CSE 1320

Week of 02/25/2019

Instructor : Donna French

More Tools for Our Toolbox

`memcmp()` and `memcpy()` copy byte by byte regardless of what is in those bytes.

`strcpy()` and `strcmp()` look for null terminators

Two new library functions

`memcpy()` and `memcmp()`

`memcpy()` is a lot like `strcpy()` except that it does not rely on a null terminator – it is given the number of bytes to copy

`memcmp()` is a lot like `strcmp()` except that it does not rely on a null terminator – it is given the number of bytes to copy

memcpy() and memcmp()

```
char Array1[100] = {"The quick fox jumps"};
char Array2[100];
char Array3[100];

memcpy(Array2, Array1, strlen(Array1));
memcpy(Array3, Array1, strlen(Array1)+1);

printf("Array1  %s\nArray2  %s\nArray3
        Array1, Array2, Array3);
```

```
Array1  The quick fox jumps
Array2  The quick fox jumps▒*
Array3  The quick fox jumps
```

Array2 does not include \0

```
(gdb) p Array2
$1 = "The quick fox jumps\252\252*\000\000\020\350\377\377\377\177\000\000\000ч\
252*\000\000<\003@\000\000\000\000\000v", '\000' <repeats 39 times>"\377, \265\3
60\000\000\000\000\000\302\000\000"
```

Array3 does include \0

```
 (gdb) p Array3
$2 = "The quick fox jumps\000\000\000\000\000\002\223\000\311>", '\000' <repeats
 11 times>"\340, \366\252\252\252*\000\000\001", '\000' <repeats 15 times>, "\00
1\000\000\000\000\000\000\000`\310Q\311>\000\000\000\000\347\377\377\377\177\000
\000\000\000\000\000\000\000\000\000v\000\000"
```

memcpy() and memcmp()

```
char Array4[100] = {"hello\0\0\0"};
char Array5[100] = {"hello\0!!"};

if (memcmp(Array4, Array5, 8) == 0)
     printf("equal\n");
else
     printf("not equal\n");

if (strcmp(Array4, Array5) == 0)
     printf("equal");
else
     printf("not equal\n");
```

memcmp() returns 0 if the two arrays are equal for the number of characters compared

\0 is not equal to !

only compares up to the \0

memcpy()

```
char Array1[100] = {"The quick fox jumps"};
char Array2[100] = {};

memcpy(&Array1[4], "brown", 5);
```

The brown fox jumps

```
memcpy(&Array1[strlen("The quick ")], "dog park", 3);
```

The brown dog jumps

```
memcpy(&Array1[14], "shakes", strlen("jumps")+1);

strcat(Array2, Array1);

printf("%s", Array2);
```

The brown dog shakes

must include `<stdlib.h>`

More Tools for Our Toolbox

Two new library functions

`atoi()` and `atof()`

`atof()` takes a null terminated string containing the ASCII representation of a floating point number as its parameter and converts the string to the corresponding value of type `float` and returns that value.

`atoi()` takes a null terminated string containing the ASCII representation of an integer number as its parameter and converts the string to the corresponding value of type `int` and returns that value.

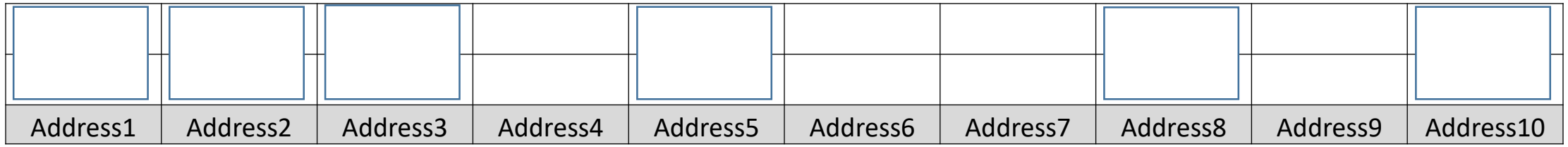
```
15                  printf("Enter a float value ");
(gdb)
16                  fgets(Input, 100, stdin);
(gdb)
Enter a float value 21.9
18                  MyFloatVar1 = atof(Input);
(gdb) p Input
$1 = "21.9\n", '\000' <repeats 94 times>
(gdb) step
23                  printf("MyFloatVar1 value is %f\n", MyFloatVar1);
(gdb) p MyFloatVar1
$2 = 21.8999996
(gdb) step
MyFloatVar1 value is 21.900000
```

```
25                  printf("\n\nEnter an integer value ");
(gdb)
26                  fgets(Input, 100, stdin);
(gdb)
Enter an integer value 12
28                  MyIntVar1 = atoi(Input);
(gdb) p Input
$3 = "12\n\000\n", '\000' <repeats 94 times>
(gdb) step
30                  printf("MyIntVar1 value is %d\n", MyIntVar1);
(gdb) p MyIntVar1
$4 = 12
(gdb) step
MyIntVar1 value is 12
```

Pointer Review

- Every variable has an address in memory

```
int VarA = 19;
int VarB = 32;
int VarC = 44;
IntVar1 = 67
IntVar2 = 23;
IntVar3 = 66;
```



Pointer Review

- A pointer can hold that address

```
int *PtrVarA = &VarA;
int *PtrVarC = &VarC;
int *PtrIntVar1 = &IntVar1;
```

VarA	VarB	VarC		IntVar3			IntVar2		IntVar1
19	32	44		66			23		67
Address1	Address2	Address3	Address4	Address5	Address6	Address7	Address8	Address9	Address10

Pointer Review

- Dereferencing the pointer gets to the contents

```
printf("Contents of PtrVarA %d", *PtrVarA);
printf("Contents of PtrVarC %d", *PtrVarC);
printf("Contents of PtrIntVar1 %d", *PtrIntVar1);
```

VarA	VarB	VarC	PtrVarA	IntVar3	PtrVarC	PtrIntVar1	IntVar2		IntVar1
19	32	44	Address1	66	Address3	Address10	23		67
Address1	Address2	Address3	Address4	Address5	Address6	Address7	Address8	Address9	Address10

Pointers

```
short *shortVarPtr = NULL;
int   *intVarPtr = NULL;
long  *longVarPtr = NULL;
char  *charVarPtr = NULL;
```

Pointers hold an address and all addresses are the same size

```
printf("The sizeof(short)     is %d\n", sizeof(short));
printf("The sizeof(int)       is %d\n", sizeof(int));
printf("The sizeof(long)      is %d\n", sizeof(long));
printf("The sizeof(char)      is %d\n", sizeof(char));
```

```
The sizeof(short)     is 2
The sizeof(int)       is 4
The sizeof(long)      is 8
The sizeof(char)      is 1
```

```
printf("The sizeof(shortVarPtr) is %d\n", sizeof(shortVarPtr));
printf("The sizeof(intVarPtr)   is %d\n", sizeof(intVarPtr));
printf("The sizeof(longVarPtr)  is %d\n", sizeof(longVarPtr));
printf("The sizeof(charVarPtr)  is %d\n\n", sizeof(charVarPtr));
```

```
The sizeof(shortVarPtr) is 8
The sizeof(intVarPtr)   is 8
The sizeof(longVarPtr)  is 8
The sizeof(charVarPtr)  is 8
```

An Array as a Pointer

Array declaration

```
int IntArray[10];
```

When the compiler processes this array declaration, it sets aside a region in memory large enough to store 10 cells of type `int`.

It also associates the address of the first cell in the array with the name `IntArray`.

Anywhere in the program that `IntArray` is used without the [], the name evaluates to the address of the first cell in the array.

```
int  *IntVarPtr = NULL;
int  IntArray[10] = {134,278,312,467,523,687,789,811,987,101};

printf("Contents of   IntArray[0] %d\n",    IntArray[0]);
printf("Address  of   IntArray    %p\n",    IntArray);

IntVarPtr = IntArray;

printf("Contents of   IntArray[0] %d\n",    IntArray[0]);
printf("Address  of   IntArray    %p\n",    IntArray);
printf("Contents of   IntVarPtr   %p\n",    IntVarPtr);
printf("Dereferencing IntVarPtr   %d\n",   *IntVarPtr);
```

Anywhere `IntArray` is used without the [], it is the address of the first cell.

```
Contents of   IntArray[0] 134
Address  of   IntArray    0x7ffffff66d30

Contents of   IntArray[0] 134
Address  of   IntArray    0x7ffffff66d30
Contents of   IntVarPtr   0x7ffffff66d30
Dereferencing IntVarPtr   134
```

arrayptrDemo.c

Pointer Arithmetic

A pointer may be incremented (++) or decremented (--)

```
IntVarPtr++          ++IntVarPtr
IntVarPtr--          --IntVarPtr
```

An integer may be added to a pointer or subtracted from a pointer

```
IntVarPtr += 2   IntVarPtr = IntVarPtr - 45
```

One pointer may be subtracted from another of the same type

```
IntVarPtr1 = IntVarPtr2 - IntVarPtr3
```

The amount of the increment/decrement is relative to the `sizeof()` the type the pointer is pointing to.

```c
#include <stdio.h>

#define MAX_CELLS 10

int main(void)
{
  int  *IntVarPtr = NULL;
  int  IntArray[MAX_CELLS] = {134,278,312,467,523,687,789,811,987,101};
  int i;

  IntVarPtr = IntArray;

  for (i = 0; i < MAX_CELLS; i++)
  {
    printf("IntArray[%d] = %d\t", i, IntArray[i]);
    printf("IntArrayPtr + %d = %d\t", i, *(IntVarPtr + i));
    printf("IntArray + %d = %d\n", i, *(IntArray + i));
  }

  return 0;
}
```

ptrarith1Demo.c

```
IntVarPtr = IntArray;

for (i = 0; i < MAX_CELLS; i++)
{
  printf("IntArray[%d] = %d\t", i, IntArray[i]);
  printf("IntArrayPtr + %d = %d\t", i, *(IntVarPtr + i));
  printf("IntArray + %d = %d\n", i, *(IntArray + i));
}
```

```
IntArray[0] = 134
IntArray[1] = 278
IntArray[2] = 312
IntArray[3] = 467
IntArray[4] = 523
IntArray[5] = 687
IntArray[6] = 789
IntArray[7] = 811
IntArray[8] = 987
IntArray[9] = 101
```

```
IntArrayPtr + 0 = 134
IntArrayPtr + 1 = 278
IntArrayPtr + 2 = 312
IntArrayPtr + 3 = 467
IntArrayPtr + 4 = 523
IntArrayPtr + 5 = 687
IntArrayPtr + 6 = 789
IntArrayPtr + 7 = 811
IntArrayPtr + 8 = 987
IntArrayPtr + 9 = 101
```

```
IntArray + 0 = 134
IntArray + 1 = 278
IntArray + 2 = 312
IntArray + 3 = 467
IntArray + 4 = 523
IntArray + 5 = 687
IntArray + 6 = 789
IntArray + 7 = 811
IntArray + 8 = 987
IntArray + 9 = 101
```

ptrarith1Demo.c

```
for (i = 0; i < MAX_CELLS; i++)
{

  printf("IntArrayPtr + %d = %d\t", i, *(IntVarPtr + i));

}
```

```
for (i = 0; i < MAX_CELLS; i++, IntVarPtr++)
{

  printf("IntArrayPtr + %d = %d\t", i, *IntVarPtr);

}
```

