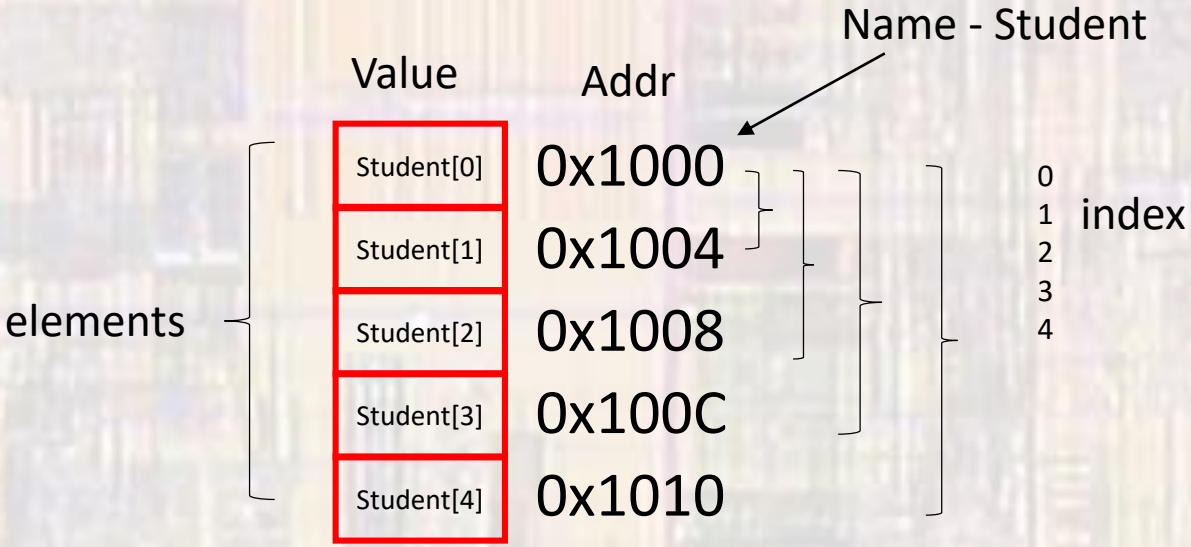
C Review II

Last updated 5/17/19

C Review II

- Array notation
  - In memory
  - Name is actually a pointer
  - Index is the offset from the name
    - not an address



C Review

- Arrays in C

  Initialization

  type arrayName[arraySize] = {comma separated list};

```
int myArray[5] = {5,4,3,2,1};          // basic
```


```
int myArray[5] = {5,4};        // partial initialization
```


```
                               // others are set to 0
int myArray[ ] = {5,4,3,2,1};          // size is taken from
```


```
                               // initialization values
int myArray[5] = {0};          // all set to 0
```

C Review II

- Arrays in C
  - Accessing elements

myArray


```
foo = myArray[3];          // foo = 2
foo = myArray[foo];        // foo = 3
```

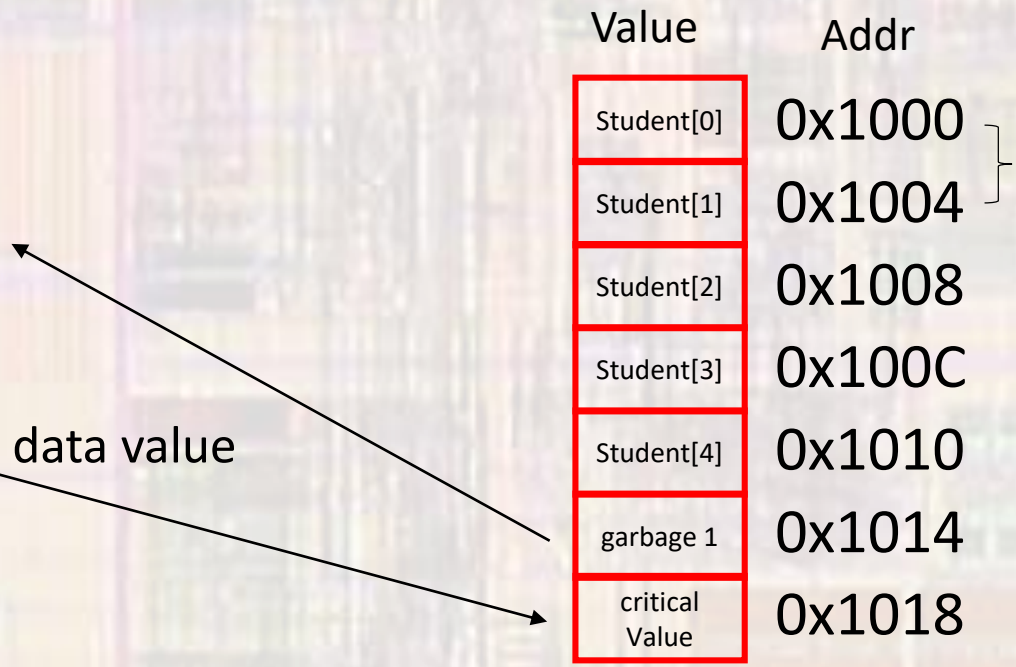
```
myArray[0] = 0;
```


```
myArray[foo + 1] = 6;
```

