Exercise 3 / Ch.7

Given the following array, write code to initialize all the elements to 0:

```c
int ed[100];
```

Hint: It can be done two different ways!
Exercise 3 / Ch.8

Given the following array, write code to initialize all the elements to 0:

```c
int ed[100];
```

• Use an initializer list.
• Use a `for` loop.
Write code to initialize all the elements of the array to zero:

```c
int ed[100] = {0};
int ed[100], i;
for (i=0; i<100; i++)
    ed[i] = 0;
```
Given the following array, write code to initialize all the elements to 42:

```cpp
int ed[100];

• Use an initializer list (?)
• Use a for loop.
```
Exercise 4 / Ch.8

Given the following array, write code to initialize all the elements to 0:

```c
int stuff[2][3];
```
Write code to initialize all the elements to 0:

```c
int stuff[2][3] = {{0,0,0},{0,0,0}};

int stuff[2][3] = {0};

int stuff[2][3],i,j;
for (i=0;i<2;i++)
    for (j=0;j<3;j++)
        stuff[i][j] = 0;
```
Exercise 5 / Ch.8

BUG BUSTER: What is wrong in this code?

```c
int x, y;
int arr[10][3];
for(x=0; x<3; x++)
    for(y=0; y<10; y++)
        arr[x][y] = 42;
```
Two-dimensional arrays reloaded!

Write code to declare and initialize a 10x10 identity matrix

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Two-dimensional arrays reloaded!

Write code to declare and initialize a 10x10 identity matrix

```
int id[10][10];
int i,j;

for (i=0; i<10; i++)
  for (j=0; j<10; j++)
    if(i==j)
      id[i][j] = 1;
    else
      id[i][j] = 0;
```
Two-dimensional arrays reloaded!

Write code to declare and initialize a 10x10 identity matrix

```
int id[10][10] = {0};
int i;

. . . . .
for (i=0; i<10; i++)
    id[i][i] = 1;
```
Which solution is more efficient?

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

```
int id[10][10] = {0};
int i;
...
for (i=0; i<10; i++)
    id[i][i] = 1;
```

```
int id[10][10];
int i,j;
...
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        if(i==j)
            id[i][j] = 1;
        else
            id[i][j] = 0;
```
Lesson 9

Pointers
A pointer is defined as a variable which contains the address of another variable (or the address of dynamically allocated memory – more on this in Ch.10)

Place an asterisk ‘*’ in front of the variable name:

```c
int i, *p;
```

Although not mandatory, it’s good programming style to start or end pointer names with `p`, or even `ptr`. 
The Size of Pointer Variables

The variable of pointer type is used to hold an address of memory.

Its size is implementation-dependent, typically:
- 4 bytes for 32-bit machines
- 8 bytes for 64-bit machines

It is independent of the variable type that it points to
- Pointers to char have the same size as pointers to long long int!
How can pointers have the same size when the variables they point to have different sizes?

short A, *A_ptr;
int B, *B_ptr;
long long D, *D_ptr;

A_ptr = &A;
B_ptr = &B;
D_ptr = &D;

This is how we declare pointers
This is how we initialize pointers

Image source: http://www.codeproject.com/Articles/15680/How-to-write-a-Memory-Scanner-using-C
How can pointers have the same size when the variables they point to have different sizes?

A: The address stored in the pointer is the address of only the first (lowest) Byte of the variable!

```c
short A, *A_ptr;
int B, *B_ptr;
long long D, *D_ptr;

A_ptr = &A;
B_ptr = &B;
D_ptr = &D;
```
Pointer or pointer variable?

It’s really pointer variable, b/c pointers are also stored in memory, just like regular variables!

For simplicity, however, we mostly say just pointer.
Where are pointer variables stored?

