Pointers in C

Jamal Hussein

jah1g12@ecs.soton.ac.uk

Electronics and Computer Science

December 1, 2015

Outline

1. Pointers
   - Pointers and Arrays
2. Strings
3. Structures
4. Unions

Pointers

- A pointer is a variable that contains the address of a variable

```
int x = 1;
int *p = &x;
int *q = p;
```

	Memory	address in memory (64bit)
x	1	0x7ffe05975b6c
	⋮	
p	0x7ffe05975b6c	0x7ffe05975b70
	⋮	
q	0x7ffe05975b6c	0x7ffe05975b78
	⋮	

Pointers

```
int x = 1, y = 2, z[10];
int *ip;          /* ip is a pointer to int */
ip = &x;          /*ip now points to x */
y = *ip;          /* y is now 1 */
*ip = 0;          /* x is now 0 */
*ip = *ip + 1;    /* x = x + 1 */
ip = &z[0];       /* ip now points to z[0] */
y = *ip + 1;      /* y = z[0] + 1 */
++*ip;   /* ++x */
(*ip)++;          /* x++ */
int *iq = ip;
```

Pointers to Pointers

```
int x = 1;
int *p = &x;
int *q = p;
int **w = &p;
```

	Memory	address in memory (64bi
x	1	0x7ffe05975b6c
	⋮	
p	0x7ffe05975b6c	0x7ffe05975b70
	⋮	
q	0x7ffe05975b6c	0x7ffe05975b78
	⋮	
w	0x7ffe05975b70	0x7ffe05975b80
	⋮	

Pointer arithmetic

- the variable pointer can be incremented (or decremented)
- the formula for computing the address of `pa + i` where `pa` has type `T*`:

  `addr(pa + i) = addr(pa) + [sizeof(T) * i]`
- `i` is scaled according to the size of the objects `pa` points to, which is determined by the declaration of `p`
- If an `int` is four bytes, for example, then `i` will be scaled by four

Pointer arithmetic

```
#include <stdio.h>
const int MAX = 3;
int main () {
        int var[] = {10, 100, 200};
        int i, *ptr = var;
        for ( i = 0; i < MAX; i++) {
                printf("Address of var[%d] = %x\n", i, ptr );
                printf("Value of var[%d] = %d\n", i, *ptr );
                ptr++;
        }
        return 0;
}
```

Pointer arithmetic

- Pointers may be compared by using relational operators: ==, <, >, <=, and >=

```
#include <stdio.h>
const int MAX = 3;
int main () {
        int var[] = {10, 100, 200};
        int i = 0, *ptr = var;
        while ( ptr <= &var[MAX - 1] ) {
                printf("Address of var[%d] = %p\n", i, ptr );
                printf("Value of var[%d] = %d\n", i, *ptr );
                ptr++;
                i++;
        }
        return 0;
}
```

Null Pointers

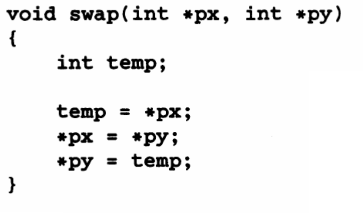
- It is always a good practice to assign a NULL value to a pointer variable

  ```
  int *ptr = NULL; /* ptr = 0 */
  ```

- we use address 0 because that memory is reserved by the operating system, so access to address 0 is not permitted
- To check for a null pointer you can use an if statement as follows:

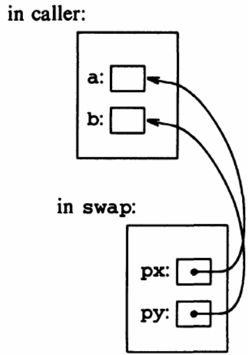
  ```
  if(ptr) /* succeeds if p is not null */
  if(!ptr) /* succeeds if p is null */
  ```