C Review II

- Array Index Range Checking
  - C does NOT check array index ranges

```
int Student[5];
…

foo = Student[5];
    sets foo = garbage

Student[6] = 12;
     overwrites critical data value
```



C Review II

- Passing array values
  - Passing array values works just like any other value

```
void fun1 (int zoo);
void fun2 (float* soo);

fun1(foo);                  // passes the value of foo to function
                            // fun1
fun1(myArray[3]);           // passes the value of myArray[3]
                            // to function fun1

fun2(&boo);                 // passes a pointer to boo (the address)
                            // to function fun2
fun2(&myFloatArray[3]);     // passes a pointer to myFoatArray
                            // element 3 (the address)
                            // to function fun2
```

C Review II

- Passing array values
  - Passing the whole array
    - If we pass all the elements of a large array to multiple functions we use up a lot of data memory
    - Instead we pass the address of the array (always use the pointer approach)
  - Remember – the name of the array is already a pointer to the beginning of the array

```
void fun3(const int ary[ ]);    // the array notation replaces the int*
                                // to tell the compiler it is expecting a
                                // pointer – use const when you do not
                                // want the function to modify the array
…
fun3(myArray);                  // function call does not have the &
                                // since the array name is already an
                                // address
```

C Review II

- Passing array values
  - Passing the whole array
    - Since we are passing the array as a pointer

      AND
    - The array already uses a pointer offset notation
      - The array is accessed using regular notation, dereferencing is not necessary

```
void fun3(int ary[ ]){
…
   for (i = 0, i<5, i++){
      sum = sum + ary[i];
   }
…
}
```

C Review II

- Structure
  - Collection of related elements
  - Not necessarily the same type
  - Sharing a single name

  - Elemental unit is called a Field
  - Looks just like a variable
    - has a type
    - takes up memory space
    - can be assigned values
    - can be read
  - Only difference is that a Field is part of a Structure

C Review II

- Structure
  - TypeDef definition
    - defines a new type
    - new elements of the structure type can be created via declaration

```
typedef struct{
   char ID[10];
   char name[26];
   float GPA;
} STUDENT;                          // type definition name
…
int foo;
STUDENT student1;
STUDENT student2;
```