They are also stored in memory, just like regular variables!
#include <stdio.h>

int main() {
    char *pc;
    int *pi;
    double *pd;

    printf("sizeof(char *) = %u\n", sizeof(char *));
    printf("sizeof(pc) = %u\n", sizeof(pc));
    printf("sizeof(pi) = %u\n", sizeof(pi));
    printf("sizeof(pd) = %u\n", sizeof(pd));
    return 0;
}

Output when the program is executed in a 32-bit machine:

    sizeof(char *) = 4
    sizeof(pc) = 4
    sizeof(pi) = 4
    sizeof(pd) = 4
Pointer Operators

& (address operator)
• To obtain the address of a variable, add the address operator ‘&’ in front of the variable name.

```c
int i = 10;
int *iPtr;
iPtr = &i; // iPtr gets address of i
```

* (indirection/dereferencing operator)
• To declare a pointer, place it in front of the identifier.
• To obtain the value of a variable, place it in front of the pointer’s name.

```c
*iPtr is the value of i
```
• Pointers can be used for assignment

```c
*iPtr = 5; // changes i to 5
```
Pointer Operators

“The two worlds”

& is the “escalator up”

* is the “escalator down”
Note well!
The asterisk in declaration is not an indirection operator

```c
int i;
int *p = &i; // initializes p, not *p
int *q, *r; // initializes two pointers
```

```c
void main(){
    int a = 42;
    int *pi = &a;
    printf("%p\n%d \n\n", pi, *pi);
}
```
Examples

```c
int i = 10;
int *iPtr;
iPtr = &i;  // iPtr gets address of i
```

![Diagram showing the relationship between iPtr and i with their corresponding memory addresses.]
Exercises

1. Show a declaration for a pointer to a type char variable. Name the pointer char_ptr.

2. If you have a type int variable named cost, how would you declare and initialize a pointer named p_cost that points to that variable?

3. Continuing with exercise 2, how would you assign the value 100 to the variable cost using both direct access and indirect access?

4. Continuing with exercise 3, how would you print the value of the pointer, plus the value being pointed to?
To do for next time:

• Read pp.188-194

• Re-Solve all today’s quizzes in notebook
Two ways to solve a lab problem

Do you remember the comma operator?

```c
void main(void)
{
    int a[50], i;
    for(i=0; i<50; i++)
        a[i] = 3*i+1;
    for(i=0; i<50; i++)
        printf("%4i", a[i]);
    printf("\n\n");
}
```

```c
void main(void)
{
    int a[50], i, num;
    for(i=0, num=1; i<50; i++, num+=3)
        a[i] = num;
    for(i=0; i<50; i++)
        printf("%4i", a[i]);
    printf("\n\n");
}
```
We want to store the squares of the first 20 integers in an array. What’s wrong with this code?
We want to store the squares of the first 20 integers in an array. Do not confuse the index with the content of the array element!

```c
int i, a[20];
for (i = 0; i < 20; i++)
    a[i * i] = i;
```

<table>
<thead>
<tr>
<th>$i^2$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>361</td>
<td></td>
</tr>
</tbody>
</table>
More 2D arrays!

Write code to declare and initialize a 10x10 bi-diagonal matrix

```c
int id[10][10] = {0};
int i,j;
...
...for (i=0; i<10; i++)
  for (j=0; j<10; j++){
    if(i==j) id[i][j] = 1;
    if(i+1==j) id[i][j] = 1;
  }
```
Write code to declare and initialize a 10x10 bi-diagonal matrix of ones

```
int id[10][10] = {0};
int i,j;
. . . . .
for (i=0; i<10; i++)
    for (j=0; j<10; j++){
        if(i==j)
            id[i][j] = 1;
        if(i+1==j)
            id[i][j] = 1;
    }
```

```
| 1 1 0 0 |
| 0 1 1 0 |
| 0 0 1 1 |
| 0 0 0 1 |
```
Write code to declare and initialize a 10x10 **tri-diagonal** matrix of ones

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Write code to declare and initialize a 10x10 **anti-identity** matrix

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```c
#include <stdio.h>

void main() {
    int i, j, *p;       //i and j as int, p as pointer to int
    i = 10;
    j = 0;
    p = &i;            //p is assigned the address of i.
}

