

BSM-308System Programming

Sakarya University-Computer Engineering

- <http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/CStuff-2/lecture.html>
- <http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Pointer-Arithmetic/index.html>
- <http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Strings-In-C/index.html>

Contents

- a) Segmentation Violations and Bus Errors
- b) Little-Endian
- c) Structures in memory
- d) Pointer arithmetic
- e) String operations

Segmentation Violations

- ❑ Memory is a giant array of bytes.
- ❑ However, certain parts of memory are inaccessible.
- ❑ For example, elements 0 - 0x1000 are generally inaccessible.
- ❑ When you try to access an inaccessible element, you create a segmentation violation.
- ❑ (In the past, we used to core dump the contents of memory We would also store it in a file called dump).
Memory is so large these days that although we can do core dumps (We do not create dump).
- ❑ Segmentation violation; When we forget to initialize a pointer and want to print its first address.

Segmentation Violations

Example:

```
pa.c
~/sist_prog/cs360-lecture-notes/CStuff-2/src

#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *s;

    s = NULL;

    printf("%d\n", s[0]);    /* Seg fault here */
    return 0;
}
```

```
File Edit View Terminal Tabs Help
unalc@unalc:~$ cd Desktop/
unalc@unalc:~/Desktop$ cd cs360/
unalc@unalc:~/Desktop/cs360$ cd CStuff-2
unalc@unalc:~/Desktop/cs360/CStuff-2$ cd bin
unalc@unalc:~/Desktop/cs360/CStuff-2/bin$ ./pa
Segmentation fault (core dumped)
unalc@unalc:~/Desktop/cs360/CStuff-2/bin$
```

Bus Errors

- ❑ Whenever you access a scalar type, its value in memory must be sequential.
- ❑ This means that if the data type is 4 bytes, it must start in a memory index whose location in memory is a multiple of 4.
- ❑ For example, if `i` is an `(int *)`, then if `i` is not a multiple of 4, this will cause errors. This error is explained by a bus error.

Little Endian

- ❑ With C, you can do "dangerous" things with memory.
- ❑ In other words, let's say you have a byte region. In this region we can hold any type you want - integers, characters, double , struct etc.
- ❑ Typically, you don't want to take advantage of this flexibility because it could get you into a lot of trouble.
- ❑ However, when writing system programs, this flexibility is often important.
- ❑ Also, it's important to understand what's going on in memory and on your machine while programs are running!

Little Endian

```
#include <stdio.h>

typedef unsigned long UL;

int main()
{
    unsigned int array[4]; /* An array of four integers. */
    unsigned int *ip;      /* An integer pointer that we're going to set to one byte beyond array */
    unsigned char *cp;     /* An unsigned char pointer for exploring the individual bytes in array */
    unsigned short *sp;    /* An unsigned short to show two-byte access. */
    int i;

    /* Set array to equal four integers, which we specify in hexadecimal. */

    array[0] = 0x12345678;
    array[1] = 0x9abcdef0;
    array[2] = 0x13579bdf;
    array[3] = 0x2468ace0;

    /* For each value of array, print it out in hexadecimal. Also print out its location in memory. */

    for (i = 0; i < 4; i++) {
        printf("Array[%d]'s location in memory is 0x%lx. Its value is 0x%x\n",
            i, (UL) (array+i), array[i]);
    }
```

```
unalc@unalc:~/Desktop/cs360/CStuff-2/bin$ ./endian
```

```
Array[0]'s location in memory is 0x7ffc1c718520. Its value is 0x12345678
Array[1]'s location in memory is 0x7ffc1c718524. Its value is 0x9abcdef0
Array[2]'s location in memory is 0x7ffc1c718528. Its value is 0x13579bdf
Array[3]'s location in memory is 0x7ffc1c71852c. Its value is 0x2468ace0
```

Little Endian

```
/* Now, print out the sixteen bytes as bytes, printing each byte's location first. */

printf("\n");
printf("Viewing the values of array as bytes:\n");
printf("\n");

cp = (unsigned char *) array;

for (i = 0; i < 16; i++) {
    printf("Byte %2d. Pointer: 0x%lx - Value: 0x%02x\n", i, (UL) (cp+i), cp[i]);
}
```

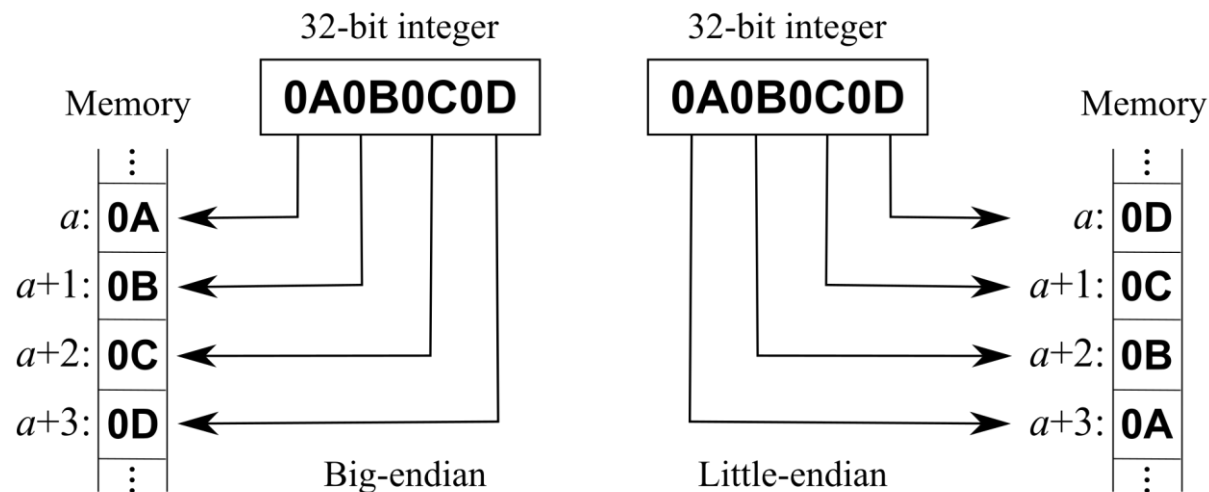
- ❑ Pay attention to the output, especially the byte values. It may seem confusing at first!
- ❑ Didn't you expect the first byte to be 0x12?

Viewing the values of array as bytes:

