in the array)
// fac: the multplication factor
{
    int i;
    for (i=0; i<n; i=i+1) {
        a[i] = fac*a[i];
    }
}

if main() {
    double a[3];
    a[0] = 1; a[1] = 2; a[2] = 3;
    multiply(3, a, 2.0)
}
```

1.18 Two-dimensional arrays, part 1

- We can store two-dimensional arrays in a one-dimensional array.
- `This is the default behaviour of numpy` (row-major order).
- In row-major order, consecutive elements of the rows of the array are contiguous in memory

```
void printMatrix(int n, int m, int* a) {
    int i, j;
    for (i = 0; i < n; i=i+1) {
        printf("\n");
        for (j = 0; j < m; j=j+1)
            printf("%i  ", a[i*m+j]);
    }
}

int main() {
    int n = 3;    // Number of rows
    int m = 4;    // Number of columns
    int a[n*m];
    int i, j;

    for (i = 0; i < n; i=i+1)
        for (j = 0; j < m; j=j+1)
            a[i*m+j] = i+j;

    printMatrix(n, m, a);
}
```

1.19 Multi-dimensional arrays, part 2

- C supports natively higher dimensional arrays
- These are implemented as pointer of pointers (array of arrays)

```
int a[2][2];    // a is a constant pointer to int pointers
                // (type `const int **`)
int i, j;

for (i = 0; i < 2; i=i+1)
```

```
{
    for ( j = 0; j < 2; j=j+1)
    {
        a[i][j] = i*j;
        printf("a[%i][%i] = %i\n", i, j, a[i][j]);
    }
}
```

Note: The memory of the array might not be contiguous (required for numpy integration).