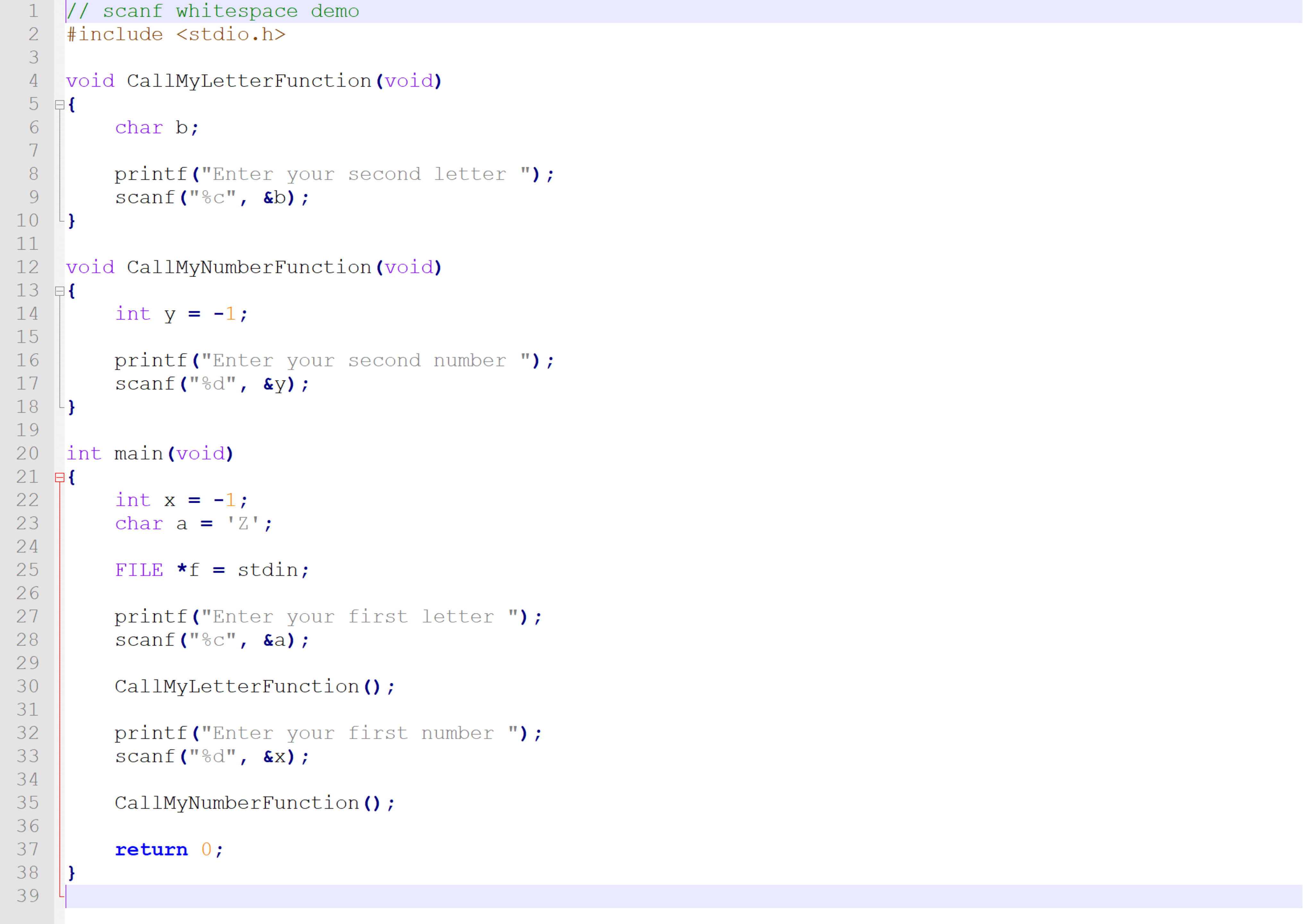
CSE 1320

Week of 02/04/2019

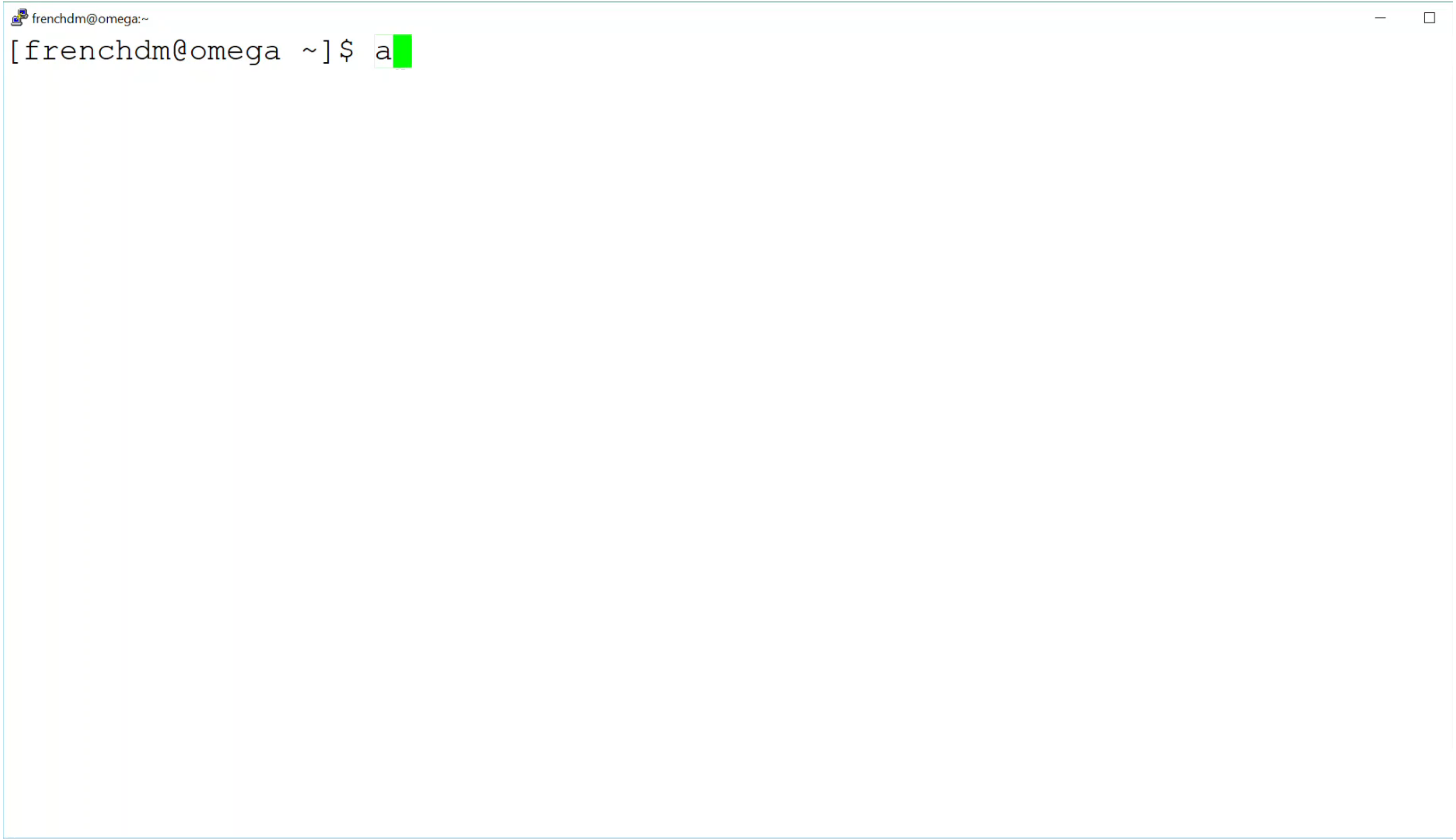
Instructor : Donna French

Plan For Today

- Pointers
- Action Items



```c
// scanf whitespace demo
#include <stdio.h>

void CallMyLetterFunction(void)
{
    char b;

    printf("Enter your second letter ");
    scanf("%c", &b);
}

void CallMyNumberFunction(void)
{
    int y = -1;

    printf("Enter your second number ");
    scanf("%d", &y);
}

int main(void)
{
    int x = -1;
    char a = 'Z';

    FILE *f = stdin;

    printf("Enter your first letter ");
    scanf("%c", &a);

    CallMyLetterFunction();

    printf("Enter your first number ");
    scanf("%d", &x);

    CallMyNumberFunction();

    return 0;
}