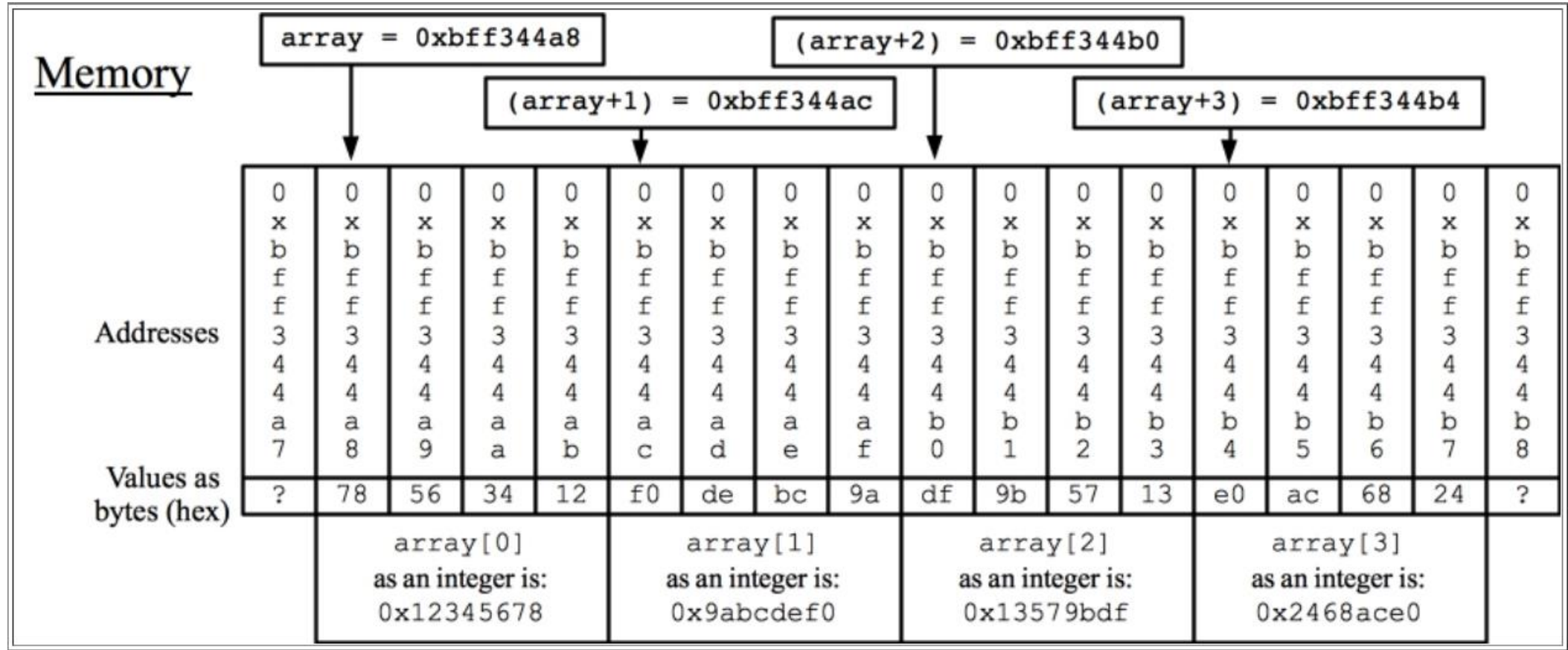
Byte	0.	Pointer:	0x7ffc1c718520	-	Value:	0x78
Byte	1.	Pointer:	0x7ffc1c718521	-	Value:	0x56
Byte	2.	Pointer:	0x7ffc1c718522	-	Value:	0x34
Byte	3.	Pointer:	0x7ffc1c718523	-	Value:	0x12
Byte	4.	Pointer:	0x7ffc1c718524	-	Value:	0xf0
Byte	5.	Pointer:	0x7ffc1c718525	-	Value:	0xde
Byte	6.	Pointer:	0x7ffc1c718526	-	Value:	0xbc
Byte	7.	Pointer:	0x7ffc1c718527	-	Value:	0x9a
Byte	8.	Pointer:	0x7ffc1c718528	-	Value:	0xdf
Byte	9.	Pointer:	0x7ffc1c718529	-	Value:	0x9b
Byte	10.	Pointer:	0x7ffc1c71852a	-	Value:	0x57
Byte	11.	Pointer:	0x7ffc1c71852b	-	Value:	0x13
Byte	12.	Pointer:	0x7ffc1c71852c	-	Value:	0xe0
Byte	13.	Pointer:	0x7ffc1c71852d	-	Value:	0xac
Byte	14.	Pointer:	0x7ffc1c71852e	-	Value:	0x68
Byte	15.	Pointer:	0x7ffc1c71852f	-	Value:	0x24

Little-Endian etc Big-Endian

- ❑ Little-Endian : Low-value bits of the data are sorted first in memory.
- ❑ Big-Endian : The higher value bits of the data are sorted first in memory.
- ❑ Memory spelling specific to processors, just like the right-to-left spelling rule in Arabic or Turkish



Little-Endian



- ❑ The address and value of each byte are listed in hexadecimal. The four pointers are then labeled array, (array+1), (array+2), and (array+3).

Little-Endian

- Now we cp ++ and print the four integers represented by ip, (ip+1), (ip+2) and (ip+3).

```
/* Finally, set the pointer ip to be one byte greater than array,  
   and then print out locations and integers. */  
  
printf("\n");  
printf("Setting the pointer ip to be one byte greater than array:\n");  
printf("\n");  
  
cp++;  
ip = (unsigned int *) cp;  
for (i = 0; i < 4; i++) {  
    printf("(ip+%d) is 0x%lx.  *(ip+%d) is 0x%x\n", i, (UL) (ip+i), i, *(ip+i));  
}
```

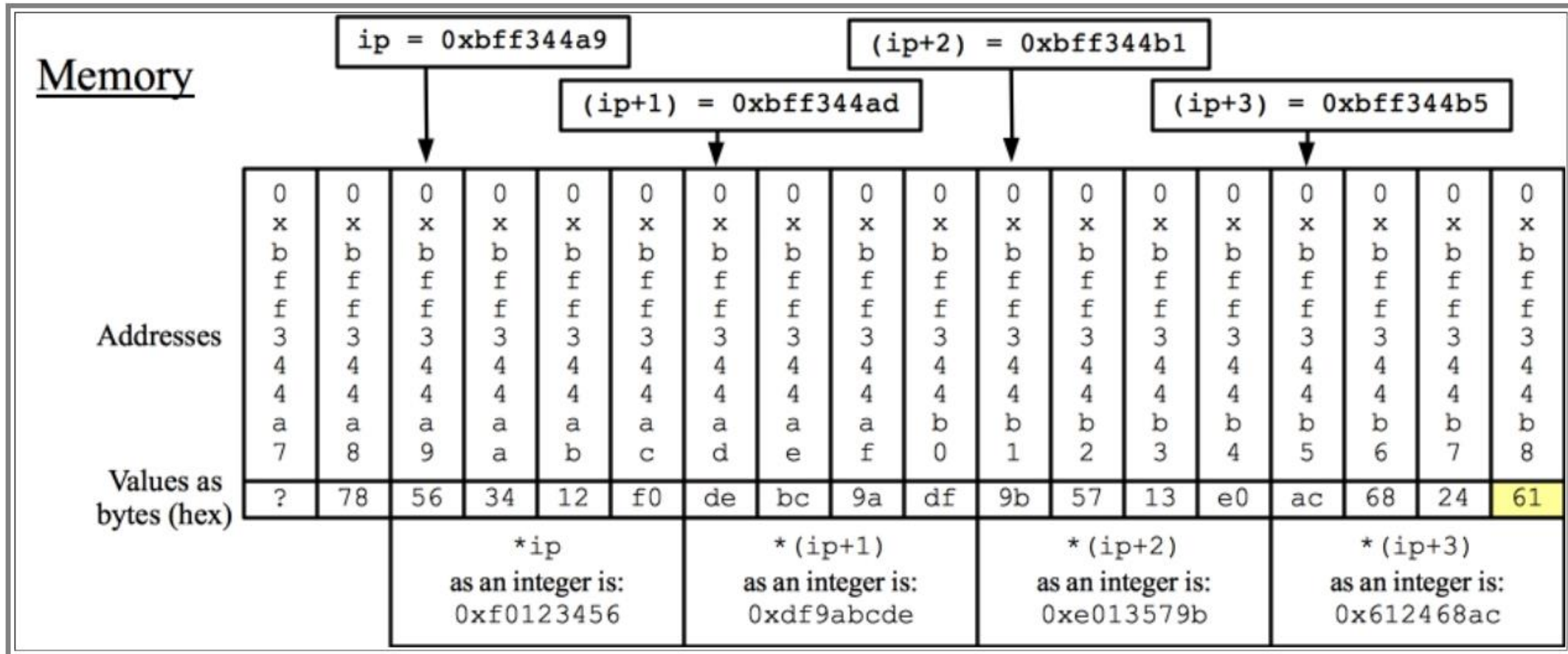
```
Setting the pointer ip to be one byte greater than array:
```

```
(ip+0) is 0x7ffda6bd4961.  *(ip+0) is 0xf0123456  
(ip+1) is 0x7ffda6bd4965.  *(ip+1) is 0xdf9abcde  
(ip+2) is 0x7ffda6bd4969.  *(ip+2) is 0xe013579b  
(ip+3) is 0x7ffda6bd496d.  *(ip+3) is 0x602468ac
```