C Review II

- Structure
  - Access

    structure.field

    ```
    student2.GPA = 2.5;
    if(student1.GPA >= 3.5){ … }
    ```

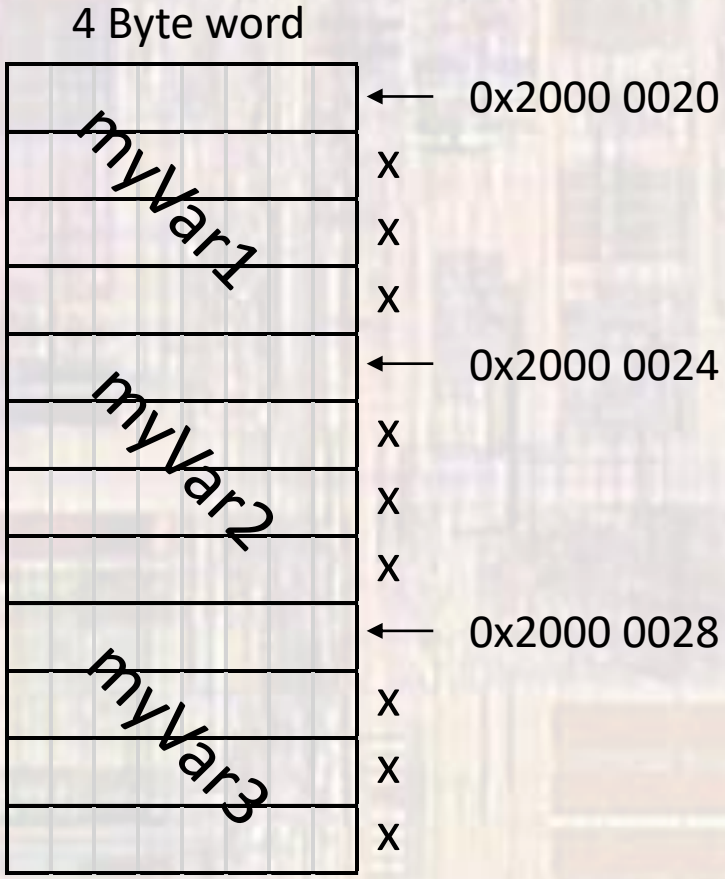
    - Access via Pointers

      ```
      STUDENT* student_ptr;        // define a pointer of STUDENT type
      student_ptr = &student1;     // student_ptr now points to student1

      (*student_ptr).GPA = 3.66;   // dereference
      student_ptr->GPA = 3.66;     // indirect selection
      ```

C Review II

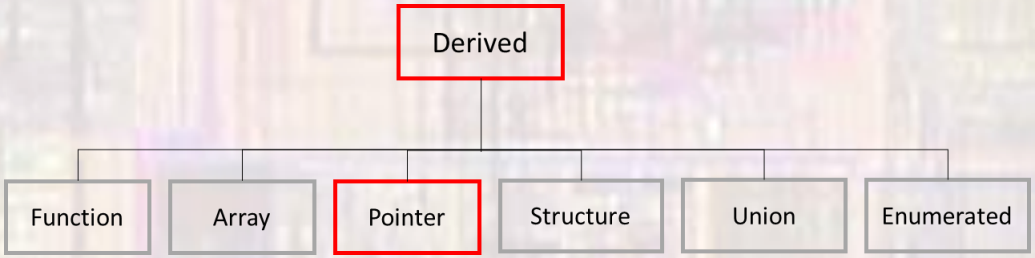
- Pointer
  - Review variables in memory

  - address for myVar1
    0x2000 0020
  - address for myVar2
    0x2000 0024
  - address for myVar3
    0x2000 0028



C Review II

- Pointer



  - A special Type
  - A variable that holds the memory location of another variable
  - Holds an address – in our case 32 bits
  - Each pointer must be tied to a specific data type
    - int, float, char, …

C Review II

Precedence	Operator	Description	Associativity
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	

- Pointer
  - To find the memory location of a variable use the "address of" operator: &

&myVar1 → 0x2000 0020

&myVar2 → 0x2000 0024

&myVar3 → 0x2000 0028

C Review II

Precedence	Operator	Description	Associativity
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	

- Pointer
  - To declare a pointer variable
    - follow the type declaration with a *

```
int* myVar1_ptr;
// declare a pointer variable with name myVar1_ptr
// that points to an integer variable

float* myVar2_ptr;
// declare a pointer variable with name myVar2_ptr
// that points to a float variable
```

C Review II

Precedence	Operator	Description	Associativity
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	

- Pointer
  - To determine the value of a variable pointed to by a pointer variable
    - precede the pointer variable with * (dereference operator)

```
*myVar1_ptr;
// provides the value held in the memory location
// pointed to by myVar1_ptr – as an int

*myVar2_ptr;
// provides the value held in the memory location
// pointed to by myVar2_ptr – as a float
```

