CMPE-013/L

Introduction to "C" Programming

Max Lichtenstein

A=? Z=?

```
int a = 1;
void main(void){
    int z = a;

    while(a < 10){
        int z = a;
        a++;
        z++;
    }

    printf("a = %d, z = %d", a, z);

}
```

a=10, z=5

a=10, z=10

a=10, z=9

Announcements

- Lab 3 is out
- Lab 1 is graded
- Quiz 1 back today

- Autograder is running, I think?
- Lab2 extension until ~~10:59~~ 5:59am today
  - fix those compiler errors!

- Another office hours change (W 3:30 -> 3:45)

Roadmap

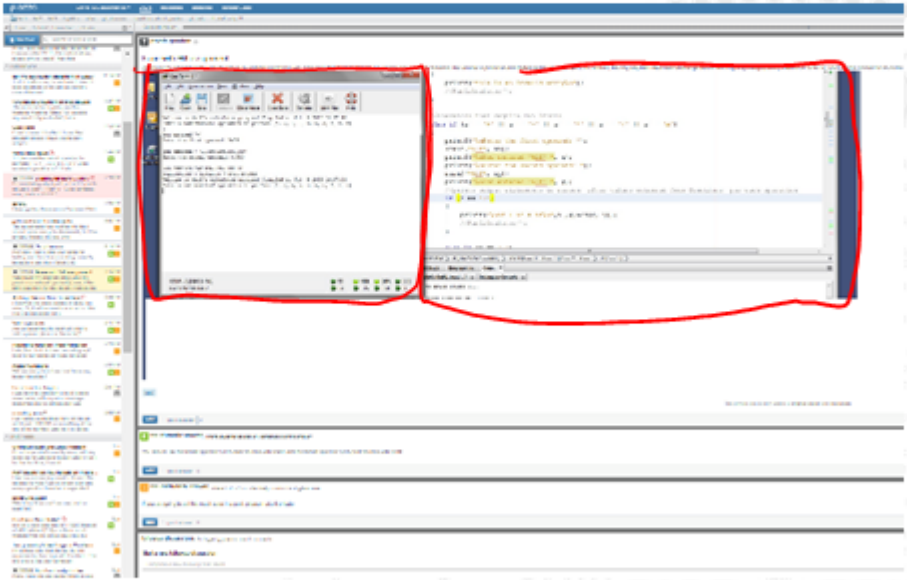
- Feedback re:
  - Piazza ettiquette
  - Goto
- Scope
  - Lifetime
  - Visibility
- 3 ways to get information out of a function
  - Pass by reference
- Pointers
- Arrays
  - And strings, apparently?

Piazza Ettiquette

!!

1) rude

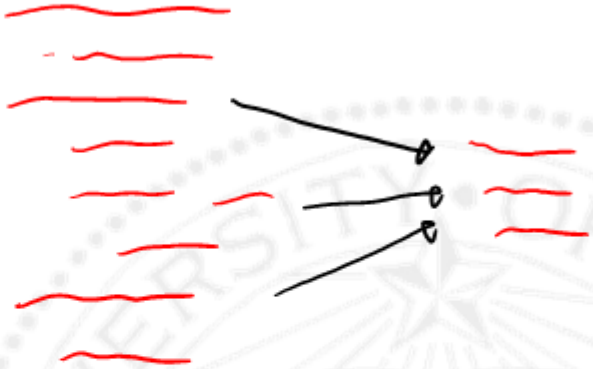
2) bad for you



Minimal Working Examples

Good for:

- you
- us
- your classmates



goto

- For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce.
- ...I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code).

goto

- The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program.