Difference between

`*IntVarPtr + i` `*(IntVarPtr + i)`

ptrarith2Demo.c

```
  for (i = 0; i < MAX_CELLS; i++, CharVarPtr++)
  {
    printf("CharArray[%d] = %c CharVarPtr = %p *CharVarPtr = %c\n",
           i, CharArray[i], CharVarPtr, *CharVarPtr);
  }
  for (i = 0; i < MAX_CELLS; i++, IntVarPtr++)
  {
    printf("IntArray[%d] = %d IntVarPtr = %p *IntVarPtr = %d\n“,
           i, IntArray[i], IntVarPtr, *IntVarPtr);
  }
  for (i = 0; i < MAX_CELLS; i++, LongVarPtr++)
  {
    printf("LongArray[%d] = %ld LongVarPtr = %p *LongVarPtr = %d\n",
           i, LongArray[i], LongVarPtr, *LongVarPtr);
  }
```

```
CharArray
{"ABC"}

IntArray
{134,278,312}

LongArray
{111,222,333}
```

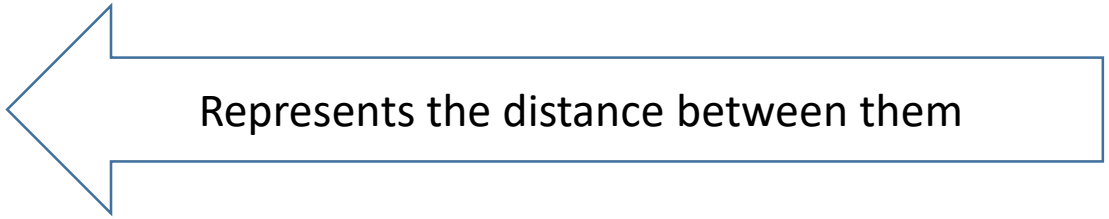
```
CharArray[0] = A CharVarPtr = 0x7fff4d0170c0 *CharVarPtr = A
CharArray[1] = B CharVarPtr = 0x7fff4d0170c1 *CharVarPtr = B
CharArray[2] = C CharVarPtr = 0x7fff4d0170c2 *CharVarPtr = C

IntArray[0] = 134 IntVarPtr = 0x7fff4d0170d0 *IntVarPtr = 134
IntArray[1] = 278 IntVarPtr = 0x7fff4d0170d4 *IntVarPtr = 278
IntArray[2] = 312 IntVarPtr = 0x7fff4d0170d8 *IntVarPtr = 312

LongArray[0] = 111 LongVarPtr = 0x7fff4d0170a0 *LongVarPtr = 111
LongArray[1] = 222 LongVarPtr = 0x7fff4d0170a8 *LongVarPtr = 222
LongArray[2] = 333 LongVarPtr = 0x7fff4d0170b0 *LongVarPtr = 333
```

Pointer arithmetic works for all different types.

ptrarith3Demo.c

```
int  IntArray[MAX_CELLS] = {134,278,312};
int  *IntVarPtr1 = IntArray;
int  *IntVarPtr2 = IntArray+1;
int  *IntVarPtr3 = IntVarPtr2+1;

printf("*IntVarPtr1 = %d\n", *IntVarPtr1);
printf("*IntVarPtr2 = %d\n", *IntVarPtr2);
printf("*IntVarPtr3 = %d\n", *IntVarPtr3);

printf("IntVarPtr3 - IntVarPtr1 = %d\n", IntVarPtr3 - IntVarPtr1);
printf("IntVarPtr1 - IntVarPtr3 = %d\n", IntVarPtr1 - IntVarPtr3);
```

```
*IntVarPtr1 = 134        IntVarPtr1 = 0x7fff03e93090
*IntVarPtr2 = 278        IntVarPtr2 = 0x7fff03e93094
*IntVarPtr3 = 312        IntVarPtr3 = 0x7fff03e93098

IntVarPtr3 - IntVarPtr1 = 2
IntVarPtr1 - IntVarPtr3 = -2
```



If I take the physical address of one house and “subtract” the physical address of a house down the street, then I would get the number of houses in between them.

If I take Setember 30th and “subtract” September 12th, then I would get the number of days in between them.

Pointer Arithmetic

Allowed operations

- A pointer may be incremented (++) or decremented (--)
- An integer may be added to a pointer or subtracted from a pointer
- One pointer may be subtracted from another of the same type

What about Pointer Addition?

```
printf("IntVarPtr1 + IntVarPtr2 = %d\n", IntVarPtr1 + IntVarPtr2);

[frenchdm@omega ~]$ gcc ptrarith4Demo.c
ptrarith4Demo.c: In function ‘main’:
ptrarith4Demo.c:23: error: invalid operands to binary +
```

- Not defined in the language. What would it mean?
- You can subtract two dates to get the number of days in between them. What would adding two dates mean?

Pointer Arithmetic

```
int Array1 = {1,2,3}
int Array2 = {4,5,6}

Array2[0] != Array1 + 3
```

Allowed operations

- A pointer may be incremented (++) or decremented (--)
- An integer may be added to a pointer or subtracted from a pointer

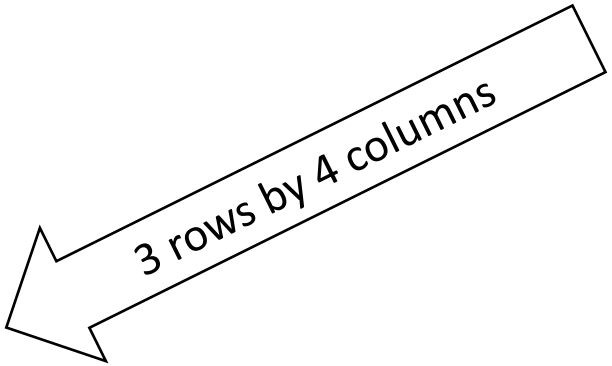
Pointer arithmetic is only used within arrays where the order of cells in memory is guaranteed.

Pointer arithmetic should not be used to travel between arrays.

Adding to/subtracting from a pointer does not guarantee the next/previous variable in your list of declarations – memory is not necessarily arranged in the order of your declarations.

Arrays of Arrays

```
int My2DArray[3][4];

int (My2DArray[3])[4];
```



	column 0	column 1	column 2	column 3
row 0	(0,0)	(0,1)	(0,2)	(0,3)
row 1	(1,0)	(1,1)	(1,2)	(1,3)
row 2	(2,0)	(2,1)	(2,2)	(2,3)

Arrays of Arrays

```
int My2DArray[ROW][COL] = {{1,2,3,4},
                           {5,6,7,8},
                           {9,10,11,12}};
```

1 (0,0)	2 (0,1)	3 (0,2)	4 (0,3)
5 (1,0)	6 (1,1)	7 (1,2)	8 (1,3)
9 (2,0)	10 (2,1)	11 (2,2)	12 (2,3)

Pointer Arithmetic with Two Dimensional Arrays

(99A81EA0) (0,0) 1	(99A81EA4) (0,1) 2	(99A81EA8) (0,2) 3	(99A81EAC) (0,3) 4
(99A81EB0) (1,0) 5	(99A81EB4) (1,1) 6	(99A81EB8) (1,2) 7	(99A81EBC) (1,3) 8
(99A81EC0) (2,0) 9	(99A81EC4) (2,1) 10	(99A81EC8) (2,2) 11	(99A81ECC) (2,3) 12

Pointer Arithmetic with Two Dimensional Arrays

Address of array `My2D`
Address of 1st element of 1st

Address of 1st element of 2nd array `*My2D+1`

Address of 1st element of 3rd array `*My2D+2`

	`*(My2D + 0) + 1`
	`(0,1)`
	2
		`*(My2D + 0) + 2`
		`(0,2)`
		3
			`*(My2D + 0) + 3`
			`(0,3)`
			4

	`*(My2D + 1) + 1`
	`(1,1)`
	6
		`*(My2D + 1) + 2`
		`(1,2)`
		7
			`*(My2D + 1) + 3`
			`(1,3)`
			8

`*(My2D + 2) + 0`
`(2,0)`
9
	`*(My2D + 2) + 1`
	`(2,1)`
	10
		`*(My2D + 2) + 2`
		`(2,2)`
		11
			`*(My2D + 2) + 3`
			`(2,3)`
			12

First element
`My2D[0][0]`
`*(*My2D)`
`**My2D`

Pointer Arithmetic with Two Dimensional Arrays

`*(*(My2D + 0) + 0)` (0,0) 1	`*(*(My2D + 0) + 1)` (0,1) 2	`*(*(My2D + 0) + 2)` (0,2) 3	`*(*(My2D + 0) + 3)` (0,3) 4
`*(*(My2D + 1) + 0)` (1,0) 5	`*(*(My2D + 1) + 1)` (1,1) 6	`*(*(My2D + 1) + 2)` (1,2) 7	`*(*(My2D + 1) + 3)` (1,3) 8
`*(*(My2D + 2) + 0)` (2,0) 9	`*(*(My2D + 2) + 1)` (2,1) 10	`*(*(My2D + 2) + 2)` (2,2) 11	`*(*(My2D + 2) + 3)` (2,3) 12

Arrays of Pointers

```
char *PtrArray[8];
```

`PtrArray` is of type `char *` so `PtrArray` is a pointer to `char`

`[8]` tells us that we have an array with 8 elements so

```
char *PtrArray[8]
```