- On some machines, this code will have a bus error because the ip is not a multiple of four. You may find the output confusing but don't worry, we'll go over it

Little-Endian

Also, the "61" in 0x612468ac is a byte not from the original string. It may have some other value - we don't really know what it should be. (In my code it turned out to be 60)



Little-Endian

- ❑ Like integers, in short it is represented by little-endian, so the smallest byte of the first short is at address 0xbff344a9 with value 0x56, and the largest byte of the first short is at address 0xbff344a8 with value 0x78:

```
/* Now, set sp to equal array. Sp is a pointer to shorts. We print out sp[0] and sp[1]. */

printf("\n");
printf("Finally printing the first four bytes of array as two shorts.\n");
printf("\n");

sp = (unsigned short *) array;
printf("Location: 0x%lx - Value as a short: 0x%04x\n", (UL) sp, sp[0]);
printf("Location: 0x%lx - Value as a short: 0x%04x\n", (UL) (sp+1), sp[1]);
printf("\n");

return 0;
}
```

Finally printing the first four bytes of array as two shorts.

Location: 0x7ffc9b3f6d60 - Value as a short: 0x5678

Location: 0x7ffc9b3f6d62 - Value as a short: 0x1234

Layout of struct

- ❑ Some machines require pointers to be aligned at certain intervals in memory.
-
- ❑ This means that integer pointers must be powers of four, double pointers must be powers of eight, and short pointers must be powers of two.
 - ❑ To meet this requirement, compilers and runtime libraries (runtime libraries) is designed with two features:
 1. Malloc () always **returns pointers that are multiples of 8** . Note that malloc () does not know the type of data that will use the memory it allocates. Therefore, it always returns multiples of 8, **just to be safe** .
 2. arranges structs so that their variables are sequential in memory , and they are aligned if the base pointer to the structure itself is a multiple of eight. This means that the compiler can put some padding into a struct and make it larger than you think it should be.

Layout of struct

```
typedef struct {  
    char b;  
    int i;  
} Char_Int;
```

```
typedef struct {  
    char b1;  
    char b2;  
    char b3;  
    char b4;  
    int i1;  
} CCCC_Int;
```

```
typedef struct {  
    char b1;  
    int i1;  
    char b2;  
    int i2;  
} C_I_C_I;
```

```
typedef struct {  
    int i;  
    char b;  
} Int_Char;
```

```
Char_Int *ci;
```

```
ci = (Char_Int *) malloc(sizeof(Char_Int)*2);  
printf("The size of a Char_Int is %ld\n", sizeof(Char_Int));  
printf("I have allocated an array, ci, of two Char_Int's at location 0x%lx\n", (UL) ci);  
printf("&(ci[0].b) = 0x%lx\n", (UL) &(ci[0].b));  
printf("&(ci[0].i) = 0x%lx\n", (UL) &(ci[0].i));  
printf("&(ci[1].b) = 0x%lx\n", (UL) &(ci[1].b));  
printf("&(ci[1].i) = 0x%lx\n", (UL) &(ci[1].i));  
printf("\n");
```

Layout of struct

- ❑ Struct in the example above uses only five bytes (one for b and four for i), the size of the structure is 8 bytes.
- ❑ When we look at the pointers, we see that where you find b is the first byte of the struct , and after b there are 4 bytes of i.
- ❑ The three bytes between b and i are not used. Why so?
- ❑ This is because the pointer i is a power of four and b's address must come before i's address (this is a compiler standard).
- ❑ The only way to do this is to omit the three bytes after b.

Layout of struct

❑ The CCCC_Int structure.

❑ As you can see, this structure makes the best use of memory -- variables take up 8 bytes and the size of the data structure is 8. The i pointer is aligned - Perfect!

```
unalc@unalc:~/Desktop/cs360/CStuff-2/bin$ ./pd
The size of a Char_Int is 8
I have allocated an array, ci, of two Char_Int's at location 0x5645d92232a0
&(ci[0].b) = 0x5645d92232a0
&(ci[0].i) = 0x5645d92232a4
&(ci[1].b) = 0x5645d92232a8
&(ci[1].i) = 0x5645d92232ac

The size of a CCCC_Int is 8
I have allocated an array, cccci, of two CCCC_Int's at location 0x5645d92236d0
&(cccci[0].b1) = 0x5645d92236d0
&(cccci[0].b2) = 0x5645d92236d1
&(cccci[0].b3) = 0x5645d92236d2
&(cccci[0].b4) = 0x5645d92236d3
&(cccci[0].i1) = 0x5645d92236d4
&(cccci[1].b1) = 0x5645d92236d8
&(cccci[1].b2) = 0x5645d92236d9
&(cccci[1].b3) = 0x5645d92236da
&(cccci[1].b4) = 0x5645d92236db
&(cccci[1].i1) = 0x5645d92236dc
```

Layout of struct

- ❑ The C_I_C_I structure is less optimally arranged; To ensure the placement of pointers i1 and i2, we must waste the three bytes after b1 and the three bytes after b2.
- ❑ If we had defined b2 right after b1, the size of the data structure would have been 12 instead of 16.

```
The size of a C_I_C_I is 16
I have allocated an array, cici, of two C_I_C_I's at location 0x5645d92236f0
&(cici[0].b1) = 0x5645d92236f0
&(cici[0].i1) = 0x5645d92236f4
&(cici[0].b2) = 0x5645d92236f8
&(cici[0].i2) = 0x5645d92236fc
&(cici[1].b1) = 0x5645d9223700
&(cici[1].i1) = 0x5645d9223704
&(cici[1].b2) = 0x5645d9223708
&(cici[1].i2) = 0x5645d922370c

The size of a Int_Char is 8
I have allocated an array, ic, of two Int_Char's at location 0x5645d9223720
&(ic[0].i) = 0x5645d9223720
&(ic[0].b) = 0x5645d9223724
&(ic[1].i) = 0x5645d9223728
&(ic[1].b) = 0x5645d922372c
```

Layout of struct

- ❑ Now look at the final output lines for `Int_Char` . It may surprise you:
- ❑ Most of us think `sizeof (Int_Char)` should be five because the `i` pointer is the start of the structure and needs to be aligned, and the `b` pointer doesn't have to worry about placement.
- ❑ This is true if we allocate only one of these structures. However, if we allocate an array of two structures, then the second one also needs to be aligned -- if the size of the structure was 5, then `ic [1].i` would not be aligned. To fix this, the size of the structure is 8 and the three bytes after `b` are wasted.

```
The size of a Int_Char is 8
I have allocated an array, ic, of two Int_Char's at location 0x5645d9223720
&(ic[0].i) = 0x5645d9223720
&(ic[0].b) = 0x5645d9223724
&(ic[1].i) = 0x5645d9223728
&(ic[1].b) = 0x5645d922372c
```

A Common Type of Error

```
/* This program shows how you lose information as you
   convert data from larger types to smaller types. */

#include <stdio.h>
#include <stdlib.h>

int main()
{
    char c;
    int i;
    int j;

    i = 10000;
    c = i;          /* We are losing information here, because 10000 cannot be stored in a byte. */
    j = c;

    printf("I: %d,   J: %d,       C: %d\n", i, j, c);
    printf("I: 0x%04x, J: 0x%04x,   C: 0x%04x\n", i, j, c);
    return 0;
}
```

A Common Type of Error

- ❑ Since `c` is a character, it cannot hold the value 10000.
- ❑ Instead it keeps the lowest byte of `i`, i.e. 16 (0x10).
- ❑ Then when you set `j` to `c`, you will see that `j` is 16.