goto

- … our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed.
- For that reason we should … make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

goto

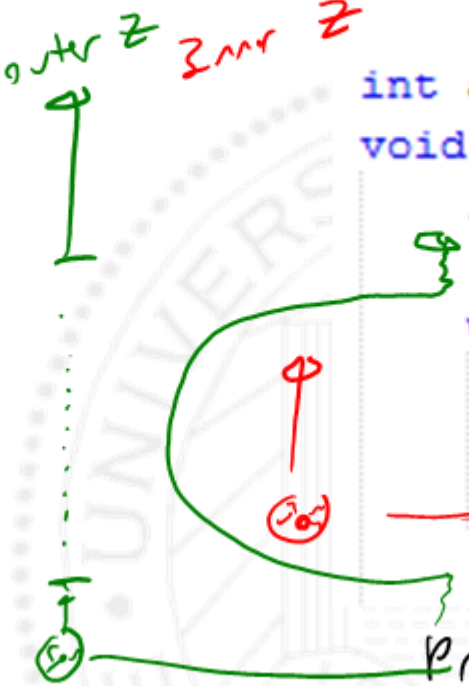
- xkcd.com/292 , Randall Munroe
- **Go To Considered Harmful**, Edsger W. Dijkstra, Letter to *Communications of the ACM* (CACM) vol. 11 no. 3, March 1968, pp. 147-148.
- http://david.tribble.com/text/goto.html

Scope

- 2 aspects:
  - Lifetime of data
    - Applies to variables
  - Visibility of names
    - Applies to anything you can *name*

Scope – Visibility of Name

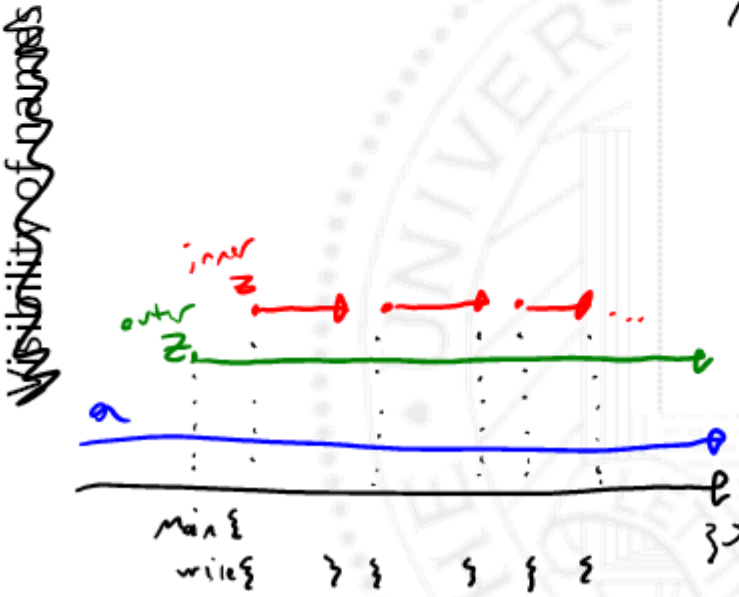
Visibility of names



```
int a = 1;
void main(void){
    int z = a;

    while(a < 10){
        int z = a;
        a++;
        z++;
    }
    printf("z=",z)
```

Scope – Lifetime of data

Lifetime of data



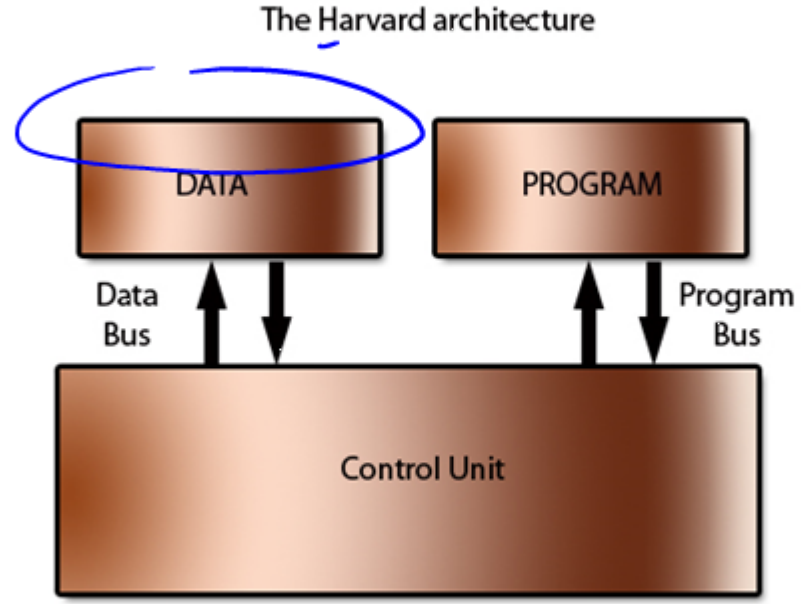

```
int a = 1;
void main(void){
   //int z = a;

    while(a < 10){
        int z = a;
        a++;
        z++;
    }
}
printf(z);
```