```



```
[frenchdm@omega ~]$ a
```

```
printf("Enter your first letter ");
scanf("%c", &a);

CallMyLetterFunction();

printf("Enter your first number ");
scanf("%d", &x);

CallMyNumberFunction();

return 0;
```

```
void CallMyLetterFunction(void)
{
      char b;

      printf("Enter your second letter ");
      scanf("%c", &b);
}

void CallMyNumberFunction(void)
{
      int y = -1;

      printf("Enter your second number ");
      scanf("%d", &y);
}
```

scanfwhitespaceDemo.c

```
printf("Enter your first letter ");
scanf(" %c", &a);

CallMyLetterFunction();

printf("Enter your first number ");
scanf("%d", &x);

CallMyNumberFunction();

return 0;
```

```
void CallMyLetterFunction(void)
{
      char b;

      printf("Enter your second letter ");
      scanf(" %c", &b);
}

void CallMyNumberFunction(void)
{
      int y = -1;

      printf("Enter your second number ");
      scanf("%d", &y);
}
```

"%c" vs " %c" vs "%d"

Using

```
        scanf("%d", …);
```

skips whitespace and special characters. `\n` is a special character; therefore, `%d` skips it.

Using

```
        scanf("%c", …);
```

`%c` does not skip `\n` because `%c` processes whitespace and special characters

Using

```
        scanf(" %c", …);
```

Putting a blank in front of the `%c` tells `scanf()` to skip whitespace and special characters

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*

*but it can be simulated using pointers –
we'll talk about that later

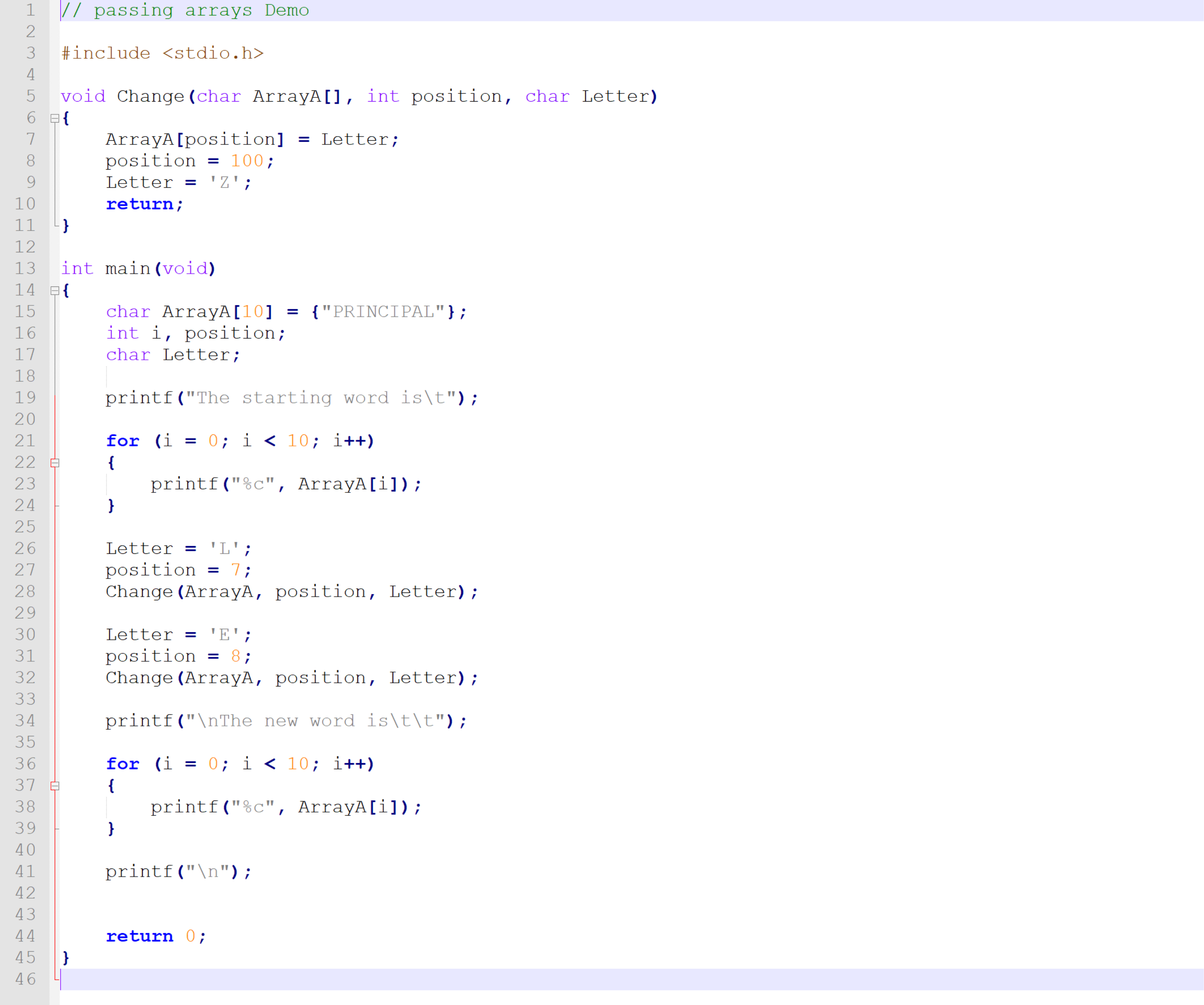
```c
int PassByValue(int MyNum)
{
     MyNum += 100;
     printf("Inside PassByValue\tMyNum     = %d\n", MyNum);
}

int main(void)
{
  int MyMainNum = 0;

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

  return 0;
}
```

simrefDemo.c

```
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



```c
// passing arrays Demo

#include <stdio.h>

void Change(char ArrayA[], int position, char Letter)
{
    ArrayA[position] = Letter;
    position = 100;
    Letter = 'Z';
    return;
}

int main(void)
{
    char ArrayA[10] = {"PRINCIPAL"};
    int i, position;
    char Letter;

    printf("The starting word is\t");

    for (i = 0; i < 10; i++)
    {
        printf("%c", ArrayA[i]);
    }

    Letter = 'L';
    position = 7;
    Change(ArrayA, position, Letter);

    Letter = 'E';
    position = 8;
    Change(ArrayA, position, Letter);

    printf("\nThe new word is\t\t");

    for (i = 0; i < 10; i++)
    {
        printf("%c", ArrayA[i]);
    }

    printf("\n");


    return 0;
}

```

sarray1Demo.c

Unnecessary Extra Variables in C

```
int BBBBB = 0;
int ZZZZZ[7] = {};
int AAAAA;

printf("Decimal to binary converter.\n");
printf("Please enter a decimal number between 0 and 255: ");
scanf("%d", &BBBBB);

