Subtleties and Common Errors of Programming in C

Sasha (Alexandre) Avreline
UBC BCS Tutor, Fall 2019

last updated: September 1, 2019

Abstract. This guide aims to take a closer look at subtleties and common errors encountered when programming in C and is intended for students taking CPSC 213 or subsequent courses at UBC.

Notes. This guide is meant as a supplement to those learning C and addresses specific situations rather than trying to provide a comprehensive introduction to the language. All examples are presented for illustrative purposes only and are not meant to give away solutions to any assiged course work problems. Any lengthy code snippets contained in this document must be referenced appropriately when used elsewhere.

Contents

1 C is Pass By Value

All function arguments in C, with the **exception of arrays** are passed in by value. This means that any operations done on those arguments remain local to the function, unless

(1) their values are returned from the function, or

(2) in addition, a pointer pointing to the memory location of the underlying variable was passed in and that location was modified.

Unlike in C++, there are no reference operators `&` in C. The ampersand's (`&`) primary uses in C are to either specify an address or its use as a bitwise AND operator.

Why are arrays so special? Well because in C, array parameters, regardless of the notation, are always passed in as pointers, see section 6 for a more in-depth discussion.

In the following example, in the function `foo1`, lines 9, 11 and 13 have absolutely no effect on the values of `p, q` and `pnt` in main, as all operations done in those lines remain local to `foo1`. However, line 10 does modify the value of `p`, as per line 26, `q` is a pointer that points to the memory location of `p`. Therefore, print statement on line 31 prints `12`.

```
 1  // passingValues1.c
 2  #include <stdio.h>
 3
 4  typedef struct Point {
 5      int x, y;
 6  } Point;
 7
 8  void foo1(int p, int *q, int a[], Point pnt) {
 9      p = p + 2;
10      *q = *q + 2;
11      q = 0;
12      a[0] = a[0] + 2;
13      pnt.x = 300;
14  }
15
16  void foo2(int p, int *q, int **r) {
17      *r = 0;
18      r = 0;
19      q = &p;
20  }
21
22  void main(void) {
23      Point pnt = {100, 200};
24      int a[] = {1, 2, 3, 4, 5};
25      int p = 10;
26      int *q = &p;
27      int **r = &q;
28
29      foo1(p, q, a, pnt);
30
31      printf("The value of p is %d\n", p);               // 12
32      printf("The value of q is %#x\n", q);              // 0x...
33      printf("The value of *q is %d\n", *q);             // 12
34      printf("The value of a[0] is %d\n", a[0]);         // 3
35      printf("The value of pnt.x is %d\n", pnt.x);       // 100
36
37      foo2(p, q, r);
38
39      printf("The value of q is %#x\n", q);              // 0
40      printf("The value of r is %#x\n", r);              // 0x...
41      printf("The value of *r is %#x\n", *r);            // 0
42  }
```

Likewise, line 12 does modify the value of a[0] as a is an array. Therefore, the print statement of line 34 prints 3.

So, as discussed, the statement q = 0 on line 11 had no effect. What if we did really want to set q to be a null pointer for some reason? One way to do so, would be to pass in a pointer pointing to q, a double pointer! This is illustrated on line 27 and in the function foo2. After the value of the double pointer r is set to 0 on line 17, q becomes a null pointer in main. However, lines 18 and 19 of foo2 have no further effect on any variables in main due to C's pass by value architecture.

In summary:

- to change the value of a regular variable in some function (without returning anything from that function), pass a pointer to that variable;
- to change the value of a pointer in some function (again without returning anything from that function), pass another pointer (i.e. a double pointer) to that variable.

Here is another example: func1 does nothing as far as changing the value of a is concerned, yet func2 does change the value of a. Likewise, func3 does nothing as far as changing the value of pointer a, yet func4 does change the value of the given pointer. Take a close look at the symmetry, similarities and differences between the four functions!

```c
// passingValues2.c
#include <stdio.h>

void func1(int a, int b) {
    a = b;
}

void func2(int *a, int b) {
    *a = b;
}

void func3(int *a, int *b) {
    a = b;
}

void func4(int **a, int *b) {
    *a = b;
}

void main(void) {
    int i = 1;
    int j = 2;
    int k = 3;
    int *x = &i;
    int *y = &k;
    int **v = &x;     // v points to x, which points to i

    func1(i, j);                            // does nothing
    printf("The value of i is %d\n", i);    // 1

    func2(x, j);                            // sets value of x to j
    printf("The value of i is %d\n", i);    // 2

    func3(x, y);                            // does nothing
    printf("The value of *x is %d\n", *x);  // 2

    func4(v, y);                            // points x to k (and not i)
    printf("The value of *x is %d\n", *x);  // 3
}
```