C Review II

- Pointer

```
int var1 = 5;                   // stored in memory location 0x2000 1010
float var2 = 12.0;              // stored in memory location 0x2000 0220
int foo1;
float foo2;

int* ptr1;                      // define a pointer to a variable of type int
float* ptr2;                    // define a pointer to a variable of type float

ptr1 = &var1;                   // set ptr1 to 0x2000 1010
ptr2 = &var2;                   // set ptr2 to 0x2000 0220

foo1 = *ptr1;                   // set foo1 to 5
foo2 = *ptr2;                   // set foo2 to 12.0
```

C Review II

- Pointers and functions
  - Pointers allow us to use called functions to change values in the calling function
  - Instead of passing variables in the parameter list (remember copies are made and then destroyed) we can pass pointers
  - Pointers allow us to modify the passed variables by memory reference

C Review II

- Pointers and functions
  - Declaration
    - Indicate that a pointer is being passed in the Formal Parameter List

```
void update_acct(float* balance_ptr, float int_rate);
```

C Review II

- Pointers and functions
  - Definition
    - Indicate that a pointer is being passed in the Formal Parameter List
    - Operate on the variables pointed to by the pointers via the dereference operator

```
void update_acct(float* balance_ptr, float int_rate){
    *balance_ptr += *balance_ptr * int_rate;
     return;
}
```

C Review II

- Pointers and functions
  - Usage
    - Pass a pointer variable in the Actual Parameter List
    - Pass the address to the variable in the Actual Parameter List

```
int main(void){
  float checking;
  float savings;
  float int_rate;
  float* check_ptr = &checking;          // ptr variable to a float variable
   …
  update_acct(check_ptr, int_rate);
  update_acct(&savings, int_rate);
  return 0;
}
```

C Review II

- Pointers and functions

```
int main(void){
  float checking;
  float savings;
  float int_rate;
  float* check_ptr = &checking;          // ptr variable to a float variable
   …
  clear_acct(check_ptr);
  clear_acct(&savings);
  return 0;
}

void clear_acct(float* balance_ptr){
  *balance_ptr =0.0;
  return;
}
```

C Review II

- Pointers and functions

```
int main(void){
   int a;
   int b;
…
   swap(&a, &b);
   return 0;
}

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

C Review II

- Pointers and functions

```
int main(void){
   int num;
   int den;
   int quo;
   int rem;
…
   divide(num, den, &quo, &rem);
   return 0;
}

void divide(int num, int den, int* quo, int* rem){
   *quo = num / den;
   *rem = num % den;
   return;
}
```

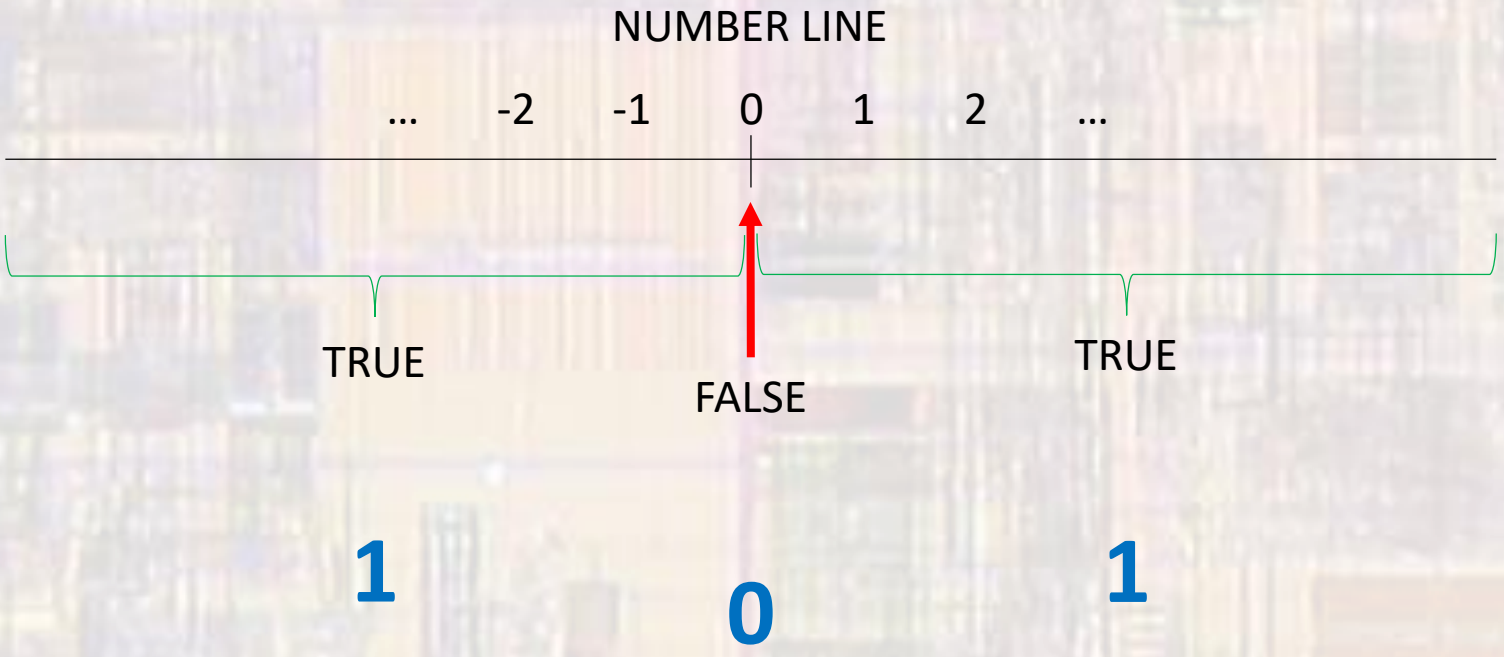
C Review II

- Pointers and functions