Static Memory vs Dynamic Memory

Half

Von Neumann

Von

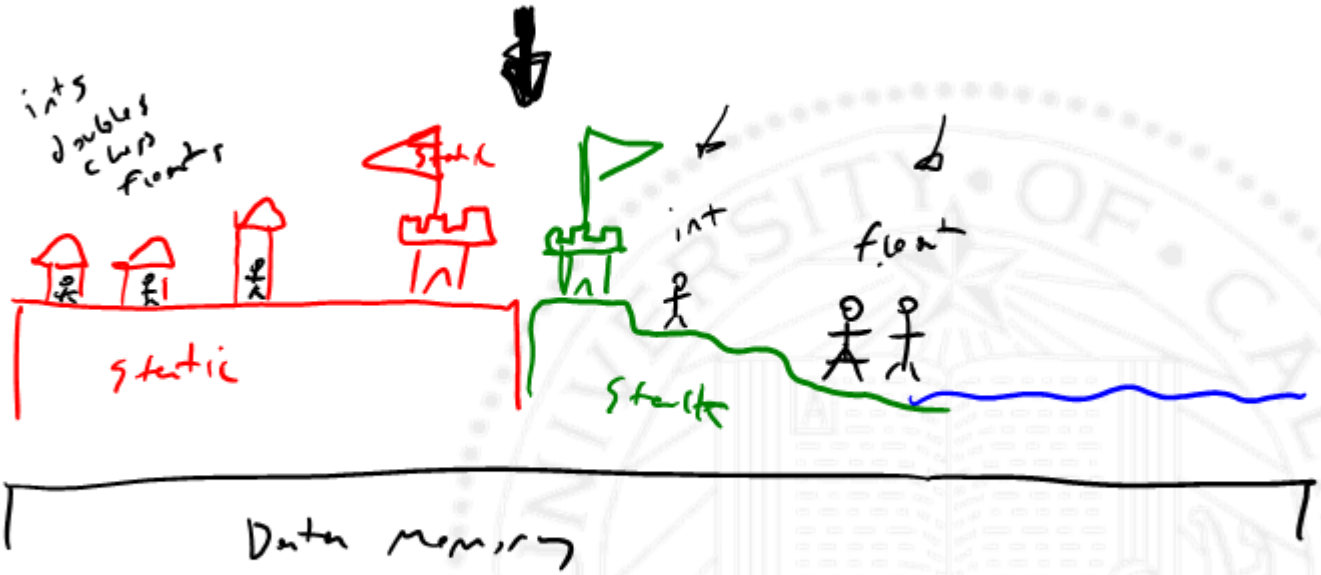


Static Memory vs Dynamic Memory

Variables "live" in data memory

- Not all data memory is identical!
  - Static memory
  - Dynamic memory
    - Heap
    - Stack

