A Review on C Programming

Table of Contents

Functions

- Functions in C must be pre-declared.

```c
#include <stdio.h>

int plus_one(int n){
    return n + 1;
}

int main(void){
    int i=10, j;
    j = plus_one(i);
    printf("i+1 is %d\n", j);
}
```

  - *void* means that the function takes no arguments.

Passing by value

- When passing an argument variable to the function, a copy of the value of that variable gets made and stored in the parameter.

```c
#include <stdio.h>

void increment(int a){
    a++;
}

int main(void){
    int i = 10;
    increment(i);
    printf("i == %d\n", i);
}
```

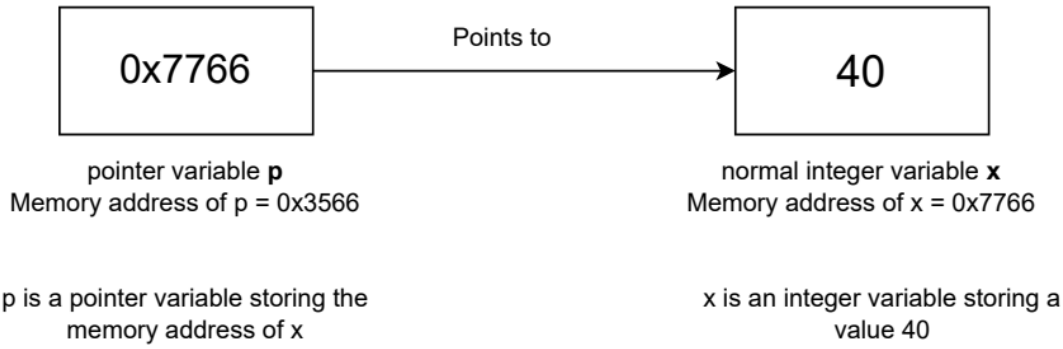
Function prototypes

- Instead of defining the function before the *main*, we can define its prototype before the main and the function (with its body) after the main.

```c
#include <stdio.h>

int foo(void);

int main(void){
    int i;
    i = foo();
    printf("%d\n", i);
}

int foo(void){
    return 1234;
}
```

Pointers

- In C, we must be explicit when we are using an object (or a value) itself or a reference to it.
- Pointers are the address of the data.
    - Just like int can hold the value 12, a pointer can hold the address of the 12.

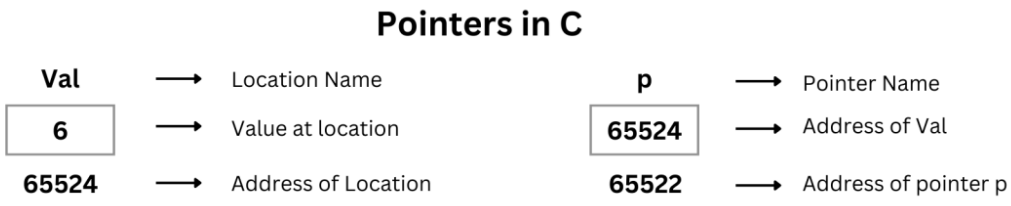


- To create a pointer in C, include an * before the variable name.
    - Declaring pointers in this way just allocates memory space.
    - So, a pointer variable will initially contain garbage.

```
int *pointerToInt;
double *pointerToDouble;
char *pointerToChar;
char **pointerToPointerToChar;
```

- To initialize a pointer variable, you must assign to it the address of something that already exists.
    - Using the & (address-of) operator:

```
int val;                /* an int variable */
int *p;                 /* a pointer to an int */

p = &val;               /* p now points to val */
```



Using pointers

- Example:

```c
#include <stdio.h>

int main(void){
    int n = 5;                  /* an int variable */
    int *p;                     /* a pointer to an int */
    p = &n;                     /* p now points to n */
    printf("%d\n", n);
    printf("%p\n", p);
    printf("%p\n", &n);
    printf("%d\n", *p);
}
```

- You can work on the value stored at the location pointed to.

```c
#include <stdio.h>

int main(void){
    int n = 5;                  /* an int variable */
    int *p;                     /* a pointer to an int */
    p = &n;                     /* p now points to n */
    printf("n = %d\n", n);
    *p = 2;
    printf("n = %d\n", n);
    *p = *p + *p;
    printf("n = %d\n", n);
}
```

- Observer the difference between $(*p)++$ and $*p++$:

```c
#include <stdio.h>

int main(void){
    int n = 5;                  /* an int variable */
    int *p;                     /* a pointer to an int */
    p = &n;                     /* p now points to n */
    printf("p = %p\n", p);
    printf("n = %d\n", *p);

    (*p)++;
    printf("p = %p\n", p);
    printf("n = %d\n", *p);

    *p++;
    printf("p = %p\n", p);
    printf("n = %d\n", *p);

}
```

Passing by reference

- When passing arguments to a function, we can pass it by value or pass by reference.