```
*  circle.c
     Calculate the area and circumference of a circle
     Created by tj
     Rev 0, 11/15/16
 */

#include <stdio.h>
#define PI 3.14159

int main(void){
  setbuf(stdout, NULL);  // disable buffering

//  Local variables
  float radius;
  float circumference;
  float area;

  //   Get input for radius
  printf("Please enter a value for radius: ");
  scanf("%f", &radius);

  //   Calculate circumference and area
  circumference = 2 * PI * radius;
  area = PI * radius * radius;

  // Output results
  printf("Circumference = %f\n", circumference);
  printf("Area = %f\n", area);

  return 0;
}
```

C Review II

- Logic in C



C Review II

- Logical Operators

```
int foo = 1;
float boo = -2.3;
char soo = ‘a’;
```

!foo	0
!boo	0
foo && boo	1
!foo && soo	0
boo || soo	1
foo || !soo	1
!(foo && !boo)	1

C Review II

- Evaluating Logical Expressions
  - Short circuit evaluation

foo || boo → stop evaluating if foo is true

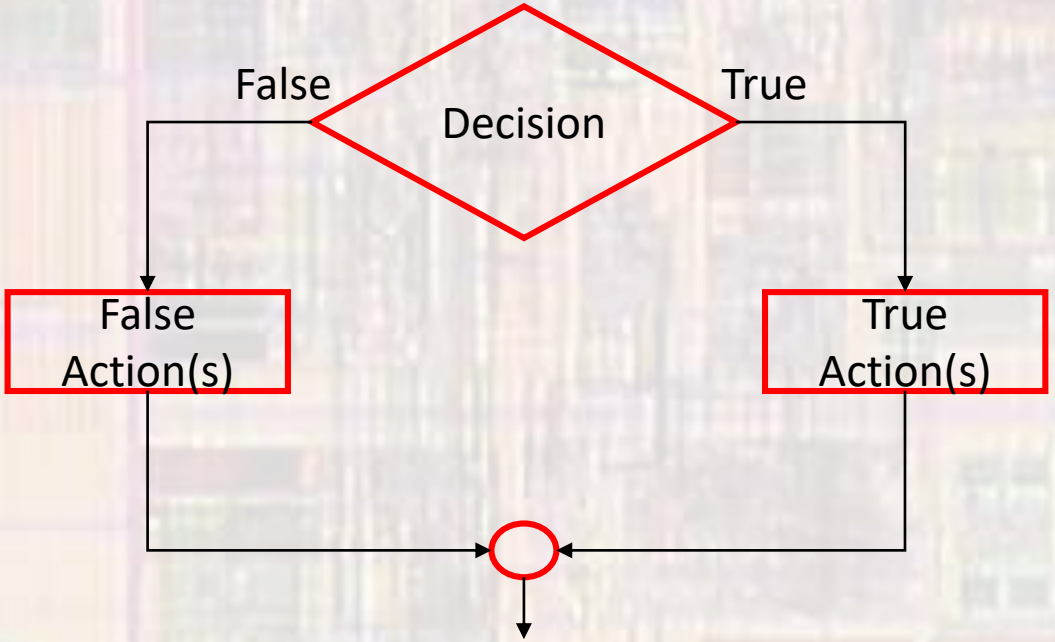
foo || boo++ → boo never gets incremented if foo is true

True || anything → true
False && anything → false

C Review II

- Two way decisions
  - if … else