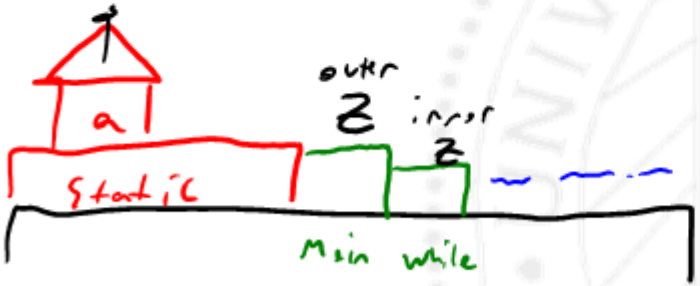
Static vs Stack
A Tale of Two Kingdoms



Max Lichtenstein

Static vs Stack

```
//int a = 1;
 void main(void){
     int z = a;

     while(a < 10){
         int z = a;
         a++;
         z++;
     }
```



Scope

Lifetime of data

Visibility of names

```
int a = 1;
void main(void){
    int z = a;

    while(a < 10){
        int z = a;
        a++;
        z++;
    }
```

Recap: what does it print?

```
int a = 2;
void main(void)
{
    int z=1;

    while (a < 3) {
        int z = 1;
        a++;
        z++;
        printf("inner z = %d\n",z);
    }

    printf("a = %d, outer z = %d\n", a, z);

}
```

inner z = 2
inner z = 2
inner z = 2
a = 3, outer z = 1

3 1
1 1
2 1

How does this work …

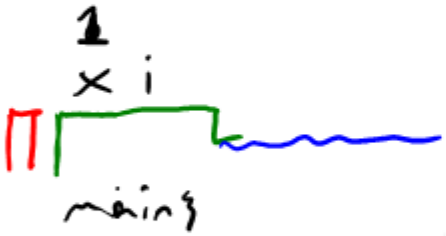
- When using functions?
- Across multiple files?

Scope in functions

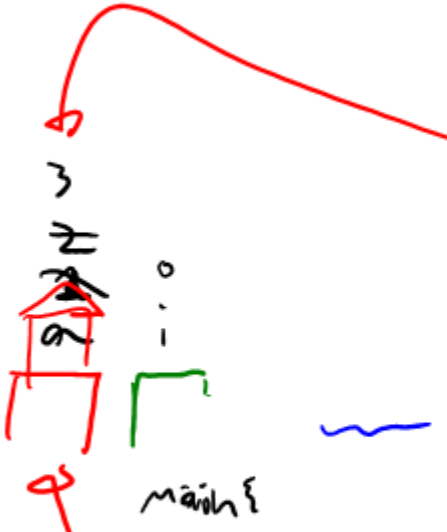
```
void foo(int x){
    x = x + 1;
};

void main(void)
{
    int x = 0;
    int i;
    for( i = 0; i<3; i++){
        foo(x);
    }

    printf("x = %d",x);

}
```

Static vs Stack

```
int foo(int a){
    a = a + 1;
    return a;
}

void main(void)
{
    int x = 0;
    int i;
    for( i = 0; i<3; i++){
     x= foo(x);
    }

    printf("x = %d",x);

}
```



Higher Scope



```
int a = 0;
void foo(void){
    a = a + 1;
};

void main(void)
{
    int i;
    for( i = 0; i<3; i++){
        foo();
    }

    printf("x = %d\n",a);

}
```

Higher Scope

```
int a;
void foo(void){
    a = a + 1;
};

void main(void)
{
    int i;
    for( i = 0; i<3; i++){
        foo();
    }

    printf("x = %d\n",a);

}
```

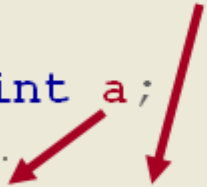
Scope

Variables Declared Within a Function

- Variables declared within a code block are local to that block.

Example

```
int x, y, z;

int Foo(int n)
{
      int a;
      …
      a += n;
}
```



The n refers to the function parameter n

The a refers to the a declared locally within the function body

Scope

Variables Declared Within a Function

- Variables declared within a block are not accessible outside that block

Example

```
int x;
int Foo(int n)
{
      int a;
      return (a += n);
}
int main(void)
{
      x = Foo(5);
      x = a;
}
```

This will generate an error. a may not be accessed outside of the scope where it was declared.

