Pointers - Chapter 5

CS 241
Data Organization using C

Instructor: **Joel Castellanos**

e-mail: joel@unm.edu

Web: http://cs.unm.edu/~joel/

Office: Farris Engineering Center
Room 2110

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

xkcd.com

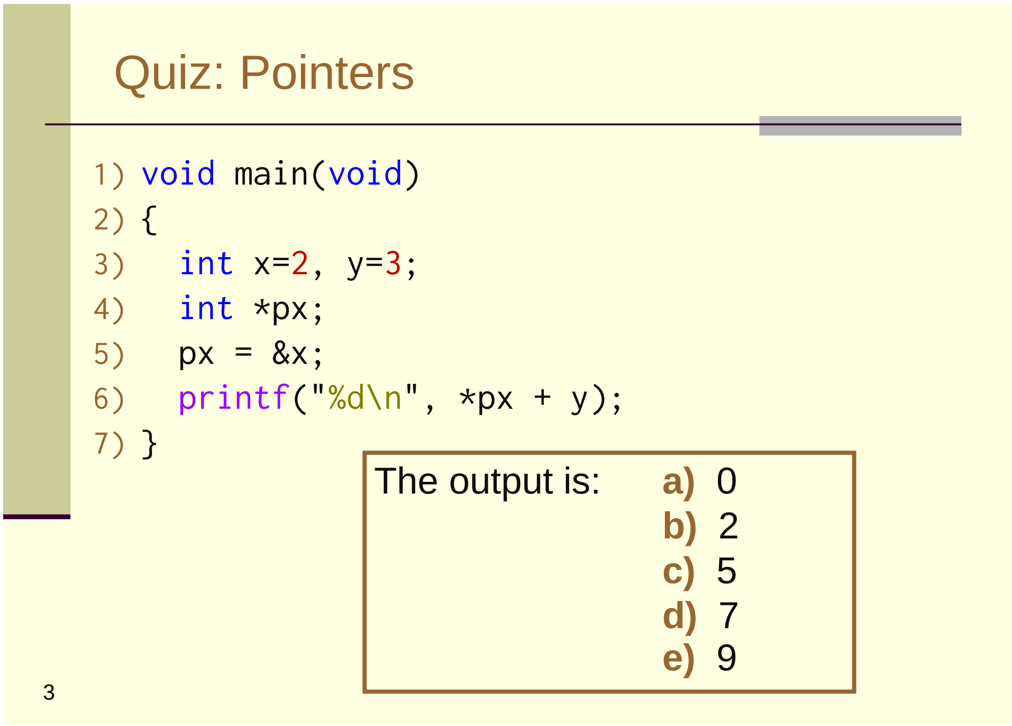
9/25/2019

Quiz: Bitwise AND Operator

```
#include <stdio.h>

void main(void)
{
  printf("%d\n", 31 & 37);
}
```

The output is:
a) 3
b) 5
c) 27
d) 31
e) 68

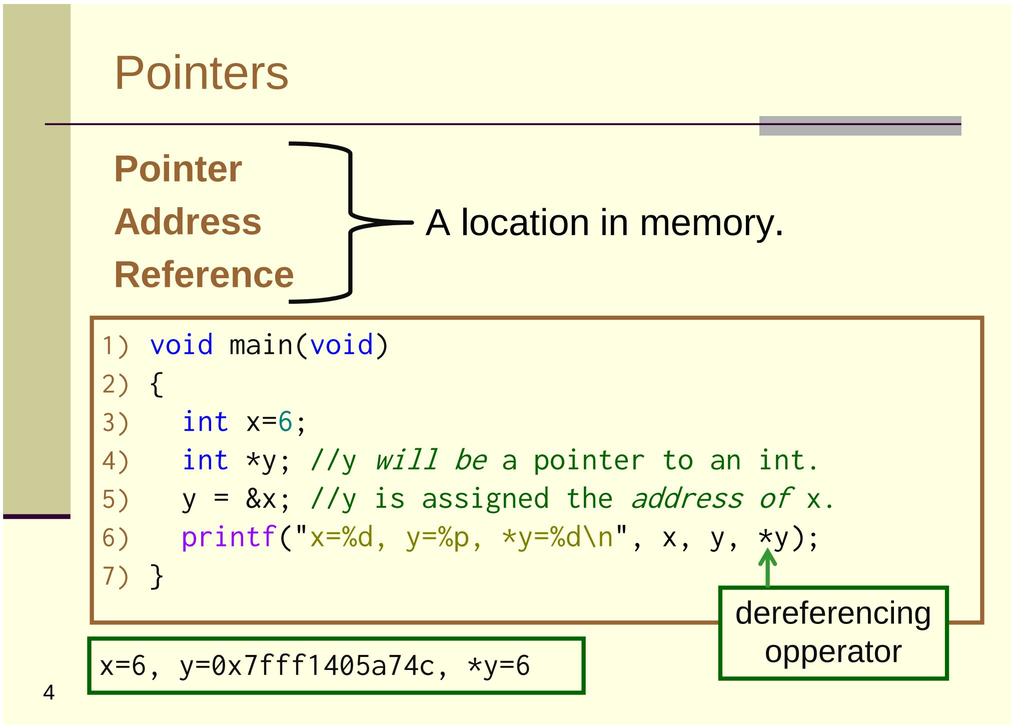
	128	64	32	16	8	4	2	1
	0	0	0	1	1	1	1	1
&	0	0	1	0	0	1	0	1
	0	0	0	0	0	1	0	1



Quiz: Pointers

```
void main(void)
{
  int x=2, y=3;
  int *px;
  px = &x;
  printf("%d\n", *px + y);
}
```

The output is:
a) 0
b) 2
c) 5
d) 7
e) 9



Pointers

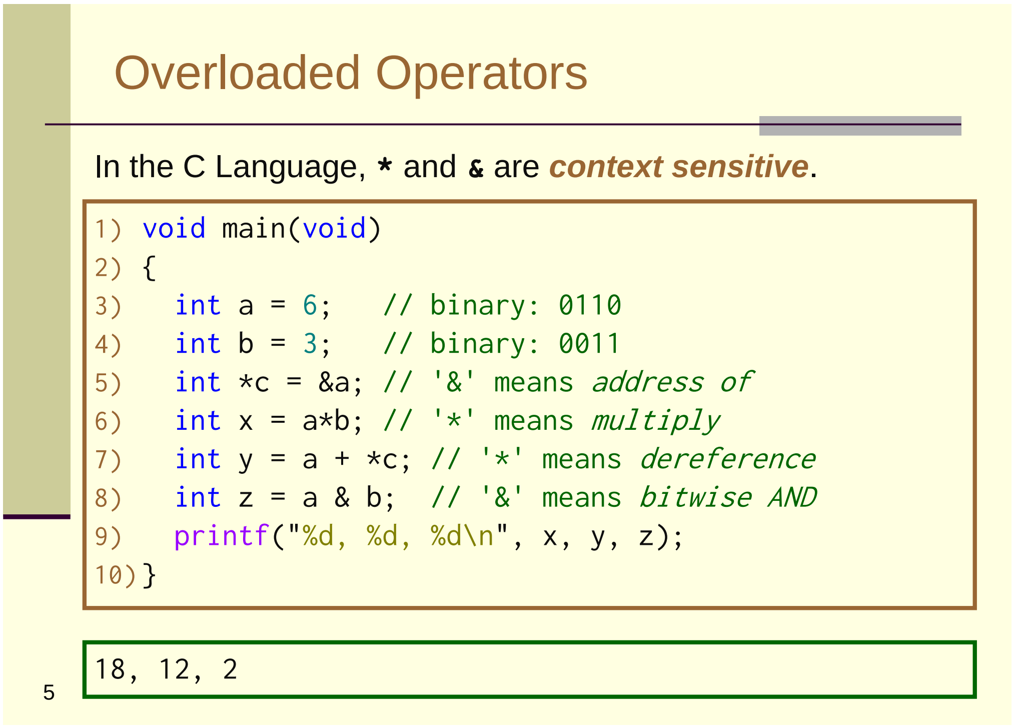
Pointer
Address
Reference

A location in memory.

```
void main(void)
{
  int x=6;
  int *y; //y will be a pointer to an int.
  y = &x; //y is assigned the address of x.
  printf("x=%d, y=%p, *y=%d\n", x, y, *y);
}
```