2 Returning Multiple Values

All functions in C are limited to returning just one value. What happens if we need to return more than one value from a function? The two approaches are: structs and pointers.

```c
 1  // returningValues.c
 2  #include <stdio.h>
 3
 4  typedef struct Point {
 5      int x, y, z;
 6  } Point;
 7
 8  Point foo1(int x, int y, int z) {
 9      x += 10;
10      y += 20;
11      z += 30;
12      Point p = {x, y, z};
13      return p;
14  }
15
16  void foo2(int *a, int *b, int *c) {
17      *a = *a + 1;
18      *b = *b + 2;
19      *c = *c + 3;
20  }
21
22  void main(void) {
23      int x = 1, y = 2, z = 3;
24
25      Point p = foo1(x, y, z);
26      x = p.x, y = p.y;
27      z = p.z;
28
29      printf("x is %d\n", x); // 11
30      printf("y is %d\n", y); // 22
31      printf("z is %d\n", z); // 33
32
33      int *a, *b, *c;
34      a = &x, b = &y, c = &z;
35
36      foo2(a, b, c);
37
38      printf("x is %d\n", x); // 12
39      printf("y is %d\n", y); // 24
40      printf("z is %d\n", z); // 36
41  }
```

Suppose we would like to update `x, y` and `z` in `foo1`. Since C is pass by value, and there are no pointers passed to `foo1`, so `foo1` must return the updated values. Therefore, those values have to be packaged into a struct. Struct is declared on lines 3-5 and a new struct is then constructed on line 11 and is returned on line 12. Lines 27-29 print updated values of `x, y` and `z`.

As an aside: declaring struct using the `typedef` notations saves the need to write the keyword `struct` whenever a new struct of that type is used.

Another way to update the values of all `x, y` and `z`, as discussed in the previous section, is to pass in pointers that point to their memory locations. In this example, pointers are declared and are initialized accordingly on lines 31-32. The values they point to are updated in the function `foo2` which has a return type of void. Lines 36-38 print the updated values of `x, y` and `z`.

3 Incrementing Pointers and Values

We are all familiar with the standard shortcuts for incrementing or decrementing a value: `i++, ++i, i--` and `--i`. Recall that placing the operator before the value increments or decrements the value before it is used. How do those operators behave when there are pointers involved?

The increment (or decrement) operator has a higher precedence than the dereference operator `*`, so writing `*p++` or `*++p` increments **the address** stored in the pointer and **not the value** that the pointer points to. Therefore, lines 8 and 12 move the pointer `p` to point to elements 2 and 3 of the array, respectively. On line 8, the pointer is post-incremented, so at that point, it is still the value of the first element of `a` that gets saved into `x`. On line 12; however, the pointer is incremented first, so it is already the value of the third element of `a` that gets saved into `y`.

If brackets are placed around the dereference operator `*`, then the dereference operator is applied first, before the increment operator. Therefore, on lines 16 and 20 it is **the value** that is pointed to by `p` gets incremented, and **not the address** that is stored in the pointer. Likewise, a post-increment happens on line 16, so the original value of `30` gets saved into `v`, while a pre-increment takes place on line 20.

```
 1  // pntInc.c
 2  #include <stdio.h>
 3
 4  void main(void) {
 5      int a[] = {10, 20, 30, 40, 50};
 6      int *p = a;
 7
 8      int x = *p++;
 9      printf("After *p++ the value of x is %d\n", x);       // 10
10      printf("After *p++ the value of *p is %d\n", *p);     // 20
11
12      int y = *++p;
13      printf("After *++p the value of y is %d\n", y);       // 30
14      printf("After *++p the value of *p is %d\n", *p);     // 30
15
16      int v = (*p)++;
17      printf("After (*p)++ the value of v is %d\n", v);     // 30
18      printf("After (*p)++ the value of *p is %d\n", *p);   // 31
19
20      int w = ++(*p);
21      printf("After ++(*p) the value of w is %d\n", w);     // 32
22      printf("After ++(*p) the value of *p is %d\n", *p);   // 32
23  }
```

As an aside: we didn't really need brackets on line 20. Writing `int w = ++*p;` would have had the same effect. Why? Precedence of operators `++` and `*`, in addition to other rules, is also determined on a right-to-left basis in C, and since `*` is further to the right in this case, it is applied first. However, please do not rely on complex analysis like this when writing complex C code. When in doubt, use brackets! In fact, for clarity, lines 8 and 12 could been equally written as `int x = *(p++);` and `int y = *(++p);`.

4 Side Effects of Incrementing a Pointer

One needs be careful of side effects whenever incrementing pointers in place: the address stored in the pointer changes, and unless this is accounted for, the program could crash or exhibit unpredictable behavior.

Consider the following example. There, `foo` is a function that makes a copy of the input string and returns the copy. There are of course functions `strdup` and `strcpy` in the C's string library that do the same, this example here is for illustrative purposes. Also for illustrative purposes, the copy is done dynamically via pointer arithmetic.