Scope

Variables Declared Within a Function

- Variables declared within a block are not accessible outside that block

Example

```
int x;
int main(void)
{
   {
     int a = 6;
   }

   x = Foo(5);
   x = a;
}
```

This will generate an error. a may not be accessed outside of the scope where it was declared.

Scope

Global versus Local Variables

Example

```
int x = 5;

int Foo(int y)
{
  int z = 1;
  return (x + y + z);
}

int main(void)
{
  int a = 2;
  x = foo(a);
  a = foo(x);
}
```

x can be seen by everybody

foo's local parameter is y
foo's local variable is z
foo cannot see main's a
foo can see x

main's local variable is a
main cannot see foo's y or z
main can see x

Scope

Parameters

- "Overloading" variable names:

n Declared Locally and Globally

```
int n;

int Foo(int n)
{
        …
        y += n;
        …
}
```

local **n** hides global **n**

n Declared Globally Only

```
int n;

int Foo(int x)
{
        …
        y += n;
        …
}
```

A locally defined identifier takes precedence

Max Lichtenstein

Scope

Parameters

Example

```
int n;

int Foo(int n)
{
      y += n;
}

int Bar(int n)
{
      z *= n;
}
```

- Different functions may use the same parameter names
- The function will only use its own parameter by that name

"Static" keyword

Static

```
int a = 1;
int foo(void){
    a++;
    return a;
}

int bar(void){
    a++;
    return a;
}

void main(void)
{
    int i, fb;
    for( i = 0; i<3; i++){
        fb = foo();
        printf("foo = %d\n", fb);
        fb = bar();
        printf("bar = %d\n", fb);
    }
}
```

foo 1
bar 2
foo 3
bar 4
⋮

```
int foo(void){
    static int a = 1;
    a++;
    return a;
}

int bar(void){
    static int a = 1;
    a++;
    return a;
}

void main(void)
{
    int i, fb;
    for( i = 0; i<3; i++){
        fb = foo();
        printf("foo = %d\n", fb);
        fb = bar();
        printf("bar = %d\n", fb);
    }
}
```

foo 1
bar 1
foo 2
bar 2
foo 3
bar 3

Storage Class Specifiers

Static Variables

- Given a permanent address in memory
- Exist for the entire life of the program
  - Created when program starts
  - Destroyed when program ends
- Global variables are always static (cannot be made automatic using **auto**)

```
int x;
int main(void)
{
   ...
```