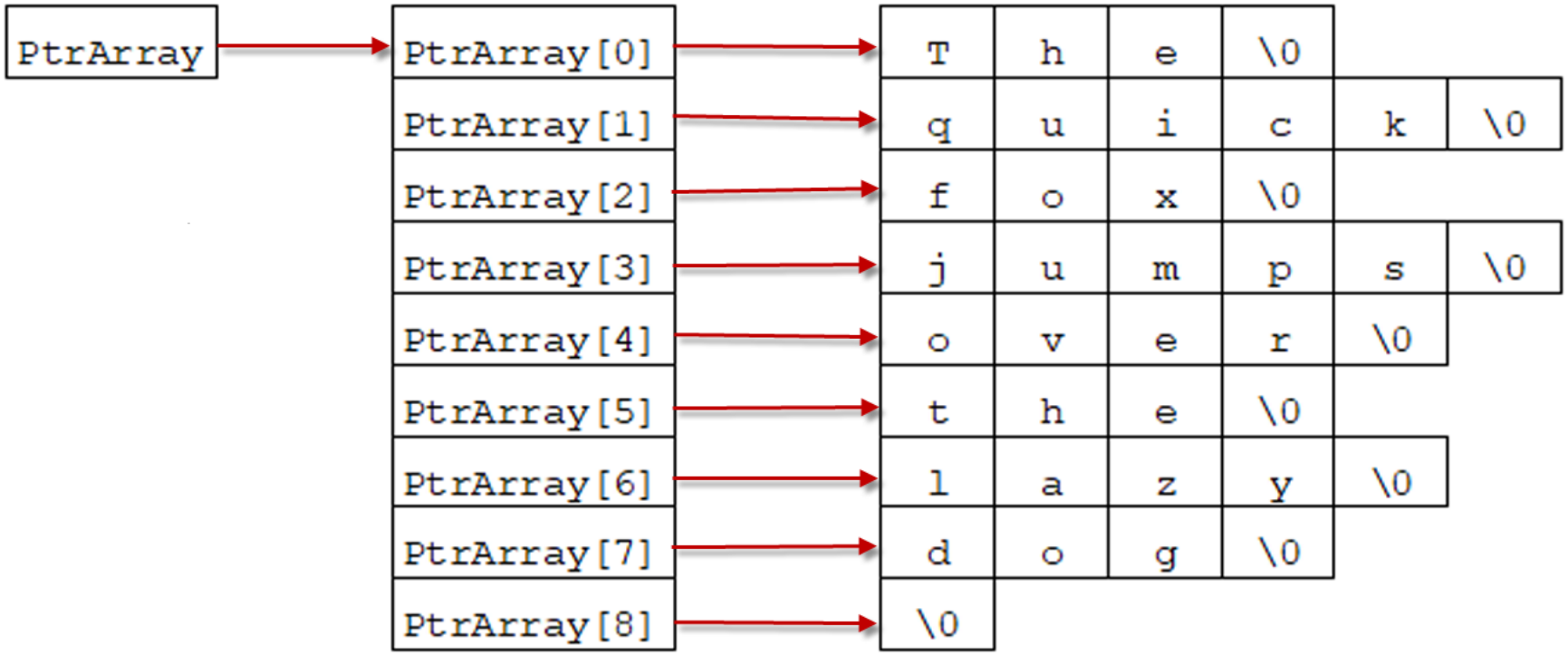
is an array of 8 pointers to `char`

Arrays of Pointers

```
char *PtrArray[8];
```

This construct is used in C to represent an array of strings.

The array name, `PtrArray`, is evaluated as the address of the first element of the array



```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

int i;

printf("sizeof(PtrArray) = %d\n"                    rArray));

for (i = 0; i < 9; i++)
{
   printf("PtrArray[%d]) =                          , PtrArray[i]);
   printf("PtrArray[%d])                            i, *(PtrArray[i]));
}
```

```
sizeof(PtrA                    = 72
```

What happens when we initialize a variable with a quoted string?

```
PtrArray[0]) = The
PtrArray[0]) = T
PtrArray[1]) = quick
PtrArray[1]) = q
PtrArray[2]) = fox
PtrArray[2]) = f
PtrArray[3]) = jumps
PtrArray[3]) = j
PtrArray[4]) = over
PtrArray[4]) = o
PtrArray[5]) = the
PtrArray[5]) = t
PtrArray[6]) = lazy
PtrArray[6]) = l
PtrArray[7]) = dog
PtrArray[7]) = d
PtrArray[8]) =
PtrArray[8]) =
```

arrayptr1Demo.c

Double Indirection

Since any type in C can have a pointer to it, we can declare a pointer to a pointer.

```
int **ptr;
```

`*ptr` is a pointer to an `int` so `**ptr` is a pointer to a pointer to an `int`.

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

char **PtrPtr = PtrArray;

int i;

printf("sizeof(PtrPtr)          %d\n"
       "sizeof(PtrArray)        %d\n",
        sizeof(PtrPtr), sizeof(PtrArray));

for (i = 0; i < 9; i++)
{
     printf("sizeof(PtrArray[%d]) = %d\n", i, sizeof(PtrArray[i]));
}
```

```
sizeof(PtrPtr)          8
sizeof(PtrArray)        72
```

```
sizeof(PtrArray[0]) = 8
sizeof(PtrArray[1]) = 8
sizeof(PtrArray[2]) = 8
sizeof(PtrArray[3]) = 8
sizeof(PtrArray[4]) = 8
sizeof(PtrArray[5]) = 8
sizeof(PtrArray[6]) = 8
sizeof(PtrArray[7]) = 8
sizeof(PtrArray[8]) = 8
```

dichar1Demo.c

dichar2Demo.c

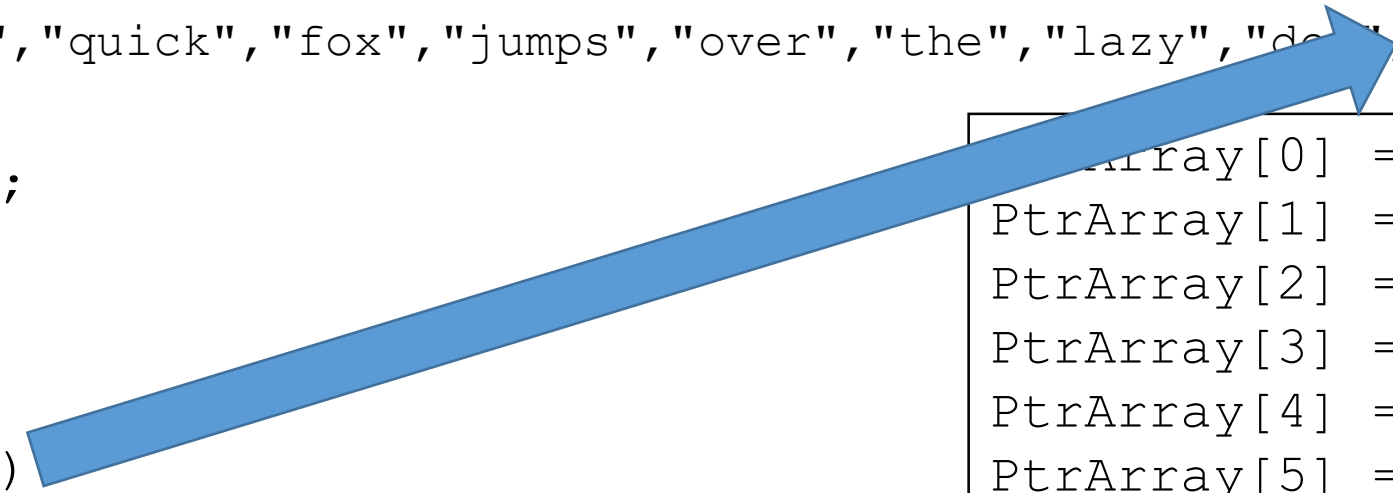
Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

char **PtrPtr = PtrArray;

int i;

for (i = 0; i < 9; i++)
{
   printf("PtrArray[%d] = %s\n",
          i, PtrArray[i]);
}

for (i = 0; i < 9; i++)
{
   printf("PtrPtr + %d = %s\n",
          i, *(PtrPtr + i));
}
```

```
PtrArray[0] = The
PtrArray[1] = quick
PtrArray[2] = fox
PtrArray[3] = jumps
PtrArray[4] = over
PtrArray[5] = the
PtrArray[6] = lazy
PtrArray[7] = dog
PtrArray[8] =
PtrPtr + 0 = The
PtrPtr + 1 = quick
PtrPtr + 2 = fox
PtrPtr + 3 = jumps
PtrPtr + 4 = over
PtrPtr + 5 = the
PtrPtr + 6 = lazy
PtrPtr + 7 = dog
PtrPtr + 8 =
```

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

char **PtrPtr = PtrArray;

int i = 0;

while (PtrArray[i] != "")
{
   printf("PtrArray[%d] = %s\n", i++, PtrArray[i]);
}

i = 0;
while (*(PtrPtr + i) != "")
{
   printf("PtrPtr + %d = %s\n", i++, *(PtrPtr + i));
}
```



```
PtrArray[0] = The
PtrArray[1] = quick
PtrArray[2] = fox
PtrArray[3] = jumps
PtrArray[4] = over
PtrArray[5] = the
PtrArray[6] = lazy
PtrArray[7] = dog
PtrPtr + 0 = The
PtrPtr + 1 = quick
PtrPtr + 2 = fox
PtrPtr + 3 = jumps
PtrPtr + 4 = over
PtrPtr + 5 = the
PtrPtr + 6 = lazy
PtrPtr + 7 = dog
```

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

char **PtrPtr = PtrArray;

int i = 0;

while (PtrArray[i] != "")
{
   printf("*PtrArray[%d] = %c\n", i++, *PtrArray[i]);
}

i = 0;
while (*(PtrPtr + i) != "")
{
   printf("**(PtrPtr + %d) = %c\n", i++, **(PtrPtr + i));
}
```

```
%s    PtrArray[i]
```

```
%s    *(PtrPtr + i)
```

```
*PtrArray[0] = T
*PtrArray[1] = q
*PtrArray[2] = f
*PtrArray[3] = j
*PtrArray[4] = o
*PtrArray[5] = t
*PtrArray[6] = l
*PtrArray[7] = d
**(PtrPtr + 0) = T
**(PtrPtr + 1) = q
**(PtrPtr + 2) = f
**(PtrPtr + 3) = j
**(PtrPtr + 4) = o
**(PtrPtr + 5) = t
**(PtrPtr + 6) = l
**(PtrPtr + 7) = d
```

dichar5Demo.c

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};
char **PtrPtr = PtrArray;


while (PtrArray[i] != "")
{
   for (j = 0; j < strlen(PtrArray[i]); j++)
   {
      printf("PtrArray[%d][%d] = %c\n",
             i, j, PtrArray[i][j]);
   }
   i++;
}
```