The while loop increments both the `str` and `cpy` pointers, copying the values pointed to by `str` into memory pointed to by `cpy`. The new string is null terminated on line 11 (null terminating the strings is of course something very important to do in C!). Then on line 12 it seems natural to return `cpy`, as that is where we stored the new string. However, after all of those increments that happened in the while loop, `cpy` now points to the end of the string. Therefore, nothing gets printed on line 18 and then the program crashes as an invalid pointer is passed into free on line 19 (i.e. a pointer that no longer points to the beginning of the memory region allocated by malloc on line 7).

```
1  // pntIncS0.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  char *foo(char *str) {
7      char *cpy = malloc((strlen(str) + 1) * sizeof(char));
8      while (*str)
9          *cpy++ = *str++;
10
11     *cpy = '\0';
12     return cpy;
13 }
14
15 int main() {
16     char *str = "abcde";
17     char *cpy = foo(str);
18     printf("Result is %s\n", cpy);
19     free(cpy);
20 }
```

There are two ways to fix this issue. First approach, which is not recommended, is to move any incremented pointers back to their original locations. In particular, instead of returning `cpy`, we could return `cpy - strlen(str)` which effectively undoes all incrementation done to `cpy` in the while loop.

However, we must also be careful to save `strlen(str)` into a variable at the beginning of `foo`, as the `str` gets moved to the end of the string by the while loop as well, and any subsequent calls to `strlen` would return `0`. The code presented below executes without any errors or memory leaks.

```c
// pntIncS1.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *foo(char *str) {
   int len = strlen(str); // !!!
   char *cpy = malloc((strlen(str) + 1) * sizeof(char));
   while (*str)
      *cpy++ = *str++;

   *cpy = ’\0’;
   return cpy - len; // !!!
}

int main() {
   char *str = "abcde";
   char *cpy = foo(str);
   printf("Result is %s\n", cpy);
   free(cpy);
}
```

A better approach, which doesn't require any manual accounting of pointer locations, is simply to save the address of the beginning of the new string into a new pointer. In the code presented below, original value of `cpy` is saved into `cpy_pnt` on line 8. Then `cpy_pnt` remains fixed, while `cpy` is incremented in the while loop. At the end, `cpy_pnt` is returned from `foo` as is and there are no issues or memory leaks.

```c
 1  // pntIncS2.c
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5
 6  char *foo(char *str) {
 7     char *cpy = malloc((strlen(str) + 1) * sizeof(char));
 8     char *cpy_pnt = cpy; // !!!
 9     while (*str)
10        *cpy++ = *str++;
11
12     *cpy = ’\0’;
13     return cpy_pnt;
14  }
15
16  int main() {
17     char *str = "abcde";
18     char *cpy = foo(str);
19     printf("Result is %s\n", cpy);
20     free(cpy);
21  }
```

A third approach is, of course, to just avoid any pointer arithmetic or any dynamic allocation. The code becomes much simpler. The main trade off is that all string sizes need to be specified when strings are declared. Some memory is wasted as string sizes are usually declared to be somewhat of an arbitrary length that is larger than expected string lengths, so that there is some additional buffer just in case (typical value of string length is 81, for an 80 character string with an extra character for the null pointer at the end). However, in many applications, this is perhaps a rather small price to pay for safer code. Sample code is presented below.

```c
// pntIncS3.c
#define STR_LEN 81
#include <stdio.h>
#include <string.h>

void foo(char *str, char *cpy) {
   int i = 0;
   while (str[i])
      cpy[i++] = str[i];

   cpy[i] = '\0';
}

int main() {
   char str[STR_LEN] = "abcde";
   char cpy[STR_LEN];
   foo(str, cpy);
   printf("Result is %s\n", cpy);
}
```

5 Avoiding Memory Leaks due to String.h Functions

In section 4, we saw how copying strings dynamically requires allocation of new memory on the heap and that such memory must be eventually freed to avoid memory leaks. What about when C's build-in string functions from `string.h` are used? Do they allocate memory that must be freed? The answer is yes.

Consider the following code which uses build-in `strdup` function to duplicate a string. If this code is ran with line 10 commented out, then Valgrind states there is a memory leak. Once line 10 is enabled, there are no more memory leaks.