Pointers and Function Arguments



```
void swap(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```



Return pointer from functions

- C allows you to return a pointer from a function
- it is not good idea to return the address of a local variable to outside of the function
    - so you would have to define the local variable as static variable

```
int * myFunction()
{
.
.
.
}
```

Return pointer from functions

```c
/* if compilation failed, use gcc -ansi or gcc -std=c89 */
#include <stdio.h>
#include <time.h>
int * getRandom(){
        static int r[10];
        int i;
        srand((unsigned)time(NULL)); /* set the seed */
        for (i = 0; i < 10; ++i)
                r[i] = rand();
        return r;
}
int main () {
        int *p; int i;
        p = getRandom();
        for ( i = 0; i < 10; i++ )
                printf("*(p + [%d]) : %d\n", i, *(p + i) );
        return 0;
}
```

Return pointer from functions (local variables)

```
char** func1(); char** func2();
int main(){
        char **ptr1 = NULL;
        char **ptr2 = NULL;
        ptr1 = func1();
        printf("[%s] :: [%p]\n", *ptr1, ptr1);
        ptr2 = func2();
        printf("[%s] :: [%p]\n", *ptr2, ptr2);
        printf("[%s] :: [%p]\n", *ptr1, ptr1);
        return 0;
}
char** func1(){
        char *p = "Linux";
        return &p;
}
char** func2(){
        char *p = "Windows";
        return &p;
}
```

Return pointer from functions (local variables)

- The output:

```
[Linux] :: [0x7ffcb085ee80]
[Windows] :: [0x7ffcb085ee80]
[Windows] :: [0x7ffcb085ee80]
```

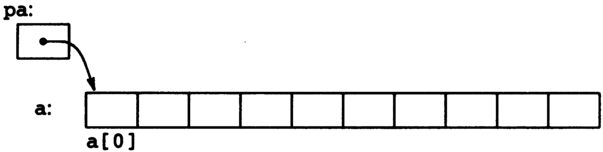
- to solve this issue, use static variables instead

```
char** func1(){
        static char *p = "Linux";
        return &p;
}
char** func2(){
        static char *p = "Windows";
        return &p;
}
```

Pointers and Arrays

- In C, there is a strong relationship between pointers and arrays

```
int a[10];
int *pa;
pa =&a[0]; /*pa = a*/
```

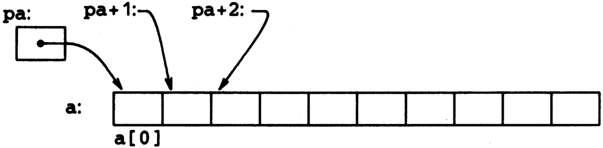


- `int x = *pa; /* same as x = a[0] */`

Pointers and Arrays

- the variable pointer can be incremented (or decremented), but the array name cannot, because it is a constant pointer
- pa+i points i elements after pa, and pa−i points i elements before

```
*(pa+1)  /*a[1]*/
*(pa+2)  /*a[2]*/
*(pa+i)  /*a[i]*/
```



Pointers and Arrays

```
        #include <stdio.h>
int main() {
        int a[10]={10,20,30,40,50,60,70,80,90,100};

        int i = 0;
        for(; i < 10; i++)
                printf("%d, ", a[i]);

        int *pa = &a[0]; /* int *p = a; */
        printf("\n");
        for(i = 0; i < 10; i++)
                printf("%d, ", *(pa+i));
        printf("\n");
        return 0;
}
```

Pointers and Arrays

- The name of an array is a synonym for the location of the initial element
- the following are equivalent

a	pa	&a[0]
*a	*pa	a[0]
a+1	pa+1	&a[1]
a+i	pa+i	&a[i]
*(a+1)	*(pa+1)	a[1]
*(a+i)	*(pa+i)	a[i]

Pointers and Arrays

