Programming in C

Dr. Neill Campbell
Neill.Campbell@bristol.ac.uk

University of Bristol

October 8, 2021

Table of Contents

Call-by-Value

```c
#include <stdio.h>

void changex(int x);

int main(void)
{
   int x = 1;

   changex(x);
   printf("%i\n", x);
   return 0;
}

void changex(int x)
{
   x = x + 1;
}
```

Execution :

1

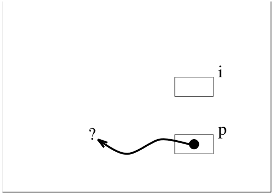
- In the program, the function cannot change the value of `v` as defined in `main()` since a **copy** is made of it.
- To allow a function to modify the value of a variable passed to it we need a mechanism known as **call-by-reference**, which uses the **address** of variables (pointers).

Call-by-Reference

- We have already seen addresses used with `scanf()`. The function call:

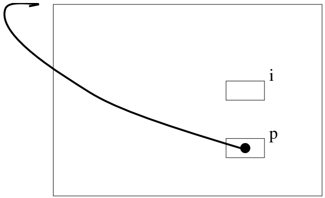
  `scanf("%i", &v);`

  causes the appropriate value to be stored at a particular address in memory.
- If `v` is a variable, then `&v` is its address, or location, in memory.
- `int i, *p;`
- Here `i` is an `int` and `p` is of type *pointer to int*.
- Pointers have a legal range which includes the special address 0 and a set of positive integers which are the machine addresses of a particular system.

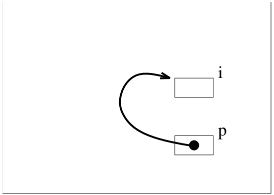


The *NULL* Pointer

- p = NULL;

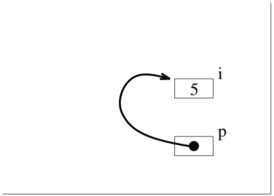


- p = &i;



Equivalence of *i* and **p*

- i = 5;




```c
#include <stdio.h>

int main(void)
{

    int i = 5;
    int* p = &i;
    printf("%i\n", *p);
    i = 17;
    printf("%i\n", *p);
    *p = 99;
    printf("%i\n", i);

    return 0;

}
```

Execution :

```
5
17
99
```

scanf Again

```c
#include <stdio.h>

int main(void)
{

    int i;
    int* p;

    p = &i;
    printf("Please Type a number : ");
    scanf("%i", &i);
    printf("%i\n", i);
    printf("Please Type a number : ");
    scanf("%i", p);
    printf("%i\n", i);

    return 0;

}
```

Execution :

```
Please Type a number : 70
70
Please Type a number : 3
3
```

- In many ways the dereference operator * is the inverse of the address operator &.

```c
float x = 5, y = 8, *p;
p = &x;
y = *p;
```

- What is this equivalent to ?

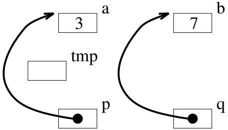
The *swap* Function

```c
#include <stdio.h>

void swap(int* p, int* q);

int main(void)
{
    int   a = 3, b = 7;

    // 3 7 printed
    printf("%i %i\n", a, b);
    swap(&a, &b);
    // 7 3 printed
    printf("%i %i\n", a, b);
    return 0;
}

void swap(int* p, int* q)
{
    int   tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

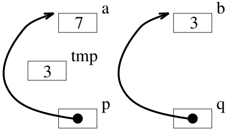
Execution :

3 7
7 3

- At beginning of function:



- At end of function:



- Remeber that the variables *a* and *b* are not in the scope of *swap()*.

Arrays are Pointers ?

- An array name by itself is simply an address (**Array Decay**).
- For instance:

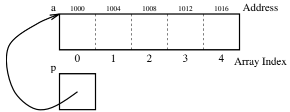
```
int a[5];
int *p;
```

  declares an array of 5 elements, and a is the address of the start of the array.
- Assigning:

```
p = a;
```

  is completely valid and the same as:

```
p = &a[0];
```



- To assign p to point to the next element, we could either :

```
p = a + 1;
p = &a[1];
```

- Notice that p = a + 1 advances the pointer **4** bytes and not 1 byte. This is because an integer is 4 bytes long and p is a pointer to an int.
- we can use the pointer p is exactly the same way as normal, i.e.:

```
*p = 5;
```

Summing an Array

```c
#include <stdio.h>

#define NUM 5

int sum(int a[]);

int main(void)
{

    int n[NUM] = {10, 12, 6, 7, 2};

    printf("%i\n", sum(n));
    return 0;
}

int sum(int a[])
{
    int sum = 0;

    for(int i=0; i<NUM; i++){
        sum += a[i];
    }
    return sum;
}
```

Execution :

37

```c
#include <stdio.h>

#define NUM 5

int sum(int a[]);

int main(void)
{

    int n[NUM] = {10, 12, 6, 7, 2};

    printf("%i\n", sum(n));
    return 0;
}

int sum(int a[])
{
    int sum = 0;

    for(int i=0; i<NUM; i++){
        sum += *(a + i);
    }
    return sum;
}
```

Execution :

37

```c
#include <stdio.h>

#define NUM 5

int sum(int* p );

int main(void)
{

    int n[NUM] = {10, 12, 6, 7, 2};

    printf("%i\n", sum(n));
    return 0;
}

int sum(int* p )
{
    int sum = 0;

    for(int i=0; i<NUM; i++){
        sum += *p;
        p++;
    }
    return sum;
}
```

Execution :

37

Pointers to Structures

- By default, structures are passed by value (copied) when used as a parameter to a function.
- But, like any other type, we could pass a pointer instead.
- The complication is that to access the elements of a structure via a pointer, we use the "->" operator, and not the ".".

```
void print_deck(card d[DECK], int n)
{
   char str[BIGSTR];
   for(int i=0; i<n; i++){
      print_card(str, &d[i]);
      printf("%s\n", str);
   }
   printf("\n");
}

#define SMALLSTR 20
void print_card(char s[], const card* p)
{
   // Note the +1 below : zero pips not used, but makes easier coding ?
   char pipnames[PERSUIT+1][SMALLSTR] = {"Zero", "One", "Two", "Three",
                                         "Four", "Five", "Six", "Seven",
                                         "Eight", "Nine", "Ten", "Jack",
                                         "Queen", "King"};
   char suitnames[SUITS][SMALLSTR] = {"Hearts", "Diamonds", "Spades", "Clubs"};
   sprintf(s, "%s of %s", pipnames[p->pips], suitnames[p->st]);
}
```

Nested Structures

```c
#include <stdio.h>

struct dateofbirth {
   unsigned char day;
   unsigned short month;
   unsigned short year;
};
typedef struct dateofbirth dob;

typedef struct {
   char* name;
   dob date;
} person;

void print_byval(person b);
void print_byref(const person* p);

int main(void)
{
   person a = {"Gary", {17, 5, 1999}};
   print_byval(a);
   print_byref(&a);
}

void print_byval(person b)
{
   printf("%s %hhu/%hi/%hi\n", b.name, b.date.day, b.date.month, b.date.year);
}

void print_byref(const person* p)
{
   printf("%s %hhu/%hi/%hi\n", p->name, p->date.day, p->date.month, p->date.year);
}
```

Execution :

```
Gary 17/5/1999
Gary 17/5/1999
```

Table of Contents

String Constants

```c
// A FAILED attempt to
// convert all 'n' chars to 'N'
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

void nify(char* s);

int main(void)
{

   nify("neill");
   return 0;

}

// In-Place : Swaps all 'n' -> 'N'
void nify(char* s)
{
   for(int i=0; s[i]; i++){
      if(s[i] == 'n'){
         s[i] = 'N';
      }
   }
}
```

- This looks (at first) like a sensible attempt to accept a string and change it *in-place* to capitalise all 'n' characters. It crashes though via a segmentation fault.
- With the usual compile flags we get no more information.
- But using:

  clang nify1.c -g3 -fsanitize=undefined -fsanitize=address -o nify1

  we find that