- Pass by reference example:

```c
#include <stdio.h>

void increment(int *p){
    *p = *p + 1;
}

int main(void){
    int i = 10;
    int *j = &i;

    printf("i = %d\n", i);
    printf("Also, i = %d\n", *j);

    increment(j);
    printf("Now, i = %d\n", i);

}
```

- We could also write it as follows:

```c
printf("i is %d\n", i); // prints "10"
increment(&i);
printf("i is %d\n", i); // prints "11"!
```

- The *NULL* pointer is used to indicate that the pointer does not point to anything.
    - This will result in undefined behavior or crash.

```c
int main(void){
    int *p = NULL;
    *p = 12;
    printf("%d\n", *p);
}
```

```
Process finished with exit code 139 (interrupted by signal 11:SIGSEGV)
```

- We can use the *sizeof*() operator with pointers.

```c
int *p;
// Prints size of an 'int'
printf("%zu\n", sizeof(int));
// p is type 'int *', so prints size of 'int*'
printf("%zu\n", sizeof(p));
// *p is type 'int', so prints size of 'int'
printf("%zu\n", sizeof(*p));
```

Constant pointer

- A pointer is said to be a constant pointer when the address that is pointing to cannot be changed.
    - **That means a constant pointer cannot point to a new address if it is already pointing to an address.**
- A constant pointer is defined as $< pointer - type > * const < name >$.
    - The code below causes an error.

```c
#include<stdio.h>

int main(void)
{
    char ch = 'c';
    char c = 'a';

    char *const ptr = &ch; // A constant pointer
    ptr = &c;              //WRONG!!!!

}
```

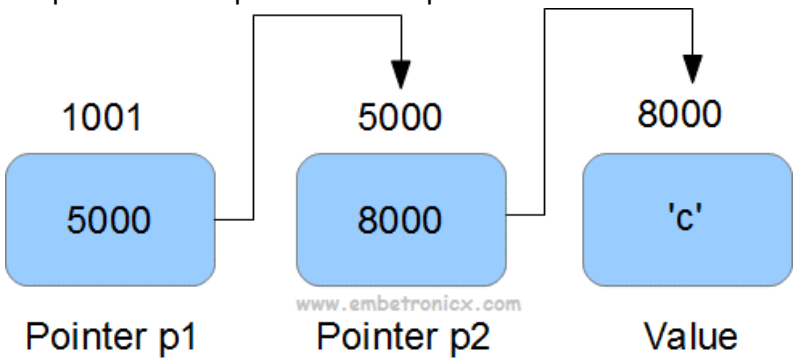
Pointer to constant

- The value pointed to by the pointer variable cannot be changed.
- A pointer to a constant is declared as: $const < pointer - type > * < name >$.
    - The code below causes an error.

```c
#include<stdio.h>

int main(void)
{
    char ch = 'c';
    const char *ptr = &ch; // A constant pointer 'ptr' pointing
    *ptr = 'a';    // WRONG!!! Cannot change the value
}
```

Pointer to pointer

- We can have pointers that point to other pointers.



- Example:

```c
#include<stdio.h>

int main(void)
{
    char **ptr = NULL;
    char *p = NULL;
    char c = 'd';

    p = &c;
    ptr = &p;

    printf("c = %c\n",c);
    printf("*p = %c\n",*p);
    printf("**ptr = %c\n",**ptr);
}
```

Function pointers

- We can have pointers to functions.
- A function pointer is declared as follows:

  $< return\ type >\ (*< pointer\ name >)\ (types\ of\ function\ arguments)$
- Example:

```c
#include<stdio.h>

void func (int a, int b)
{
    printf("a = %d\n",a);
    printf("b = %d\n",b);
}

void main(void)
{
    void(*fptr)(int,int); // Function pointer
    fptr = func;          // Assign address to function pointer

    func(2,3);
    fptr(2,3);
}
```

- Function pointers can be useful to implement:
  - Callback mechanism
  - Array of functions
- A callback function is a function that is called by using a function pointer.