AAAAA = BBBBB;
ConvertDecimaltoBinary(AAAAA, ZZZZZ);
printf("Decimal %d converts to binary ", BBBBB);
PrintBinary(ZZZZZ);
```

Segmentation Fault

What is a segmentation fault?

In computing, a **segmentation fault** (often shortened to **segfault**) or access violation is a **fault**, or failure condition, raised by hardware with memory protection, notifying an operating system (OS) the software has attempted to access a restricted area of memory (a memory access violation).

For more details and other common examples of causes of segmentation faults

Segmentation fault – Wikipedia

```
[frenchdm@omega ~]$ 
```

Two-Dimensional Arrays

A two-dimensional array can be thought of as a matrix of elements.

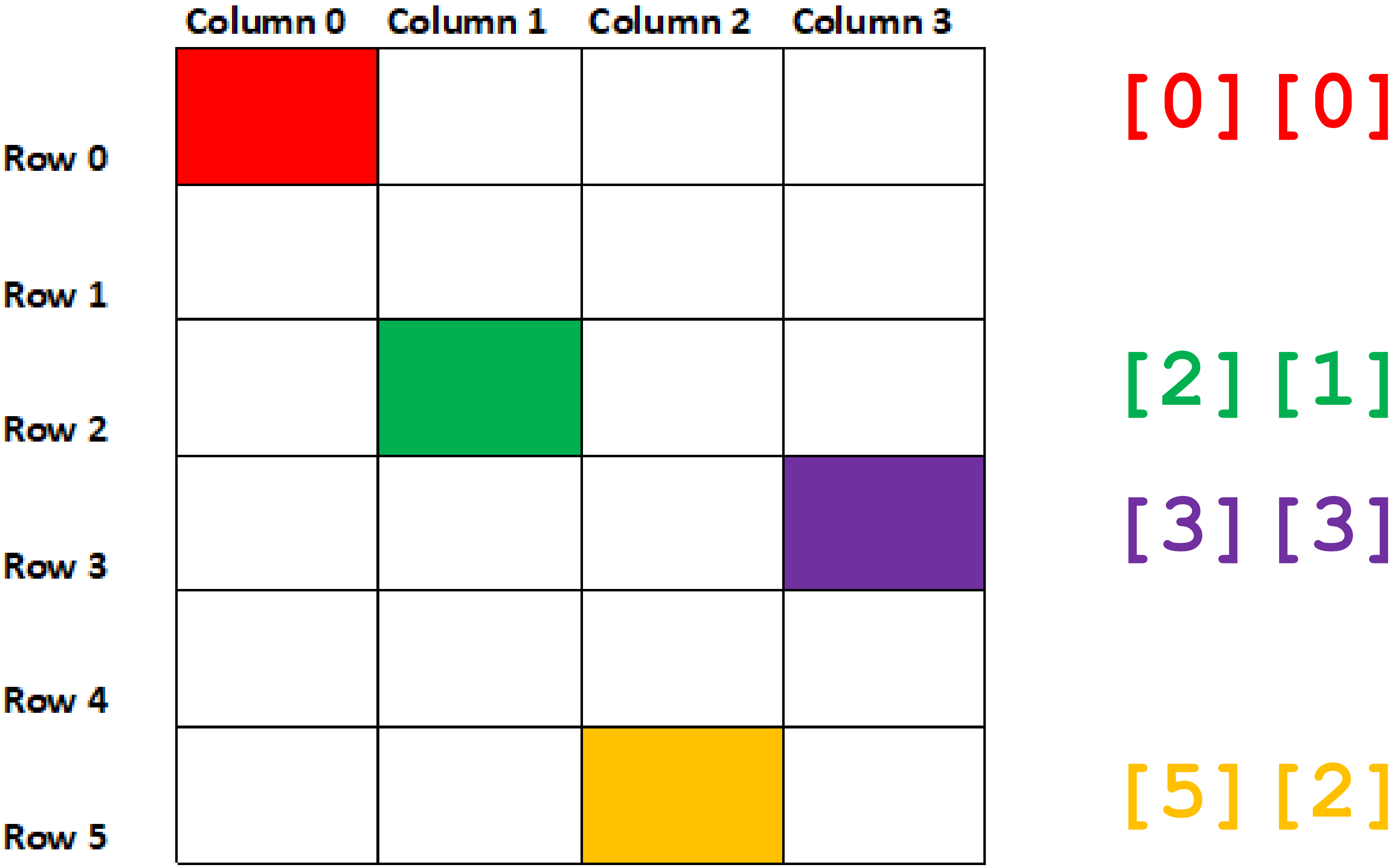
	Column 0	Column 1	Column 2	Column 3
Row 0				
Row 1				
Row 2				
Row 3				
Row 4				
Row 5				

Two-Dimensional Arrays

Indices start at 0 and positions are referred to by row, column

	Column 0	Column 1	Column 2	Column 3
Row 0				
Row 1				
Row 2				
Row 3				
Row 4				
Row 5				

Two-Dimensional Arrays



	Column 0	Column 1	Column 2	Column 3
Row 0	🟥			
Row 1				
Row 2		🟩		
Row 3				🟪
Row 4				
Row 5			🟨	

[0] [0]

[2] [1]

[3] [3]

[5] [2]

Two-Dimensional Arrays

```
int My2DArray[6][4];
```

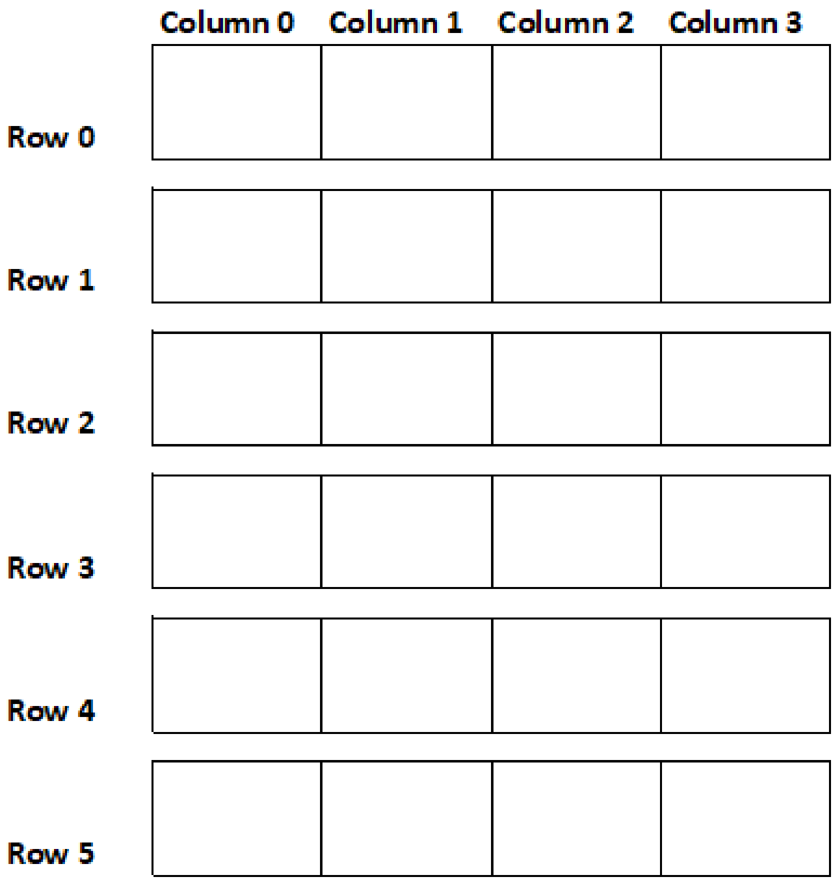
	Column 0	Column 1	Column 2	Column 3
Row 0	🟥			
Row 1				
Row 2		🟩		
Row 3				🟪
Row 4				
Row 5			🟨	

```
My2DArray[0][0];

My2DArray[2][1];

My2DArray[3][3];