  ```c
  s[i] = 'N';
  ```

  is the culprit.
- It turns out that in main() we have passed a **constant** string to the function. This is in a part of memory that we have read-only permission.

Local Variables

```c
1  // A FAILED attempt to
2  // convert all 'n' chars to 'N'
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <assert.h>
7
8  #define LINE 500
9
10 char* nify(char* s);
11
12 int main(void)
13 {
14
15    char* s1 = nify("inconveniencing");
16    char* s2 = nify("neill");
17    assert(strcmp(s2, "Neill")==0);
18    assert(strcmp(s1, "iNcoNveNieNciNg")==0);
19    return 0;
20
21 }
22
23 // Local copy : Swaps all 'n' -> 'N'
24 char* nify(char* s)
25 {
26    char t[LINE];
27    strcpy(t, s);
28    for(int i=0; t[i]; i++){
29       if(t[i] == 'n'){
30          t[i] = 'N';
31       }
32    }
33    return t;
34 }
```

- Now we try to create a copy of the string, and return a pointer to it.
- With the usual compile flags we're told:

  nify2.c: In function ‘nify’:
  nify2.c:33:11: warning: function returns address of local variable [-Wreturn-local-addr]
  33 | return t;

- The string t is local to nify().
- What happens in this memory when outside the scope of this function is completely undefined.

Static Variables

```c
// A FAILED attempt to
// convert all 'n' chars to 'N'
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define LINE 500

char* nify(char* s);

int main(void)
{

   char* s1 = nify("inconveniencing");
   char* s2 = nify("neill");
   assert(strcmp(s2, "Neill")==0);
   assert(strcmp(s1, "iNcoNveNieNciNg")==0);
   return 0;

}

// Local copy : Swaps all 'n' -> 'N'
char* nify(char* s)
{
   static char t[LINE];
   strcpy(t, s);
   for(int i=0; t[i]; i++){
      if(t[i] == 'n'){
         t[i] = 'N';
      }
   }
   return t;
}
```

- We could just make the local string a static and return it's address couldn't we?
- This only works if we're very careful with the order in which we use the strings.
- This code fails because, in `main()`, by the time we `strcmp(s1, "iNcoNveNieNciNg")` the contents of `s1` have been overwritten by `"Neill"`.
- The pointers `s1` and `s2` are the same.

Using *malloc()*

- We must use malloc() instead.
- void* malloc(int n);
  allocates *n* bytes and returns a pointer to the allocated memory. The memory is not initialized.
- Now, when our function is called, a dedicated chunk of memory is allocated.
- This memory is always in scope until free() is used on it.
- We must free the memory somewhere though, otherwise memory leaks develop.
- We will see calloc() (and perhaps realloc()) later.

```c
#include <stdlib.h>
#include <string.h>
#include <assert.h>
char* nify(char* s);

int main(void)
{

   char* s1 = nify("inconveniencing");
   char* s2 = nify("neill");
   assert(strcmp(s2, "Neill")==0);
   assert(strcmp(s1, "iNcoNveNieNciNg")==0);
   free(s1);
   free(s2);
   return 0;

}

// malloc : Swaps all 'n' -> 'N'
char* nify(char* s)
{
   int l = strlen(s);
   char* t  = (char*)malloc(l+1);
   if(t==NULL){
      exit( EXIT_FAILURE );
   }
   strcpy(t, s);
   for(int i=0; t[i]; i++){
      if(t[i] == 'n'){
         t[i] = 'N';
      }
   }
   return t;
}
```

Variable Length Arrays