```
PtrArray[0][0] = T   PtrArray[4][0] = o
PtrArray[0][1] = h   PtrArray[4][1] = v
PtrArray[0][2] = e   PtrArray[4][2] = e
PtrArray[1][0] = q   PtrArray[4][3] = r
PtrArray[1][1] = u   PtrArray[5][0] = t
PtrArray[1][2] = i   PtrArray[5][1] = h
PtrArray[1][3] = c   PtrArray[5][2] = e
PtrArray[1][4] = k   PtrArray[6][0] = l
PtrArray[2][0] = f   PtrArray[6][1] = a
PtrArray[2][1] = o   PtrArray[6][2] = z
PtrArray[2][2] = x   PtrArray[6][3] = y
PtrArray[3][0] = j   PtrArray[7][0] = d
PtrArray[3][1] = u   PtrArray[7][1] = o
PtrArray[3][2] = m   PtrArray[7][2] = g
PtrArray[3][3] = p
PtrArray[3][4] = s
```

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};
char **PtrPtr = PtrArray;

i = 0;
while (*(PtrPtr + i) != "")
{
   for (j = 0; j < strlen(*(PtrPtr + i)); j++)
   {
      printf("*(*(PtrPtr + %d) + %d) = %c\n",
             i, j, *(*(PtrPtr + i)+j));
   }
   i++;
}
```

```
*(*(PtrPtr + 4) + 0) = o
*(*(PtrPtr + 4) + 1) = v
*(*(PtrPtr + 4) + 2) = e
*(*(PtrPtr + 4) + 3) = r
*(*(PtrPtr + 5) + 0) = t
*(*(PtrPtr + 5) + 1) = h
*(*(PtrPtr + 5) + 2) = e
*(*(PtrPtr + 6) + 0) = l
*(*(PtrPtr + 6) + 1) = a
*(*(PtrPtr + 6) + 2) = z
*(*(PtrPtr + 6) + 3) = y
*(*(PtrPtr + 7) + 0) = d
*(*(PtrPtr + 7) + 1) = o
*(*(PtrPtr + 7) + 2) = g
```

Double Indirection

C does not put a limit on the number of levels of indirection.

```
long ***ThisIsRidiculous;
```

Pointer to a pointer to a pointer to a `long`

```
char *****ThisIsMoreRidiculous;
```

Pointer to a pointer to a pointer to a pointer to a pointer to a `char`

Double Indirection

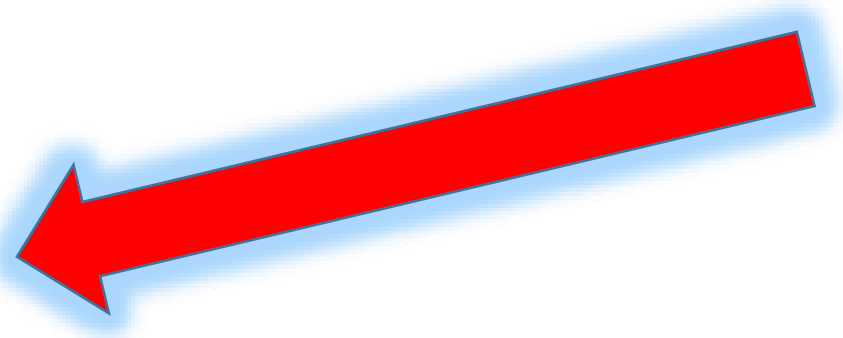
```
#include <stdio.h>

int main(void)
{
  int VarA = 10;
  int *VarAPtr = &VarA;
  int **Ptr2VarAPtr = &VarAPtr;

  printf("VarA = %d\n", VarA);
  printf("*VarAPtr = %d\n", *VarAPtr);
  printf("**Ptr2VarAPtr = %d\n", **Ptr2VarAPtr);

  return 0;
}
```

```
(gdb) p VarA
$1 = 10
(gdb) p &VarA
$2 = (int *) 0x7fffffffe7a4
(gdb) p VarAPtr
$3 = (int *) 0x7fffffffe7a4
(gdb) p &VarAPtr
$4 = (int **) 0x7fffffffe798
(gdb) p Ptr2VarAPtr
$5 = (int **) 0x7fffffffe798
```

```
VarA = 10
*VarAPtr = 10
**Ptr2VarAPtr = 10
```

The `do-while` Statement

```
do
{
      statement
}
while (expression);
```



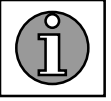
Looping structure

- `statement` will always execute at least once
- `expression` will be evaluated after `statement` executes
- loop repeated if `expression` is nonzero
- a semicolon is required after `expression`

The `do-while` Statement

```
int AskAgain = 1;

while (AskAgain)
{
   printf("Please enter a decimal "
          "number between 0 and 255 ");
   scanf("%d", &DecNum);

   if (DecNum >= 0 && DecNum <= 255)
      AskAgain = 0;
   else
   {
      AskAgain = 1;
      printf("\nYou entered a number "
             "not between 0 and 255\n\n");
   }
}
```

```
int AskAgain;

do
{
   printf("Please enter a decimal "
          "number between 0 and 255 ");
   scanf("%d", &DecNum);

   if (DecNum >= 0 && DecNum <= 255)
      AskAgain = 0;
   else
   {
      AskAgain = 1;
      printf("\nYou entered a number "
             "not between 0 and 255\n\n");
   }
}
while (AskAgain);
```



The `switch` Statement

multiway decision statement

```
switch (expression)
{
      case c1:
            statement;
            break;
      case c2:
            statement;
            break;
      .
      .
      .
      default:
            statement
}
```

`expression`

`expression` must have one of the integer types.

`case const_expr: statement`

`const_expr` must be a constant expression and must have one of the integer types.

There can be multiple `case` labeled statements but each `const_expr` must have distinct value.

`default: statement`

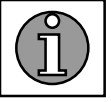
optional

Executed if none of the `case` statements are executed.

The `switch` Statement

```
if (MenuChoice == 1)
{
   printf("strlen() example\n");
}
else if (MenuChoice == 2)
{
   printf("strcpy() example\n");
}
else if (MenuChoice == 3)
{
   printf("strcat() example\n");
}
else
   printf("Invalid choice\n");
```

```
switch (MenuChoice)
{
   case 1:
      printf("strlen() example\n");
      break;
   case 2:
      printf("strcpy() example\n");
      break;
   case 3:
      printf("strcat() example\n");
      break;
   default:
      printf("Invalid menu choice\n");

}
```



The `switch` Statement

```
switch (MenuChoice)
{
   case 1:
      printf("strlen() example\n");
      break;
   case 2:
      printf("strcpy() example\n");
      break;
   case 3:
      printf("strcat() example\n");
      break;
   default:
      printf("Invalid menu choice\n");

}
```

```
switch (MenuChoice)
{
   case 1:
      printf("strlen() example\n");
   case 2:
      printf("strcpy() example\n");
   case 3:
      printf("strcat() example\n");
   default:
      printf("Invalid menu choice\n");
}
```

casebreakDemo.c

Altering the Flow of Control

`continue` and `break`

used to alter the flow of control

- `while` loop
- `for` loop
- `do-while` loop

`break` can also be used with `switch`

The `continue` Statement

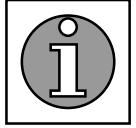
```
continue;
```

- used inside a loop
- when encountered, it causes control to pass to the point after the last statement in the loop body instead of executing the next statement

The `break` Statement

```
break;
```

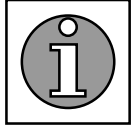
- used inside a loop or `switch`
- when encountered, it causes the loop to terminate and control to pass to the point immediately after the loop



```
while (SecretNumber)
{
  printf("Enter a secret number between 1 and 10 : ");
  scanf("%d", &SecretNumber);

  if (SecretNumber < 1 || SecretNumber > 10)
  {
    printf("\n\nYou did not enter a number "
           "between 1 and 10.  Try again.\n\n\n");
    continue;
  }
  else
  {
    break;
  }
}

printf("\n\nPlayer 2\n\n");
```

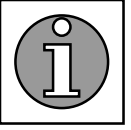
```
Enter a secret number between 1 and 10 : 11

You did not enter a number between 1 and 10.  Try again.

Enter a secret number between 1 and 10 : 2

Player 2
```

whilebreakDemo.c



```
do
{
  printf("Pick a number between 1 and 10 ");

  scanf("%d", &GuessedNumber);

  if (GuessedNumber < 1 || GuessedNumber > 10)
  {
    printf("\nYou did not enter a number between 1 and 10.  Try again.\n");
    continue;
  }

  if (GuessedNumber == SecretNumber)
  {
    printf("\n\nYou guessed the secret number!\n\n");
    break;
  }
  else
  {
    printf("\n\nThe number you entered is not the secret number.\n\n");
  }
}
while (GuessedNumber);

printf("\n\nBye!  Thanks for playing.\n");
```

whilebreakDemo.c



```
printf("Do you want to print even or odd numbers (E/O) ");
scanf("%s", &EvenOdd);
printf("\n\nEnter start of range ");
scanf("%d", &Start);
printf("\n\nEnter end of range ");
scanf("%d", &End);
```

```
if (EvenOdd == 'O')
{
   for (i = Start; i <= 100; i++)
   {
      if (i > End)
         break;

      if (i & 1)
         printf("i = %d\n", i);
      else
         continue;
   }
}
```

```
else  /* assume E */
{
   for (i = Start; i <= 100; i++)
   {
      if (i > End)
         break;

      if (!(i & 1))
         printf("i = %d\n", i);
      else
         continue;
   }
}
```

forcontinue1Demo.c

```
while (!DISCREAD (ordfd,(short *)&gstOrdhdr,sizeof (gstOrdhdr)))
{
   /* Do not process invoices that are in process or marked as */
   /* duplicates (status 'D')                                  */
   if (gstOrdhdr.xinvoice == 'A' ||
       gstOrdhdr.xinvoice == 'B' ||
       gstOrdhdr.xinvoice == 'C' ||
       gstOrdhdr.xinvoice == 'D')
   {
       continue;
   }

   /* Order will be written to the transmit file so now add it to */
   /* the =srt_tbl in order to detect duplicates.  If it is a     */
   /* duplicate, then skip this order.                            */
   if (insert_dup_check())
   {
      continue;
   }
```

```
while (!DISCREAD (ordfd,(short *)&gstOrdhdr,sizeof (gstOrdhdr)))
{

   if (time_is_up())    /* it's time to finish up */
   {
      x = NBR_STATS;
      break;
   }


   if (nDone)           /* no room for order so quit*/
   {
      nDone = 0;
      break;
   }


}
```

```
/* Based on value of x, set status. */
char get_stat (short x)
{
   switch  (x)
   {
      case 0:
         status = '5';
         break;
      case 1:
         status = 'L';
         break;
      case 2:
         status = 'T';
         break;
      case 3:
         status = '6';
         break;
      default:
         status = 'T';
         break;
   }
   return status;
}
```

The `return` Statement

```
return;
return expression;
```

- `return` statement is used in `main()` to terminate the program
- `return` statement can also be used to terminate execution of a function
- when `return` is executed, it causes control to pass from the function back to the position where it was called
- used to provide a point of exit from a function other than at the end of the function
- allows a function to return a value

```
/* Return TRUE if it's time to stop running.                          */

short time_is_up(void)
{
    TIME (time_tbl);

    if( !cutoff_hour )
       return FALSE;      /* Only one run */

    if( time_tbl[3] > cutoff_hour )
       return TRUE;

    if( (time_tbl[3] == cutoff_hour) && (time_tbl[4] > cutoff_minute))
       return TRUE;

    return FALSE;
}
```

```
/*   Don't go to delay if there is not enough time left to make another run  *
     after returning from delay.                                              */
short no_time_left(void)
{
    short future_hour = 0;
    short future_minute = 0;

    if( !cutoff_hour )  /* only one run */
       return TRUE;

    future_minute = (short)(( time_tbl[4] + delay_time) % 60);
    future_hour  = (short)(time_tbl[3] + (time_tbl[4] + delay_time)/ 60);
    if( future_hour > cutoff_hour)
       return TRUE;

    if( (future_hour == cutoff_hour) && (future_minute > cutoff_minute) )
       return TRUE;

    return FALSE;
}
```

The `exit()` Library Function

The `exit()` function takes a single parameter of type `int`.

