RICHARD ROBINSON

C, UNIX, & JAVA

Contents

Intro to C 5

Types 7

Pointers 9

Structures 11

Application of Structs 13

Even More Pointers 15

Files 17

Intro to C

The Basics

In C, there only exist multiline `/**/` comments. As well, there is no garbage collection, classes, exception, nor strings. The basic C program syntax is

```
#include <stdio.h>
main() {
    /* body */
    return 0;
}
```

The `#include` tag acts as a macro for whatever file is called. In C, the null character `\0` is critical, as it is appended to all arrays of characters and therefore represents the end of a string. A byte is 8 bits is 4 binary characters.

Basic I/O

The basic input and output commands in C are `getchar()`, `putchar()`, `printf()`, and `scanf()`;

- The `int getchar()` command reads one character at a time and returns `EOF` at end of line, defined as `-1`.
- The `int putchar(c)` commands returns `c` to the standard output or `EOF` if an error occurs.
- The `printf("str")` command outputs the string with optional arguments.
- The `scanf("\%x", &x)` command reads the input, assigning `\%x` to `x`. The `x` type is a hexadecimal integer.

To parse a string by character in C, a while loop of the form

```
int c;
while ((c = getchar()) != EOF) { }
```

is used. Other types in addition to `x` are `c, d, f, lf, s`. The basic data types in C are `char` (8 bits), `int` (16/32 bits), `float` (4 bytes), and `double` (8 bytes). A string is an array of characters such ended by null, with syntax

```
char varname[] = "string"; /* or */
char *varname = "string" /* or */
char varname[] = {'chr1', 'chr2', ..., '\0'}
```

Because of the nature of strings, `printf()` is equivalent to

```c
int printf(char *format, args);
```

Printf and Scanf

The format code `"\%0n.df"` represents the following characteristics:

- The `n` is the total allotment of characters for the number, printing the characters from the right.
- The `0` replaces all the leading whitespace characters allotted before the number with 0.
- The `.d` is the number of decimal places to be included; the decimal point counts as a character for `n`.
- The `f` represents the number is of type float.

The default number of decimal places is 6. As well, a negative `n` value adds whitespace to the right instead of left. To read an integer from the input, the syntax

```c
int num; scanf("%d", &num);
```

is used. The function stops reading upon EOF or a failed input occurs. Additionally, it returns the number of successfully matched inputs, and returns 0 for an error. Additionally, file-wise scanning and printing are given via

- `prog < infile`: prog reads chars from infile
- `prog > infile`: prog writes chars to infile
- `prog1 | prog2`: input of prog2 is output of prog1

Example

An example snippet to calculate the number of words in the input is shown below:

```c
#define TRUE 1;
#define FALSE 0;
for ( int new_word = TRUE; (c = getchar() ) != EOF;) {
   if (c == '␣' || c == '\n') new_word = TRUE; // works for multiple whitespaces
   else if (new_word == TRUE) {
       new_word = FALSE; ++num_words; // count number of whitespace groups
   }
}
```

Types

Strings

Arrays have their sizes explicitly with index ranges of $[0, n-1]$ defined with syntax

```
type arr[n] = {el1, el2, ...} /* remaining elements are zero */
type arr[ ] = {el1, el2, ...} /* sets length to number of elements */
```

As aforementioned, a string is an array of chars with syntax

```
char str[] = "str"; /* length = n+1 = 4 */
x = str[n]; /* equal to '/0' */
```

The enumeration constants allow for constant arrays of expressions in a single definition, using the syntax

```
enum boolean { FALSE, TRUE} \* FALSE == 0, TRUE == 1 *\
enum months { JAN = 1, FEB, MAR} \* JAN == 1, FEB == 2, ... *\
```

In arithmetic between variables of different types, the smaller-bit type is converted into the other type. Additionally, char may be used in arithmetic expressions in accordance with the ASCII table. If a larger-bit type is forced to a shorter-bit type, the remaining bits are truncated; for example:

```
float x = 2.7;
int i = x; /* i = floor(x) = 2 */
```

Qualifiers

The `short` qualifier prefixed to int is 16 bits. The `long` qualifier to int or double is 32 bits or 12/16 bytes, respectively. The `signed`, `unsigned` qualifiers give a range $[-(n-1), n]$ and $[0, 2n-1]$, respectively. By default, chars are unsigned and ints are signed. Using `<limits.h>`, `<float.h>`, the `sizeof(type)` command returns the size of type.