Pointer Arithmetic

❑ We can declare an array statically by placing [size] in the variable declaration.

❑ **iarray** of ten integers :

```
int iarray[10];
```

❑ iarray elements within square brackets. In particular, the size of the iarray is not stored anywhere -- you have to keep track of it yourself.

❑ In effect iarray is a pointer to the first element of the array. In other words, there are 40 bytes allocated for the array (because integers are four bytes each) and iarray points to the first of them.

❑ If we want, we can set a second pointer to the iarray and print the iarray's elements by incrementing the pointer and dereferencing it.

Pointer Arithmetic

□ We will print 5 quantities in the for loop. Let's examine;

```
/* This program sets a pointer to an array, and then dereferences each of the
   elements of the array using the pointer and pointer arithmetic. It prints
   the pointers in hexadecimal while it does so. */

#include <stdio.h>
#include <stdlib.h>

typedef unsigned long (UL);

int main()
{
    int iarray[10];
    int *ip;
    int i;

    /* Set the 10 elements of iarray to be 100 to 109, and print the array's address. */

    for (i = 0; i < 10; i++) iarray[i] = 100+i;
    printf("iarray = 0x%lx\n", (UL) iarray);
```

Pointer Arithmetic

```
/* Set ip equal to array, and then print the 10 elements using both iarray and ip.  
   The following quantities will be printed for each element:
```

- The index i (goes from 0 to 9)
- The value of iarray[i] (goes from 100 to 109)
- The pointer ip (will start at iarray and increment by four each time).
- What *ip points to (this will be 100 to 109 again)
- Pointer arithmetic: (ip-array) -- this will be the value of i.

```
*/
```

```
ip = iarray;
```

```
for (i = 0; i < 10; i++) {  
    printf("i=%d.  ",          i          );  
    printf("iarray[i]=%d.  ",   iarray[i]   );  
    printf("ip = 0x%lx.  ",     (UL) ip     );  
    printf("*ip=%d.  ",        *ip         );  
    printf("(ip-iarray)=%ld.\n", (UL) (ip-iarray));  
    ip++;  
}
```

```
return 0;
```

```
}
```


Pointer Arithmetic

```
unalc@unalc:~/Desktop/cs360/Pointer-Arithmetic/bin$ ./iptr
```

```
iarray = 0x7ffeb9dcc340
```

i=0.	iarray[i]=100.	ip = 0x7ffeb9dcc340.	*ip=100.	(ip-iarray)=0.
i=1.	iarray[i]=101.	ip = 0x7ffeb9dcc344.	*ip=101.	(ip-iarray)=1.
i=2.	iarray[i]=102.	ip = 0x7ffeb9dcc348.	*ip=102.	(ip-iarray)=2.
i=3.	iarray[i]=103.	ip = 0x7ffeb9dcc34c.	*ip=103.	(ip-iarray)=3.
i=4.	iarray[i]=104.	ip = 0x7ffeb9dcc350.	*ip=104.	(ip-iarray)=4.
i=5.	iarray[i]=105.	ip = 0x7ffeb9dcc354.	*ip=105.	(ip-iarray)=5.
i=6.	iarray[i]=106.	ip = 0x7ffeb9dcc358.	*ip=106.	(ip-iarray)=6.
i=7.	iarray[i]=107.	ip = 0x7ffeb9dcc35c.	*ip=107.	(ip-iarray)=7.
i=8.	iarray[i]=108.	ip = 0x7ffeb9dcc360.	*ip=108.	(ip-iarray)=8.
i=9.	iarray[i]=109.	ip = 0x7ffeb9dcc364.	*ip=109.	(ip-iarray)=9.

PointerArithmetic

- ❑ Addresses expressed in hex vary from machine to machine.
-
- ❑ But the relationship between them will always be the same.
 - ❑ When this program starts running, the operating system has set it up so that the 40 bytes starting with 0x7fff5fbfdc40 are where the iarray is stored. Therefore iarray is equal to 0x7fff5fbfdc40.
 - ❑ iarray[0] is four bytes starting at 0x7fff5fbfdc40, then iarray[1] must be four bytes starting at 0x7fff5fbfdc44.
 - ❑ Therefore ip equals 0x7fff5fbfdc44 in the second iteration of the for loop.
 - ❑ Adding one to ip actually adds four to the value of the pointer. This is called "**pointer arithmetic**"
 - when you add x to a pointer, it actually adds **sx to it** , where s is the size of the data the pointer points to.

Pointer Arithmetic

- ❑ The for loop is also a bit confusing.
- ❑ `ip` is equal to `0x7fff5fbfdc44`, so you would think `(ip - iarray)` would be equal to four. It's not, because the compiler does pointer arithmetic -- from the compiler's perspective, when you say "`ip - iarray`" you're asking for the number of elements between `ip` and `iarray`.
- ❑ This will be the difference between pointers divided by the size of the element. In this case, $(0x7fff5fbfdc44 - 0x7fff5fbfdc40) / 4$ equals one.

Pointer Arithmetic

```
/* This program sets a pointer to an array, and then dereferences each of the
   elements of the array using the pointer and pointer arithmetic.  It prints
   the pointers in hexadecimal while it does so. */

#include <stdio.h>
#include <stdlib.h>

typedef unsigned long (UL);

int main()
{
    int iarray[10];
    int *ip;
    int i;

    /* Set the 10 elements of iarray to be 100 to 109, and print the array's address. */

    for (i = 0; i < 10; i++) iarray[i] = 100+i;
    printf("iarray = 0x%lx\n", (UL) iarray);
```

Pointer Arithmetic

```
/* Set ip equal to array, and then print the 10 elements using both iarray and ip.  
   The following quantities will be printed for each element:
```

- The index i (goes from 0 to 9)
- The value of iarray[i] (goes from 100 to 109)
- The pointer ip (will start at iarray and increment by four each time).
- What *ip points to (this will be 100 to 109 again)
- Pointer arithmetic: (ip-array) -- this will be the value of i.

```
*/
```

```
ip = iarray;
```

```
for (i = 0; i < 10; i++) {  
    printf("i=%d.   ",          i                );  
    printf("iarray[i]=%d.   ",  iarray[i]         );  
    printf("ip = 0x%lx.   ",    (UL) ip           );  
    printf("*ip=%d.   ",      *ip                );  
    printf("(ip-iarray)=%ld.\n", (UL) (ip-iarray));  
    ip++;  
}
```

```
return 0;
```

```
}
```