dereferencing opperator

```
x=6, y=0x7fff1405a74c, *y=6
```

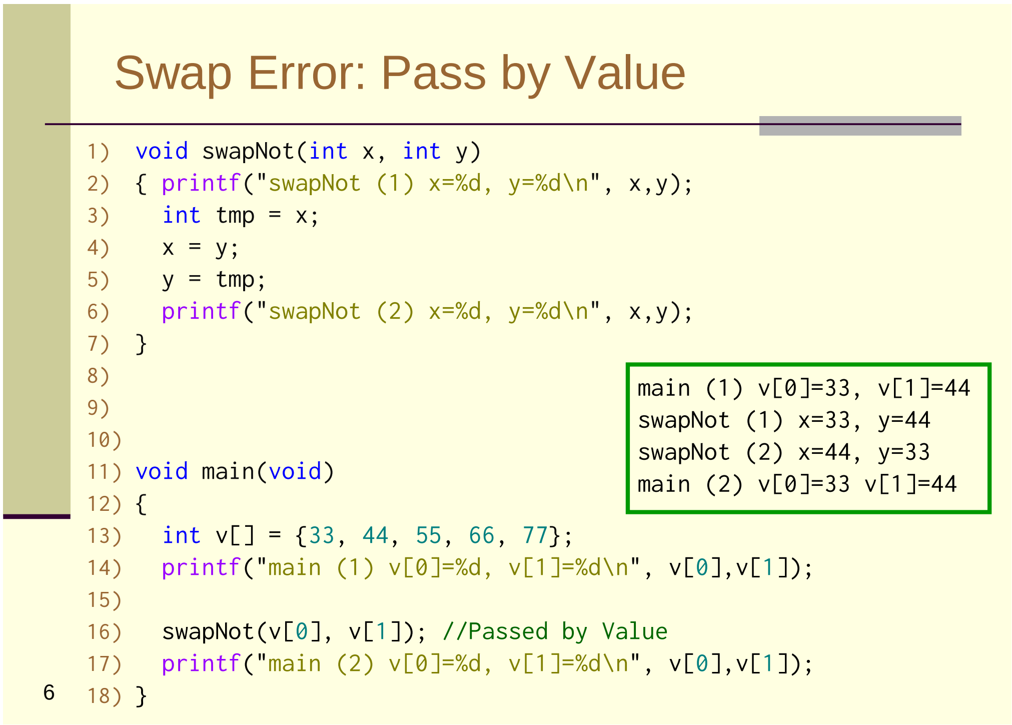


Overloaded Operators

In the C Language, * and & are ***context sensitive***.

```
void main(void)
{
  int a = 6;   // binary: 0110
  int b = 3;   // binary: 0011
  int *c = &a; // '&' means address of
  int x = a*b; // '*' means multiply
  int y = a + *c; // '*' means dereference
  int z = a & b;  // '&' means bitwise AND
  printf("%d, %d, %d\n", x, y, z);
}
```

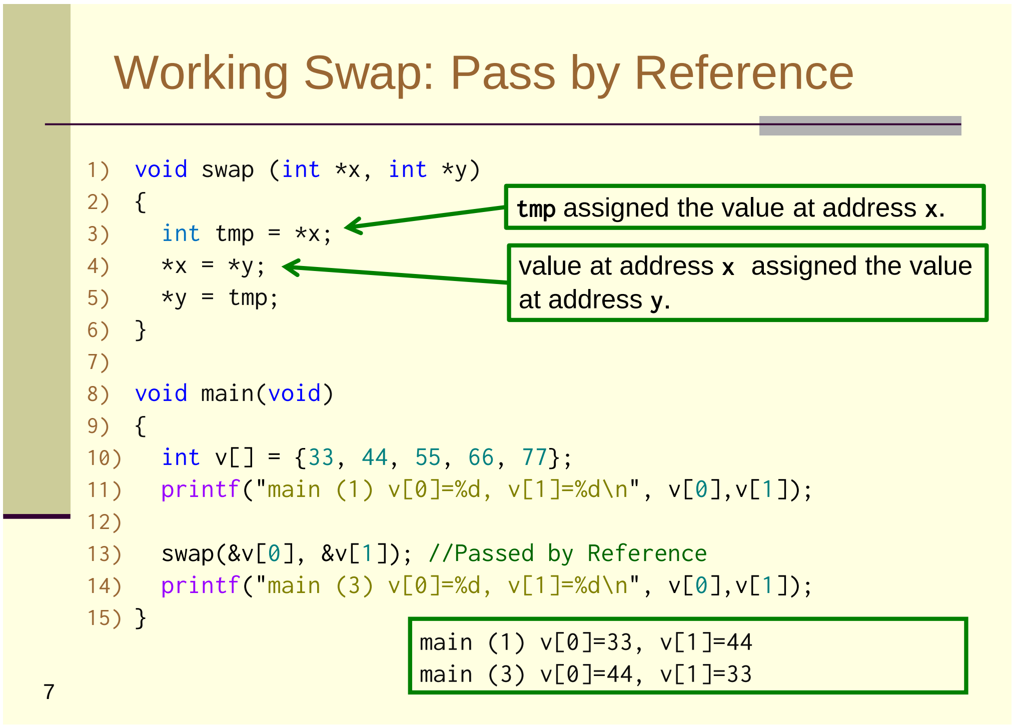
```
18, 12, 2
```



Swap Error: Pass by Value

```
void swapNot(int x, int y)
{ printf("swapNot (1) x=%d, y=%d\n", x,y);
  int tmp = x;
  x = y;
  y = tmp;
  printf("swapNot (2) x=%d, y=%d\n", x,y);
}



void main(void)
{
  int v[] = {33, 44, 55, 66, 77};
  printf("main (1) v[0]=%d, v[1]=%d\n", v[0],v[1]);

  swapNot(v[0], v[1]); //Passed by Value
  printf("main (2) v[0]=%d, v[1]=%d\n", v[0],v[1]);
}
```

```
main (1) v[0]=33, v[1]=44
swapNot (1) x=33, y=44
swapNot (2) x=44, y=33
main (2) v[0]=33 v[1]=44
```



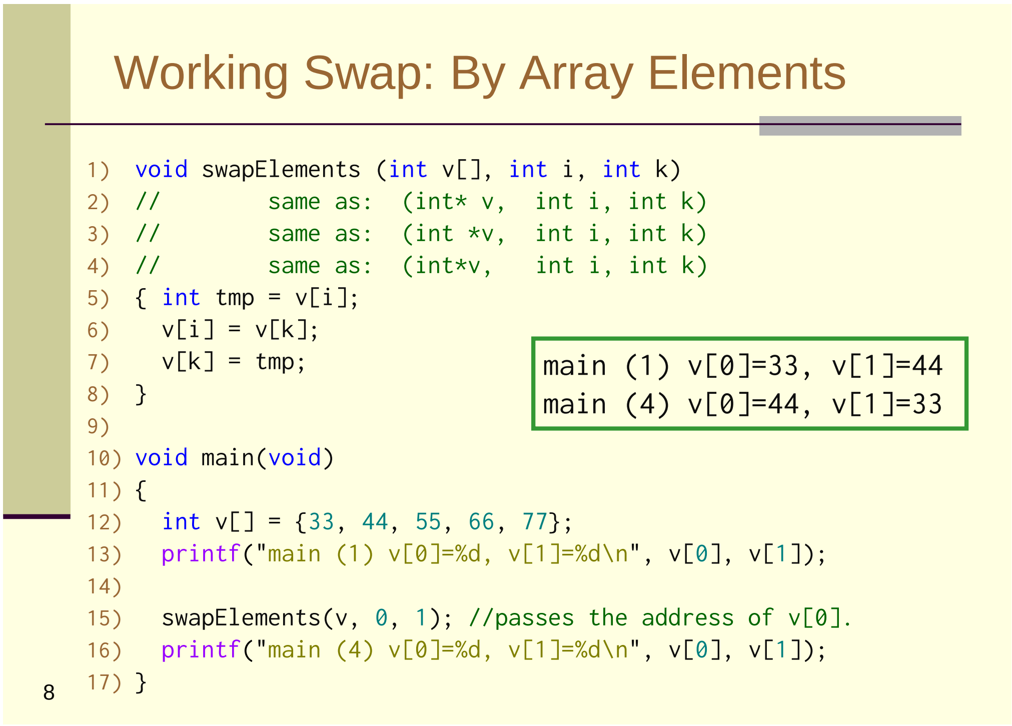
Working Swap: Pass by Reference

```
void swap (int *x, int *y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

void main(void)
{
  int v[] = {33, 44, 55, 66, 77};
  printf("main (1) v[0]=%d, v[1]=%d\n", v[0],v[1]);

  swap(&v[0], &v[1]); //Passed by Reference
  printf("main (3) v[0]=%d, v[1]=%d\n", v[0],v[1]);
}
```

tmp assigned the value at address **x**.

value at address **x** assigned the value at address **y**.

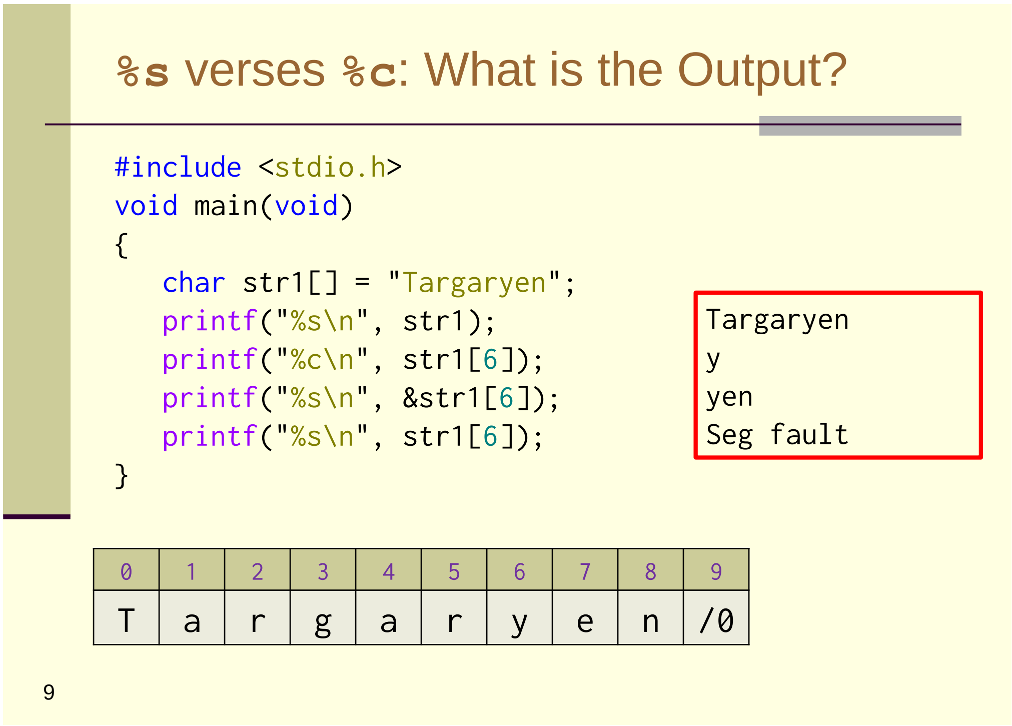
```
main (1) v[0]=33, v[1]=44
main (3) v[0]=44, v[1]=33
```