- when executed, `exit()` causes the program to terminate
- control is returned to the operating system
- `<stdlib.h>` must be `include`d in your program

```
exit(0);
exit(1);
exit(255);
```

```
if ( nError = STARTOPENS((short *)"COMMENT ", &cmtfd,
                                                R_O+SHARED, 1, dataset) )
{
   sprintf(gszMsg, "opn_files() Error %d opening COMMENT file \n", nError);
   fnProcessError();
   SENDEMAIL((short *)&gstErrorEmail);
   msgabend (gszMsg, (short)nError, 0);
}

if ( nError = STARTOPENS((short *)"CUSTNAME", &cstfd,
                                                R_O+SHARED, 1, dataset) )
{
   sprintf(gszMsg, "opn_files() Error %d opening CUSTNAME file \n", nError);
   fnProcessError();
   SENDEMAIL((short *)&gstErrorEmail);
   msgabend (gszMsg, (short)nError, 0);
}
```

Library function with `exit(-1)` as its last statement

Library function with `exit(-1)` as its last statement

`return 0` vs `exit(0)` in `main()`

In `main()`,

what is the difference between using `return 0` and `exit(0)`?

```
int main(void)
{
   return 0;
}
```

```
int main(void)
{
   exit(0);
}
```

`return` is a statement and `exit()` is a function. From a standard C perspective, there is no difference. There are, however, a few unusual circumstances where using `exit()` instead of `return` at the end of `main()` will cause undefined behavior; therefore, it is good practice to use `return` rather than `exit()` in `main()`.

return 0 vs exit(0) in main()

```
int main(void)
{
  int Code;
  char RetExit;

  printf("Return or Exit? (R/E)? ");
  scanf("%s", &RetExit);
  RetExit = toupper(RetExit);

  printf("Enter a code ");
  scanf("%d", &Code);

  if (RetExit == 'E')
  {
    exit(Code);
  }
  else
  {
    return Code;
  }
}
```

Code is displayed in octal in debug

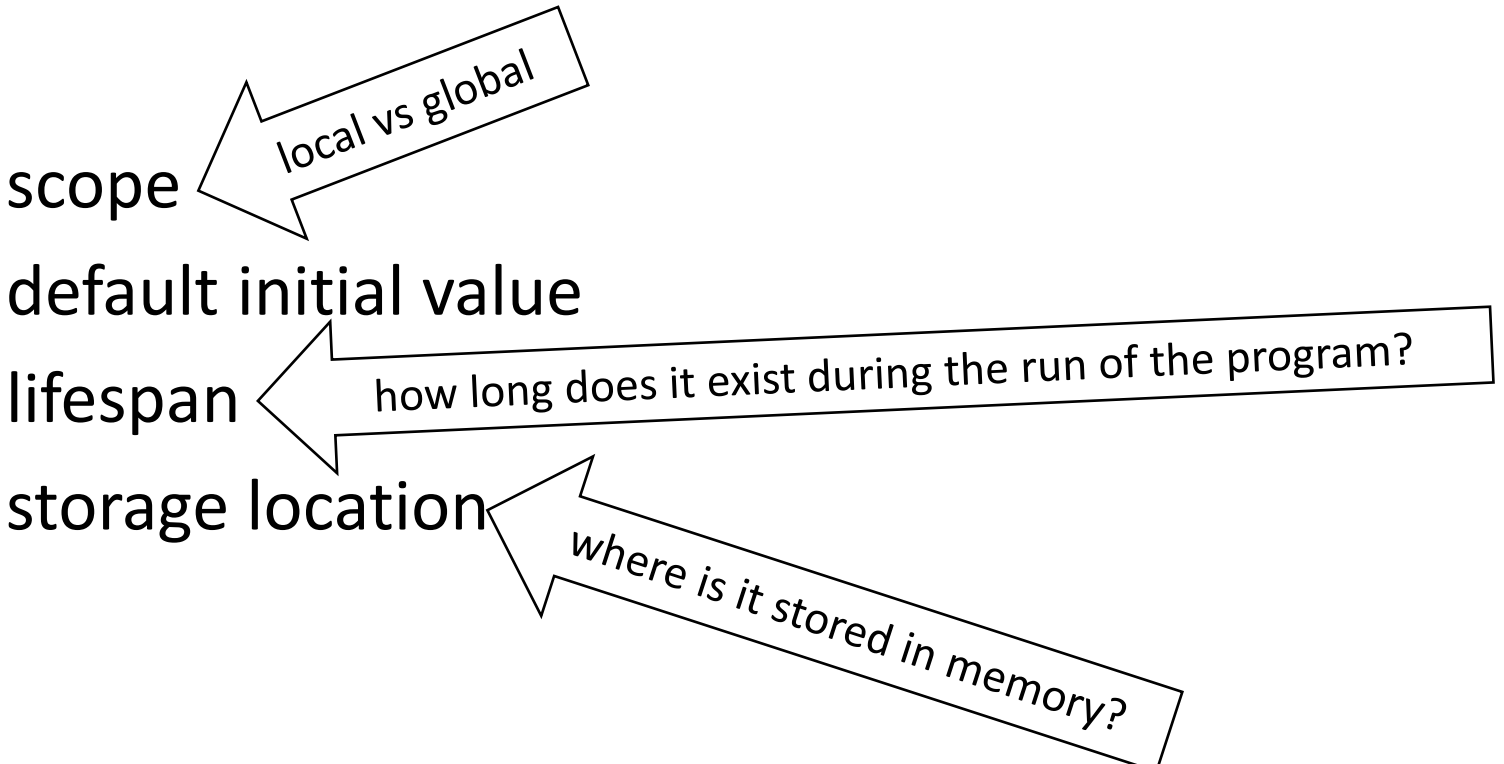
Running in debug…

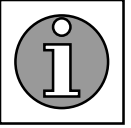
```
Return or Exit? (R/E)? e
Enter a code 8

Program exited with code
010.
```

Automatic versus Static Variables

Storage Class

Storage classes are used to describe the features of a variable or function.





Automatic versus Static Variables

Automatic Variables

```
auto int IntVar;
   int IntVar;
```

- default storage class
- automatic variables are created each time its function is called and destroyed when the execution of its function terminates
- when an automatic variable is created without being initialized, it is not given an initial value – may contain garbage.
- when an automatic variable is created with an initialization, the initialization is done each time the variable is created,

Automatic versus Static Variables

`auto`

scope

- inside function - local

default initial value

- contain garbage until explicitly initialized

lifespan

- created when function called and destroyed when function exits

storage location

- stack

Automatic versus Static Variables

Static Variables

```
static int IntVar;
```

- static variables exist the whole time the program is executing
- memory space is allocated when program starts and is deallocated when program ends
- static variables are given the default initial value of 0
- if an initializer is used, then the variable is initialized once at the beginning of the program

Automatic versus Static Variables

`static`

scope

inside function - local

default initial value

0

lifespan

created when program starts and ends when program ends

storage location

data segment

Automatic versus Static Variables

`auto`

scope
- inside function - local

default initial value
- contain garbage until explicitly initialized

lifespan
- created when function called and destroyed when function exits

storage location
- stack

`static`

scope
- inside function - local

default initial value
- 0

lifespan
- created when program starts and ends when program ends

storage location
- data segment

```
void CallMyFunction(void)
{
   static int staticVar1;
   static int staticVar2 = 100;

   int autoVar1;
   int autoVar2 = 100;

   printf("Value of staticVar1 = %d\n", staticVar1++);
   printf("Value of staticVar2 = %d\n", staticVar2++);
   printf("Value of autoVar1   = %d\n", autoVar1++);
   printf("Value of autoVar2   = %d\n", autoVar2++);
}
```

`staticVar1` is initialized to 0 for us and `staticVar2` is set to 100 once and both retain their values between function calls and are not reset.