For integer constants in base 8, 16, long, unsigned, the prefixes and postfixes are `0-`, `0x-`, `-L`, `-u`, respectively. Additionally, float constants allow for scientific `e` notation. The default type is double. The `const` qualifier acts like `final` in Java; that is, it indicates the variable will not be changed, with syntax

```
const type var = value;
type funcz( const type[] );
```

Casting & Boolean

The cast operator casts a variable to a higher type without changing its value so as to allow precise arithmetic between low-bit types. For example,

```
int x = 9, y = 2; double i;
i = x / y; /* i = floor(9/2) */
i = x / (double)y /* i = 9/2 */
```

Bitwise operators operate on individual bits of a value, and include `&, |, ^, ~` (and, or, xor, not), not to be confused with their logical counterparts. Additionally, the bit shift operators `x<<y, x>>y` are equivalent to the decimal multiplication or division by 2^y, respectively. For example,

```
(101010 == 42) << 3 == (101010000 == 336);
```

In right shifting, the new bits are filled with 0. Signed numbers are undefined for shifting. For example,

```
/* get n bits from position p of x */
unsigned getbits(unsigned x, int p, int n) {
    return (x >> (p + 1 - n)) & ~(~0 << n);
}
getbits(42, 5, 3); /* is 5 = 101 */
```

Pointers

Introduction

A pointer stores the address of another variable, which itself stores a value. This means a pointer literally points to another variable. Its most common application is in scanning input, with syntax

```
scanf( "%f", &float_var);
```

Another example of using pointers to store variables is in using the general form `ptr = &var` as follows:

```
char c, *p; /* initializes p as a pointer */
c = getchar();
p = &c; /* pointer p points to c */
printf("%c", *p); /* prints *p = c */
```

The `&` symbol is the address operator, which gets the address of the variable it has. Pointers are declared via `type *ptr`. That is, `*ptr == var == value` iff

```
ptr = &var; *ptr = value;
```

Usage

Pointers are useful for efficiency as they only store addresses, not value. For example,

```
void swap(int *px, int *py) {
    int temp = *px;
    *px = *py;
    *py = temp;
}
void main() { swap(&a, &b); }
```

In an array, `arr == arr[0]` and `&a[i] == a + i`. The identifier of an array is equivalent to the address of its first element; that is,

```
int *p; p = arr; /* p =  &arr[0] */
x = *pa; /* same as x = arr[0] */
int y = *(p + 1) /* address of the number after p */
pa++; /* same as p = &a[1] */
```

Specifically, the array indexing syntax `a[i]` is equivalent to `*(a+i)`. For example, the following snippets use pointers as it need not store the values themselves:

```
int strlen(char *s) {
    for (int n = 0; *s != '\0'; s++) { n++; }
    return n;
}
```

```
int strlen(char *s) {
    for (char *p = s; *p != '\0'; p++) { }
    return p - s;
}
```

Address Arithmetic

Arithmetic operators for pointer addresses work as follows assuming p, q are pointers and n is an integer:

```
p +/- n; /* moves p forwards / backwards by 4*n  */
p + q; /* illegal if q > p */
q - p + n; /* illegal if q < p */
if (p == NULL) /* same as if (!p) */
```

As well, adding two pointers is illegal, as is adding a float or double to a pointer. It is also illegal to assign a pointer of one type to one of a different type without casting. With regards to strings, `char *str` is a pointer, not an array and thus may be modified. Therefore, the following functions are equivalent:

```
void strcpy(char *s, char *t) {
    for (int i = 0; (s[i] = t[i]) != 0; i++ ) {}
}
```

```
void strcpy(char *s, char *t) {
    while ((*s++ = *t++) != '\0');
}
```

Dynamic Memory

The header `<stdlib.h>` in conjunction with the syntax

```
int *arr;
arr = *malloc(int n); /* typically n = n * sizeof(int) */
free(arr);
```

allocates the needed memory dynamically. Additionally, it returns a pointer to at least n bytes available, and null if allocation was not successful. Note that the allocated memory is not initialized. The calloc function with syntax

```
*calloc(int n, int s);
*calloc(n, sizeof(int)); /* same as malloc(n*sizeof(int)) */
```

acts similarly, but differs in that it allocates an array of n elements with individual size s, and initializes the memory to 0. Consequently, the realloc and free functions with syntax

```
*realloc(void *ptr, int n);
free(void *ptr)
```

can resize a perviously allocated block of memory such that `ptr` was previously returned from another function, and releases the previously allocated memory, respectively.

Structures

Basics

A C structure defines a type, similar to classes in Java.The basic structure creation syntax is given by

```
struct name { type var; }
```

and is called by `struct name new_var`. In this case, `name` is the structure tag and `var` is a member. Members may share names across structures. The value of members are accessed via `name.var`. For nested structures, the syntax is recursively used. Combining functions and structures may be used like the following:

```
struct point makepoint(int x, int y) /* analogous to Java constructor */
{
    struct point temp; /* different point */
    temp.x = x; temp.y  = y;
    return temp; /* temp = (x, y) */
}
```

The function may then be called via

```
struct point makepoint(int, int);
struct point coord = makepoint(xcoord, ycoord);
```

Struct Funcs

A function which has a structure as an argument is known as a struct func. The structure parameters are passed be values like other types, and a copy of the structure is sent to the function. For example, a struct func for adding two points is

```
struct point addpoint(struct point p1, struct point p1) {
    p1.x += p2.x; p1.y += p2.y;
    return p1;
}
```

Struct funcs can also have pointers. If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the entire structure. For example,

```
struct point *pt; /* new pointer of type point */
struct point origin = makepoint(0,0); /* from previous snippet */
pt = &origin;
```

```
(*pt).x, (*pt).y; /* origin coordinates */
```

A shorthand notation for `(*pt).var` is `pt->var`.