My2DArray[5][2];
```

Two-Dimensional Arrays

Two-dimensional arrays are more accurately described as an array of arrays.

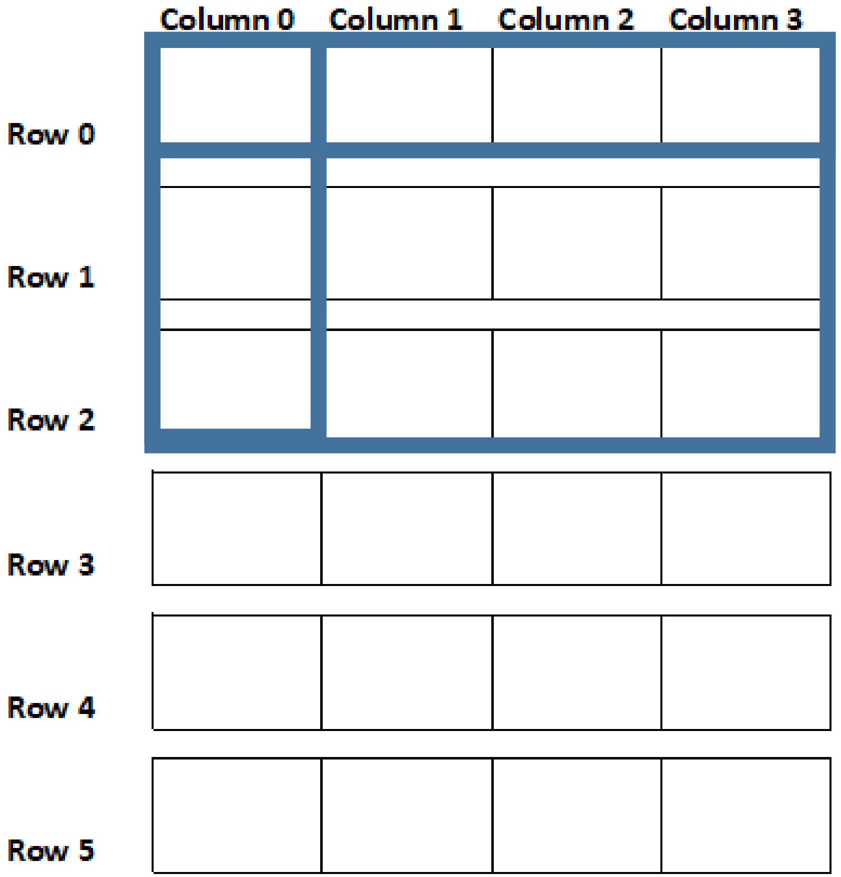
```
int (My2DArray[6])[4];
```



Multi-Dimensional Arrays

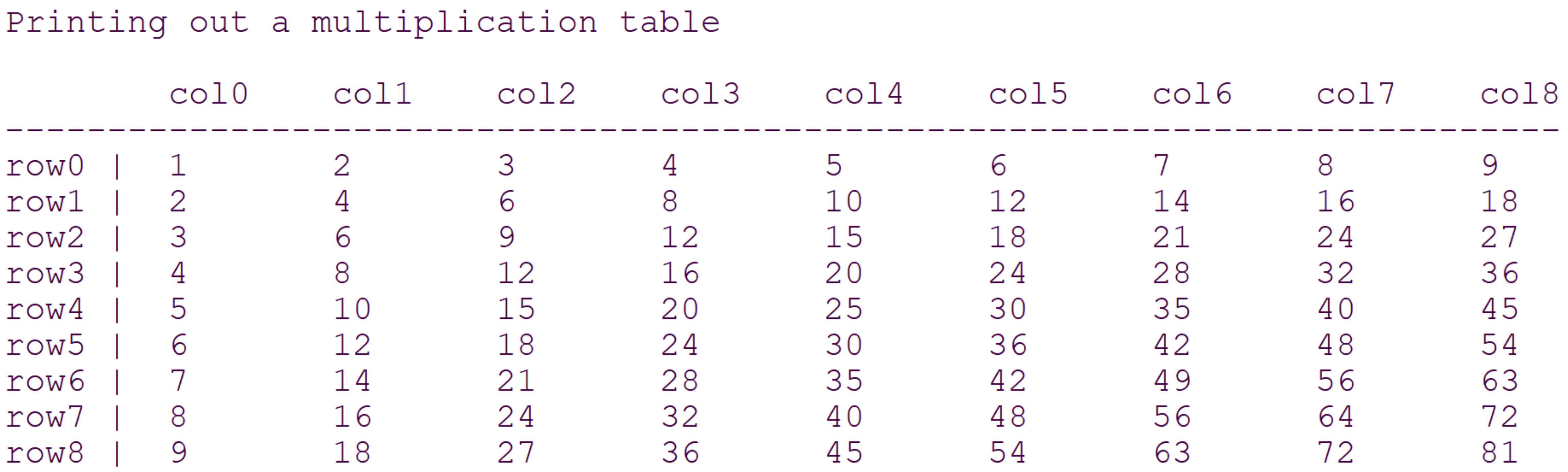
Multi-dimensional arrays are more accurately described as arrays of arrays.

```
int My2DArray[6][4];

int (My2DArray[6])[4];

int My2DArray[3][4];
int My2DArray[3][1];
int My2DArray[1][4];
```



2d1Demo.c

Two-Dimensional Arrays



```
Printing out a multiplication table

        col0    col1    col2    col3    col4    col5    col6    col7    col8
-----------------------------------------------------------------------------
row0 |  1       2       3       4       5       6       7       8       9
row1 |  2       4       6       8       10      12      14      16      18
row2 |  3       6       9       12      15      18      21      24      27
row3 |  4       8       12      16      20      24      28      32      36
row4 |  5       10      15      20      25      30      35      40      45
row5 |  6       12      18      24      30      36      42      48      54
row6 |  7       14      21      28      35      42      49      56      63
row7 |  8       16      24      32      40      48      56      64      72
row8 |  9       18      27      36      45      54      63      72      81
```

2d1Demo.c

```
// Declare a two dimensional array and initialize to NULLs
int MultTable[9][9] = {};

for (i = 1; i <= 9; i++)
{
   for (j = 1; j <= 9; j++)
   {
      MultTable[i-1][j-1] = i * j;
   }
}
```

```
{{0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0}}
```

```
MultTable[0][0] = 1 * 1;
```

```
{{1, 2, 3, 4, 5, 6, 7, 8, 9},
 {2, 4, 6, 8, 10, 12, 14, 16, 18},
 {3, 6, 9, 12, 15, 18, 21, 24, 27},
 {4, 8, 12, 16, 20, 24, 28, 32, 36},
 {5, 10, 15, 20, 25, 30, 35, 40, 45},
 {6, 12, 18, 24, 30, 36, 42, 48, 54},
 {7, 14, 21, 28, 35, 42, 49, 56, 63},
 {8, 16, 24, 32, 40, 48, 56, 64, 72},
 {9, 18, 27, 36, 45, 54, 63, 72, 81}}
```

2d1Demo.c

```
/* print heading */
printf("\tcol0\tcol1\tcol2\tcol3\tcol4\tcol5\tcol6\tcol7\tcol8\n");

for (i = 0; i <= 75; i++)
   printf("-");

printf("\n");

for (i = 0, k = 1; i < 9; i++, k
{
   printf("row%d |\t", k-1);
   for (j = 0; j < 9; j++)
   {
      printf("%d\t", MultTable[i][j]);
   }
   printf("\n");
}
```

Print the dashes under the columns

Print "row x | " at start of each row

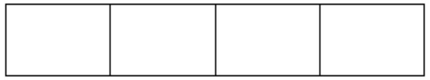
Print one row of table

Bring cursor back to start of line