Working Swap: By Array Elements

```
void swapElements (int v[], int i, int k)
//        same as:  (int* v,  int i, int k)
//        same as:  (int *v,  int i, int k)
//        same as:  (int*v,   int i, int k)
{ int tmp = v[i];
  v[i] = v[k];
  v[k] = tmp;
}

void main(void)
{
  int v[] = {33, 44, 55, 66, 77};
  printf("main (1) v[0]=%d, v[1]=%d\n", v[0], v[1]);

  swapElements(v, 0, 1); //passes the address of v[0].
  printf("main (4) v[0]=%d, v[1]=%d\n", v[0], v[1]);
}
```

```
main (1) v[0]=33, v[1]=44
main (4) v[0]=44, v[1]=33
```

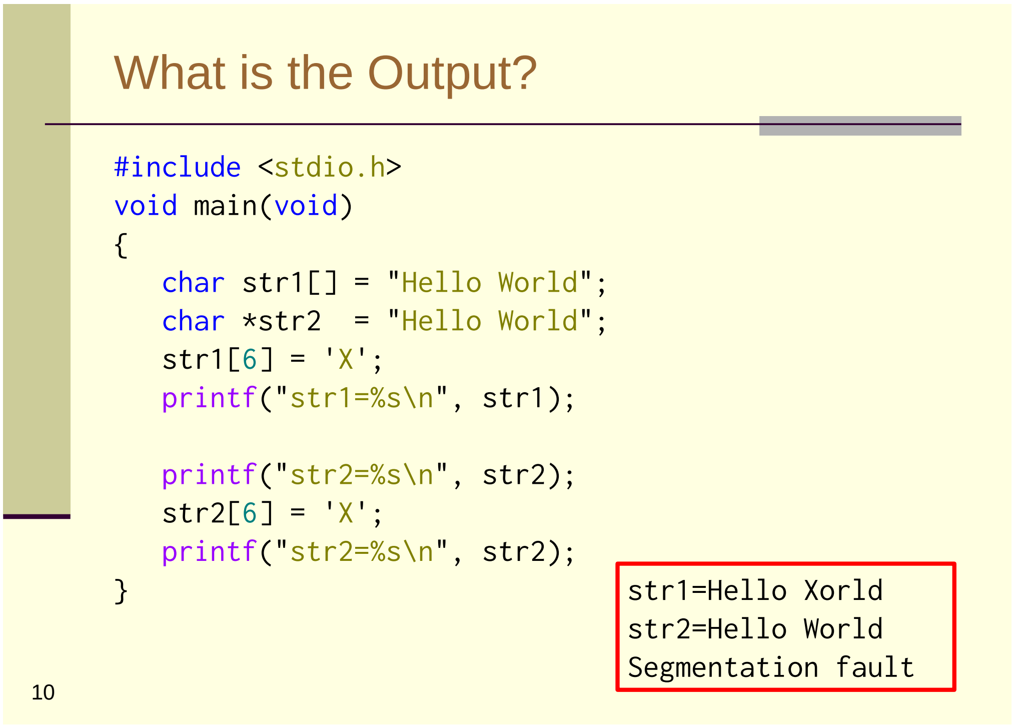


%s verses %c: What is the Output?

```
#include <stdio.h>
void main(void)
{
   char str1[] = "Targaryen";
   printf("%s\n", str1);
   printf("%c\n", str1[6]);
   printf("%s\n", &str1[6]);
   printf("%s\n", str1[6]);
}
```

```
Targaryen
y
yen
Seg fault
```

0	1	2	3	4	5	6	7	8	9
T	a	r	g	a	r	y	e	n	/0



What is the Output?

```
#include <stdio.h>
void main(void)
{
   char str1[] = "Hello World";
   char *str2  = "Hello World";
   str1[6] = 'X';
   printf("str1=%s\n", str1);

   printf("str2=%s\n", str2);
   str2[6] = 'X';
   printf("str2=%s\n", str2);
}
```

```
str1=Hello Xorld
str2=Hello World
Segmentation fault
```

Address Arithmetic

```
int n=17;
int*   a = &n;
short* b = (short*)&n;
char*  c = (char*)&n;

printf("%p %p %p\n", a, b, c);
a++;  b++;  c++;

printf("%p %p %p\n", a, b, c);
printf("%d\n", n);
```

The values at ***a**, ***b** and ***c** are undefined and may be seg fault.

```
0x7fff1985494c 0x7fff1985494c 0x7fff1985494c
0x7fff19854950 0x7fff1985494e 0x7fff1985494d
17
```

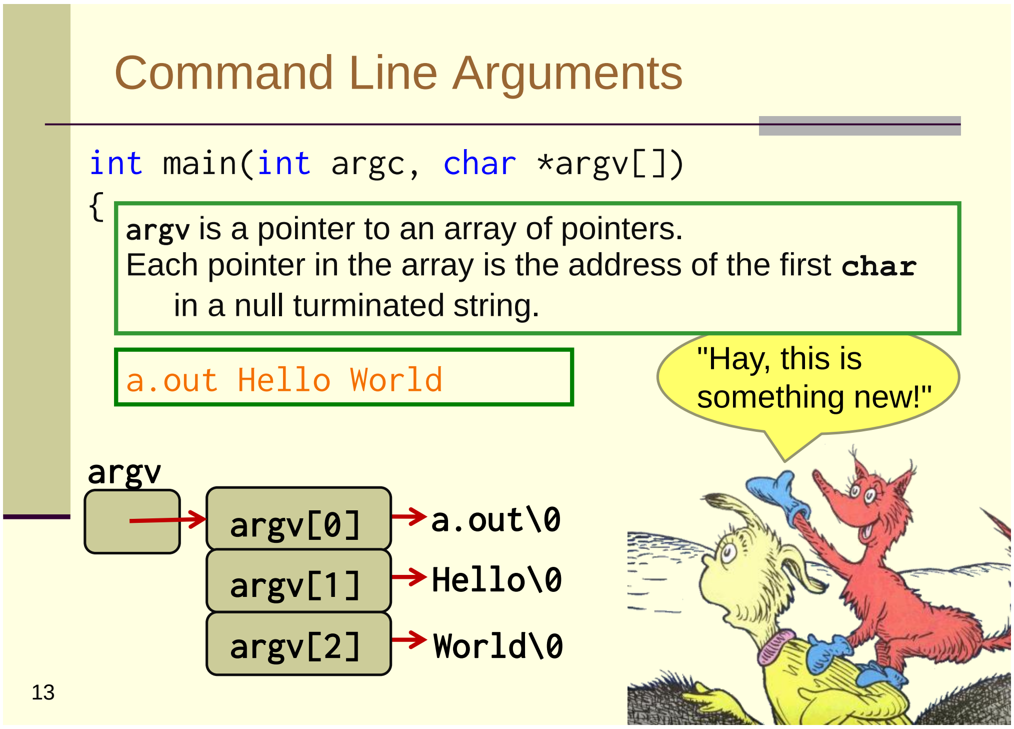
String Length by Index & Address Arithmetic

```
int strLen(char s[])
{
  int i=0;
  while (s[i]) i++;
  return i;
}
```

```
int strLen2(char *s)
{
  char *p = s;
  while (*p) p++;
  return p - s;
}
```

```
s[i]: Machine Code
  get s
  get i
  add
  get *topofstack
```

```
*p: Machine Code
  get p
  get *topofstack
```



Command Line Arguments

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

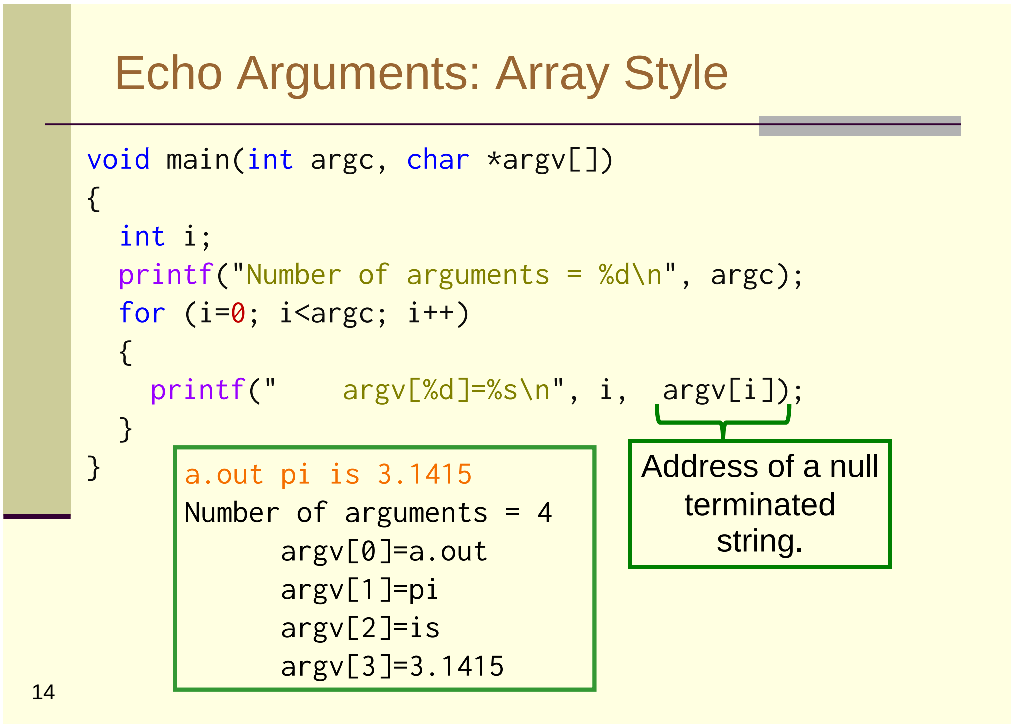
argv is a pointer to an array of pointers.
Each pointer in the array is the address of the first **char** in a null turminated string.

```
a.out Hello World
```