Pointer Arithmetic

```
unalc@unalc:~/Desktop/cs360/Pointer-Arithmetic/bin$ ./sptr
iarray = 0x7ffea4cc6100
i=0.  iarray[i]={100.00,200.00}.  ip = 0x7ffea4cc6100.  *ip={100.00,200.00}.  (ip-iarray)=0.
i=1.  iarray[i]={101.00,201.00}.  ip = 0x7ffea4cc6110.  *ip={101.00,201.00}.  (ip-iarray)=1.
i=2.  iarray[i]={102.00,202.00}.  ip = 0x7ffea4cc6120.  *ip={102.00,202.00}.  (ip-iarray)=2.
i=3.  iarray[i]={103.00,203.00}.  ip = 0x7ffea4cc6130.  *ip={103.00,203.00}.  (ip-iarray)=3.
i=4.  iarray[i]={104.00,204.00}.  ip = 0x7ffea4cc6140.  *ip={104.00,204.00}.  (ip-iarray)=4.
i=5.  iarray[i]={105.00,205.00}.  ip = 0x7ffea4cc6150.  *ip={105.00,205.00}.  (ip-iarray)=5.
i=6.  iarray[i]={106.00,206.00}.  ip = 0x7ffea4cc6160.  *ip={106.00,206.00}.  (ip-iarray)=6.
i=7.  iarray[i]={107.00,207.00}.  ip = 0x7ffea4cc6170.  *ip={107.00,207.00}.  (ip-iarray)=7.
i=8.  iarray[i]={108.00,208.00}.  ip = 0x7ffea4cc6180.  *ip={108.00,208.00}.  (ip-iarray)=8.
i=9.  iarray[i]={109.00,209.00}.  ip = 0x7ffea4cc6190.  *ip={109.00,209.00}.  (ip-iarray)=9.
```

- As you can see, pointers differ from how iptr works. However, the relationship between them is the same and every time you increase ip , its value increases by 16 because the size of the struct is 16 bytes.

String- strcpy ()

```
char * strcpy (char *s1, const char *s2);
```

- ❑ Strcpy () assumes s2 is a null/ null- terminated string and s1 is a (char *) with enough characters to hold s2, including the trailing null character .
- ❑ Strcpy () then copies s2 to s1.
- ❑ It also returns s1.

String Example

```
/* Initialize three strings using strcpy() and print them. */

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

int main()
{
    char give[5];
    char him[5];
    char six[5];

    strcpy(give, "Give");
    strcpy(him, "Him");
    strcpy(six, "Six!");

    printf("%s %s %s\n", give, him, six);
    return 0;
}
```

```
/* What happens when you call strcpy and didn't allocate enough memory? */

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

typedef unsigned long UL;

int main()
{
    char give[5];
    char him[5];
    char six[5];

    /* Print the addresses of the three arrays. */

    printf("give: 0x%lx  him: 0x%lx  six: 0x%lx\n", (UL) give, (UL) him, (UL) six);

    /* This is the same as before -- nice strcpy() statements, and then print. */

    strcpy(give, "Give");
    strcpy(him, "Him");
    strcpy(six, "Six!");
    printf("%s %s %s\n", give, him, six);

    /* Now, this strcpy() is copying a string that is too big. */

    strcpy(him, "T.J. Houshmandzadeh");
    printf("%s %s %s\n", give, him, six);

    return 0;
}
```


String Example

- ❑ There's clearly a problem with the example above -- the string "TJ Houshmandzadeh " is much larger than five characters.
- ❑ Some compilers will compile this, but some others may have trouble with it.

```
UNIX> gcc -o bin/strcpy2 src /strcpy2.c
src /strcpy2.c: In function 'main':
src /strcpy2.c:21: warning: call to __builtin___strcpy_chk will always overflow destination buffer
UNIX>
```

- ❑ This is a smart compiler. However, compilers are not all-seeing and all-knowing. We can trick it by writing our own expressions around strcpy () -- now it won't fix the problem.

	----4 bytes----				
	0	1	2	3	(I'm drawing this in big endian)
six----->	'S'	'i'	'x'	'l'	0xbffffe040
	0				0xbffffe044
					0xbffffe048
					0xbffffe04c
him----->	'H'	'i'	'm'	0	0xbffffe050
					0xbffffe054
					0xbffffe058
					0xbffffe05c
give----->	'G'	'i'	'v'	'e'	0xbffffe060
	0				0xbffffe064
					0xbffffe068
					0xbffffe06c

Now, we make the call **strcpy(him, "T.J. Houshmandzadeh")**. What happens is that the entire string is copied to **him**,

	----4 bytes----				
	0	1	2	3	(I'm drawing this in big endian)
six----->	'S'	'i'	'x'	'l'	0xbffffe040
	0				0xbffffe044
					0xbffffe048
					0xbffffe04c
him----->	'T'	'.'	'J'	'.'	0xbffffe050
	' '	'H'	'o'	'u'	0xbffffe054
	's'	'h'	'm'	'a'	0xbffffe058
	'n'	'd'	'z'	'a'	0xbffffe05c
give----->	'd'	'e'	'h'	0	0xbffffe060
	0				0xbffffe064
					0xbffffe068
					0xbffffe06c

String

```
unalc@unalc:~/Desktop/cs360/Strings-In-C/bin$ ./strcpy2
give: 0x7fff910e8319  him: 0x7fff910e831e  six: 0x7fff910e8323
Give Him Six!
Give T.J. Houshmandzadeh Houshmandzadeh
*** stack smashing detected ***: terminated
Aborted (core dumped)
unalc@unalc:~/Desktop/cs360/Strings-In-C/bin$ ./strcpy3
give: 0x7ffdd5d266d9  him: 0x7ffdd5d266de  six: 0x7ffdd5d266e3
Give Him Six!
Give T.J. Houshmandzadeh Houshmandzadeh
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

```
/* This is the same as strcpy2.c, but I write a procedure to call strcpy(), so that
   even a smart compiler won't figure out that I have a problem. */

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

typedef unsigned long UL;

void my_strcpy(char *s1, char *s2)
{
    strcpy(s1, s2);
}

int main()
{
    char give[5];
    char him[5];
    char six[5];

    printf("give: 0x%lx  him: 0x%lx  six: 0x%lx\n", (UL) give, (UL) him, (UL) six);

    strcpy(give, "Give");
    strcpy(him, "Him");
    strcpy(six, "Six!");

    printf("%s %s %s\n", give, him, six);

    my_strcpy(him, "T.J. Houshmandzadeh");

    printf("%s %s %s\n", give, him, six);
    return 0;
}
```

String- strcat ()

```
char * strcat (char *s1, const char *s2);
```

- ❑ Strcat () assumes that s1 and s2 are both null/ null terminated strings.
- ❑ Strcat () then concatenates s2 to the end of s1.
- ❑ Strcat () assumes there is enough space in s1 to hold these extra characters. Otherwise, you'll start crushing the memory you didn't allocate.

```
UNIX> bin/ strcat
Give
Give Him
Give Him Six!
UNIX>
```

```
/* Using strcpy() and strcat() to create the string "Give Him Six!" incrementally. */
#include <stdio.h>
#include <string.h>

int main()
{
    char givehimsix[15];

    strcpy(givehimsix, "Give");
    printf("%s\n", givehimsix);
    strcat(givehimsix, " Him");
    printf("%s\n", givehimsix);
    strcat(givehimsix, " Six!");
    printf("%s\n", givehimsix);
    return 0;
}
```

String- strcat ()

- ❑ Strcat () assumes that s1 and s2 are both null/ null terminated strings.
- ❑ Strcat () then concatenates s2 to the end of s1.
- ❑ Strcat () assumes there is enough space in s1 to hold these extra characters. Otherwise, you'll start crushing the memory you didn't allocate.