```
        col0    col1    col2    col3    col4    col5    col6    col7    col8
----------------------------------------------------------------------------
row0 |  1       2       3       4       5       6       7       8       9
row1 |  2       4       6       8       10      12      14      16      18
row2 |  3       6       9       12      15      18      21      24      27
row3 |  4       8       12      16      20      24      28      32      36
row4 |  5       10      15      20      25      30      35      40      45
row5 |  6       12      18      24      30      36      42      48      54
row6 |  7       14      21      28      35      42      49      56      63
row7 |  8       16      24      32      40      48      56      64      72
row8 |  9       18      27      36      45      54      63      72      81
```

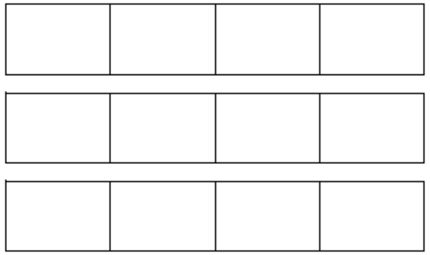
Arrays with More Than Two Dimensions

one dimension



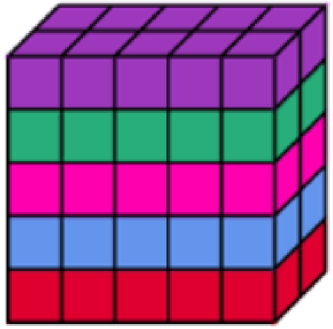
two dimensions

array of 1D arrays



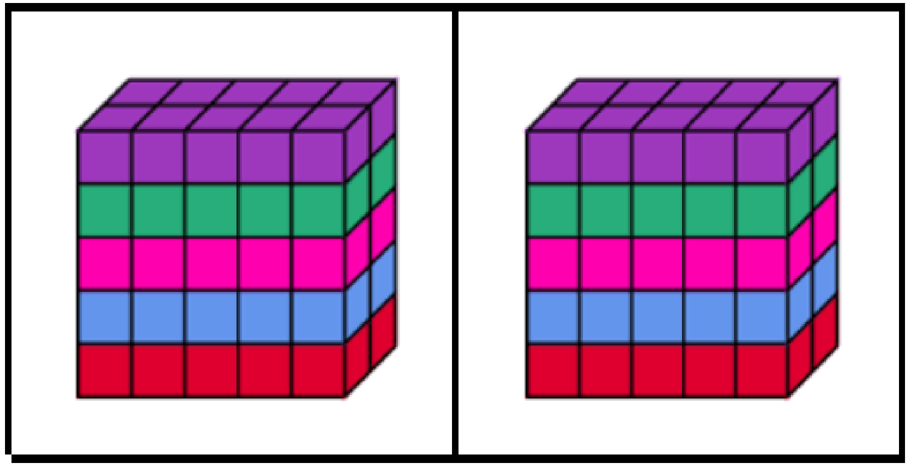
three dimensions

array of 2D arrays

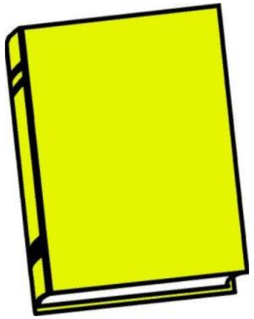


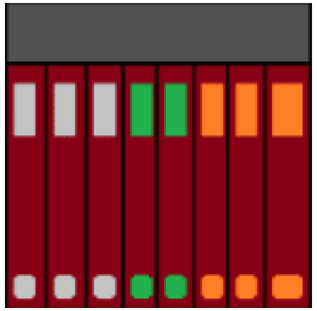
four dimensions

array of 3D arrays



If a single book is an array of pages,
then a shelf of books is an array of arrays (2D array).





If a shelf of books is a 2D array,
then a bookcase is an array of 2D arrays (3D array).

If a bookcase is a 3D array,
then a set of bookcases is an array of 3D arrays (4D array).

So how many dimensions could a library have?

Initializing Multidimensional Arrays

```
int My2DArray[3][2] = {};

int My2DArray[3][2] = {{1, 2},
                       {3, 4},
                       {5, 6}};

int My2DArray[][2]  = {{1, 2},
                       {3, 4},
                       {5, 6}};
```

only works if initializing at the time of declaration – compiler can count the number of items

```
char Alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

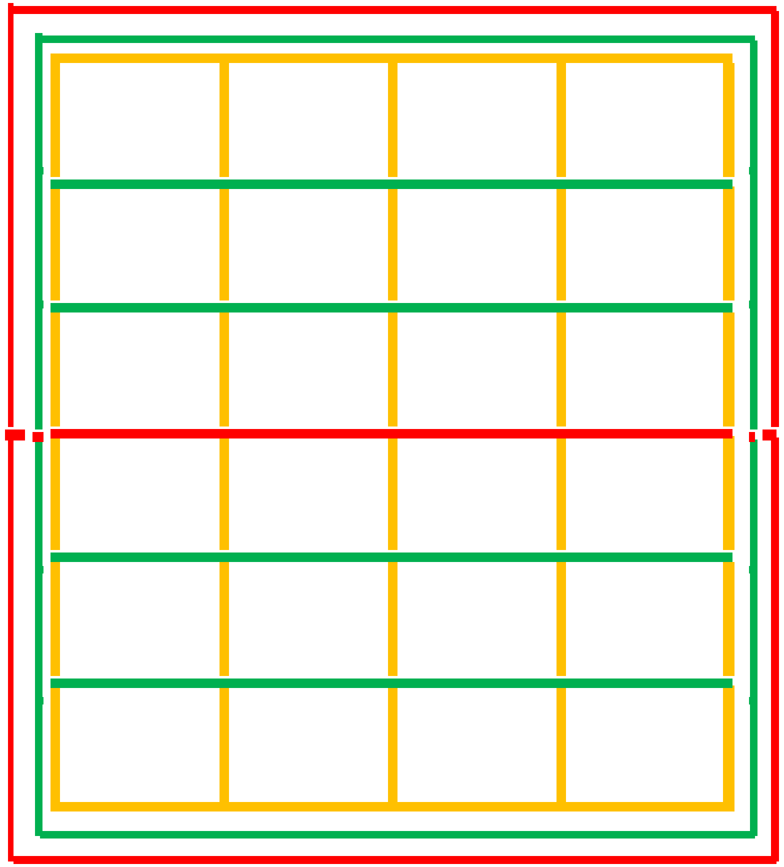
Arrays with More Than Two Dimensions

```
float My3DArray[2]       = {0, 1};
```

```
float My3DArray[2][3]    = {{0.0, 0.1, 0.2},
                            {1.0, 1.1, 1.2}};
```

Arrays with More Than Two Dimensions

```
float My3DArray[2][3][4] = {
                            {  {0.00,0.01,0.02,0.03},
                               {0.10,0.11,0.12,0.13},
                               {0.20,0.21,0.22,0.23}
                            }
                            {  {1.00,1.01,1.02,1.03},
                               {1.10,1.11,1.12,1.13},
                               {1.20,1.21,1.22,1.23}
                            }
                           };
```



Multidimensional Arrays as Parameters to Functions

When passing a one dimensional array to a function, we don’t specify the number of elements in the function parameter because we are passing the address of the array.

```
void ConvertDecimalToBinary(int BinaryArray[])
```

The programmer needs to know how many elements are in the array in order to not go beyond the bounds of the array.