```
…
if (expression)
    statement 1
else
    statement 2
…
```



C Review II

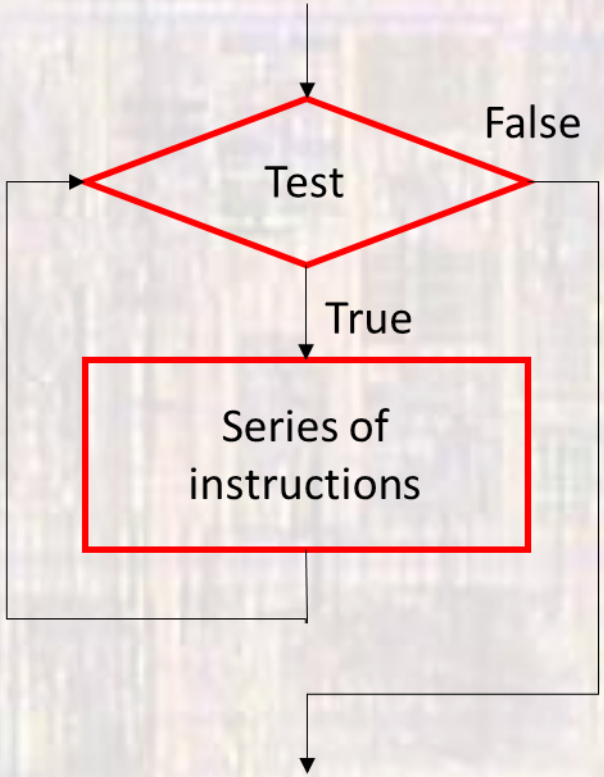
- Two way decisions

```
if (j == 1)
        a++;
else
        a--;


if (j <= 5){
        a++;                    // compound statement
        b = a + 3;
}
else
        a--;
```

C Review II

- While loop

```
while(expression)
  statement;

while(expression){
  statements;
}
```



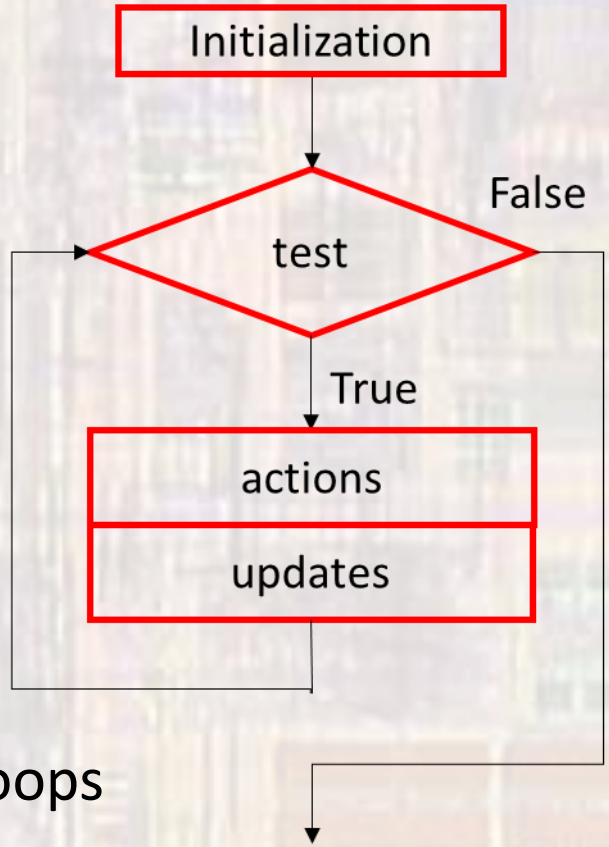
- execute statements while expression is true

C Review II

- For loop