```
char * strcat (char *s1, const char *s2);
```

```
/* Using strcpy() and strcat() to create the string "Give Him Six!" incrementally. */  
  
#include <stdio.h>  
#include <string.h>  
  
int main()  
{  
    char givehimsix[15];  
  
    strcpy(givehimsix, "Give");  
    printf("%s\n", givehimsix);  
    strcat(givehimsix, " Him");  
    printf("%s\n", givehimsix);  
    strcat(givehimsix, " Six!");  
    printf("%s\n", givehimsix);  
    return 0;  
}
```

```
unalc@unalc:~/Desktop/cs360/Strings-In-C/bin$ ./strcat  
Give  
Give Him  
Give Him Six!
```

String- strcat ()

❑ src /strcat2.c. Can you explain why the output is like this?

```
unalc@unalc:~/Desktop/cs360/Strings-In-C/bin$ ./strcat2
give: 0x7ffe7ccf5800  him: 0x7ffe7ccf57f6  six: 0x7ffe7ccf57fb
Give Him Six!
mandzadeh T.J. Houshmandzadeh Houshmandzadeh
mandzadeh Help! T.J. Houshmandzadeh Help! Houshmandzadeh Help!
```

❑ *address locations have changed

String-strcat ()

[illegible]

- ❑ C-style string operations are slightly more difficult than C++-style string operations.
- ❑ For example, let's say you want to create an array containing a certain number of **js**.
- ❑ `malloc ()` first to allocate the array.

```

/* Trying to use strcat() like C++ string concatenation. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    char *s;
    int i;
    int n;

    if (argc != 2) { fprintf(stderr, "usage: strcat3 number\n"); exit(1); }

    n = atoi(argv[1]);
    s = (char *) malloc(sizeof(char)*(n+1));
    strcpy(s, "");

    for (i = 0; i < n; i++) strcat(s, "j"); /* Here's the strcat() call, wh

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

```

```

/* Create a string with a given number of j's by using string concatenation. */

#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv)
{
    int i, n;
    string s;

    if (argc != 2) { fprintf(stderr, "usage: makej number\n"); exit(1); }
    n = atoi(argv[1]);

    for (i = 0; i < n; i++) s += "j";    // Here is the string concatenation.
    cout << s << endl;
    return 0;
}

```

String-strcat ()

- ❑ Let's try it on a really big number.
- ❑ Here, let's redirect standard output to `/dev/null`,

[illegible]

However, try them on a really big number. Here, I'm going to redirect standard output to **/dev/null**, which throw

```

UNIX> time sh -c "bin/makej 1000 > /dev/null"
0.002u 0.004s 0:00.01 0.0%      0+0k 0+0io 0pf+0w      # Blink of an eye.
UNIX> time sh -c "bin/makej 10000 > /dev/null"
0.002u 0.004s 0:00.00 0.0%      0+0k 0+0io 0pf+0w      # Blink of an eye.
UNIX> time sh -c "bin/makej 100000 > /dev/null"
0.004u 0.004s 0:00.01 0.0%      0+0k 0+0io 0pf+0w      # Blink of an eye.
UNIX> time sh -c "bin/strcat3 1000 > /dev/null"
0.002u 0.004s 0:00.00 0.0%      0+0k 0+0io 0pf+0w      # Blink of an eye.
UNIX> time sh -c "bin/strcat3 10000 > /dev/null"
0.039u 0.004s 0:00.04 75.0%      0+0k 0+0io 0pf+0w      # A little slower
UNIX> time sh -c "bin/strcat3 100000 > /dev/null"
3.468u 0.005s 0:03.47 99.7%      0+0k 0+0io 0pf+0w      # Nearly 100 times slower!

```

- ❑ ➤ Redirections will be discussed in detail later.
- ❑ **/dev/null**
- ❑ It is a special file found on every Linux system. However, unlike many other virtual files, it is used for writing rather than reading. Anything you write to /dev/ null is thrown away (like a trash can). In a UNIX system it is known as a null device.

String-strcat ()

- ❑ Do you see the problem? C++ string preserves the length of the string, so concatenation is fast.
 - ❑ In contrast, `strcat ()` has to find the end of the string on each call, making the program $O(n^2)$.
-
- ❑ Since we know where the end of the string is, we can fix it.

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

int main(int argc, char **argv)
{
    char *s;
    int i;
    int n;

    if (argc != 2) { fprintf(stderr, "usage: strcat4 number\n"); exit(1); }

    n = atoi(argv[1]);
    s = (char *) malloc(sizeof(char)*(n+1));
    strcpy(s, "");

    for (i = 0; i < n; i++) strcat(s+i, "j"); /* The only changed line */

    printf("%s\n", s);
    return 0;
}
```

[illegible]

- **command > file** : Sends standard output to <file>
- **command < input** : Feeds a command input from <input>

```
unalc@unalc:~/Desktop/cs360/Strings-In-C/bin$ ./strlen
Give Him Six!
4 3 4
```

String-strlen()

- ❑ Strlen () assumes s is a null-terminated string. Returns the number of characters before the null character. I mean its length!

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

int main()
{
    char give[5];
    char him[5];
    char six[5];

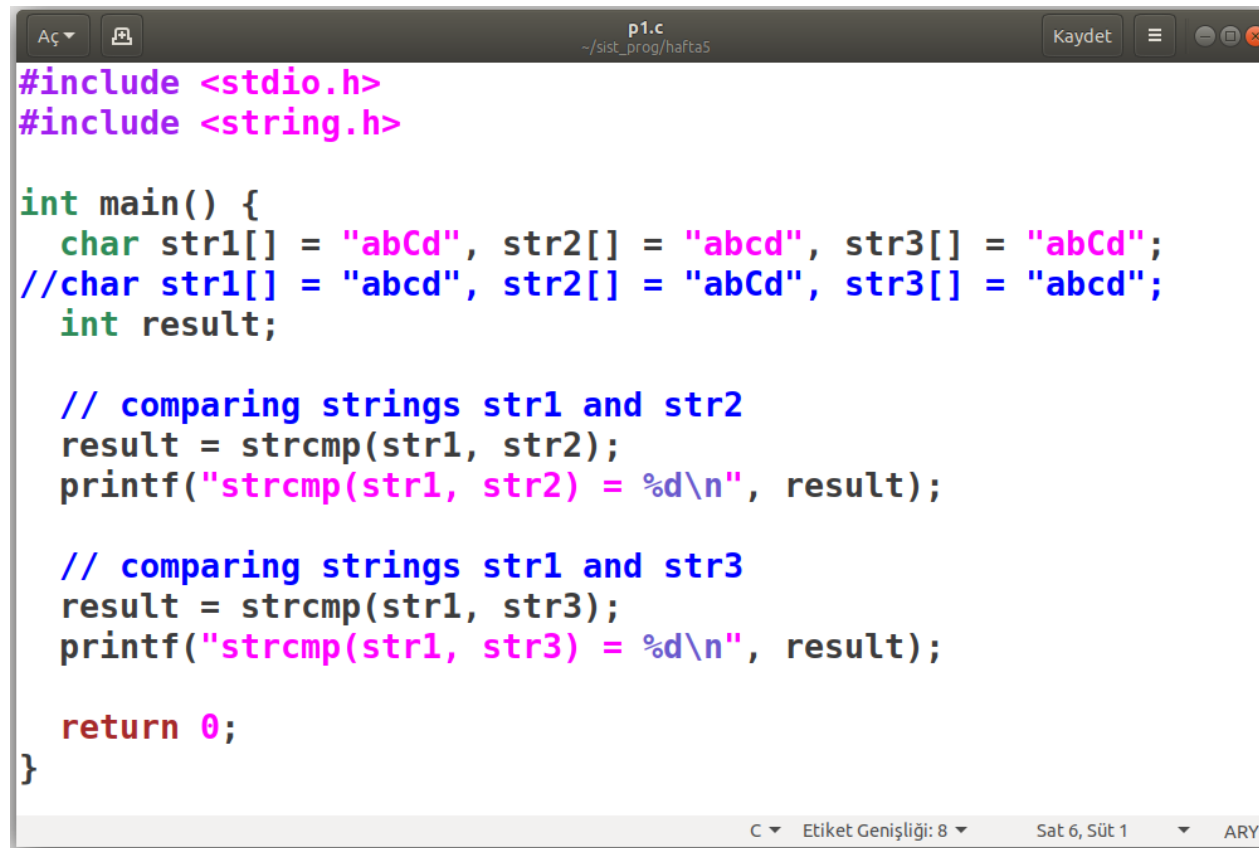
    strcpy(give, "Give");
    strcpy(him, "Him");
    strcpy(six, "Six!");