This phenomenon is an example when free must be called *without* a corresponding existence of an explicit malloc call anywhere in the code (i.e. the malloc happens inside the implementation of the `strdup` function)!

```c
1  // strMemLeak1.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int main() {
7     char *str = "abcde";
8     char *cpy = strdup(str);
9     printf("Result is %s\n", cpy);
10    free(cpy);
11 }
```

To avoid such hassles and any manual memory management, we could, once again, use statically declared strings and the function `strcpy`. This function doesn't allocate any new memory, since it expects to be given a destination location as an argument.

```c
// strMemLeak2.c
#define STR_LEN 81
#include <stdio.h>
#include <string.h>

int main() {
   char str[STR_LEN] = "abcde";
   char cpy[STR_LEN];
   strcpy(cpy, str);
   printf("Result is %s\n", cpy);
}
```

6 Pointers vs Arrays

As mentioned in section 1, pointers and arrays in C are closely related. In particular, array parameters, regardless of the notation, are always passed in as pointers. In the following code, four different versions of function foo are presented. All version calculate the sum of elements of the array `a`:

- `foo1` accepts input `a` as an *array* and uses *array notation* to compute the sum,
- `foo2` accepts input `a` as a *pointer* and uses *array notation* to compute the sum,
- `foo3` accepts input `a` as a *pointer* and uses *pointer arithmetic* to compute the sum,
- `foo4` accepts input `a` as an *array* and uses *pointer arithmetic* to compute the sum.

All functions compile and compute the sum without errors. In `foo3` and `foo4` the for loop does not have an initial condition, since `a` already points to the start of the array. It is important to determine the end of the array and store it into the variable `aEnd` prior to the start of the loop, as the value `a + n` will increase whenever `a` is incremented in the loop.

```c
// arrayPnt1.c
#include <stdio.h>
#include <string.h>

int foo1(int a[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum;
}

int foo2(int *a, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum;
}

int foo3(int *a, int n) {
    int sum = 0;
    int *aEnd = a + n;
    for (; a < aEnd; a++)
        sum += *a;
    return sum;
}

int foo4(int a[], int n) {
    int sum = 0;
    int *aEnd = a + n;
    for (; a < aEnd; a++)
        sum += *a;
    return sum;
}

void main() {
    int a[] = {1, 2, 3, 4, 5};
    int n = 5;

    printf("Foo1 sum is %d\n", foo1(a, n));
    printf("Foo2 sum is %d\n", foo2(a, n));
    printf("Foo3 sum is %d\n", foo3(a, n));
    printf("Foo4 sum is %d\n", foo4(a, n));
}
```

Stepping away from arguments to a function, where array and pointers are essentially the same, let us look at some of the differences between pointers are arrays. Array elements could be accessed using pointer notation as shown on lines 17-18 below, but pointer arithmetic is not appropriate with array variables, such as on line 21.

However, as discussed on the previous page, passing an array variable to a function enables pointer arithmetic to be done, as is done in `foo` below. Also, explicitly casting an array to a pointer enables pointer arithmetic to be done as well, as is shown on lines 24-27 below.

```
 1 // arrayPnt2.c
 2 #include <stdio.h>
 3 #include <string.h>
 4 
 5 int foo(int a[], int n) {                       // OK!
 6     int sum = 0;
 7     int *aEnd = a + n;
 8     for (; a < aEnd; a++)
 9         sum += *a;
10     return sum;
11 }
12 
13 void main() {
14     int a[] = {1, 2, 3, 4, 5};
15     int n = 5;
16 
17     for (int i = 0; i < n; i++)                 // OK!
18         printf("val is %d\n", *(a + i));
19 
20     int *aEnd = a + n;
21     for (; a < aEnd; a++)                       // does not compile
22         printf("val is %d\n", *a);
23 
24     int *b = a;
25     int *bEnd = b + n;
26     for (; b < bEnd; b++)                       // OK!
27         printf("val is %d\n", *b);
28 
29     printf("Sum is %d\n", foo(a, n));
30 }
```

Next, since arrays are statically declared, their values cannot be re-assigned. For example, line 2 in the following code does not compile. Re-assigning pointers on the other hand is totally fine. For example, lines 4 and 5 compile and any of the foo functions from the previous page could determine the sum of `b` to now be `40`.

```
1 int a[] = {1, 2, 3, 4, 5};
2 a = {6, 7, 8, 9, 10};                       // does not compile
3 
4 int* b = (int[]) {1, 2, 3, 4, 5};
5 b = (int[]) {6, 7, 8, 9, 10};               // OK!
```

Similar behavior is exhibited in the following example with strings, which are just character arrays in C. In particular, strings statically declared in C cannot be re-assigned; however, some ways to modify such strings are:

(1) on an element-by-element basis, for example via `str[1] = 'a';`, or

(2) by implicitly casting them to a pointer when passing to C's string library functions (see line 7 below and also the last example of section 5).

```c
char str1[] = "abcde";
str1 = "fghij";                         // does not compile

char* str2 = "abcde";
str2 = "fghij";                         // OK!

strcpy(str1, str2);                     // OK!    str1 is cast to char* when passed into strcpy
                                        //        strcpy copies str2 directly into str1
```