`autoVar1` is system trash and `autoVar2` is set to 100 every time the function is called.

```
int i;

for (i = 0; i < 3; i++)
{
   CallMyFunction();
}
```

```
Value of staticVar1 = 0
Value of staticVar2 = 100
Value of autoVar1   = -920532032
Value of autoVar2   = 100

Value of staticVar1 = 1
Value of staticVar2 = 101
Value of autoVar1   = -920532031
Value of autoVar2   = 100
```

System trash

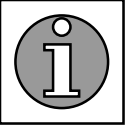
System trash incremented

```
Value of staticVar1 = 2
Value of staticVar2 = 102
Value of autoVar1   = -920532030
Value of autoVar2   = 100

Address of staticVar1 = 0x600a54
Address of staticVar2 = 0x600a44
Address of autoVar1   = 0x7fff26ebc28c
Address of autoVar2   = 0x7fff26ebc288
```

System trash incremented

data segment

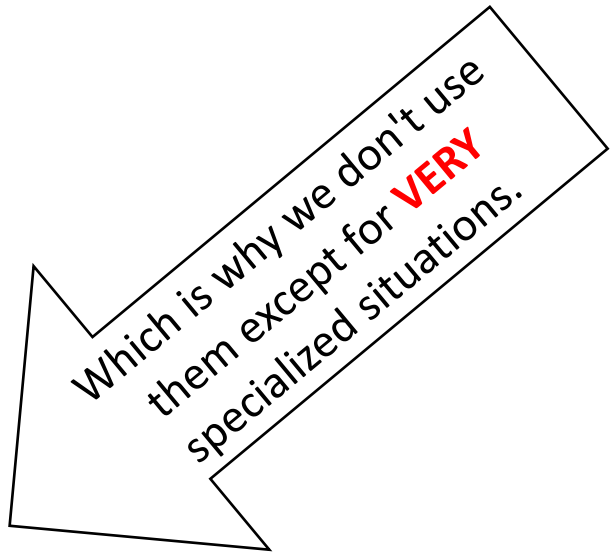
stack



Automatic versus Static Variables

Register Variables

```
register int i;
```



- programmer requests that a variable be placed in a register
- usually indicates that a variable will be used frequently
  - improve speed and performance – indicies and loop counters
- no guarantee that the variable will be placed in the register
- very limited in availability and size
- illegal to use the address operator & with the name of a register variable

```
printf("%p", &i);
error: address of register variable ‘i’ requested
```

```
void print_it(void)
{
   register int i;
   int x;

   i = 12345;
   x = 98765;
   printf("i = %d", i);
}
```

calling `print_it()` from `main()`

```
print_it () at registerDemo.c:9
9                 i = 12345;
(gdb)
10                x = 98765;
(gdb) p &i
Address requested for identifier
"i" which is in register $rsi
(gdb)
```

register

```
void print_it(void)
{
   int i;
   int x;

   i = 12345;
   x = 98765;
   printf("i = %d", i);
}
```

calling `print_it()` from `main()`

```
print_it () at registerDemo.c:10
10                i = 12345;
(gdb)
11                x = 98765;
(gdb) p &i
$3 = (int *) 0x7fffffffe788
```

stack memory

Global versus Local Variables

Local Variables

- only known inside the function block or compound statement block in which they were defined

- can be legally referenced at any point from its declaration to the closing braces for that block or function

Global versus Local Variables

Global Variables

- variable that can be referenced by more than one function
- defined outside function or compound statement blocks
- global variables are defined before all functions in a source code file
- global variables can be referenced by all functions in that file
- global variables are in existence during the full execution time of the program

```
int Pongo;
int Perdita;

void Dog(int Puppy)
{
  int Patch;
  int Lucky;
  Pongo = Perdita;
}

void Spots(int Puppy)
{
  int Rolly;
  int Penny;
  Pongo = Perdita;
}

int main(void)
{
  int Freckles;
  int Pepper;

  Dog(Freckles);
  Spots(Pepper);

  Pongo = Perdita;
  Freckles = Lucky;
  Pepper = Penny;

  return 0;
}
```

Global

```
lvsgDemo.c:30: error: 'Lucky' undeclared
lvsgDemo.c:31: error: 'Penny' undeclared
```

lvsgDemo.c

Global versus Local Variables

CAUTION

Global variables should be used with discretion.

All functions can access global variables and change their values.

The effect of a function changing a variable from outside its scope is called a

side effect

Every change to a global variables is a side effect.

sideeffectDemo.c

```
int X = 0;  /* Global version of X */

void SetXFunction(void)
{
  X = 987;
}

void PrintXFunction(void)
{
  printf("PrintXFunction()\tX = %d\n", X);
}

void NewSetXFunction(int *NewX)
{
  X = 567;
}
```

```
int main(void)
{
  /* Local version of X */
  int X = 123;

  printf("main()       X = %d\n", X);
  SetXFunction();
  printf("main()       X = %d\n", X);
  PrintXFunction();
  NewSetXFunction(&X);
  PrintXFunction();
  printf("main()       X = %d\n", X);

  return 0;
}
```

```
main()                    X = 123
main()                    X = 123
PrintXFunction()          X = 987
PrintXFunction()          X = 567
main()                    X = 123
```

Passing Parameters to Functions

Two basic methods of passing parameters to functions

- *pass by value*
  - parameter is called *value parameter*
  - a copy is made of the current value of the parameter
  - operations in the function are done on the copy – the original does not change

- *pass by reference*
  - parameter is called a *variable parameter*
  - the address of the parameter's storage location is known in the function
  - operations in the function are done directly on the parameter

Passing Parameters to Functions

In C

all parameters are passed by value

the ability to pass by reference does not exist

Pass by reference can be simulated

- pass the address of the variable
- address cannot be modified
- contents of address can be modified

```
int main(void)
{
  int MyMainNum = 0;

  printf("Before PassByValue call\tMyMainNum = %d\n", MyMainNum);
  PassByValue(MyMainNum);
  printf("After  PassByValue call\tMyMainNum = %d\n", MyMainNum);

  printf("Before PassByRef   call\tMyMainNum = %d\n", MyMainNum);
  PassByRef(&MyMainNum);
  printf("After  PassByRef   call\tMyMainNum = %d\n", MyMainNum);

  return 0;
}
```

simrefDemo.c

A copy is passed

```
int PassByValue(int MyNum)
{
      MyNum += 100;
      printf("Inside PassByValue\tMyNum     = %d\n", MyNum);
}
```

The address of the actual variable is passed

```
int PassByRef(int *MyNumPtr)
{
      *MyNumPtr += 100;
      printf("Inside PassByRef\tMyRefNum  = %d\n", *MyNumPtr);
}
```

simrefDemo.c

```
int MyMainNum = 0;

printf("Before PassByValue call\tMyMainNum = %d\n", MyMainNum);
PassByValue(MyMainNum);
printf("After  PassByValue call\tMyMainNum = %d\n", MyMainNum);

int PassByValue(int MyNum)
{
     MyNum += 100;
     printf("Inside PassByValue\tMyNum     = %d\n", MyNum);
}
```

```
Before PassByValue call MyMainNum = 0
Inside PassByValue      MyNum     = 100
After  PassByValue call MyMainNum = 0
```

simrefDemo.c

```
int MyMainNum = 0;

printf("Before PassByRef   call\tMyMainNum = %d\n", MyMainNum);
PassByRef(&MyMainNum);
printf("After  PassByRef   call\tMyMainNum = %d\n", MyMainNum);

int PassByRef(int *MyNumPtr)
{
     *MyNumPtr += 100;
     printf("Inside PassByRef\tMyNumPtr  = %d\n", *MyNumPtr);
}

Before PassByRef   call MyMainNum = 0
Inside PassByRef        MyRefNum  = 100
After  PassByRef   call MyMainNum = 100
```

simrefDemo.c

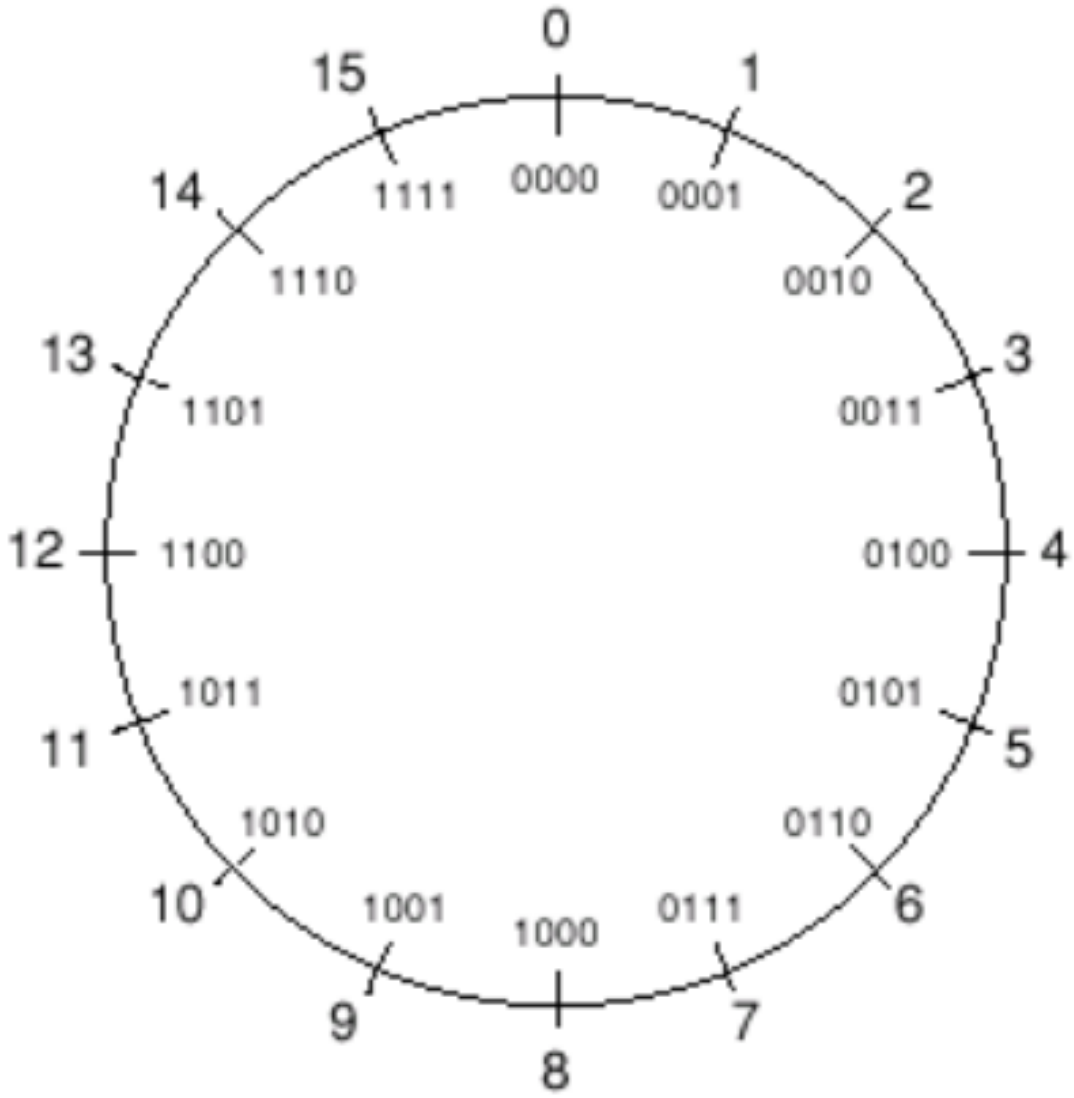
frenchdm@omega:~/StudentCode

```
[frenchdm@omega StudentCode]$ g
```

4-bit Adding in Binary

$$\begin{array}{r} 0110_2 \\ +\ 1001_2 \\ \hline 1111_2 \end{array} \qquad \begin{array}{r} 0101_2 \\ +\ 1101_2 \\ \hline 0010_2 \end{array}$$

$$\begin{array}{r} 0101_2 \\ +\ 1001_2 \\ \hline 1110_2 \end{array} \qquad \begin{array}{r} 1101_2 \\ +\ 1101_2 \\ \hline 1010_2 \end{array}$$