What if we wanted to move the creation of the multiplication table to its own function and the printing of the multiplication table to its own function? We could also allow the user to choose the size of the table.

```
int MultTable[MAX_ROW][MAX_COL] = {};
```

`MAX_ROW` and `MAX_COL` are both set to 9

```
printf("How many rows (1-9)? ");
scanf("%d", &row);
```

Don't need a `getchar()` in between. Why?

```
printf("How many columns (1-9)? ");
scanf("%d", &col);


FillOutMultTable(MultTable, row, col);

PrintMultTable(MultTable, row, col);
```

Passing the address of the array by using the name and the size of the array in `row` and `col`

2d2Demo.c

```
How many rows (1-9)? 5
```

```
printf("How many rows (1-9)? ");
scanf("%d", &row);
```

```
How many columns (1-9)? 5
```

```
printf("How many columns (1-9)? ");
scanf("%d", &col);
```

```
(gdb) p row
$7 = 5

(gdb) p col
$8 = 5
```

2d2Demo.c

```
void FillOutMultTable(int MultTable[][MAX_COL], int row, int col)
{
     int i, j;

     for (i = 1; i <= row; i++)
     {
          for (j = 1; j <= col; j++)
          {
               MultTable[i-1][j-1] = i * j;
          }
     }

     return;
}
```

Don't need the first dimension but do need any additional dimensions.

First dimension is not needed because it is just an address but when you add more dimensions, the compiler needs to know where each column starts and ends in order to properly divide up the contiguous memory reserved for the array.

2d2Demo.c

Multidimensional Arrays as Parameters to Functions

Calling the functions

```
FillOutMultTable(MultTable, row, col);
PrintMultTable(MultTable, row, col);
```

Pass the name of the array which is the address of the array

The functions themselves

```
void FillOutMultTable(int MultTable[][MAX_COL], int row, int col)
void PrintMultTable(int MultTable[][MAX_COL], int row, int col)
```

Must pass the size of every dimension after the first one

Getting Started with Pointers

Computer Memory and Addresses

Computer Memory and Addresses

Boxes

of

many

different

sizes

Every box has a unique address

Computer Memory and Addresses

- When you rent a PO box, the Post Office decides where your box is – you don’t choose.

- You are only given a spot that is already empty.

- PO boxes come in different sizes.

Computer Memory and Addresses

- In general, upper-level languages give the programmer little or no control over the assignment of memory addresses
  - You don’t pick your PO box and you don’t pick where your variables go in memory. Space is reserved for you but you do not choose. If a particular box is already being used, then your spot will be somewhere else.

- The programmer controls what is stored in memory but not where it is stored.
  - You control what is in your PO box based what type of mail you receive.
  - You decide how big your PO box will be and you decide how much memory will be used based on the variable types you choose.

Computer Memory and Addresses

- Every PO box has an address and every address is unique.

- the `&` used by `scanf()` refers to the address of the variable

```
scanf("%d", &MyVar);
```

By using the `&`, we are telling `scanf()` where to put the value it reads by giving it the address of the variable.

Computer Memory and Addresses

`%p`

- conversion specification for printing the memory address assigned by the computer for the location of the variable

- form of output can vary with computer systems

- `%x`

  hexadecimal
- `%o`

  octal

```
char CharVar1;
char CharVar2;
int  IntVar1;
int  IntVar2;
long LongVar1;
long LongVar2;
```

```
printf("The address of CharVar1 is %p\t%x\t%o\n\n",
       &CharVar1, &CharVar1, &CharVar1);
printf("The address of CharVar2 is %p\t%x\t%o\n\n",
       &CharVar2, &CharVar2, &CharVar2);

printf("The address of IntVar1 is %p\t%x\t%o\n\n",
       &IntVar1, &IntVar1, &IntVar1);
printf("The address of IntVar2 is %p\t%x\t%o\n\n",
       &IntVar2, &IntVar2, &IntVar2);

printf("The address of LongVar1 is %p\t%x\t%o\n\n",
       &LongVar1, &LongVar1, &LongVar1);
printf("The address of LongVar2 is %p\t%x\t%o\n\n",
       &LongVar2, &LongVar2, &LongVar2);
```

varadd1Demo.c

```
                                                          hex              octal
The address of CharVar1 is 0x7fffa67cbaff          a67cbaff       24637135377
                                           1 byte
The address of CharVar2 is 0x7fffa67cbafe          a67cbafe       24637135376

The address of IntVar1 is 0x7fffa67cbaf8     4     a67cbaf8       24637135370
The address of IntVar2 is 0x7fffa67cbaf4   bytes   a67cbaf4       24637135364

The address of LongVar1 is 0x7fffa67cbae8    8     a67cbae8       24637135350
The address of LongVar2 is 0x7fffa67cbae0  bytes   a67cbae0       24637135340

The address of CharVar1 is 0x7fffec05e88f           ec05e88f       35401364217
The address of CharVar2 is 0x7fffec05e88e           ec05e88e       35401364216

The address of IntVar1 is 0x7fffec05e888            ec05e888       35401364210
The address of IntVar2 is 0x7fffec05e884            ec05e884       35401364204

The address of LongVar1 is 0x7fffec05e878           ec05e878       35401364170
The address of LongVar2 is 0x7fffec05e870           ec05e870       35401364160
```

varadd1Demo.c

```
int i;
int Choice = 0;
int MyIntArray[2] = {0,0};

printf("Choice is currently %d at %p\t", Choice, &Choice);

for (i = 0; i <= 2; i++)
{
   MyIntArray[i] = i;
   printf("MyIntArray[%d] = %d\t%p\n", i, MyIntArray[i], &MyIntArray[i]);
   printf("Choice is currently %d at %p\t", Choice, &Choice);
}
```

```
Choice is currently 0 at 0x7fff02cfaf68 MyIntArray[0] = 0      0x7fff02cfaf60

Choice is currently 0 at 0x7fff02cfaf68 MyIntArray[1] = 1      0x7fff02cfaf64

Choice is currently 0 at 0x7fff02cfaf68 MyIntArray[2] = 2      0x7fff02cfaf68

Choice is currently 2 at 0x7fff02cfaf68
```

```
int i;
int MyIntArray[2] = {0,0};
int Choice = 0;

printf("Choice is currently %d at %p\t", Choice, &Choice);

for (i = 0; i <= 2; i++)
{
   MyIntArray[i] = i;
   printf("MyIntArray[%d] = %d\t%p\n", i, MyIntArray[i], &MyIntArray[i]);
   printf("Choice is currently %d at %p\t", Choice, &Choice);
}
```

Declaring `Choice` after `MyIntArray`

```
Choice is currently 0 at 0x7fff3c10511c MyIntArray[0] = 0       0x7fff3c105120

Choice is currently 0 at 0x7fff3c10511c MyIntArray[1] = 1       0x7fff3c105124

Choice is currently 0 at 0x7fff3c10511c MyIntArray[2] = 2       0x7fff3c105128

Choice is currently 0 at 0x7fff3c10511c
```