```
#include <stdio.h>
int main()
{
        int a[10]={10,20,30,40,50,60,70,80,90,100};
        int *pa = a;
        printf("%d, %d, %d, %d\n", a[0], *a, *pa, pa[0]);
        printf("%d, %d, %d, %d\n",
                a[1], *(a+1), *(pa+1), pa[1]);
        pa += 2;
        printf("%d, %d, %d, %d\n",
                a[2], *(a+2), *pa, pa[0]);

        return 0;
}
```

Pointers to Functions

- functions are not variables, but
- in C, it is possible to define pointers to functions,
  ```
  int (*func)(int, int);
  ```
- like pointer to variables, function pointers can be assigned, placed in arrays, passed to functions, returned by functions, ...
  ```
  func = max;
  ```

Pointers to Functions

```c
#include <stdio.h>
int max (int, int);
int main ()
{
        int (*func)(int, int);
        func = max; /* or func = &max */
        printf("%d\n", func(3, 4));
        return 0;
}
int max(int a, int b)
{
        return a>b ? a : b;
}
```

Pointers to constants and constant pointers

- A constant pointer is a pointer that cannot change the address its holding

```
int * const ptr;
```

- A pointer to constant is a pointer through which we cannot change the value of the variable it points to

```
const int * ptr;
```

- we could have both in one definition, cconstant pointer to a constant

```
const int * const ptr;
```

Pointers to constants and constant pointers

```c
#include <stdio.h>
int main (){
        int x = 100, y = 200;
        const int * const ptr = &x;
        *ptr += 1;
        ptr = &y;
        printf("%d\n", *ptr);
        return 0;
}
```

When the code was compiled:

```
$ gcc constpointer.c -o constpointer
constpointer.c: In function 'main':
constpointer.c:5:7: error: assignment of read-only location '*ptr'
*ptr += 1;
     ^
constpointer.c:6:6: error: assignment of read-only variable 'ptr'
ptr = &y;
    ^
```

Strings

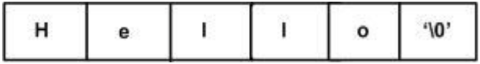
- string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'
- Thus a **null-terminated string** contains the characters that comprise the string followed by a null

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- the above definition can be written as follows:

```
char greeting[] = "Hello";
```

- The C compiler automatically places the '\0' at the end of the string when it initializes the array



H	e	l	l	o	'\0'

Strings (string.h)

Function	Purpose
`strcpy(s1, s2);`	Copies string s2 into string s1
`strcat(s1, s2);`	Concatenates string s2 onto the end of string s1
`strlen(s1);`	Returns the length of string s1
`strcmp(s1, s2);`	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2 (lexicographical)
`strchr(s1, ch);`	Returns a pointer to the first occurrence of character ch in string s1
`strstr(s1, s2);`	Returns a pointer to the first occurrence of string s2 in string s1

Strings

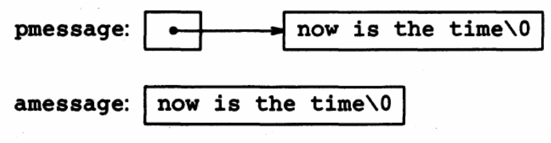
```
#include <stdio.h>
#include <string.h>
int main () {
        char str1[12] = "Hello";
        char str2[12] = "World";
        char str3[12];
        int  len;
        strcpy(str3, str1);
        printf("strcpy( str3, str1) :  %s\n", str3 );
        strcat( str1, str2);
        printf("strcat( str1, str2):   %s\n", str1 );
        len = strlen(str1);
        printf("strlen(str1) :  %d\n", len );
        return 0;
}
```

Strings

- There is an important difference between these definitions:

```
char amessage[] = "now is the time";
char *pmessage = "now is the time";
```

- Individual characters within the array may be changed but `amessage` will always refer to the same storage
- `pmessage` points to a string constant and may be modified to point elsewhere, but the result is undefined if you try to modify the string contents



Array of Strings

- You can also use an array of pointers to character to store a list of strings as follows