Global variable is always static

Storage Class Specifiers

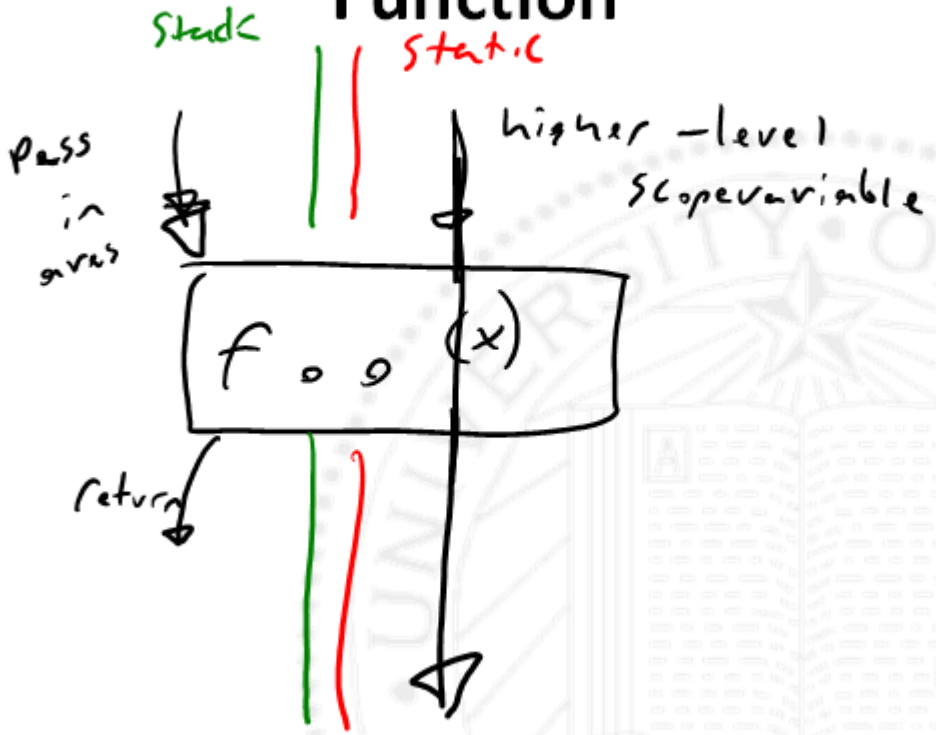
`static` Keyword with Variables

- A variable declared as **`static`** inside a function retains its value between function calls (not destroyed when exiting function)
- Function parameters cannot be **`static`** with some compilers (XC32)

```
int Foo(int x)
{
    static int a = 0;
    ...
    a += x;
    return a;
}
```

a will remember its value from the last time the function was called.
If given an initial value, it is only initialized when first created – not during each function call

2 Ways to get Information out of a Function



Max Lichtenstein

Limitations of pass/return and Global vars

Higher-Scope

- Name conflicts
- hard to visualize

Pass-return

- only 1 value can return
- can only pass in so many values

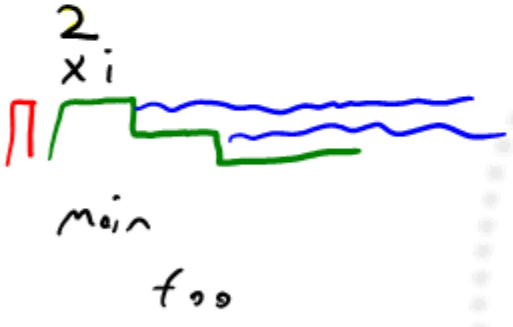
Pass by reference

& = reference operator

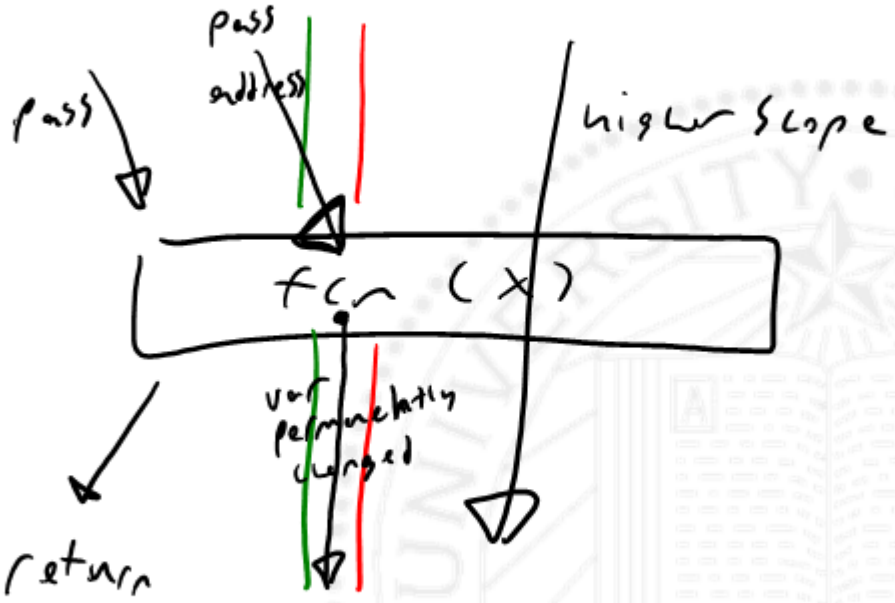
* = de-ref operator

0x100
0x104
0x108

```
void foo(int * a){
    *a = *a + 1;
};

void main(void)
{
    int x = 0;
    int i;
    for (i = 0; i < 3; i++) {
        foo(&x);
    }

    printf("x = %d\n", x);

}
```