"Hay, this is something new!"

argv

argv	→ argv[0]	→ a.out\0
	argv[1]	→ Hello\0
	argv[2]	→ World\0

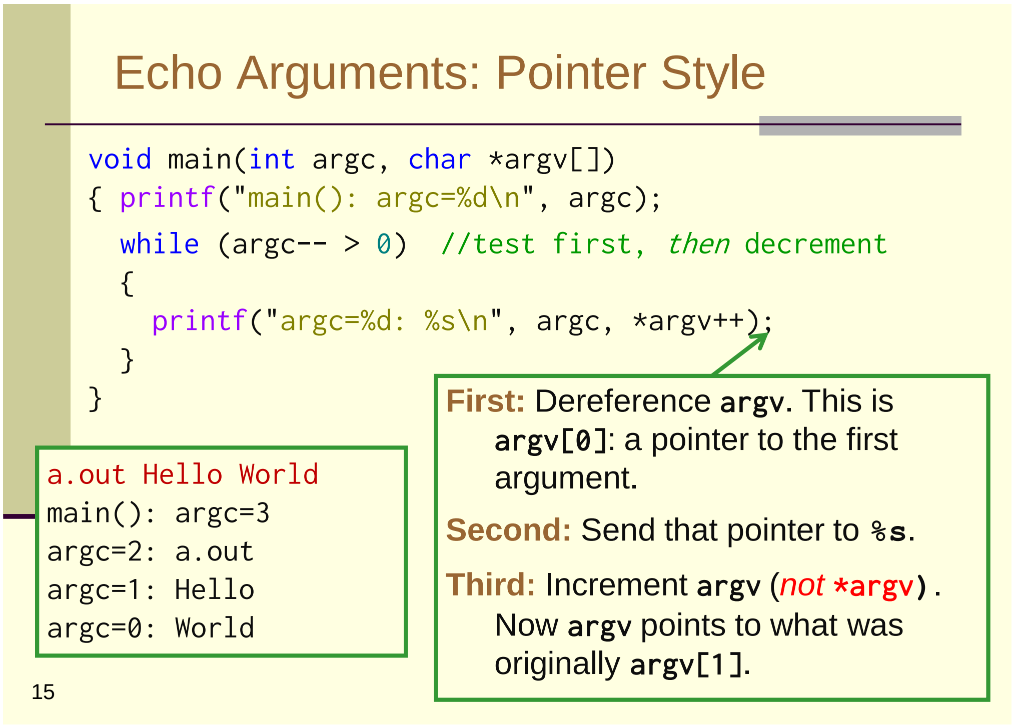


Echo Arguments: Array Style

```
void main(int argc, char *argv[])
{
  int i;
  printf("Number of arguments = %d\n", argc);
  for (i=0; i<argc; i++)
  {
    printf("    argv[%d]=%s\n", i,  argv[i]);
  }
}
```

Address of a null terminated string.

```
a.out pi is 3.1415
Number of arguments = 4
       argv[0]=a.out
       argv[1]=pi
       argv[2]=is
       argv[3]=3.1415
```



Echo Arguments: Pointer Style

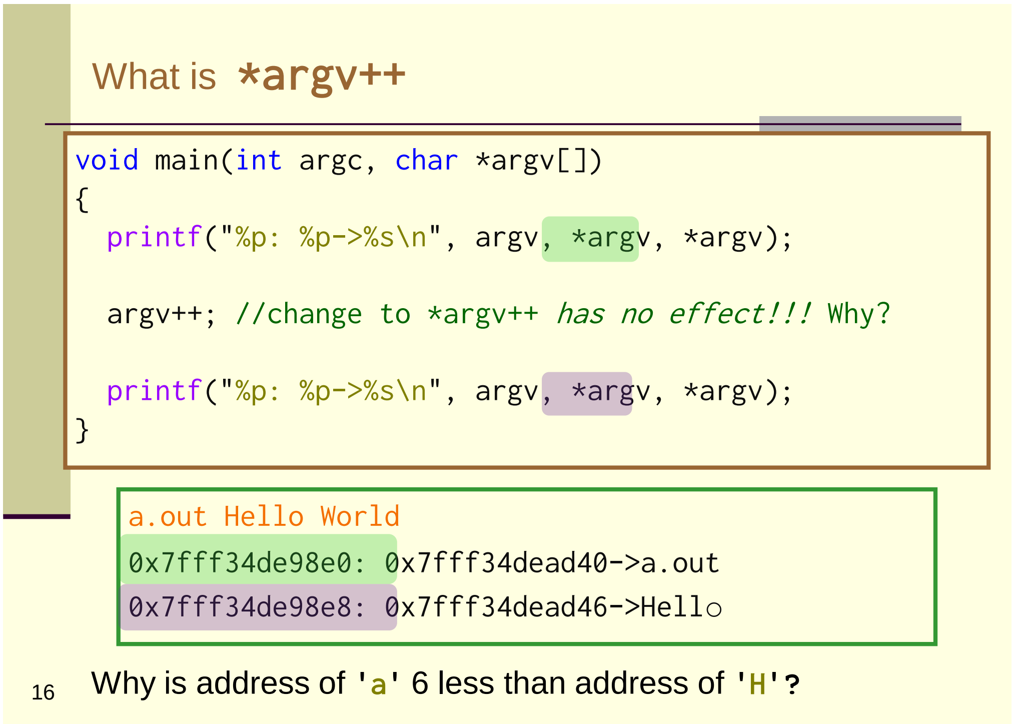
```
void main(int argc, char *argv[])
{ printf("main(): argc=%d\n", argc);
  while (argc-- > 0)  //test first, then decrement
  {
    printf("argc=%d: %s\n", argc, *argv++);
  }
}
```

```
a.out Hello World
main(): argc=3
argc=2: a.out
argc=1: Hello
argc=0: World
```

First: Dereference **argv**. This is **argv[0]**: a pointer to the first argument.

Second: Send that pointer to **%s**.

Third: Increment **argv** (*not* ***argv**). Now **argv** points to what was originally **argv[1]**.

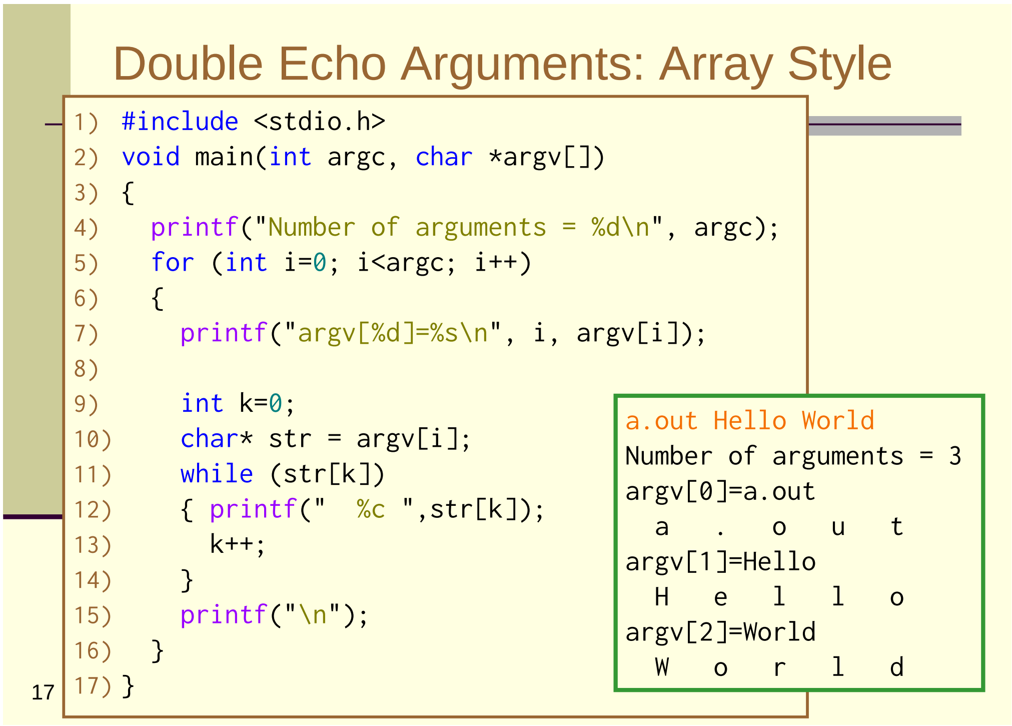


What is ***argv++**

```
void main(int argc, char *argv[])
{
  printf("%p: %p->%s\n", argv, *argv, *argv);

  argv++; //change to *argv++ has no effect!!! Why?

  printf("%p: %p->%s\n", argv, *argv, *argv);
}
```

```
a.out Hello World
0x7fff34de98e0: 0x7fff34dead40->a.out
0x7fff34de98e8: 0x7fff34dead46->Hello
```