7 Fancy and Silly Loop Constructs

In the previous section, in functions `foo3` and `foo4` we saw loop constructs without initial values. In fact we can take this idea further and write for loops without any expressions at all inside the `for` expression. The following code is written in a silly way, but computes the sum of `a` just fine.

```c
// loops1.c
#include <stdio.h>

void main() {
    int a[] = {1, 2, 3, 4, 5};
    int n = 5, sum = 0;

    for (int i = 0; i < n; i++)
        sum += a[i];

    printf("Sum is %d\n", sum);

    sum = 0;

    int j = 0;
    for (;;) {
        if (j >= n)
            break;
        sum += a[j];
        j++;
    }

    printf("Sum is %d\n", sum);
}
```

Recall that the `break` keyword causes the loop to exit right away. The `continue` keyword causes the current iteration of the loop to exit, but the the subsequent iterations of the loop will continue to run.

It is also possible to construct loops with multiple initial values, as well as multiple terminating conditions and multiple update expressions. In the following example, lines 5-13 and lines 15-22 do the exact same work and print the exact same values, yet the first for loop uses just one index `i`, while the second one uses two indices `i` and `j`.

```c
// loops2.c
#include <stdio.h>

void main() {
    puts("\n");
    int a[] = {1, 2, 3, 4, 5};
    int n = 5;

    for (int i = 1; i < n; i++)
        a[i] += a[i - 1];

    for (int i = 0; i < n; i++)
        printf("Final value of a[%d] is %d\n", i, a[i]);

    puts("\n");
    int b[] = {1, 2, 3, 4, 5};

    for (int i = 1, j = 0; i < n; i++, j++)
        b[i] += b[j];

    for (int i = 0; i < n; i++)
        printf("Final value of b[%d] is %d\n", i, b[i]);
}
```

There is also the `goto` construct in C, which results in an unconditional jump to some predetermined label, as is shown in the code snippet below. In this code, while loop will exit and line 5 will be skipped. Frequent usage of `continue` and `goto` keywords is considered poor coding style.

```c
1  while (1) {
2      goto mylabel;
3  }
4
5  printf{"some text");
6
7  mylabel:
8      printf{"some more text \n");
```

8 Arguments to Main

C's main function takes in two arguments:

1. `int argc` is an integer that stores the number of command line arguments passed to the program, when the program was called from the command line
2. `char **argv` is an array that contains those command line arguments as strings (i.e. as char pointers, so `argv` is an array of char pointers often declared as `char *argv[]`)

The first element of `argv` array is usually name of the program itself, and thus the value of `argc` is one greater than the actual number of arguments supplied. So, for example, typing into the command line `programName 1 23 4 5` results in the following values for the arguments to main:

- `argc: 5`
- `argv: {‘‘programName’’,‘‘1’’,‘‘23’’,‘‘4’’,‘‘5’’}`

If the function expects numeric arguments, since arguments are stored as strings in `argv`, the `strtol` function must be used to convert them into long integers. The `strtol` function requires the following arguments:

- the string to be converted into the long integer
- a char** usually denoted ep (stands for “error pointer”) into which strtol will save any left over non-numeric characters
- numeric base to be used in the conversion (i.e. 2, 8, 10, 16)

If, after the call to strtol, the value of ep is null, then conversion was successful and the command line argument was indeed a number; otherwise ep is not null and will contain any left over non-numeric characters.

Overall, there are three steps to be done in main with respect to the arguments:

1. check the value of argc to see if the correct number of arguments has been supplied when the program was called,
2. convert the values of argv into long integers using strtol, if integer arguments are expected,
3. use the arguments

Below are two examples of these steps. In the first case, args, the array that is used to store converted arguments is allocated statically, while in the second case, this array is allocated dynamically, based on the value of argc. In this case the program expects anywhere from 2 to 4 arguments. Since the first value of the argv array is usually the name of the program, care must be taken with respect to loop indices when getting values from this array (as well as with the value of argc which is one greater than the actual number of arguments).

```c
// mainArgs1.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

    // Check no.
    if (argc < 2 + 1 || argc > 4 + 1) {
        printf("Invalid no. of arguments\n");
        return -1;
    }

    char *ep;
    int args[4];

    // Convert
    for (int i = 0; i < argc - 1; i++) {
        args[i] = strtol(argv[i + 1], &ep, 10);
        if (*ep) {
            printf("An argument is not a number, x is %d, value of *ep is %s\n",
                   args[i], ep);
            return -1;
        }
    }

    // Use args
    for (int i = 0; i < argc - 1; i++)
        printf("Argument %d is %d\n", i, args[i]);

    return 0;
}
```

```
// mainArgs2.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

   // Check no.
   if (argc < 2 + 1 || argc > 4 + 1) {
      printf("Invalid no. of arguments\n");
      return -1;
   }

   char *ep;
   int *args = malloc(argc * sizeof(int));

   // Convert
   for (int i = 0; i < argc - 1; i++) {
      args[i] = strtol(argv[i + 1], &ep, 10);
      if (*ep) {
         printf("An argument is not a number, x is %d, value of *ep is %s\n",
               args[i], ep);
         free(args);
         return -1;
      }
   }

   // Use the args
   for (int i = 0; i < argc - 1; i++)
      printf("Argument %d is %d\n", i, args[i]);

   free(args);
   return 0;
}
```

9 Malloc, Calloc, Realloc and Buffers

The following calls to `malloc` and `calloc` are almost identical and will allocate the same amount of space for your needs. The only major difference is that `calloc` sets the entire allocated memory space to "0", while `malloc` does not. Therefore, `malloc` is generally faster and is of course more commonly used.