Draw a memory map showing all 3 variables!
```
Example memory map for two integers and one pointer

\[
p = &i;
\]

If the address of \( i \) is \texttt{0x00E81E20}, the value of \( p \) is \texttt{0x00E81E20}

The program performs the indirection operations:

\[
*p = 20;
j = *p;
\]
printf("The value of i is %d\n", i);
printf("The value of j is %d\n", j);
printf("The address of i is %p\n", &i);
printf("The value of p is %p\n", p);
printf("The value of *p is %d\n", *p);

Output:
The value of i is 10
The value of j is 0
The address of i is 00E81E20
The value of p is 00E81E20
The value of *p is 10
*p = 20;  //The memory pointed to by p is assigned 20
j = *p;   //j is assigned the value pointed to by p

printf("After indirection operations\n");
printf("The value of i is %d\n", i);
printf("The value of j is %d\n", j);
printf("The address of i is %p\n", &i);
printf("The value of p is %p\n", p);
printf("The value of *p is %d\n", *p);

Output:

Draw a memory map showing all 3 variables!

After indirection operations
The value of i is 20
The value of j is 20
The address of i is 00E81E20
The value of p is 00E81E20
The value of *p is 20
Example memory map for two integers and one pointer

*p = 20;
j = *p;

BEFORE

AFTER
5. Show how to assign the address of a float value called radius to a pointer.

6. Show two ways to assign the value 100 to the third element of data[].

7. Write a function named `sumarrays()` that accepts two arrays as arguments, totals all values in both arrays, and returns the total to the calling program.

8. Use the function created in exercise 7 in a simple program.

9. Write a function named `addarrays()` that accepts two arrays that are the same length and returns the sum of their correspondingly indexed elements.
Array storage and pointers

**Figure 9.7**
Array storage for different array types.

```c
short x[6];
```

```
1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011

```

```c
float expenses[3];
```

```
1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261

```
An array of type `long long int` has 10 elements, and it is stored in memory starting at address 1000. What is the address of the last Byte of the array?
Array storage and pointers

An array of type **long long int** has 10 elements, and it is stored in memory starting at address 1000. What is the address of the last Byte of the array?

A: $1000 + 10 \times 8 - 1 = 1079$
An array of type **long long int** has 10 elements, and it is stored in memory starting at address 1000. What is the address of the **last element** of the array?
Array storage and pointers

An array of type `long long int` has 10 elements, and it is stored in memory starting at address 1000. What is the address of the last element of the array?

A: $1000 + (10 - 1) \times 8 = 1072$
**Uninitialized pointers**

What happens if we declare a pointer, but do not initialize it?

```c
int *pi;
```

Nothing ... until we try to dereference it!

Never dereference uninitialized pointers!
void main()
{
    int *pi;
    printf("%p \n\n", pi);
}

void main()
{
    int *pi;
    printf("%p%n%d \n\n", pi, *pi);
}
Attempt to alleviate the uninitialized pointer problem: Null pointers

The C99 standard mandates the `null pointer` as a macro:
• A null pointer is guaranteed to compare unequal to a pointer to any object
• Conversion of a null pointer to another pointer type yields a null pointer of that type
• Any two null pointers shall compare equal.

The C11 standard introduces (optionally!) `nullptr` as a keyword!

OK, but this still does not say what is the null pointer ...
Let’s try to see one!
Null pointers

The C99 standard requires a macro NULL to be defined in \texttt{stddef.h} (implementation-dependent).

This is what I found in MSVC:

```c
/* Define NULL pointer value */

#ifndef NULL
  #ifdef __cplusplus
    #define NULL 0
  #else
    #define NULL ((void *)0)
  #endif
#endif

#endif
```
In MSVS 2012, \texttt{nullptr} only works with .cpp files!
In C99 and C11, global and static pointers are initialized by default to null pointers, but auto pointers are not