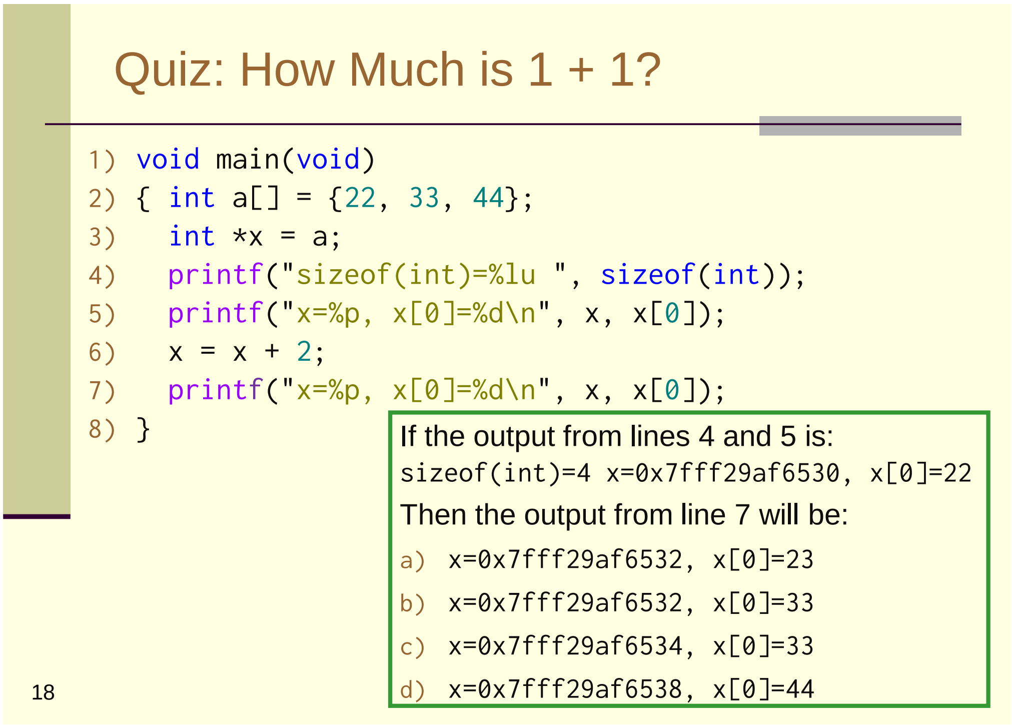
Why is address of **'a'** 6 less than address of **'H'**?



Double Echo Arguments: Array Style

```c
#include <stdio.h>
void main(int argc, char *argv[])
{
  printf("Number of arguments = %d\n", argc);
  for (int i=0; i<argc; i++)
  {
    printf("argv[%d]=%s\n", i, argv[i]);

    int k=0;
    char* str = argv[i];
    while (str[k])
    { printf("  %c ",str[k]);
      k++;
    }
    printf("\n");
  }
}
```

```
a.out Hello World
Number of arguments = 3
argv[0]=a.out
  a   .   o   u   t
argv[1]=Hello
  H   e   l   l   o
argv[2]=World
  W   o   r   l   d
```



Quiz: How Much is 1 + 1?

```c
1) void main(void)
2) { int a[] = {22, 33, 44};
3)   int *x = a;
4)   printf("sizeof(int)=%lu ", sizeof(int));
5)   printf("x=%p, x[0]=%d\n", x, x[0]);
6)   x = x + 2;
7)   printf("x=%p, x[0]=%d\n", x, x[0]);
8) }
```

If the output from lines 4 and 5 is:

```
sizeof(int)=4 x=0x7fff29af6530, x[0]=22
```

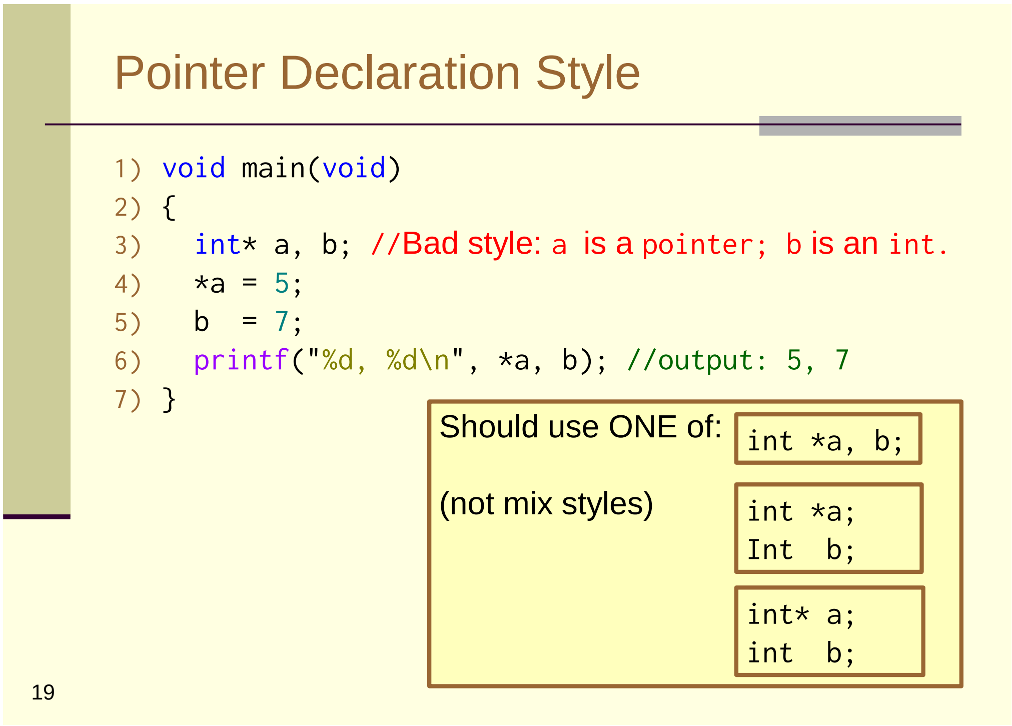
Then the output from line 7 will be:

a) x=0x7fff29af6532, x[0]=23

b) x=0x7fff29af6532, x[0]=33

c) x=0x7fff29af6534, x[0]=33

d) x=0x7fff29af6538, x[0]=44



Pointer Declaration Style

```
void main(void)
{
  int* a, b; //Bad style: a  is a pointer;  b is an int.
  *a = 5;
  b  = 7;
  printf("%d, %d\n", *a, b); //output: 5, 7
}
```

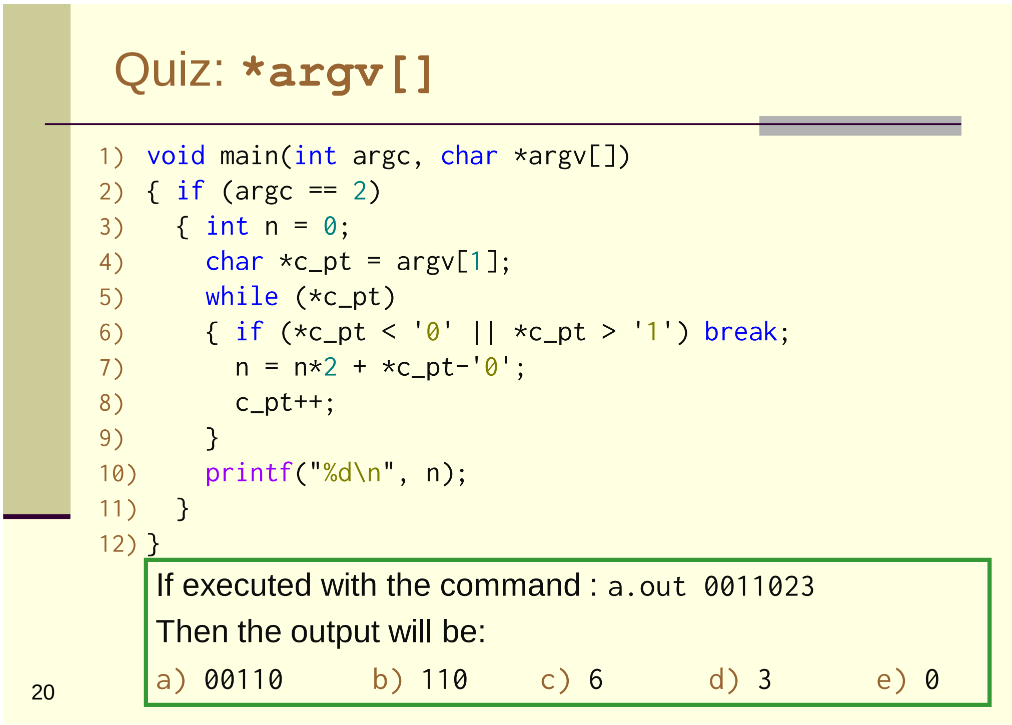
Should use ONE of:

(not mix styles)

```
int *a, b;
```

```
int *a;
Int  b;
```

```
int* a;
int  b;
```