3 Ways to get Information out of a Function



Max Lichtenstein

Breaktime

```
void ping(int * a){
    *a = *a + 1;
};

void pong(int b){
    b = b - 1;
}

void main(void){
    int x = 0;
    int i;
    for (i = 0; i < 10; i++) {
        ping(& x);
        pong(& x);
    }

    printf("x = %d\n", x);
}
```

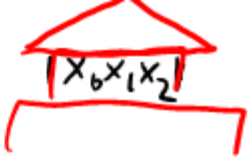
Pointers Recap

Only 4 data types?

- C has 4 data types
- Wait…

"What am I?"

Arrays



Arrays

Definition

Arrays are variables that can store many items of the same type. The individual items known as **elements**, are stored sequentially and are uniquely identified by the array **index** (sometimes called a **subscript**).

- Arrays:
  - May contain any number of elements
  - Elements must be of the same type
  - The index is zero based
  - Array size (number of elements) must be specified at declaration

Arrays

How to Create an Array

Arrays are declared much like ordinary variables:

Syntax

```
type arrayName[size];
```

- *size* refers to the number of elements
- *size* can be a constant OR specified at runtime (c99)

Example

```
int a[10];

char s[25];

char str[x];
```

str[] = {1, 2, 3}

Arrays

How to Initialize an Array at Declaration

Arrays may be initialized with a list when declared:

Syntax

```
type arrayName[size] = {item1, …, itemn};
```

- The items must all match the *type* of the array

Example

```
int a[5] = {10, 20, 30, 40, 50};

char b[5] = {'a', 'b', 'c', 'd', 'e'};
```

Arrays

How to Use an Array

Arrays are accessed like variables, but with an index:

Syntax

```
arrayName[index]
```

- *index* may be a variable or a constant
- The first element in the array has an index of 0
- C does not provide any bounds checking

Example

```
int i, a[10];    // An array that can hold 10 integers

for(i = 0; i < 10; i++) {
    a[i] = 0;    // Initialize all array elements to 0
}
a[4] = 42;       // Set fifth element to 42
```

Arrays

Creating Multidimensional Arrays

Add additional dimensions to an array declaration:

Syntax

```
type arrayName[size1]...[sizen];
```

- Arrays may have any number of dimensions
- Three dimensions tend to be the largest used in common practice

Example

```
int a[10][10];          // 10x10 array for 100 integers

float b[10][10][10];    // 10x10x10 array for 1000 floats
```

Arrays

Initjalizing Multidjmensional Arrays at Declaration

Arrays may be initialized with lists within a list:

Syntax

```
type arrayName[size0]…[sizen] =
                        {{item,…,item},
                                ⋮
                         {item,…,item}};
```

Example

```
char a[3][3] = {{'X', 'O', 'X'},
                {'O', 'O', 'X'},
                {'X', 'X', 'O'}};

int b[2][2][2] = {{{0, 1},{2, 3}},{{4, 5},{6, 7}}};
```

Arrays

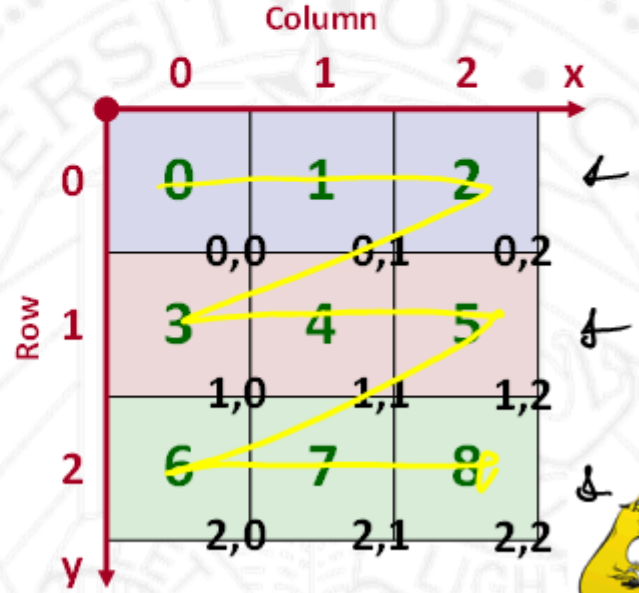
Visualizing 2-Dimensional Arrays

```
int a[3][3] = {{0, 1, 2},
               {3, 4, 5},
               {6, 7, 8}};
```