- `malloc(n * sizeof(X))`
- `calloc(n, sizeof(X))`

The `realloc` function is used to expand the allocated memory region, when the space initially allocated by `malloc` or `calloc` has been exhausted. The `realloc` function takes in a pointer to the existing memory space and the *total* size of the expanded memory region. It returns a (potentially totally different!) pointer to the allocated memory region. Therefore, it is very important to update the value of the pointer to the allocated memory region, to the value returned by `realloc`; otherwise, subsequent calls to `realloc` or `free` will fail. The `realloc` function would have automatically freed any existing pointers it was given, had it changed the location of the allocated memory region.

On the following page there is a basic example that illustrates a typical use of `realloc`. Memory for a string of length `BUF_SIZE` is initially allocated by `malloc` on line 9 and characters are added to this string from the command line input via the `getchar` method, until the return key is pressed. If the number of characters typed in exceeds BUF_SIZE, then realloc is called on line 21 to expand the size of the buffer.

```
 1  // mem.c
 2  #define BUF_SIZE 10
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5
 6  int main() {
 7      int count = 0;
 8      int cur_size = BUF_SIZE;
 9      char *text = malloc(cur_size * sizeof(char));
10
11      printf("Type in your input\n");
12
13      while (1) {
14          char c = getchar();
15
16          if (c == ’\n’)
17              break;
18
19          if (count >= cur_size) {
20              cur_size += BUF_SIZE;
21              text = realloc(text, cur_size * sizeof(char));
22          }
23
24          text[count++] = c;
25      }
26
27      text[count] = ’\0’;
28      printf("The text is %s\n", text);
29      free(text);
30  }
```

10 Casting Void Pointers

Descendants of C such as C++ and Java support generic programming as well as class and function templates. For example, in C++ or Java, it is possible to construct a template class for a queue of some generic type T and then create specific instances of this queue where T is an integer, a string, etc. In fact, all of C++'s standard library containers are generic templates, which is why this library is called the "standard template library".

In C, the idea of generic programming is partially supported via void pointers: void*. Essentially, void pointers are often used to bypass C's type check system and to have functions which are open to accepting arguments of any type. This approach allows to construct function wrappers that wrap around groups of similar functions. Having said that, void pointers are acceptable only places in such as argument declaration and ***must be cast*** to specific types when those arguments are actually used. All this is best illustrated using an example.

In the following example, there is a struct called "Student" that would be used to store information that is contained in a student's academic record. Essentially, the components of the struct behave sort of like fields would in an object oriented language. Then there are the setter functions for each of the first four fields, which are very similar to each other, expect that some set fields that are of the integer type, and others set fields that are of the string type. The goal is to group those setter functions and to allow them to be called via the `setField` wrapper which also performs some console logging tasks that are common to all of the setter functions.

```c
// voidPnt.c
#include "stdio.h"

typedef struct Student {
    char **first_name;
    char **last_name;
    int *year;
    int *gpa;
    int *id;
} Student;
char *fields[] = {"FirstName", "LastName", "Year", "GPA"};
enum FIELDS {FIRSTNAME = 0, LASTNAME = 1, YEAR = 2, GPA = 3};

void setFirstName(Student *s, void *fn) {
    s->first_name = (char **)fn;
}

void setLastName(Student *s, void *ln) {
    s->last_name = (char **)ln;
}

void setYear(Student *s, void *year) {
    s->year = (int *)year;
}

void setGpa(Student *s, void *gpa) {
    s->gpa = (int *)gpa;
}

typedef void (*FunctionTemplate)(Student *s, void *val);
FunctionTemplate functions_array[] = {setFirstName, setLastName, setYear, setGpa};

void setField(int i, Student *s, void *val) {
    printf("Changing %s for student no: %d.\n", fields[i], *(s->id));
    functions_array[i](s, val);
    printf("Procedure %s completed for student no: %d.\n", fields[i], *(s->id));
}

int main() {
    char *fn = "John";
    char *ln = "Doe";
    int year = 2;
    int gpa = 80;
    int id = 9999;
    Student s = {&fn, &ln, &year, &gpa, &id};

    char *new_name = "Jane";
    setField(FIRSTNAME, &s, &new_name);
    printf("%s\n", *(s.first_name));

    int new_year = 3;
    setField(YEAR, &s, &new_year);
    printf("%d\n", *(s.year));

    int new_gpa = 90;
    setField(GPA, &s, &new_gpa);
    printf("%d\n", *(s.gpa));

    return 0;
}
```

To do so, it is critical that all setter functions take in the same number of arguments of the same type. Here is where void pointers come in handy. The functions that set first and last name need the student record and the new names as arguments. The functions that set year and GPA need the student record and the new year or new GPA as arguments. Names are strings, year and GPA are integers. In order for all functions to share the same argument list, the second argument must have `void*` type.