- The following two programs are equivalent.

```c
#include<stdio.h>

void callback_fn(void) {
    printf("In callback function\n");
}

void test_loop(void) {
    for (int i = 0; i < 6; i++) {
        if (i == 5) {
            callback_fn();
        }
        printf("i = %d\n", i);
    }
}

void main() {
    test_loop();
}
```

```c
#include<stdio.h>

// Callback Function which has no argument and no return value
void callback_fn( void ){
    printf("In callback function\n");
}

void test_loop( void (*fn)(void) ){
    for( int i = 0; i < 6; i++ ){
        if(i == 5){
            // callback execution
            (*fn) ();
        }
        printf("i = %d\n", i);
    }
}

void main(){
    // Registering the callback
    void (*fn_ptr)( void ) = &callback_fn;

    // calling the function with the function pointer
    test_loop(fn_ptr);
}
```

Arrays and pointers

- A pointer to an array means a pointer to the first element of the array.

```c
#include<stdio.h>


void main() {
    int a[5] = {11, 22, 33, 44, 55};
    int *p;

    p = &a[0];
    printf("%p\n", p);
    printf("%d\n", *p);
    printf("%d\n", *p);
}
```

Passing arrays to functions.

- The following functions are the same thing.

```c
#include<stdio.h>

void times2(int *a, int len){
    for (int i = 0; i < len; ++i) {
        printf("%d ", a[i]*2);
    }
    printf("\n");
}

void times3(int a[], int len){
    for (int i = 0; i < len; ++i) {
        printf("%d ", a[i]*3);
    }
    printf("\n");
}

void times4(int a[5], int len){
    for (int i = 0; i < len; ++i) {
        printf("%d ", a[i]*4);
    }
    printf("\n");
}

void main() {
    int a[5] = {11, 22, 33, 44, 55};
    times2(a, 5);
    times3(a, 5);
    times4(a, 5);
}
```

- Changing arrays in functions.

```c
#include<stdio.h>

void double_array(int *a, int len){
    for (int i = 0; i < len; ++i) {
        a[i] *= 2;
    }
}
void main() {
    int a[5] = {11, 22, 33, 44, 55};
    double_array(a, 5);

    for (int i = 0; i < 5; ++i) {
        printf("%d ", a[i]);
    }
}
```

- Passing multi-dimensional arrays.
    - **The compiler needs to know all the dimensions except the first one.**

```c
#include<stdio.h>

void print_2d_array1(int a[][3], int rows, int cols){
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}

void print_2d_array2(int a[2][3], int rows, int cols){
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}

void main() {
    int a[2][3] = {
            {1, 2, 3},
            {10, 11, 12}
    };

    print_2d_array1(a, 2, 3);
    print_2d_array2(a, 2, 3);
}
```

Strings

- Strings are treated as pointers in C.

```
char *s = "Hello, world!";
```

  - **The variable points to the first character, namely "H".**

- Print it using "%s" format specific.

```
char *s = "Hello, world!";
printf("%s\n", s); // "Hello, world!"
```

- We can define strings explicitly as arrays.

```
char s[14] = "Hello, world!";
// Or
char s[] = "Hello, world!";
```

- We can access the characters of a string using [] operator.

```
#include<stdio.h>

void main() {
    char *s1 = "String 1";
    char s2[] = "String 2";

    size_t s1_len = sizeof(s1);
    // -1 for the null character
    size_t s2_len = sizeof (s2) - 1;

    for (int i = 0; i < s1_len; ++i) {
        printf("%c ", s1[i]);
    }
    printf("\n");
    for (int i = 0; i < s2_len; ++i) {
        printf("%c ", s2[i]);
    }
}
```

- If a string is declared as a pointer, then it is *immutable*.

```
char *s = "Hello, world!";
s[0] = 'z'; // BAD NEWS: tried to mutate a string literal!
```

```
Process finished with exit code 138 (interrupted by signal 10:SIGBUS)
```

- If a string is declared as an array, then it is *mutable*.

```
char s[] = "Hello, world!";
s[0] = 'z';
printf("%s\n", s);
```

- Getting the length of a string using *strlen*().

```c
#include<stdio.h>
#include <string.h>

void main() {
    char *s = "Hello, world!";
    size_t len = strlen(s);
    printf("The length of the string=%zu\n", len);
}
```

- You can't copy a string through the assignment operator (=).
    - All that does is make a copy of the pointer to the first character, so you end up with two pointers to the same string:

```c
#include<stdio.h>
#include <string.h>

void main() {
    char s[] = "Hello, world!";
    char *t = s;
    t[0] = 'z';
    printf("%s\n", s);
}
```

- To copy a string, use the *strcpy*() function.

```c
#include<stdio.h>
#include <string.h>

void main() {
    char s[] = "Hello, world!";
    char t[100];

    strcpy(t, s);

    t[0] = 'z';
    printf("s = %s\n", s);
    printf("t = %s\n", t);
}
```