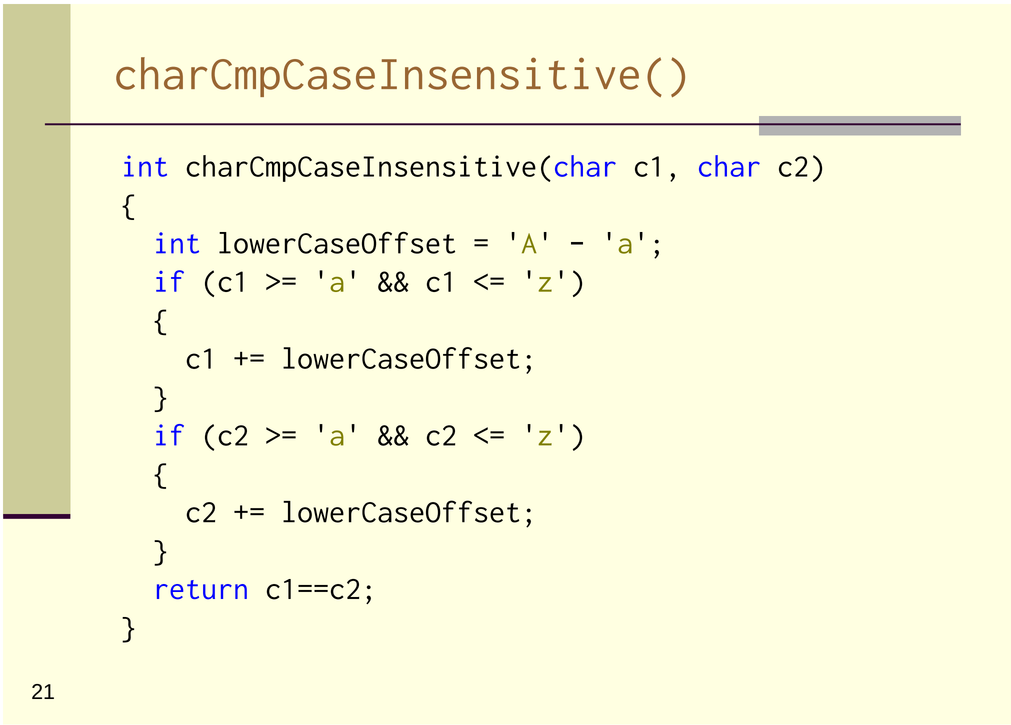
Quiz: ***argv[]**

```
void main(int argc, char *argv[])
{ if (argc == 2)
  { int n = 0;
    char *c_pt = argv[1];
    while (*c_pt)
    { if (*c_pt < '0' || *c_pt > '1') break;
      n = n*2 + *c_pt-'0';
      c_pt++;
    }
    printf("%d\n", n);
  }
}
```

If executed with the command : `a.out 0011023`

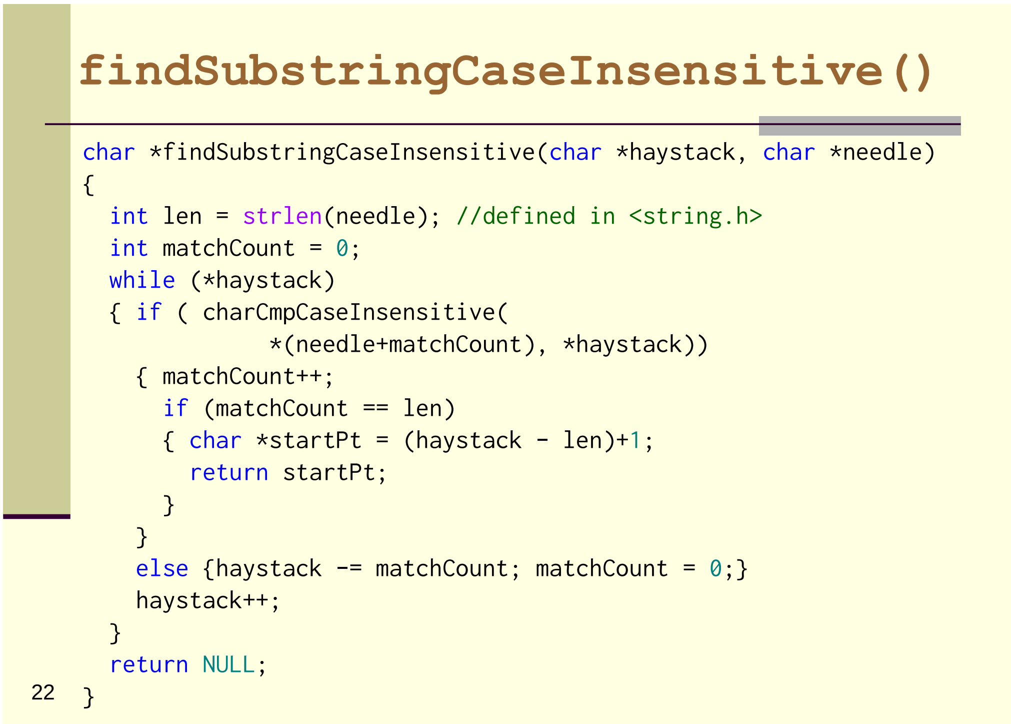
Then the output will be:

a) 00110 b) 110 c) 6 d) 3 e) 0



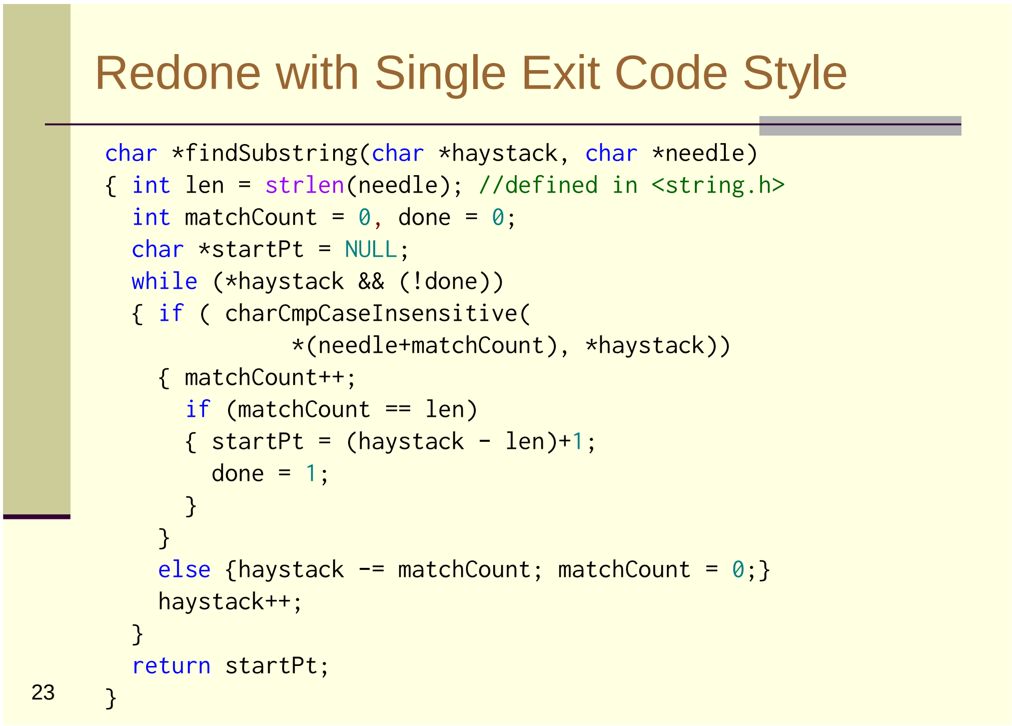
charCmpCaseInsensitive()

```
int charCmpCaseInsensitive(char c1, char c2)
{
  int lowerCaseOffset = 'A' - 'a';
  if (c1 >= 'a' && c1 <= 'z')
  {
    c1 += lowerCaseOffset;
  }
  if (c2 >= 'a' && c2 <= 'z')
  {
    c2 += lowerCaseOffset;
  }
  return c1==c2;
}
```