    printf("%s %s %s\n", give, him, six);
    printf("%ld %ld %ld\n", strlen(give), strlen(him), strlen(six));
    return 0;
}
```

String-strcmp () and strncmp ()

- ❑ **Strcmp ()** performs a lexical comparison of two string expressions.
 - ❑ *0 if equal,*
 - ❑ *negative if s1 is less than s2,*
 - ❑ *otherwise it returns a positive number.*
- ❑ strcmp () quite a bit because it's the easiest way to compare two strings .
- ❑ **Strncmp ()** stops the comparison after n characters if the null character has not been reached yet

String-strcmp () and strncmp ()



```
p1.c
~/sist_prog/hafta5
Kaydet

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

int main() {
    char str1[] = "abCd", str2[] = "abcd", str3[] = "abCd";
    //char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
    int result;

    // comparing strings str1 and str2
    result = strcmp(str1, str2);
    printf("strcmp(str1, str2) = %d\n", result);

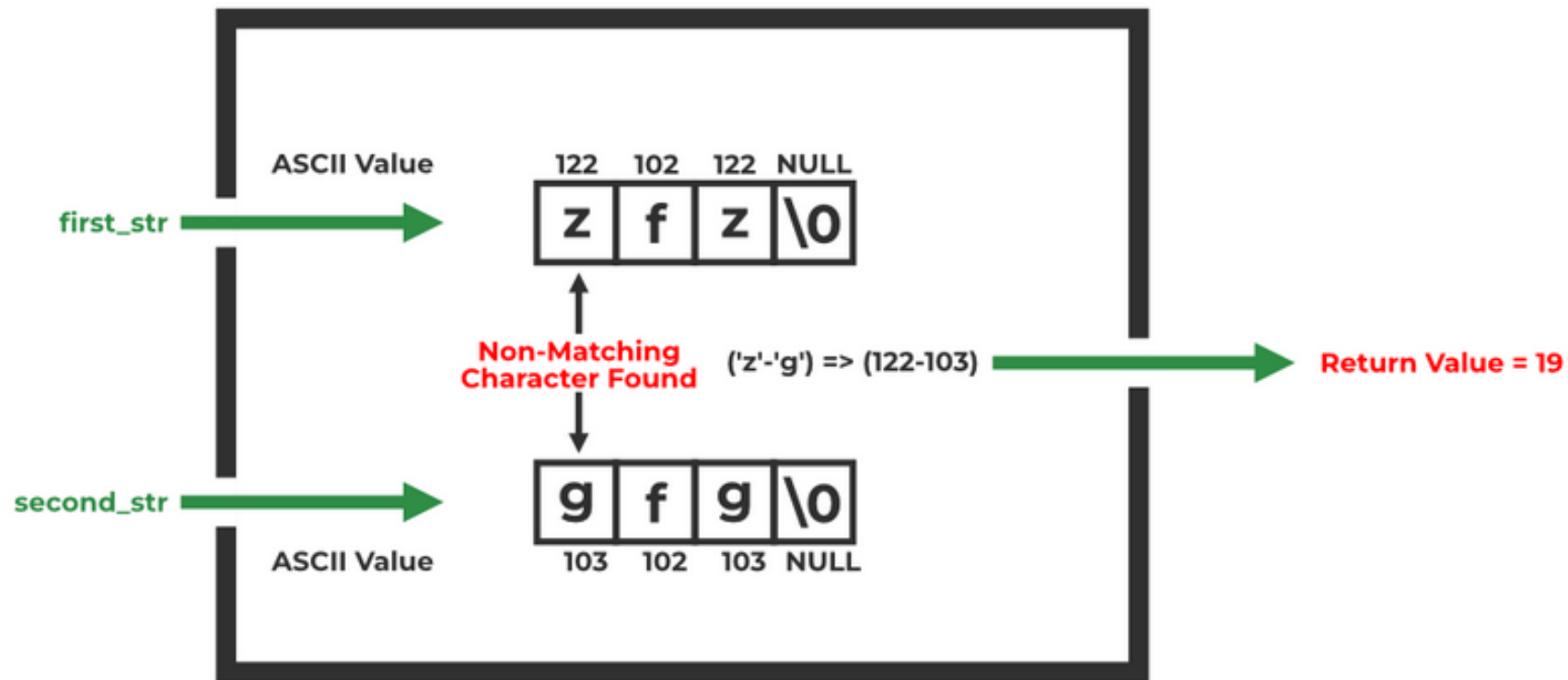
    // comparing strings str1 and str3
    result = strcmp(str1, str3);
    printf("strcmp(str1, str3) = %d\n", result);

    return 0;
}
```

C Etiket Geniřlięi: 8 Sat 6, Süt 1 ARY

String-strcmp () and strncmp ()

strcmp("zfg", "gfg");



<https://www.geeksforgeeks.org/strcmp-in-c/>

String-strncmp () and strncmp ()

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

int main () {
    char str1[20];
    char str2[20];
    int result;

    strcpy(str1, "hello");
    strcpy(str2, "heLLo WORLD");

    //This will compare the first 4 characters
    result = strncmp(str1, str2, 4);

    if(result > 0) {
        printf("ASCII value of first unmatched character of str1 is greater than str2");
    } else if(result < 0) {
        printf("ASCII value of first unmatched character of str1 is less than str2");
    } else {
        printf("Both the strings str1 and str2 are equal");
    }

    return 0;
}
```

String-strchr ()

`char * strchr (const char *s, int c);`

- ❑ **Strchr ()** is a "find" operation for single characters in C strings.
- ❑ `c` is an integer, but is treated as a character. `Strchr ()` returns a pointer to the first occurrence of the character `s` equal to `c`. If `s` does not contain `c`, it returns `NULL`.

```
/* Use strchr() to determine if each line of standard input has a space. */  
  
#include <stdio.h>  
#include <string.h>  
  
int main()  
{  
    char line[100];  
    char *ptr;  
  
    while (fgets(line, 100, stdin) != NULL) {  
        ptr = strchr(line, ' ');  
        if (ptr == NULL) {  
            printf("No spaces\n");  
        } else {  
            printf("Space at character %ld\n", ptr-line);  
        }  
    }  
    return 0;  
}
```

String-strchr ()

□ With a little modification we can show the locations of all the characters;