```c
// This code is not allowed by the -Wvla flag
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define WORD 500

int main(void)
{

    char s[WORD] = "String";
    int n = strlen(s) + 1;
    char t[n];
    // Deep copy: character by character
    strcpy(t, s);
    printf("%s %s\n", s, t);
    return 0;
}
```

- Here we duplicate a string into t.
- This is known as a variable length array.
- However, we will always use the -Wvla with the compiler to prevent them.
- There are a number of reasons for this:
  - Some C++ compilers don't accept it.
  - The memory comes off the stack not the heap, and you have no idea if the allocation has worked (it'll just crash if not)
  - https://nullprogram.com/blog/2019/10/27/
- None of these is a problem if we use malloc().

Memory Leaks

```c
// This leaks - but it's not obvious
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define WORD 500

int main(void)
{

    char s[WORD] = "String";
    int n = strlen(s);
    /* malloc() returns a pointer to memory that
       you have access to. Note forcing cast. */
    char* t = (char*) malloc(n+1);
    // If no space, returns NULL
    assert(t != NULL);
    // Deep copy: character by character
    strcpy(t, s);
    printf("%s %s\n", s, t);
    return 0;
}
```

- This code appears to work correctly.
- However, it actually **leaks**. The memory allocated was never free()'d.
- This is best found by running the program valgrind.

```
String String
==474==
==474== HEAP SUMMARY:
==474==     in use at exit: 7 bytes in 1 blocks
==474==   total heap usage: 2 allocs, 1 frees, 1,031 bytes allocated
==474==
==474== LEAK SUMMARY:
==474==    definitely lost: 7 bytes in 1 blocks
```

free()

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

#define WORD 500

int main(void)
{

    char s[WORD] = "String";
    int n = strlen(s);
    /* malloc() returns a pointer to memory that
       you have access to. Note forcing cast. */
    char* t = (char*) malloc(n+1);
    /* If no space, returns NULL */
    assert(t != NULL);
    /* Deep copy: character by character */
    strcpy(t, s);
    printf("%s %s\n", s, t);
    /* All malloc'd memory must be freed
       to prevent memory leaks */
    free(t);
    return 0;
}
```

- This code is now correct.

```
String String
==475==
==475== HEAP SUMMARY:
==475==     in use at exit: 0 bytes in 0 blocks
==475==   total heap usage: 2 allocs, 2 frees, 1,031 bytes allocated
==475==
==475== All heap blocks were freed -- no leaks are possible
```

Structures with Self-Referential Pointers

```c
// Store a list of numbers
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct data {
   int num;
   struct data* next;
};
typedef struct data data;

int main(void)
{
   data c = {5 , NULL};
   data b = {17, &c};
   data a = {11, &b};

   // print first number
   printf("%i\n", a.num);
   data* p = &a;
   // Can also get to it via p
   printf("%i\n", p->num);
   // Pointer chasing : The Key concept
   p = p->next;
   // We're accessing b, without using it's name
   printf("%i\n", p->num);
   p = p->next;
   // And c
   printf("%i\n", p->num);

   return 0;
}
```

- The structure contains a pointer to a something of it's own type (even before we've fully defined the struture itself).
- Here, if p points to a, then `p->next->next` point to c.

Linked Lists

```c
// Store a list of numbers (length unknown)
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define MAXNUM 20
#define ENDNUM 10

struct data {
   int num;
   struct data* next;
};
typedef struct data data;

void addtolist(data* tail);
void printlist(data* st);

int main(void)
{
   data *p, *start;
   start = p = calloc(1, sizeof(data));
   assert(p);
   p->num = rand()%MAXNUM;
   // Add other numbers to the list
   do{
      addtolist(p);
      p = p->next;
   }while(p->num != ENDNUM);
   printlist(start);
   // Need to free up list - not shown here ...
   return 0;
}
```

```c
// Create some new space and store number in it
void addtolist(data* tail)
{
   tail->next = calloc(1, sizeof(data));
   assert(tail);
   tail->next->num = rand()%MAXNUM;
}

void printlist(data* st)
{
   while(st != NULL){
      printf("%i ", st->num);
      st = st->next;
   };
   printf("\n");
}
```

Execution :

3 6 17 15 13 15 6 12 9 1 2 7 10