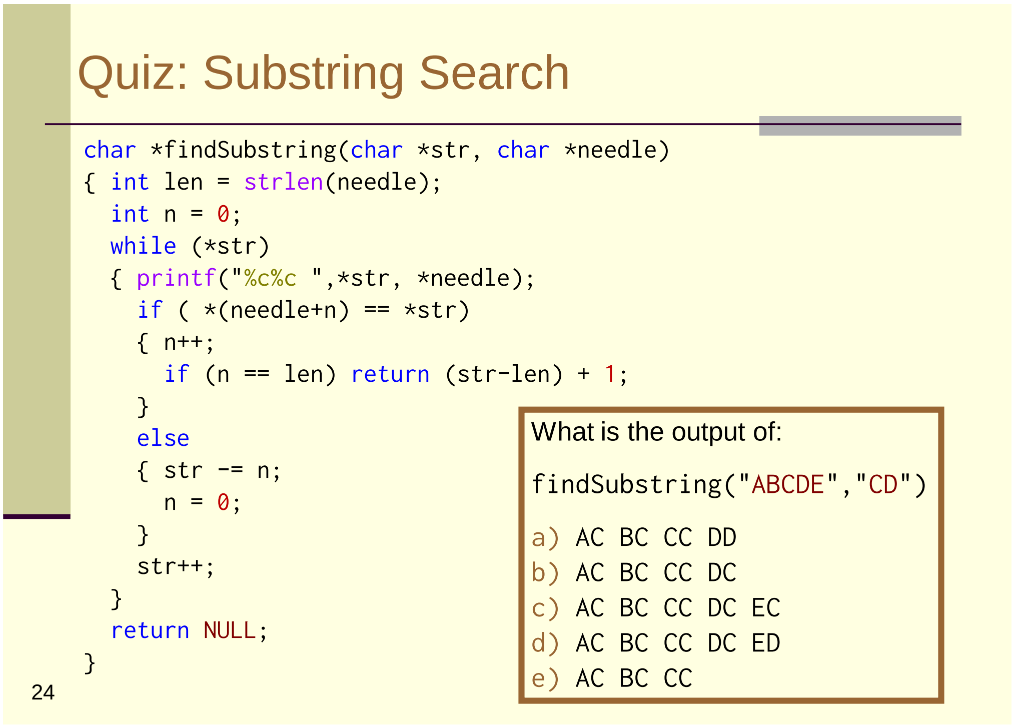
findSubstringCaseInsensitive()

```
char *findSubstringCaseInsensitive(char *haystack, char *needle)
{
  int len = strlen(needle); //defined in <string.h>
  int matchCount = 0;
  while (*haystack)
  { if ( charCmpCaseInsensitive(
              *(needle+matchCount), *haystack))
    { matchCount++;
      if (matchCount == len)
      { char *startPt = (haystack - len)+1;
        return startPt;
      }
    }
    else {haystack -= matchCount; matchCount = 0;}
    haystack++;
  }
  return NULL;
}
```



Redone with Single Exit Code Style

```
char *findSubstring(char *haystack, char *needle)
{ int len = strlen(needle); //defined in <string.h>
  int matchCount = 0, done = 0;
  char *startPt = NULL;
  while (*haystack && (!done))
  { if ( charCmpCaseInsensitive(
              *(needle+matchCount), *haystack))
    { matchCount++;
      if (matchCount == len)
      { startPt = (haystack - len)+1;
        done = 1;
      }
    }
    else {haystack -= matchCount; matchCount = 0;}
    haystack++;
  }
  return startPt;
}
```



Quiz: Substring Search

```
char *findSubstring(char *str, char *needle)
{ int len = strlen(needle);
  int n = 0;
  while (*str)
  { printf("%c%c ",*str, *needle);
    if ( *(needle+n) == *str)
    { n++;
      if (n == len) return (str-len) + 1;
    }
    else
    { str -= n;
      n = 0;
    }
    str++;
  }
  return NULL;
}
```

What is the output of:

`findSubstring("ABCDE","CD")`

a) AC BC CC DD
b) AC BC CC DC
c) AC BC CC DC EC
d) AC BC CC DC ED
e) AC BC CC

Quiz: Substring Search

```
char *findSubstring(char *str, char *needle)
{ int len = strlen(needle);
  int n = 0;
  while (*str)
  { printf("%c%c ",*str, *(needle+n));
    if ( *(needle+n) == *str)
    { n++;
      if (n == len) return (str-len) + 1;
    }
    else
    { str -= n;
      n = 0;
    }
    str++;
  }
  return NULL;
}
```

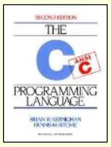
What is the output of:

`findSubstring("ACDCDEF","CDE"))`

a) AC CC DC CC DC CC DC EC
b) AC CC DC CC DC CC DD EE
c) AC CC DD CE DE CE DE EE
d) AC CC DD CE DC CC DD EC
e) AC CC DD CE DC CC DD EE

scanf(...): read from stdin

```
#include <stdio.h>

void main(void)
{ int n, m, a;
  float x;
  scanf("%d %d %f %d", &n, &m, &x, &a);

  printf("%d %d %f %d\n", n, m, x, a);
}
```



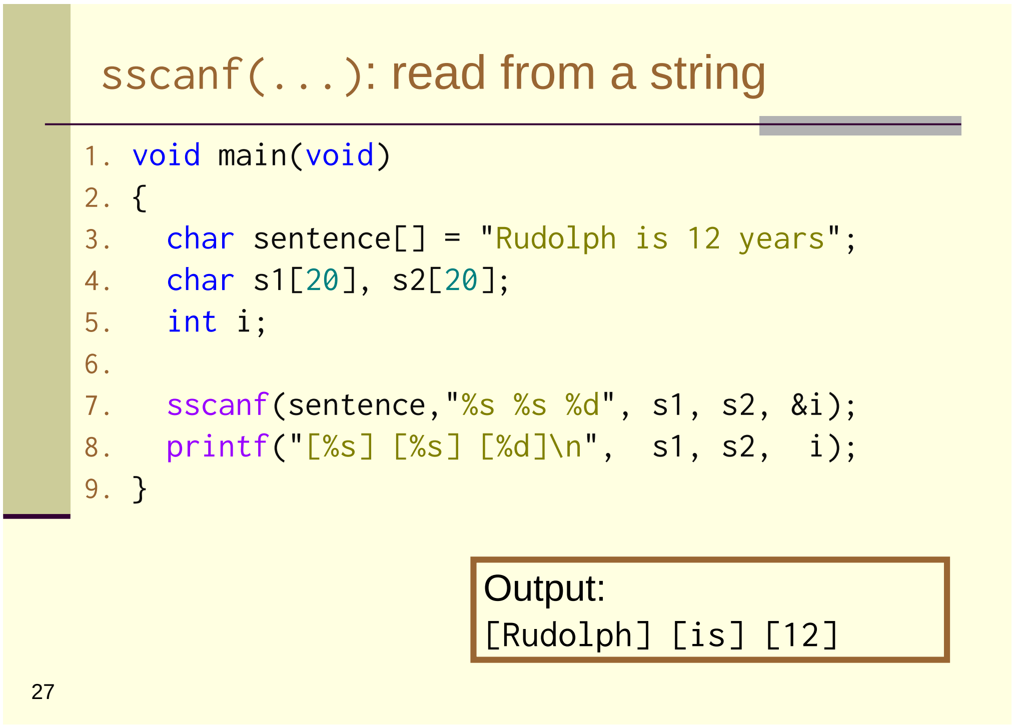
Kernighan & Ritchie
7.4 Formatted Input

Input:

```
2 49 3.1415
128
```

Output:

```
2 49 3.141500 128
```

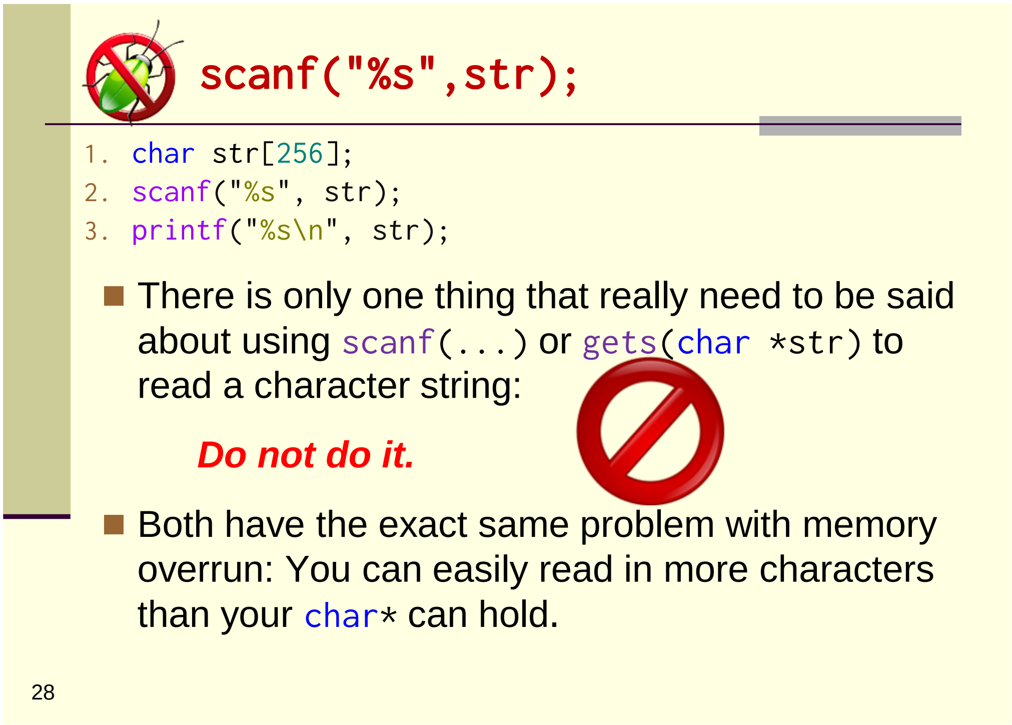


sscanf(...): read from a string

```
void main(void)
{
  char sentence[] = "Rudolph is 12 years";
  char s1[20], s2[20];
  int i;

  sscanf(sentence,"%s %s %d", s1, s2, &i);
  printf("[%s] [%s] [%d]\n",  s1, s2,  i);
}
```

Output:

```
[Rudolph] [is] [12]
```



scanf("%s",str);

```
char str[256];
scanf("%s", str);
printf("%s\n", str);
```

- There is only one thing that really need to be said about using `scanf(...)` or `gets(char *str)` to read a character string:

  Do not do it.

- Both have the exact same problem with memory overrun: You can easily read in more characters than your `char*` can hold.

fgets: Get a String From a Stream

SYNOPSIS

```
    #include <stdio.h>
    char *fgets(char *s, int n, FILE *stream);
```

DESCRIPTION

The fgets() function shall read bytes from stream into the array pointed to by **s**, until **n-1** bytes are read, or a <newline> is read and transferred to s, or an end-of-file condition is encountered. The string is then terminated with a null byte.