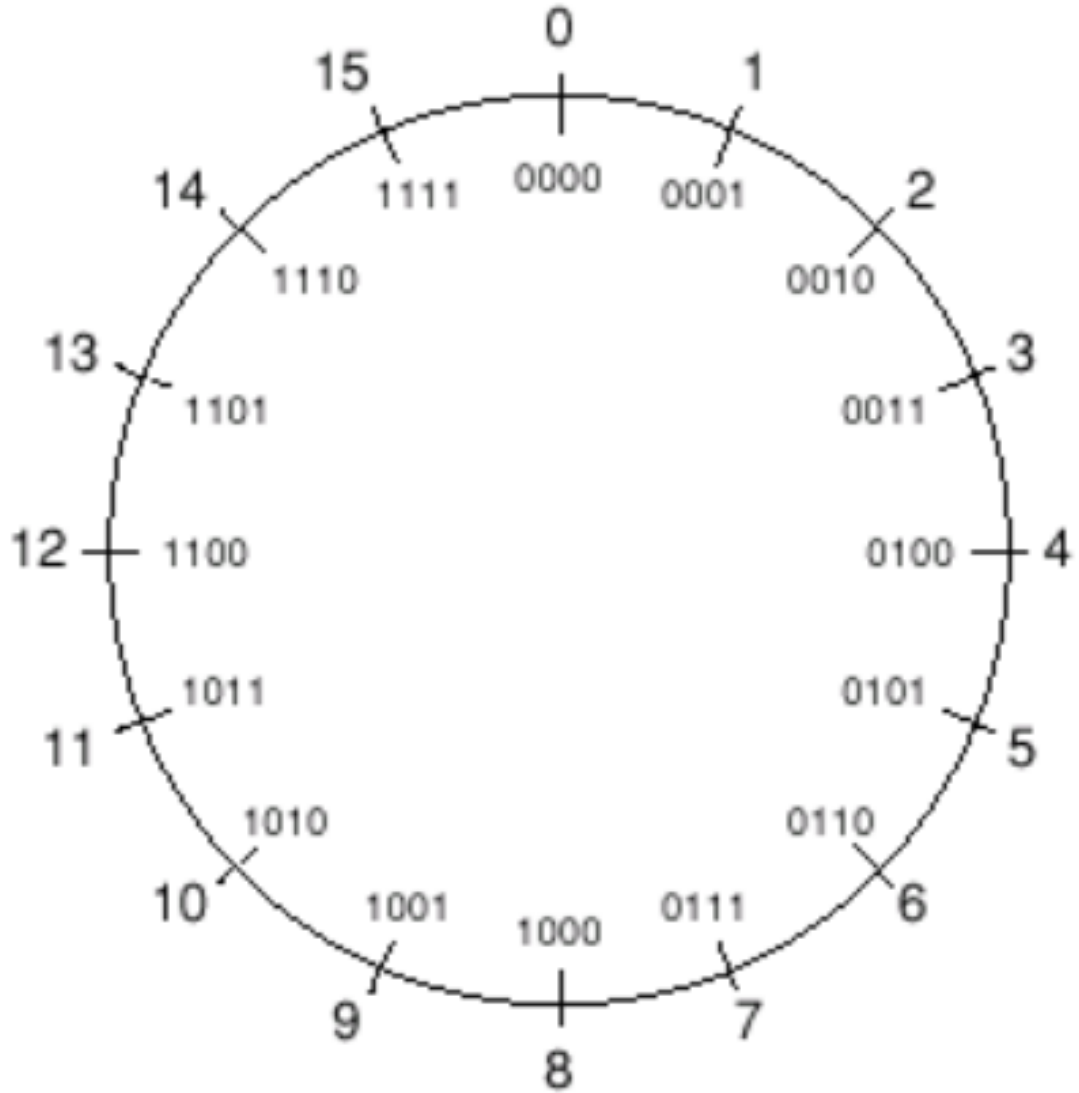
Two's Complement

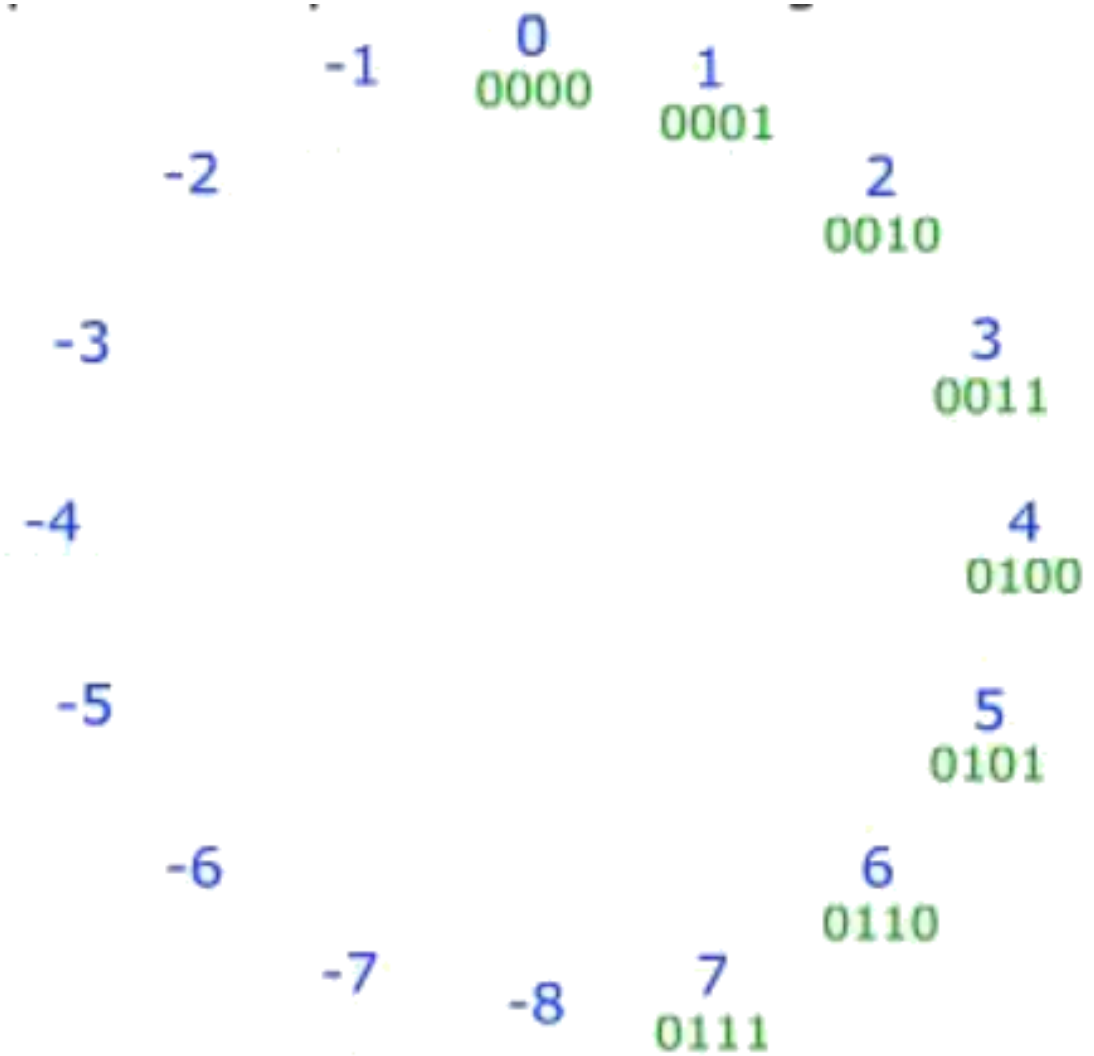
How can we represent negative numbers in binary?

There are several ways

- using a sign bit
- one's complement
- two's complement

Two's complement is the most commonly used technique because it's relatively easy to implement in hardware


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111


0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
-1
-2
-3
-4
-5
-6
-7
-8

Two's Complement

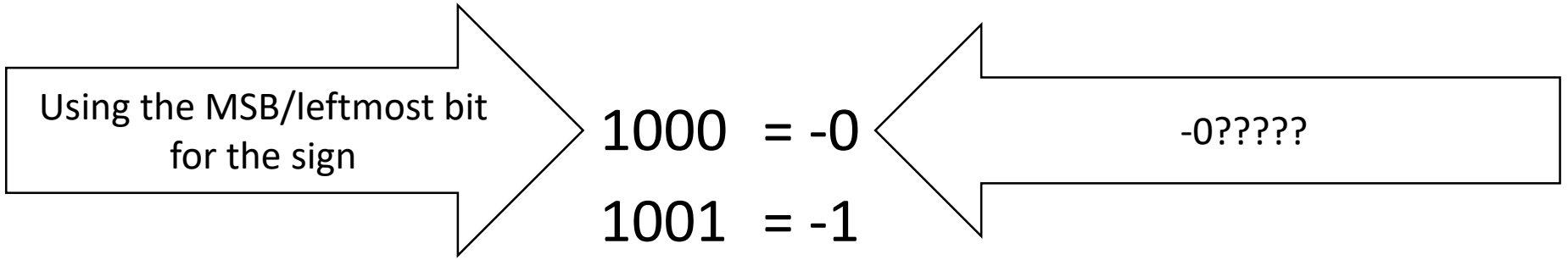
Positive 5 in binary

$0101_2 = \quad 2^0+2^2 = \quad 1+4 = 5$

So if we use the first bit to determine the sign, then negative 5 in binary would be

$1101_2 = -5$

Two's Complement



0000	= 0
0001	= 1
0010	= 2
0011	= 3
0100	= 4
0101	= 5
0110	= 6
0111	= 7

1000	= -0
1001	= -1
1010	= -2
1011	= -3
1100	= -4
1101	= -5
1110	= -6
1111	= -7

Two's Complement

0000 = 0	1000 = -0
0001 = 1	1001 = -1
0010 = 2	1010 = -2
0011 = 3	1011 = -3
0100 = 4	1100 = -4
0101 = 5	1101 = -5
0110 = 6	1110 = -6
0111 = 7	1111 = -7

Using MSB to hold the sign

```
    5          0101₂
+ (-5)       + 1101₂
------       ------
    0          0010₂
```

2_{10} 5+(-5) = 2????

```
    7          0111₂
+ (-3)       + 1011₂
------       ------
    4          0010₂
```

2_{10} 7+(-3) = 2????

```
    6          0110₂
+ (-2)       + 1010₂
------       ------
    4          0000₂
```

0_{10} 6+(-2) = 0????