Pointers

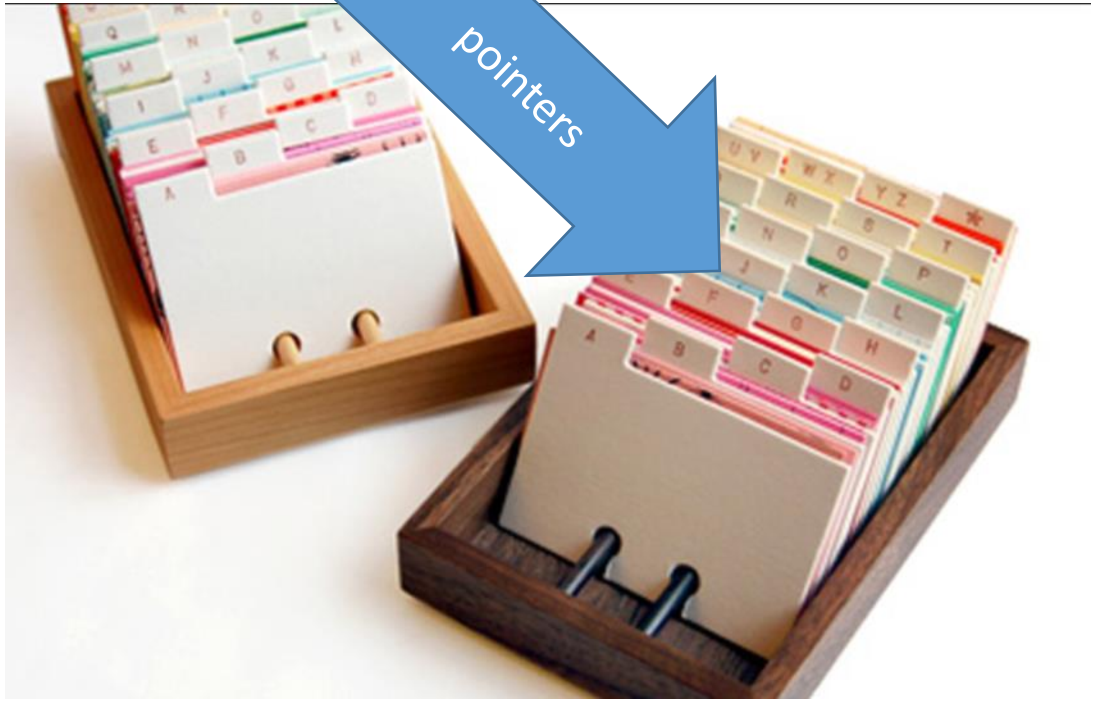
What is a pointer?

- another technique to determine the address of a variable
- stores the address of a memory location
- pointer variable points to another variable
  - it stores the address of the memory location allocated for values of the other variable

These address cards hold/contain an address – not what's at the address.

Pointers

Memory locations have addresses and pointers can hold those addresses.



pointers

memory locations

Pointers

A variable name directly references a value

```
int IntVarA = 8765;
```

A pointer indirectly references a value

```
int *IntVarAPtr = &IntVarA;
```

Pointer variables contain *memory addresses* as their values. Normally, a variable *directly* contains a specific value.

Pointers

- pointers are considered to be separate data types
  - pointer to `char`   pointer to `int`
  - pointer to `float`   pointer to `double`

```
char *charptr
int *intptr
float *floatptr
double *doubleptr
```

- every data type has a corresponding pointer type

```
int *IntPtr
```

the legal values for `IntPtr` are the addresses of integers

- Referencing a value through a pointer is called **indirection**.

- double indirection
  - pointer to pointer

```
char **dicharptr
```

Pointers

Unary operator * is used to create pointer type

regular variable

```
      int MyIntVar1;
```

pointer variable

```
      int *MyIntVarPtr1;
```

```
int* MyIntVarPtr;
int*MyIntVarPtr;
```

`MyIntVarPtr` is a pointer to `int`

```
int *DogPtr, CatPtr, BirdPtr;
```

Is this a valid declaration?

`CatPtr` and `BirdPtr` are not pointers

```
#include <stdio.h>

int main(void)
{
     int MyInt = 123;
     int *MyIntPtr;

     printf("The contents of MyInt    is %d\n", MyInt);
     printf("The address  of MyInt    is %p\n", &MyInt);

     // Storing the address of MyInt in MyIntPtr
     MyIntPtr = &MyInt;

     return 0;
}
```

The address operator (`&`) is a unary operator that obtains the memory address of its operand.

pointer1Demo.c

```
(gdb) break main
Breakpoint 1 at 0x4004a0: file pointer1Demo.c, line 7.
(gdb) run
Starting program: /home/f/fr/frenchdm/a.out

Breakpoint 1, main () at pointer1Demo.c:7
7               int MyInt = 123;
(gdb) step
10              printf("The contents of MyInt    is %d\n", MyInt);
(gdb) p MyInt
$1 = 123
(gdb) step
The contents of MyInt    is 123
11              printf("The address  of MyInt    is %p\n", &MyInt);
(gdb) p &MyInt
$2 = (int *) 0x7fffffffe7a4
(gdb) step
The address  of MyInt    is 0x7fffffffe7a4
14              MyIntPtr = &MyInt;
(gdb) step
16              return 0;
(gdb) p MyIntPtr
$3 = (int *) 0x7fffffffe7a4
```

pointer1Demo.c

Pointer Initialization and the NULL pointer

When a pointer is declared, the compiler sets aside memory for the value of the pointer (an address) but it does not initialize the pointer.

The programmer must assign/initialize the pointer to a legal memory address.

BE CAREFUL!!

- don’t write outside of your allowable memory space
- don’t erase data needed by the operating system or other programs

NULL should be used to indicate that a pointer does not point at a legal memory address.

```
IntVarPtr1 = NULL;
```

nullpointer1Demo.c

```
int  IntVar1 = 66, *IntVarPtr1;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr1);
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7fff91e16bd4
Contents of   IntVarPtr1   (nil)
```

```
IntVarPtr1  = &IntVar1;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7fff91e16bd4
Contents of   IntVarPtr1   0x7fff91e16bd4
```

nullpointer1Demo.c

```
IntVarPtr1 = NULL;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr1);
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7fff91e16bd4
Contents of   IntVarPtr1   (nil)
```

What is NULL and how is it defined?

As a matter of style, many programmers prefer not to have unadorned 0's scattered through their programs, some representing numbers and some representing pointers. Therefore, the preprocessor macro `NULL` is defined (by several headers, including `<stdio.h>` and `<stddef.h>`) as a null pointer constant,
typically `0` or `((void *)0)`. A programmer who wishes to make explicit the distinction between 0 the integer and 0 the null pointer constant can then use NULL whenever a null pointer is required.

Using NULL is a stylistic convention only; the preprocessor turns NULL back into 0 which is then recognized by the compiler, in pointer contexts, as before.

Dereferencing a Pointer Variable

Printing the addresses of variables

could be useful for debugging

not often a permanent part of a program

We are more interested in the value pointed to by a pointer

the value can accessed by pointer operations

the value can be changed by pointer operations

Dereferencing a Pointer Variable

The unary `*` operator is commonly referred to as the

indirection operator or dereferencing operator

This *dereference* operator * is used to get to the contents of the address stored in `IntPtr`.

```
printf("The address in IntPtr is pointing to value %d", *IntPtr);
```

When `*IntPtr` is used in any other expression other than a declaration, it refers to the contents of the current address in `IntPtr`.

This is called *dereferencing* the pointer.

```c
int MyInt = 123;
int *MyIntPtr = NULL;

printf("The contents of MyInt    is %d\n", MyInt);
printf("The address  of MyInt    is %p\n", &MyInt);
printf("The address  of MyIntPtr is %p\n", &MyIntPtr);
```

```
The contents of MyInt    is 123
The address  of MyInt    is 0x7fff8bef8b2c
The address  of MyIntPtr is 0x7fff8bef8b20
```

```c
// Storing the address of MyInt in MyIntPtr
printf("\n\nStoring the address of MyInt in MyIntPtr...\n\n");
MyIntPtr = &MyInt;
```

```
Storing the address of MyInt in MyIntPtr...
```

```c
printf("The contents of MyIntPtr is %p\n", MyIntPtr);
printf("Dereferencing MyIntPtr....  %d\n", *MyIntPtr);
```

```
The contents of MyIntPtr is 0x7fff8bef8b2c
Dereferencing MyIntPtr....  123
```