```c
void main(){
    static int *pi;
    printf("%p \n\n", pi);
}
```
In C99 and C11, global and static pointers are initialized by default to null pointers, but auto pointers are not ...

... However, this does not solve the problem, since the OS does not allow access to the low memory area!
int a[6] = {1,2,3,4,5,6};

If the first Byte of a was assigned the memory address 2000 (decimal), calculate:

• The address of the first Byte of the element storing the value 6.
• The address of the last Byte of the array.
Pointers and Arrays

The **name** of the array is a pointer to the first element of the array.

In C, the relationship `(data == &data[0])` is true

```c
int array[100], *p_array;
/* additional code goes here */
p_array = &array;
```
What will the following expressions evaluate to?

1: \( x == 1000 \)
2: \( x[0] == 1000 \)
3: \( x[1] == 1002 \)
4: \( \text{expenses} == 1250 \)
5: \( \&\text{expenses}[0] == 1250 \)
6: \( \&\text{expenses}[1] == 1254 \)
/* Demonstrates the relationship between addresses and */
/* elements of arrays of different data types. */

#include <stdio.h>

/* Declare a counter and three arrays. */
int ctr;
short array_s[10];
float array_f[10];
double array_d[10];

int main( void )
{

QUIZ

How are `array` and `p_array`?

• Similar?
• Different?

```c
int array[100], *p_array;
p_array = array;
```
Printing memory maps for arrays

/* Print the table heading */

printf("\t\tShort\t\tFloat\t\tDouble\n");
printf("\n=");  
printf("="); 

/* Print the addresses of each array element. */

for (ctr = 0; ctr < 10; ctr++)
  printf("\nElement %d: \t%p \t%p \t%p", ctr, 
         &array_s[ctr], &array_f[ctr], &array_d[ctr]);

printf("\n=");  
printf("=");  

return 0;
<table>
<thead>
<tr>
<th></th>
<th>Short</th>
<th>Float</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element 0</td>
<td>4210896</td>
<td>4210752</td>
<td>4210816</td>
</tr>
<tr>
<td>Element 1</td>
<td>4210898</td>
<td>4210756</td>
<td>4210824</td>
</tr>
<tr>
<td>Element 2</td>
<td>4210900</td>
<td>4210760</td>
<td>4210832</td>
</tr>
<tr>
<td>Element 3</td>
<td>4210902</td>
<td>4210764</td>
<td>4210840</td>
</tr>
<tr>
<td>Element 4</td>
<td>4210904</td>
<td>4210768</td>
<td>4210848</td>
</tr>
<tr>
<td>Element 5</td>
<td>4210906</td>
<td>4210772</td>
<td>4210856</td>
</tr>
<tr>
<td>Element 6</td>
<td>4210908</td>
<td>4210776</td>
<td>4210864</td>
</tr>
<tr>
<td>Element 7</td>
<td>4210910</td>
<td>4210780</td>
<td>4210872</td>
</tr>
<tr>
<td>Element 8</td>
<td>4210912</td>
<td>4210784</td>
<td>4210880</td>
</tr>
<tr>
<td>Element 9</td>
<td>4210914</td>
<td>4210788</td>
<td>4210888</td>
</tr>
</tbody>
</table>
### Pointer Arithmetic

The following arithmetic operations can be performed on pointers:

- **Add an integer to a pointer**

  ```
  p = p+i;
  // or even
  p += i;
  ```

The following calculation is performed:

```
0x008C955C + 2*sizeof(int) = 0x008C9564
```
Pointer Arithmetic

- Subtract an integer from a pointer

```c
p = p - i;
```

//or even

```c
p -= i;
```

The following calculation is performed:

```c
0x008C955C - 2 * sizeof(int) = 0x008C9554
```
• Increment/decrement pointer (++ or --)

• Subtract pointers from each other (p2-p1)
  • p1 and p2 must point to the same data type.
  • p2-p1 gives the **number of elements** from p1 to p2 (counting only one end!)
  • `ptrdiff_t` is a signed integer type defined in `stddef.h`