```
for(exp1; exp2; exp3)
  statement;
```

exp1 -> initialization
exp2 -> test
exp3 -> update



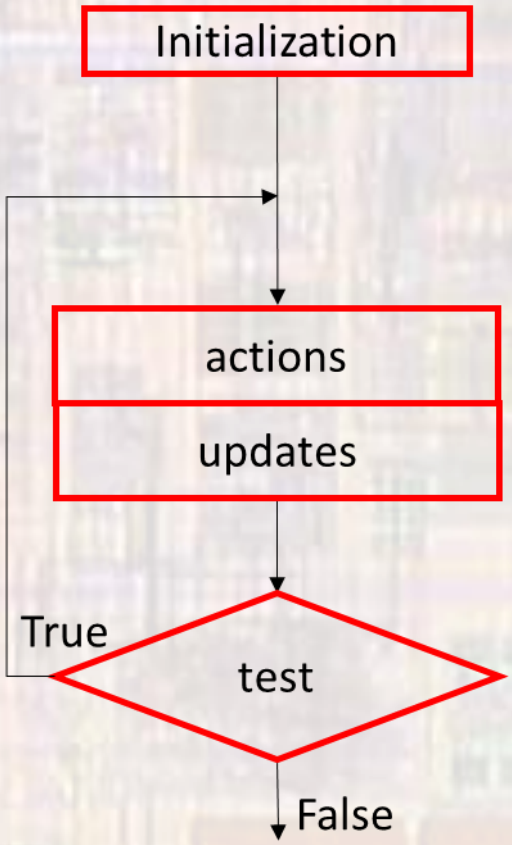
- Typically used in counter controlled loops

C Review II

- Do-While Loop

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

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



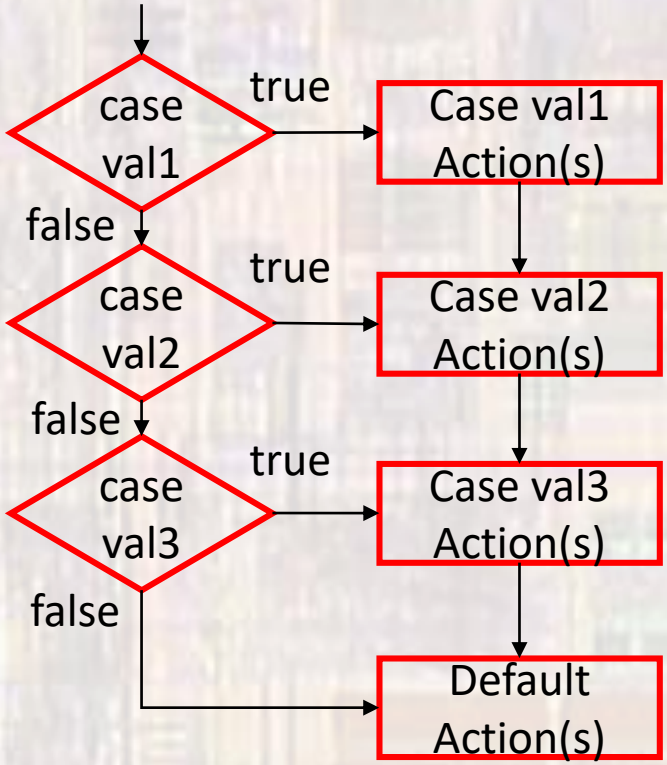
- Often used in data validation situations

C Review II

- Switch Statement
  - If… else allows a 2 way decision
  - Switch allows for n-way decisions