Two's Complement

0000	= 0	1111	= -0
0001	= 1	1110	= -1
0010	= 2	1101	= -2
0011	= 3	1100	= -3
0100	= 4	1011	= -4
0101	= 5	1010	= -5
0110	= 6	1001	= -6
0111	= 7	1000	= -7

one's complement
invert all bits

```
    5          0101₂
+ (-5)       + 1010₂
------       ------
    0          1111₂
```

all off by 1 place

-0_{10} 5+(-5)=-0????

```
    7          0111₂
+ (-3)       + 1100₂
------       ------
    4          0011₂
```

3_{10} 7+(-3) = 3????

```
    6          0110₂
+ (-2)       + 1101₂
------       ------
    4          0011₂
```

3_{10} 6+(-2) = 3????

Two's Complement

So one's complement gets us close to representing a negative binary number

using the MSB for the sign

only off by 1

uses a -0 and +0

1111 = -0

1110 = -1

1101 = -2

1100 = -3

1011 = -4

1010 = -5

1001 = -6

1000 = -7

Two's Complement

One's complement

0000	= 0	1111	= -0
0001	= 1	1110	= -1
0010	= 2	1101	= -2
0011	= 3	1100	= -3
0100	= 4	1011	= -4
0101	= 5	1010	= -5
0110	= 6	1001	= -6
0111	= 7	1000	= -7

Two's Complement

0000	= 0	1111	= -1
0001	= 1	1110	= -2
0010	= 2	1101	= -3
0011	= 3	1100	= -4
0100	= 4	1011	= -5
0101	= 5	1010	= -6
0110	= 6	1001	= -7
0111	= 7	1000	= -8

Took out the -0

Added 1 to one's complement

Two's Complement

0000	= 0	1111	= -1
0001	= 1	1110	= -2
0010	= 2	1101	= -3
0011	= 3	1100	= -4
0100	= 4	1011	= -5
0101	= 5	1010	= -6
0110	= 6	1001	= -7
0111	= 7	1000	= -8

two's complement

```
    5          0101₂
+ (-5)       + 1011₂
------       ------
    0          0000₂          0₁₀
```

```
    7          0111₂
+ (-3)       + 1101₂
------       ------
    4          0100₂          4₁₀
```

```
    6          0110₂
+ (-2)       + 1110₂
------       ------
    4          0100₂          4₁₀
```

Two's Complement

How to calculate the two's complement of a number

0000 = 0	1111 = -1
0001 = 1	1110 = -2
0010 = 2	1101 = -3
0011 = 3	1100 = -4
0100 = 4	1011 = -5
0101 = 5	1010 = -6
0110 = 6	1001 = -7
0111 = 7	1000 = -8

Take

5_{10} which is 0101_2

invert it 1010_2

and add 1

$$\begin{array}{r} 1010_2 \\ +\quad 1 \\ \hline 1011_2 \end{array}$$

-5_{10}

Take

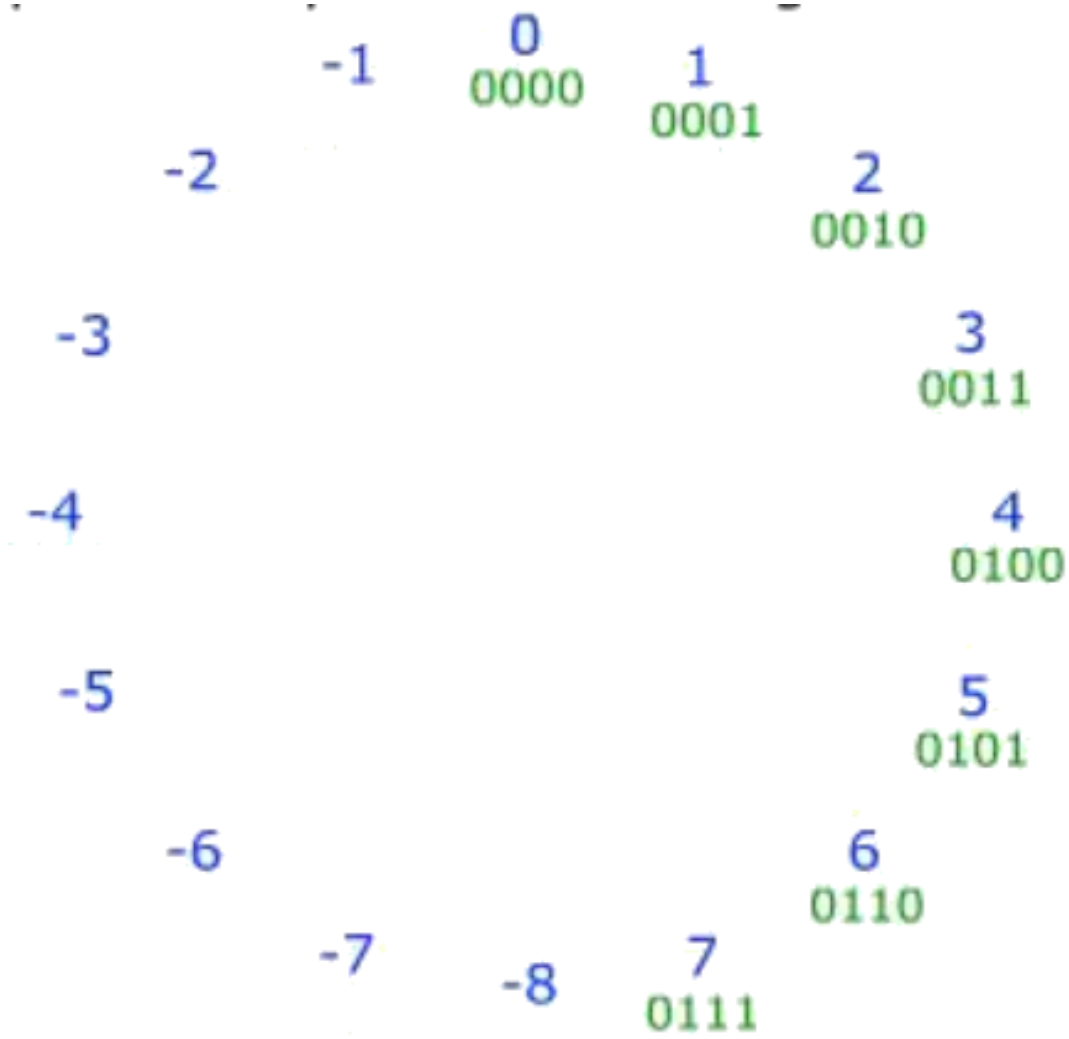
3_{10} which is 0011_2

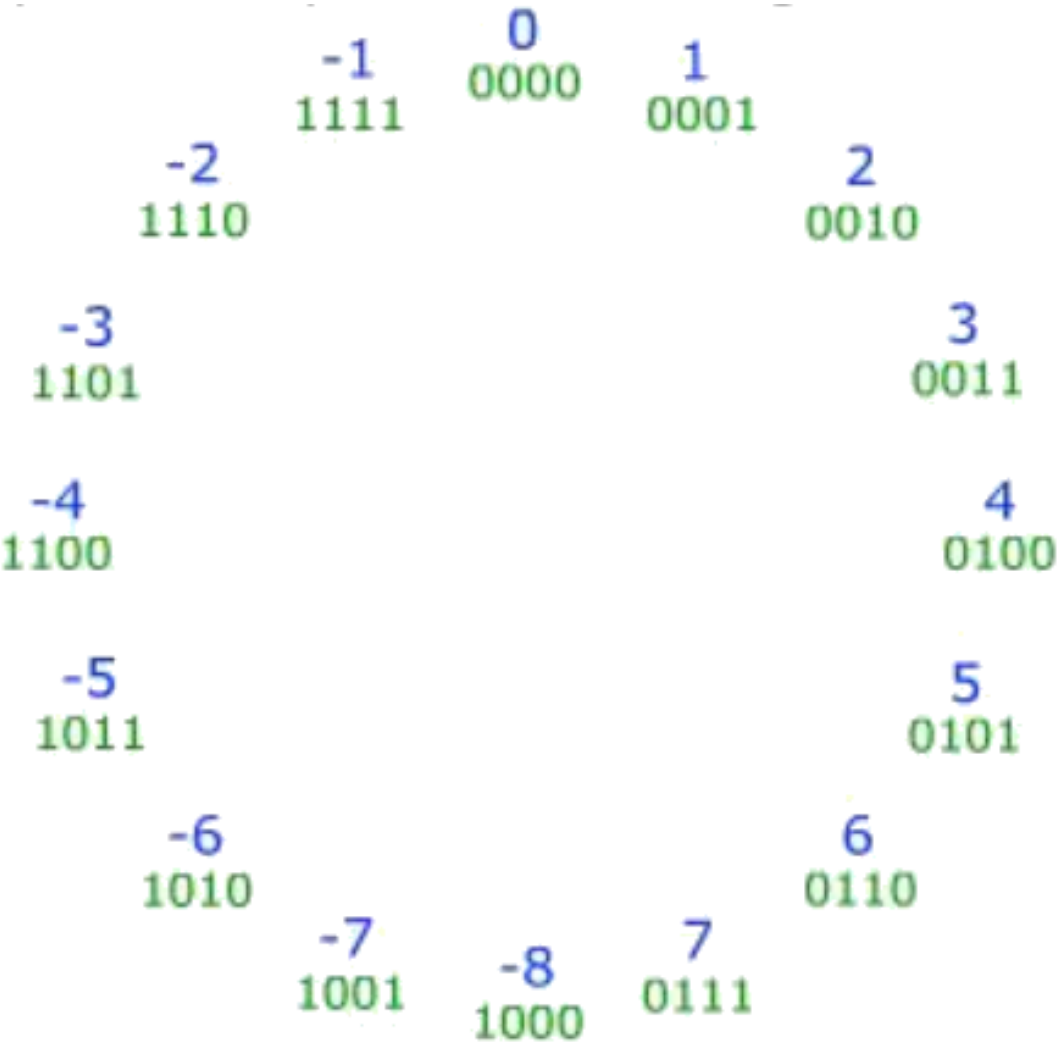
invert it 1100_2

and add 1

$$\begin{array}{r} 1100_2 \\ +\quad 1 \\ \hline 1101_2 \end{array}$$

-3_{10}


0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
-1
-2
-3
-4
-5
-6
-7
-8


0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
-1
1111
-2
1110
-3
1101
-4
1100
-5
1011
-6
1010
-7
1001
-8
1000