```c
#include <stddef.h>
...
int a[6], *p1, *p2;
ptrdiff_t n;
p1 = &a[1];
p2 = &a[4];
n = p2-p1;       // n is 3
```
These are all the legal arithmetic operations on pointers:
• Add pointer with \texttt{int}
• Sub. \texttt{int} from pointer
• Incr.
• Decr.
• Subtr. ptrs. of same type (a.k.a. \textit{differencing})

No multiplication, division, integer division or remainder!
int a[6] = {100,200,300,400,500,600}, *p, *q;

p = q = &a[0];
printf("%d", *p);    // 100

p += 3;
printf("%d", *p);    // 400
printf("%d", *(p+1)); // 500
printf("%d", *(p-1)); // 300
printf("%d", *p + 1); // 401

*p = 11;           // sets a[3] = 11
*(p+2) = 12;        // sets a[5] = 12
*(p-2) = 13;        // sets a[1] = 13

q++;
printf("%p", p-q);  // ???

Contents of array at the end:

<table>
<thead>
<tr>
<th></th>
<th>a[0] = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-2</td>
<td>0x008C9554</td>
</tr>
<tr>
<td></td>
<td>a[1] = 13</td>
</tr>
<tr>
<td></td>
<td>0x008C9558</td>
</tr>
<tr>
<td>a[2]</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>0x008C9560</td>
</tr>
<tr>
<td>a[3]</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>0x008C9564</td>
</tr>
<tr>
<td>a[4]</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>a[5] = 12</td>
</tr>
</tbody>
</table>
Pointer Comparisons

All the numeric comparisons also work on pointers

```
==  !=  <  <=  >  >=
```

(They only make sense if we compare pointers to elements of the same array!)
What does the increment apply to?

```c
char dest[100], src[100] = "original string";
char *dp = &dest[0], *sp = &src[0];

while(*sp != '\0')
    *dp++ = *sp++;

*dp++ = *sp++ is equivalent to *dp=*sp; dp++; sp++;

The pre-increment form *++p increments p, then dereferences it.
```
Pointers and Arrays: subscript notation vs. pointer notation

Based on our knowledge of pointer arithmetic, we can now perform array subscripting operations using either the array name or some other pointer(s)!
Pointers and Arrays: 
subscript notation vs. pointer notation

```c
int a[6] = {1,2,3,4,5,6};
int *pa = a;
```

The 3rd element of array `a` can be accessed in 4 ways:

- `a[2]`
- `*(a+2)`
- `pa[2]`
- `*(pa+2)`
6. Show two ways to assign the value 100 to the third element of data[].
Can also be written in pointer notation: int *num_array

The length of the array is passed as a separate argument.
Silly question:
Why do we use two different names for the array?
The function definition

29: int largest(int num_array[], int length)
30: {
31:     int count, biggest;
32:     for ( count = 0; count < length; count++)
33:     {
34:         if (count == 0)
35:             biggest = num_array[count];
36:         if (num_array[count] > biggest)
37:             biggest = num_array[count];
38:     }
39: }
40:
41:     return biggest;
42: }
Alternate way of telling the function where the array ends:

There is no *length* argument, but the last array value is a special one (a.k.a. *sentinel*).
Practice

7. Write a function named `sumarrays()` that accepts two arrays as arguments, totals all values in both arrays, and returns the total to the calling program.
When processing arrays using pointers, care must be taken to stay within the range defined for the array.

In C, this is exclusively the programmer’s responsibility!

```c
#include <stdio.h>

int main()
{
    int a[4] = {10, 20, 30, 40}, *intPtr;

    intPtr = &a[3];
    *intPtr = 400;
    intPtr++;
    *intPtr = 500;
    printf("%d\n\n", *intPtr);
}
```

Array bounds!
Pointers are awesome! 😊
Pointers are awesome! 😊

... but remember:

With great power comes great responsibility
To do for next time:

• Read pp.194 - 7

• Re-Solve all today’s quizzes in notebook
Homework for Ch. 9:

End-of-chapter #9

Not from text: Create an array of 10 integers and initialize it with elements of your choice. (OK to use `rand` if you want.) Write a function that prints the array in reverse order. The function does not return anything. Test the function in the main program.

Due Friday, March 4