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

int main()
{
    char line[100];
    char *ptr;

    while (fgets(line, 100, stdin) != NULL) {
        ptr = strchr(line, ' ');
        if (ptr == NULL) {
            printf("No spaces\n");
        } else {
            while (ptr != NULL) {
                printf("Space at character %ld\n", ptr-line);
                ptr = strchr(ptr+1, ' ');
            }
        }
    }
    return 0;
}
```

UNIX> **bin/strchr2**

Jim

No spaces

Jim Plank

Space at character 3

Jim Plank

Space at character 3

Space at character 4

Give Him Six!!!

Space at character 0

Space at character 1|

Space at character 6

Space at character 7

Space at character 8

Space at character 12

Space at character 13

Space at character 14

<CNTL-D>

UNIX>


```
unalc@unalc:~/Desktop/cs360/Strings-In-C/bin$ ./strchr
system programming lecture
Space at character 6
software eng.
Space at character 8
a b c d e
Space at character 1
^C
```

```
unalc@unalc:~/Desktop/cs360/Strings-In-C/bin$ ./strchr2
system programming lecture
Space at character 6
Space at character 18
software eng. department
Space at character 7
Space at character 12
a b c d e f g
Space at character 1
Space at character 3
Space at character 5
Space at character 7
Space at character 9
Space at character 11
```

String-scanf ()

- ❑ `scanf ()` is like `printf ()`. However, instead of writing the parameters to the terminal, it reads them from the terminal (or whatever the standard input is).
- ❑ `scanf ()` confuses people is that there are no reference variables in C, so you have to use pointers.
- ❑ If you write `"%d"` in the format string, `scanf ()` will read an integer. The parameter you need to pass is a pointer to the integer you want to read.
- ❑ Storage for the integer must be available. `scanf ()` will read the integer from standard input and fill in four bytes of the integer.

String-scanf ()

□ scanf (), I tell it to read an integer from standard input and fill those four bytes with that integer. Scanf () returns the number of successful reads it has made.

```
/* Read a single integer from standard input using scanf. */

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i;

    if (scanf("%d", &i) == 1) {
        printf("Just read i: %d (0x%x)\n", i, i);
    } else {
        printf("Scanf() failed for some reason.\n");
    }
    exit(0);
}
```

```
UNIX> bin/scanf1
10
Just read i: 10 (0xa)
UNIX> bin/scanf1
Fred
Scanf() failed for some reason.
UNIX> bin/scanf1
15.999999999999
Just read i: 15 (0xf)
UNIX> bin/scanf1
-15.999999999999
Just read i: -15 (0xffffffff1)
UNIX> bin/scanf1
<CNTL-D>
Scanf() failed for some reason.
UNIX> echo "" | bin/scanf1
Scanf() failed for some reason.
UNIX> echo 15fred | bin/scanf1
Just read i: 15 (0xf)
UNIX>
```

String-scanf ()

- ❑ It compiles (although some compilers can tell it's wrong and warn you).
- ❑ Whether the error occurs or not is a matter of luck.

```
int main()
{
    int *i;

    printf("i = 0x%lx\n", (unsigned long) i);
    if (scanf("%d", i) == 1) {
        printf("Just read i: %d (0x%x)\n", *i, *i);
    } else {
        printf("Scanf() failed for some reason.\n");
    }
    exit(0);
}
```

```
UNIX > echo 10 | bin/scanf2
i = 0x7fff5fc01052
Bus error
UNIX>
```

String-scanf ()

- ❑ The following program (src /scanf3.c) uses scanf () to read a string from standard input and then print individual characters :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char s[10];
    int i;

    if (scanf("%s", s) != 1) exit(0);

    for (i = 0; s[i] != '\0'; i++) {
        printf("Character: %d: %3d %c\n", i, s[i], s[i]);
    }
    exit(0);
}
```

```
UNIX> echo "Jim-Plank" | bin/scanf3
Character: 0:  74 J
Character: 1: 105 i
Character: 2: 109 m
Character: 3:  45 -
Character: 4:  80 P
Character: 5: 108 l
Character: 6:  97 a
Character: 7: 110 n
Character: 8: 107 k
UNIX>
```

String-Sscanf ()

- ❑ Sscanf () is just like scanf (), except that it takes an additional string as its first parameter and "reads" from that string instead of standard input .
- ❑ Returns the number of correct matches it made. Therefore, it is quite suitable for converting strings to integers and double variables.

```
#include <stdio.h>

int main()
{
    char buf[1000];
    int i, h;
    double d;

    while (fgets(buf, 1000, stdin) != NULL) {
        if (sscanf(buf, "%d", &i) == 1) {
            printf("When treated as an integer, the value is %d\n", i);
        }
        if (sscanf(buf, "%x", &h) == 1) {
            printf("When treated as hex, the value is 0x%x (%d)\n", h, h);
        }
        if (sscanf(buf, "%lf", &d) == 1) {
            printf("When treated as a double, the value is %lf\n", d);
        }
        if (sscanf(buf, "0x%x", &h) == 1) {
            printf("When treated as a hex with 0x%x formatting, the value is 0x%x (%d)\n", h, h);
        }
        printf("\n");
    }
}
```

UNIX> bin/sscanf1

10

When treated as an integer, the value is 10

When treated as hex, the value is 0x10 (16)

When treated as a double, the value is 10.000000

55.9

When treated as an integer, the value is 55

When treated as hex, the value is 0x55 (85)

When treated as a double, the value is 55.900000

.5679

When treated as a double, the value is 0.567900

a

When treated as hex, the value is 0xa (10)

0x10

When treated as an integer, the value is 0

When treated as hex, the value is 0x10 (16)

When treated as a double, the value is 16.000000

When treated as a hex with 0x%x formatting, the value is 0x10 (16)

UNIX>

String - Strdup ()

- ❑ Creates a copy of the string by allocating memory.
- ❑ malloc (), if you are done with the copy, you should call free () afterwards to avoid memory leaks.

```
char *strdup(const char *s);
```

```
char *strdup(const char s)
{
    return strcpy(malloc(strlen(s)+1), s);
}
```

Others

- **strrchr ()** finds the last occurrence of a character
- **strstr ()** finds a substring (Substring).
- **strcasestr ()** finds a substring (Substring) but ignores case.
- **strsep ()** helps you separate strings with delimiters.
- **strncpy ()** does a restricted strcpy .
- **memcpy ()** copies one region of memory to another.
- **memcmp ()** performs a byte-by- byte comparison of two regions of memory .
- **bzero ()** sets a region of memory such that each byte is zero.

strsep ()

```
p3.c ~/sist_prog/hafta5 Kayde
Aç
#include <stdio.h>
#include <string.h>

int main()
{
    char *string,*found;

    string = strdup("Hello there, People!");
    printf("Original string: '%s'\n",string);

    while( (found = strsep(&string," ")) != NULL )
        printf("%s\n",found);

    return(0);
}
```

strstr ()

```
p4.c
~/sist_prog/hafta5
Kaydet

int main()
{
    // Take any two strings
    char s1[] = "Home Sweet Home";
    char s2[] = "Sweet";
    char* p;

    // Find first occurrence of s2 in s1
    p = strstr(s1, s2);

    // Prints the result
    if (p) {
        printf("String found\n");
        printf("First occurrence of string '%s' in\n", s2, s1, p);
    } else
        printf("String not found\n");

    return 0;
}
```

Assignments

- 1- Write a program in C to replace the spaces in a string with a specific character.**
- 2- Write a C program to count each character in a given string.**
- 3-Write a C program to find the repeated character in a string.**

C-String operation more example with solution

<https://www.w3resource.com/c-programming-exercises/string/index.php>