Row, Column

`a[y][x]`

Row 0	`a[0][0] = 0;`
	`a[0][1] = 1;`
	`a[0][2] = 2;`
Row 1	`a[1][0] = 3;`
	`a[1][1] = 4;`
	`a[1][2] = 5;`
Row 2	`a[2][0] = 6;`
	`a[2][1] = 7;`
	`a[2][2] = 8;`



Arrays Are Pointers

```
a[0][0] = 0;
a[0][1] = 1;
a[0][2] = 2;
a[1][0] = 3;
a[1][1] = 4;
a[1][2] = 5;
a[2][0] = 6;
a[2][1] = 7;
a[2][2] = 8;
```

```
int a[3][3] = {
    {0, 1, 2},
    {3, 4, 5},
    {6, 7, 8}
};
```

```
int * b = a;
printf("b[0] = %d\n", b[0]);
b = b + 1;
printf("*(a + 1) = %d\n", *b);
b = b + 5;
printf("*(a + 6) = %d\n", *b);
```

Arrays

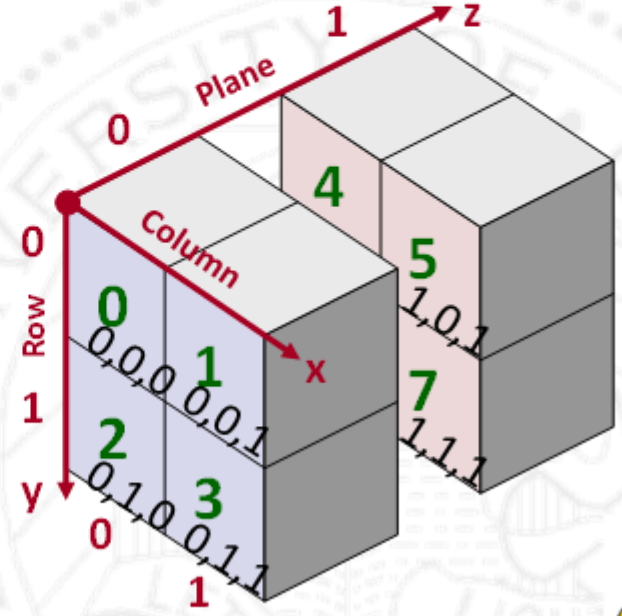
Visualizing 3-Dimensional Arrays

```
int a[2][2][2] = {{{0, 1},{2, 3}},
                  {{4, 5},{6, 7}}};
```

Plane, Row, Column

`a[z][y][x]`

Plane 0	`a[0][0][0] = 0;`
	`a[0][0][1] = 1;`
	`a[0][1][0] = 2;`
	`a[0][1][1] = 3;`
Plane 1	`a[1][0][0] = 4;`
	`a[1][0][1] = 5;`
	`a[1][1][0] = 6;`
	`a[1][1][1] = 7;`



Strings

```
void main(void)
{
  // char hello_array[10] = "Hello\n";

  // char world_array[10] = {'W','o','r','l','d','\n'};

  // printf(hello_array);
  // printf(world_array);
}
```

What does it print?

```
void main(void)
{
    char hwString[15] = "Hello World!";

    printf(hwString + 4);

    char * gbString[15] = "Goodbye World :(";

    printf(gbString + 4);

}
```