```
#include <stdio.h>
int main () {
        char *names[] = {
                "C", "C++", "Java", "Python",
        };
        int i = 0;
        for ( i = 0; i < 4; i++){
                printf("Value of names[%d] = %s\n",
                        i, names[i] );
        }
        return 0;
}
```

String IO

- reading string:
  - using scanf(): `scanf("%s", str);`
  - reading it character by character using `getchar()`
  - using `gets()` but it is unsafe and dangerous, never use it. It continues reading until `'\n'` or `EOF`
- printing strings:
  - using printf(): `printf("%s\n", str);`
  - printing it character by character using `putchar()`
  - using `puts(str)` to print the entire string

String IO

```
#include <stdio.h>
int main () {
        char *str;
        int i = 0;
        char c;
        while ((c = getchar()) != '\n'){
                str[i++] = c;
        }
        i = 0;
        while (str[i] != '\0'){
                putchar(str[i++]);
        }
        putchar(' ');
        puts(str);
        /* gets(str2); unsafe and dangerous */
        scanf("%s", str);
        printf("str is \"%s\"\n", str);
        return 0;
}
```

Structures

- structure is a user defined data type that allows to combine data items of different kinds
- Structures are used to represent a record
- To define a structure, you must use the `struct` statement
- The format of the struct statement:

```
struct [structure tag] {
member definition;
member definition;
...
member definition;
} [one or more structure variables];
```

- To access any member of a structure, we use the member access operator (.)

Structures

```
#include <stdio.h>
struct Books {
        char  title[50];
        char  author[50];
        int   book_id;
};
int main( ) {
        struct Books book1;
        strcpy( book1.title, "Security Coding in C and C++");
        strcpy( book1.author, "Robert C. Seacord");
        book1.book_id = 123456;
        printf( "book 1 title : %s\n", book1.title);
        printf( "book 1 author : %s\n", book1.author);
        printf( "book 1 book_id : %d\n", book1.book_id);
        return 0;
}
```

Structures as Function Arguments

- You can pass a structure as a function argument

```c
#include <stdio.h>
#include <string.h>
struct Books { ... };
void printBook(struct Books book);
int main( ) {
        struct Books book1;
        ...
        printBook(book1);
        return 0;
}
void printBook(struct Books book){
        printf( "title : %s\n", book.title);
        printf( "author : %s\n", book.author);
        printf( "book_id : %d\n", book.book_id);
}
```

Structures as Function Arguments

- You can pass a structure to a function as a pointer
- use (->) to access structure's members

```
struct Books {...};
void printBook(struct Books *book);
int main( ) {
        struct Books book1;
        ...
        printBook(&book1);
        return 0;
}
void printBook(struct Books *book){
        printf( "Book 1 title : %s\n", book->title);
        printf( "Book 1 author : %s\n", book->author);
        printf( "Book 1 book_id : %d\n",
                book->book_id); /* (*book).book_id */
}
```

Bit Fields

- Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium
- Typical examples include:
  - Packing several objects into a machine word. e.g. 1 bit flags can be compacted
  - Reading external (non-standard) file formats, e.g., 9-bit integers

```
struct packed_struct {
        unsigned int f1:1;
        unsigned int f2:1;
        unsigned int f3:1;
        unsigned int f4:1;
        unsigned int type:4;
        unsigned int my_int:9;
} pack;
```

Unions

- union is a special data type available in C that enables you to store different data types in the same memory location
- You can define a union with many members, but only one member can contain a value at any given time.

```
union Data {
        int  i;
        float  f;
        char   str[20];
} data;
```

Unions

```
#include <stdio.h>
#include <string.h>
union Data {
        int i;
        float f;
        char  str[20];
};
int main(){
        union Data data;
        data.i = 10;
        data.f = 220.5;
        strcpy( data.str, "C Programming");
        printf( "data.i : %d\n", data.i);
        printf( "data.f : %f\n", data.f);
        printf( "data.str : %s\n", data.str);
        return 0;
}
```