Initialization & Pointers

Structures may be initialized with either of the following syntax, for arrays and multidemsnional arrays, respectively:

```
struct custom_type struct_name = {values};
struct custom_type struct_name[] = { {values1}, {vals2}, ... };
```

In the following example, the ++ operator is used for a struct ptr to increment it by the correct amount, the structure size, so as to get the next element of the array of structs, given by:

```
struct key keytab[NKEYS], *p; /* initializes keytab and *p */
for (p = keytab; p < keytab + NKEYS; p ++) {
    int count = p -> count; char[] word = p -> word;
}
```

Application of Structs

Lists

A simple linked list is created via the syntax

```
struct list {
    int data; struct list *next;
}
```

In a typical linked list, the pointer `head` points to the first element, and the last element pointer is `NULL`. A fully implemented linked list can be made via:

```
struct list {
    int data; struct list *next;
}
*head, *p, *last, int i;

head = (struct list*) malloc (sizeof(struct list));
head -> data = -1; head -> next = NULL;
last = head;
for (scanf("%d", &i); condition; scanf("%d", &i)) {
    head = (struct list*) malloc (sizeof(struct list));
    p -> data = i; p -> next = NULL;
    last -> next = p; last = p;
}
```

Using this example, an application of it to search for a number in the list is:

```
scanf("%d", &i);
for (p = head; p != NULL; p = p -> next) {
    if (p -> data == i) { } /* matches */
}
```

Typedefs

Typedefs are used for creating new data type names. For example, to create a typedef for a length, the following is used:

```
typedef type TypedefName;
TypedefName var;
```

In conjunction with structures, typedefs can be global types in C via

```
typedef struct { int x, y; }
typedef_name;
typedef_name a, b, x, ...;
```

Errors

Common errors when using pointers with arrays and in general include:

- Uninitialized pointers; for example, `int *p = val`
- Null pointer dereferencing; for example, `x = (int*)malloc(sizeof(int)); *x = val`
- Overriding existing pointers (memory leaks)

Even More Pointers

Pointers with Arrays

An array of pointers has syntax

```
type *arr[] = {val1, val2, ...};
```

In this case, the array is an array of pointers to `type`, wherein each element itself is a pointer to `type`. Consequently, pointers to arrays have syntax of

```
type arr[i] = val;
char *p1, *(p2)[i];
p1 = a; /* ptr to a[0] */
p2 = &a; /* ptr to a */
```

Multidimensional arrays must have the second of their sizes explicitly given even when initialized with values. In this case, `a[i]` is a pointer to the *i*th row. For example,

```
int *p;
int arr[][2] = { {vals1}, {valsn} }; /* sizeof(arr) = n_elements * m bits */
p = arr[1];
*p; *(p+1); /* row 2, element 1; row 2, element 3 */
```

Additionally, an example of a function of pointer arrays is

```
char *month(int i) {
    static char *name[] = { "Jan", "Feb", ...};
    return name[i]
}
```

Pointers vs Multidimensional Arrays

The main difference between the MD array `int arr[i][j]` and `int *p[i]` is that in the former, ij locations are allocated, whereas in the latter only i pointers are allocated and initialization must be done explicitly; if each element of p points to an array of j elements, the total size is ij ints + i pointers.

The pointer version is advantageous in that the rows of the array may be of different lengths, saving space. Specifically, if n is the number of chars in the longest string of the array, then for

```
char *p[] = {"str1", "str2", ...};        char arr[][n] = {"str1", "str2", ...};
```

the former dynamically allocates memory per string, whereas the latter sets each memory to n even if the length of a string is $< n$.

Command Line Args

Thus far, the main method has been defined as `main()`. However, the typical definition is

```
main(int argc, char *argv[]) { }
```

in which `argc` is the number of args, `argv` is a pointer to the array containing the args such that `argv[0]` is a pointer to a string with the program's name, and `argv[argc]` is a NULL pointer. For example, for the program

```
main(int argc, char *argv[]) {
    printf(argc); /* number of arguments */
    for (int i = 0; i < argc; i++) {printf(argv[i]);}
}
```

the output for the command line input `a.out arg1 arg2`, the output is `n args = 3; a.out, arg1, arg2`. A typical sample is the `echo` command from `echo.c`, in which the command `echo str` outputs `str` via

```
main(int argc, char *argv[]) {
    while (--argc > 0) printf(*++argv, (argc > 1) ? "␣" : "");
    return 0;
}
```

There are several command line declarations which may be (even more) complicated, including:

- `char **argv`; /* argv is ptr to ptr to char */
- `int (*p)[n]` /* p is ptr to array[n] of int */
- `int *p[n]` /* p is array[n] of ptr to int */
- `int *arr[]` /* arr is func returning ptr to int *

Files