pointer2Demo.c

Dereferencing a Pointer Variable

A pointer variable can be used on either side of an assignment

```
int *IntVarPtr1  = &IntVar1;

*CharVarPtr1 = *CharVarPtr1 | 32;

*IntVarPtr1  = 100;

*LongVarPtr1 = *IntVarPtr1 + 1000;
```

```
char CharVar1 = 'A',      *CharVarPtr1 = &CharVar1;
int  IntVar1  = 66,       *IntVarPtr1  = &IntVar1;
long LongVar1 = 66 + ' ', *LongVarPtr1 = &LongVar1;
```

```
printf("Contents of   CharVar1          %c\n",   CharVar1);
printf("Address  of   CharVar1          %p\n",  &CharVar1);
printf("Contents of   CharVarPtr1       %p\n",   CharVarPtr1);
printf("Dereferencing CharVarPtr1(%%c)  %c\n",  *CharVarPtr1);
printf("Dereferencing CharVarPtr1(%%d)  %d\n",  *CharVarPtr1);
```

```
Contents of   CharVar1         A
Address  of   CharVar1         0x7fff7c26feff
Contents of   CharVarPtr1      0x7fff7c26feff
Dereferencing CharVarPtr1(%c)  A
Dereferencing CharVarPtr1(%d)  65
```

deref1Demo.c

```
char CharVar1 = 'A',      *CharVarPtr1 = &CharVar1;
int  IntVar1  = 66,       *IntVarPtr1  = &IntVar1;
long LongVar1 = 66 + ' ', *LongVarPtr1 = &LongVar1;
```

```
printf("Contents of   IntVar1          %d\n",   IntVar1);
printf("Address  of   IntVar1          %p\n",  &IntVar1);
printf("Contents of   IntVarPtr1       %p\n",   IntVarPtr1);
printf("Dereferencing IntVarPtr1(%%c)   %c\n",  *IntVarPtr1);
printf("Dereferencing IntVarPtr1(%%d)   %d\n",  *IntVarPtr1);
```

```
Contents of   IntVar1          66
Address  of   IntVar1          0x7fff7c26fef8
Contents of   IntVarPtr1       0x7fff7c26fef8
Dereferencing IntVarPtr1(%c)   B
Dereferencing IntVarPtr1(%d)   66
```

deref1Demo.c

```
char CharVar1 = 'A',      *CharVarPtr1 = &CharVar1;
int  IntVar1  = 66,       *IntVarPtr1  = &IntVar1;
long LongVar1 = 66 + ' ', *LongVarPtr1 = &LongVar1;


printf("Contents of   LongVar1         %ld\n",  LongVar1);
printf("Address  of   LongVar1         %p \n", &LongVar1);
printf("Contents of   LongVarPtr1      %p \n",  LongVarPtr1);
printf("Dereferencing LongVarPtr1      %ld\n", *LongVarPtr1);
printf("Dereferencing LongVarPtr1      %c\n",  *LongVarPtr1);
```

```
Contents of   LongVar1         98
Address  of   LongVar1         0x7fff7c26fef0
Contents of   LongVarPtr1      0x7fff7c26fef0
Dereferencing LongVarPtr1      98
```

deref1Demo.c

```
CharVar1 = ‘A’
IntVar1 = 66
LongVar1 = 98
```

```
*CharVarPtr1 = *CharVarPtr1 | 32;
*IntVarPtr1  = 100;
*LongVarPtr1 = *IntVarPtr1 + 1000;

printf("Contents of   CharVar1          %c\n",   CharVar1);
printf("Dereferencing CharVarPtr1(%%c)  %c\n",  *CharVarPtr1);

printf("Contents of   IntVar1           %d\n",   IntVar1);
printf("Dereferencing IntVarPtr1(%%c)   %c\n",  *IntVarPtr1);

printf("Contents of   LongVar1          %ld\n",  LongVar1);
printf("Dereferencing LongVarPtr1       %c\n",  *LongVarPtr1);
```

```
Contents of   CharVar1         a
Dereferencing CharVarPtr1(%c)  a

Contents of   IntVar1          100
Dereferencing IntVarPtr1(%d)   100

Contents of   LongVar1         1100
Dereferencing LongVarPtr1      1100
```

deref1Demo.c

Operator Precedence

- Unary operators `&` and `*`, when used with pointers, have equal precedence with each other and the other unary operators

- Expressions combining them are evaluated from left to right

- Unary operators have higher precedence than the binary operators

```
IntVar2 = *IntVarPtr1 + *&IntVar1;
```

```
int  IntVar1  = 25,  *IntVarPtr1  = &IntVar1;
int  IntVar2  = 100, *IntVarPtr2  = &IntVar2;
```

```
printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Contents of   IntVar2      %d\n",    IntVar2);
printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
printf("Dereferencing IntVarPtr2   %d\n",   *IntVarPtr2);
```

```
Contents of   IntVar1      25
Contents of   IntVar2      100
Dereferencing IntVarPtr1   25
Dereferencing IntVarPtr2   100
```

```
IntVar2 = *IntVarPtr1 + *&IntVar1;
printf("IntVar2 = *IntVarPtr1 + *&IntVar1;\n\n");
```

```
IntVar2 = *IntVarPtr1 + *&IntVar1;
```

deref2Demo.c

IntVar1 = 25

```
IntVar2 = *IntVarPtr1 + *&IntVar1;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Contents of   IntVar2      %d\n",    IntVar2);
printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
printf("Dereferencing IntVarPtr2   %d\n",   *IntVarPtr2);
```

```
Contents of   IntVar1      25
Contents of   IntVar2      50
Dereferencing IntVarPtr1   25
Dereferencing IntVarPtr2   50
```

```
IntVar3 = *&*&*&*&*&*&*&*&*&*&*&IntVar1 * *&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*IntVarPtr1;

printf("Contents of IntVar3 = %d\n", IntVar3);
```

```
Contents of IntVar3 = 625
```

deref2Demo.c

```
IntVar3 = *&*&*&*&*&*&*&*&*&*&*&IntVar1 * *&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*IntVarPtr1;

printf("Contents of IntVar3 = %d\n", IntVar3);
```

Contents of IntVar3 = 625

```
IntVar3 = *&*&*&*&*&*&*&*&*&*&*&IntVar1 * *&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&IntVarPtr1;

deref2Demo.c: In function ‘main’:
deref2Demo.c:30: error: invalid operands to binary *
```

```
IntVar3 = *&*&*&*&*&*&*&*&*&*&*IntVar1 * *&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*IntVarPtr1;

deref2Demo.c: In function ‘main’:
deref2Demo.c:30: error: invalid type argument of ‘unary *’
```

deref2Demo.c

```
int IntVar1 = 66;
int *IntVarPtr1 = &IntVar1;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr1);
printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7ffffa2d1ee4
Contents of   IntVarPtr1   0x7ffffa2d1ee4
Dereferencing IntVarPtr1   66
```

```
IntVarPtr1 = NULL;

printf("Contents of   IntVar1      %d\n",    IntVar1);
printf("Address  of   IntVar1      %p\n",   &IntVar1);
printf("Contents of   IntVarPtr1   %p\n",    IntVarPtr1);
printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
```

```
Contents of   IntVar1      66
Address  of   IntVar1      0x7ffffa2d1ee4
Contents of   IntVarPtr1   (nil)
Segmentation fault
```

nullpointer2Demo.c

```
25              printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
(gdb) step

Program received signal SIGSEGV, Segmentation fault.
0x000000000040064a in main () at nullpointer2Demo.c:25
25              printf("Dereferencing IntVarPtr1   %d\n",   *IntVarPtr1);
(gdb) step

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
```

What is a segmentation fault?

In computing, a **segmentation fault** (often shortened to **segfault**) or access violation is a **fault**, or failure condition, raised by hardware with memory protection, notifying an operating system (OS) the software has attempted to access a restricted area of memory (a memory access violation).

For more details and other common examples of causes of segmentation faults

Segmentation fault – Wikipedia