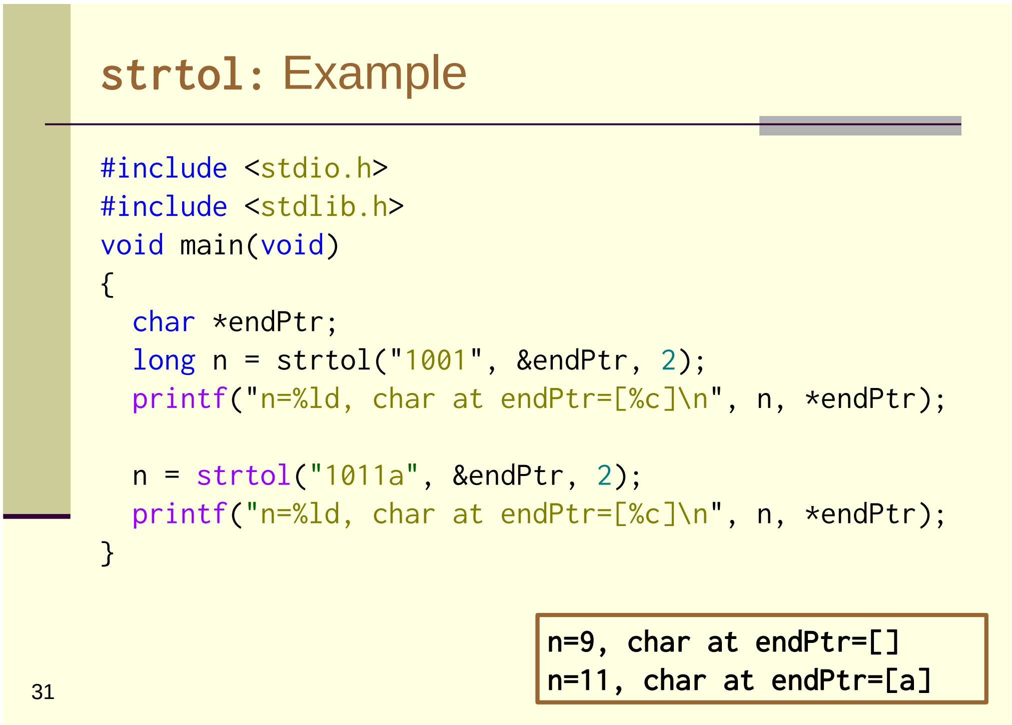
strtol: Convert String to Long

SYNOPSIS

```
    #include <stdlib.h>
    long strtol(const char *nptr,
                    char **endptr, int base);
```

DESCRIPTION

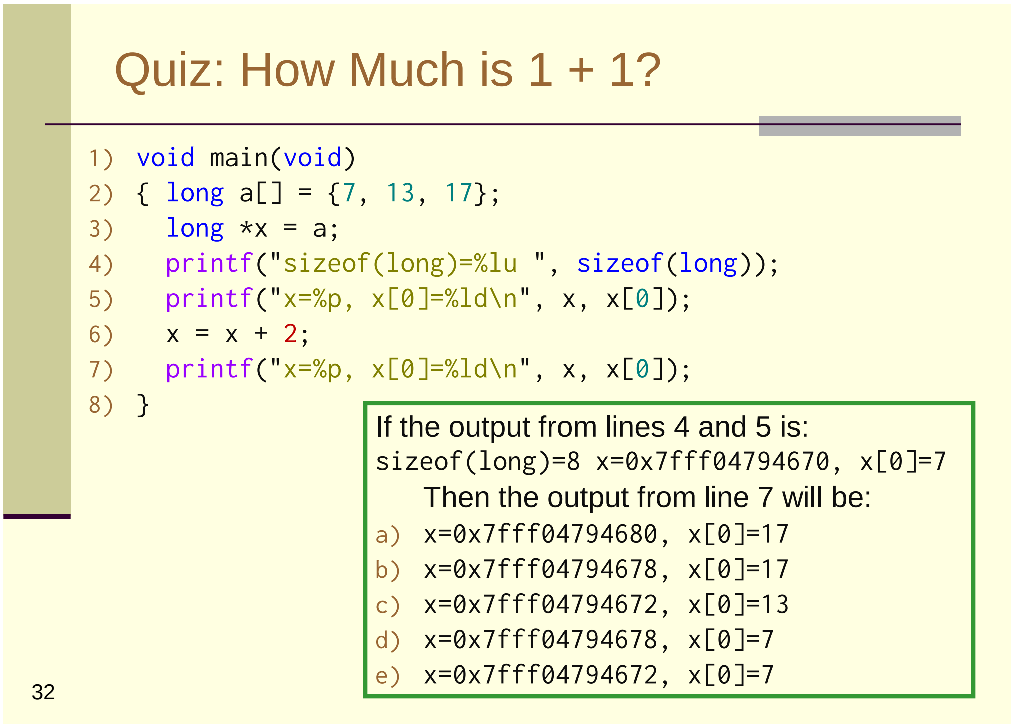
- The strtol() function converts the string pointed to by nptr to a long int representation.
- The first unrecognized character ends the string. A pointer to this unrecognized character is stored in the object addressed by endptr
- If base ([0, 36]) is non-zero, its value determines the set of recognized digits.



strtol: Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  char *endPtr;
  long n = strtol("1001", &endPtr, 2);
  printf("n=%ld, char at endPtr=[%c]\n", n, *endPtr);

  n = strtol("1011a", &endPtr, 2);
  printf("n=%ld, char at endPtr=[%c]\n", n, *endPtr);
}
```

```
n=9, char at endPtr=[]
n=11, char at endPtr=[a]
```



Quiz: How Much is 1 + 1?

```
1) void main(void)
2) { long a[] = {7, 13, 17};
3)   long *x = a;
4)   printf("sizeof(long)=%lu ", sizeof(long));
5)   printf("x=%p, x[0]=%ld\n", x, x[0]);
6)   x = x + 2;
7)   printf("x=%p, x[0]=%ld\n", x, x[0]);
8) }
```

If the output from lines 4 and 5 is:

`sizeof(long)=8 x=0x7fff04794670, x[0]=7`

Then the output from line 7 will be:

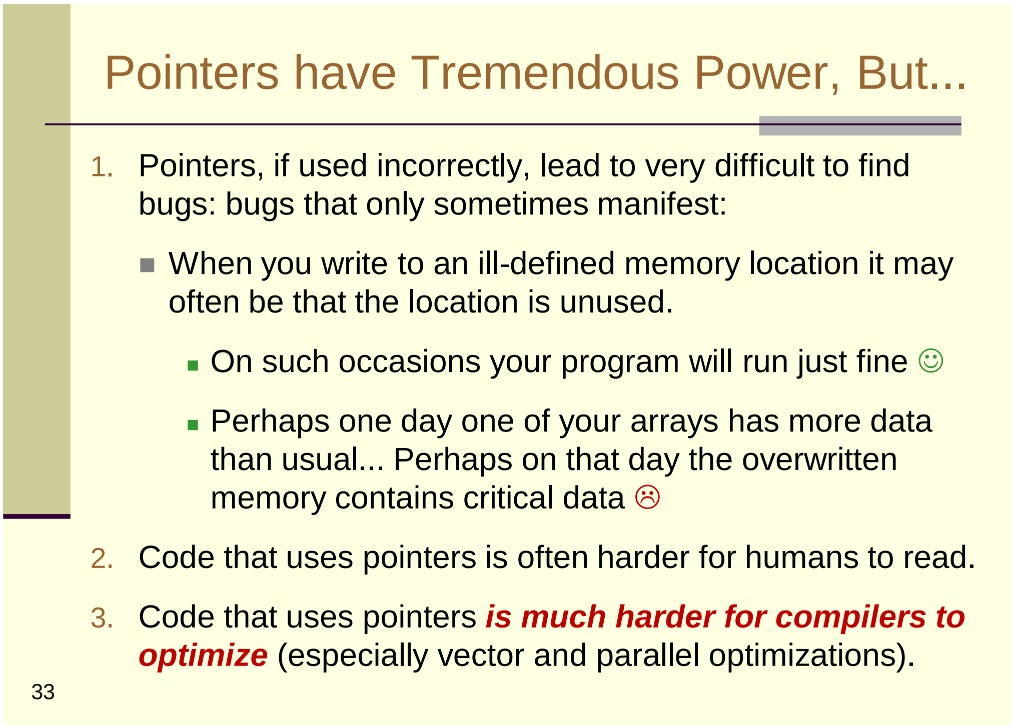
a) `x=0x7fff04794680, x[0]=17`

b) `x=0x7fff04794678, x[0]=17`

c) `x=0x7fff04794672, x[0]=13`

d) `x=0x7fff04794678, x[0]=7`

e) `x=0x7fff04794672, x[0]=7`



Pointers have Tremendous Power, But...

1. Pointers, if used incorrectly, lead to very difficult to find bugs: bugs that only sometimes manifest:
    - When you write to an ill-defined memory location it may often be that the location is unused.
        - On such occasions your program will run just fine ☺
        - Perhaps one day one of your arrays has more data than usual... Perhaps on that day the overwritten memory contains critical data ☹
2. Code that uses pointers is often harder for humans to read.
3. Code that uses pointers ***is much harder for compilers to optimize*** (especially vector and parallel optimizations).