As soon as all setter functions are set up to take in the same argument list, they could be grouped into an array of function pointers as done on lines 30-31. Moreover, having an enum on line 12 that converts field names into integers allows for the setter functions to be called in a very readable way, such as `setField(FIRSTNAME, &s, &new_name)`. The `setField` method wraps some console logging commands, common to all setter functions calls, around the actual function call.

Of course, as mentioned, in the actual setter functions it is then very important to cast the void pointers into the types those functions are meant to work with. For example, in the `setFirstName` function, the cast is done via `(char **)fn`. In practice, many compilation errors are simply from forgetting to cast void pointers prior to using them.

11 An Example: Splitting a String

In the final section of this guide, three different implementations of a longer example are presented to illustrate some of the ideas discussed.

The task at hand is to split a string into segments that are separated by some specific delimiter or token. A specific example could be splitting up a fully qualified domain name, such as `www.example.com` into the individual components which are `www`, `example` and `com` in this case (the token here is ".""). Or it could be splitting up a file path into individual components: `/path/to/dir` into `path`, `to` and `dir` (the token here is "/"). String manipulation in C is always a challenge, so this example presents another opportunity to practice that.

The approach will be to use the `strtok` function from the `string.h` library. The function takes two arguments: the string to be split and the token. Then it returns the first component of the string, up to the first discovered token. In order to get subsequent components, the `strtok` function is repeatedly called, this time with `null` as the first argument.

It should be noted that `strtok` is a destructive function, in the sense that the given string will be destroyed. In particular, the way `strtok` works is by replacing all discovered tokens with null characters, which forms separate null-terminated strings out of the components. Therefore, prior to using `strtok`, it is wise to make a copy of the string you are working with.

In the **first approach** to this task, everything is done statically. In particular, it is assumed that there will be at most 10 token-separated components in the given string and that each of those components will be at most 80 characters long. Those values are used to declare a two dimensional static array of chars, called `components`, where the discovered components will be saved to. The `split_str` function takes in the input string, the token and the `components` array to store the results. Note that when a two dimensional array is declared in the argument list to this function, there is no need to specify the first dimension, but specifying second dimension is a must.

A copy of the input string is made on lines 8-9. The char token is converted to a string on line 10 (in particular, a null-terminating char is added), this is needed as `strtok` requires a `char*` as the second argument. Initial call to `strtok` happens on line 13, and subsequent calls are done on line 16, inside the while loop. The while loop continues while there are non-null results returned from `strtok` (i.e. while new components are discovered). The results are copied into appropriate locations in the `components` array via the `strcpy` function.

```
 1  // splitStr1.c
 2  #define NO_OF_COMPONENTS 10
 3  #define STR_LEN 81
 4  #include <stdio.h>
 5  #include <string.h>
 6
 7  int split_str(char *input_str, char token, char components[][STR_LEN]) {
 8         char input_str_copy[STR_LEN];
 9         strcpy(input_str_copy, input_str);
10         const char *token_string = (char[]){token, '\0'};
11
12         int i = 0;
13         char *result = strtok(input_str_copy, token_string);
14         while (result) {
15                strcpy(components[i++], result);
16                result = strtok(NULL, token_string);
17         }
18
19         return i;
20  }
21
22  int main() {
23         char components[NO_OF_COMPONENTS][STR_LEN];
24         char *str = "www.example.com";
25         char token = '.';
26
27         int i = split_str(str, token, components);
28         printf("Input string is %s\n", str);
29
30         for (int j = 0; j < i; j++)
31                printf("Component is %s\n", components[j]);
32  }
```

In the **second approach**, things are done dynamically. This time, no assumptions are being made about how many token-separated components there will be in the given string, nor about how long they will be. As before, the `components` array is a two dimensional char array; however, in this case, it is initialized dynamically via `malloc` in the `split_str` function.

In order to work with the `components` array dynamically and not waste any space, we must know how many token-separated components there are in the given string. Therefore, the number of tokens in the given string is first counted in the while loop on lines 10-12. Then, on line 16, this count is used to allocate an array of char pointers that is large enough to hold the result. Note that here it is the `sizeof(char *)` [4 bytes] and not the `sizeof(char)` [1 byte] that is passed to `malloc`.

Why is the length of the allocated array $i + 2$? Well if there are i tokens discovered in the while loop on lines 10-12, there will be $i + 1$ token separated components (i.e. in www.example.com there are two tokens and three components). As for $i+2$, the last spot in the array will be used to store a null pointer (see line 25). This way, the calling function (and any other function that makes use of the result) could determine the length of the array, without the length being explicitly passed back. In fact, this construction is applied in the while loops that start on lines 38 and 43. Essentially, the construction here is similar to the construction of C's null-terminated string.

The rest of the code is quite similar to the one in the previous approach. In fact, lines 7, 15, 19, 20 and 22 are exactly the same. Pointer arithmetic is used in all of the while loops; therefore, the pointer to the start of the `components` array is saved into a different pointer prior to the start of the loops (lines 17 and 37) – c.f. section 4. Also in this approach `strdup` is used to duplicate the input string instead of `strcpy` and the memory from `strdup` is freed on line 26 – c.f. section 5. The actual component strings that are duplicated into the `components` array on line 21 are freed on line 44.

```
 1  // splitStr2.c
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5
 6  char **split_str(char *input_str, char token) {
 7      int i = 0;
 8      char *input_str_copy = input_str;
 9
10      while (*input_str_copy++)
11          if (*input_str_copy == token)
12              i++;
13
14      input_str_copy = strdup(input_str);
15      const char *token_string = (char[]){token, '\0'};
16      char **components = malloc((i + 2) * sizeof(char *));
17      char **components_pnt = components;
18
19      char *result = strtok(input_str_copy, token_string);
20      while (result) {
21          *components++ = strdup(result);
22          result = strtok(NULL, token_string);
23      }
24
25      *components = NULL;
26      free(input_str_copy);
27      return components_pnt;
28  }
29
30  int main() {
31      char *str = "www.example.com";
32      char token = '.';
33
34      char **components = split_str(str, token);
35      printf("Input string is %s\n", str);
36
37      char **components_pnt = components;
38      while (*components)
39          printf("Component is %s\n", *components++);
40
41      // free memory
42      components = components_pnt;
43      while (*components)
44          free(*components++);
45      free(components_pnt);
46  }
```

In the **third approach**, the goal is to address two issues with the dynamic approach that was just presented:

1. there is an extra while loop that first counts the number of components, and
2. the `components` array that stores the result gets created and initialized inside the `split_str` function (unlike in the first approach where this array was created and initialized in main)

The approach developed here may not be more efficient than the second one and is presented more for illustrative and educational purposes.

Use of `realloc` would addresses the first concern: the initial assumption will be that there will be no discovered components. So, on line 28, `malloc` allocates an array large enough just to store the null pointer that terminates the array. Then, as each additional component is discovered, `realloc` increase the array's size by one on line 15. The while loop on lines 13-16 no longer uses pointer arithmetic as the `components` pointer must be updated to point to a potentially new memory region after each call to `realloc` and should not be independently incremented.

```c
 1  // splitStr3.c
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5
 6  void split_str(char *input_str, char token, char ***components_addr) {
 7      int i = 0;
 8      char *input_str_copy = strdup(input_str);
 9      const char *token_string = (char[]){token, '\0'};
10      char **components = *components_addr;
11
12      char *result = strtok(input_str_copy, token_string);
13      while (result) {
14          components[i] = strdup(result);
15          components = realloc(components, (++i + 1) * sizeof(char *));
16          result = strtok(NULL, token_string);
17      }
18
19      components[i] = NULL;
20      *components_addr = components;
21      free(input_str_copy);
22  }
23
24  int main() {
25      char *str = "www.example.com";
26      char token = '.';
27
28      char **components = malloc(sizeof(char *));
29      char ***components_addr = &components;
30      split_str(str, token, components_addr);
31      printf("Input string is %s\n", str);
32
33      char **components_pnt = components;
34      while (*components)
35          printf("Component is %s\n", *components++);
36
37      // free memory
38      components = components_pnt;
39      while (*components)
40          free(*components++);
41      free(components_pnt);
42  }
```

Use of a triple pointer addresses the second concern. The `components` array is a double char pointer and the value of this pointer is changed inside the `split_str` function, due to calls to `realloc`. Therefore, to allow `split_str` to keep the void return type, `components` must be passed into it by pointer, which makes the type of this argument a triple pointer (double pointer plus one) – c.f. sections 1 and 2. The value of the triple pointer is updated on line 20.

Beyond those two changes, everything in the code presented in this approach is similar or identical to the code presented in the second approach.

References

[1] Kochan, Stephen. Programming in C, Fourth edition. Addison-Wesley Professional: 2014.

[2] “The C++ Resource Network”, http://www.cplusplus.com/