```
…
switch(variable){
   case val1: statement;
   case val2: statement;
   case val3: statement;
   default: statement;
}
```

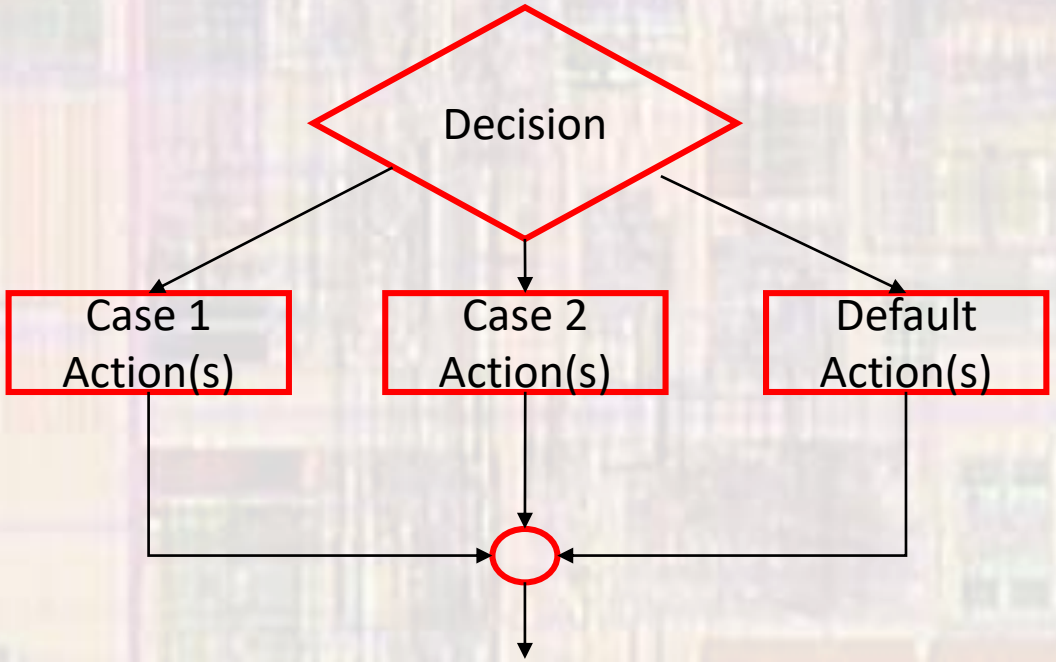
variable must be an integral value



C Review II

- Switch Statement
  - Switch with break

```
switch(variable){
 case val1:     statement;
                statement;
                break;
 case val1:     statement;
                statement;
                break;
 default:       statement;
                statement;
                break;
}
```



C Review II

- else if…
  - Actually not a new command
  - Special case of nested if

```
if(expr1){
  …
}
else if(expr2){
  …
}
else if(expr3){
  …
}
else{
  …
}
```

```
if(expr1){
  …
}
else
  if(expr2){
  …
  }
  else
    if(expr3){
    …
    }
    else{

    }
  }
}
```

- exprX should be different cases of the same test

C Review II

- Instructions and Memory

C code
a = b + c

Assembly Code		Machine Code
ldr	r0, [sp, #4]	0x9801
ldr	r1, [sp]	0x9900
adds	r0, r0, r1	0x1840
str	r0, [sp, #8]	0x9002

Why are these 16 bits?

Memory

	Memory Address
x x x x x x x x	
x	
x	
x	
0 0 0 0 0 0 0 1	0x0000 336A
1 0 0 1 1 0 0 0	
0 0 0 0 0 0 0 0	0x0000 336C
1 0 0 1 1 0 0 1	
0 1 0 0 0 0 0 0	0x0000 336E
0 0 0 1 1 0 0 0	
0 0 0 0 0 0 1 0	0x0000 3370
1 0 0 1 0 0 0 0	

C Review II

- Variables and Memory

```
uint32_t   a = 1;      uint32:   2000ffd8
char       e = 'a';    char:     2000ffdc
uint32_t   aa = 1;     uint32:   2000ffe0  ← word aligned
uint16_t   b = 2;      uint16:   2000ffe4
char       f = 'a';    char:     2000ffe6
uint16_t   bb = 2;     uint16:   2000ffe8  ← word aligned
uint8_t    c = 3;      uint8:    2000ffea
uint8_t    cc = 3;     uint8:    2000ffeb
float      d = 1.123;  float:    2000ffec  ← word aligned
char       g = 'a';    char:     2000fff0
float      dd = 1.123; float:    2000fff4  ← word aligned
char       h = 'a';    char:     2000fff8
char       ee = 'a';   char:     2000fff9
```

0x2000ffd8 <Memory Rendering 3>

8-Bit Hex - C Style

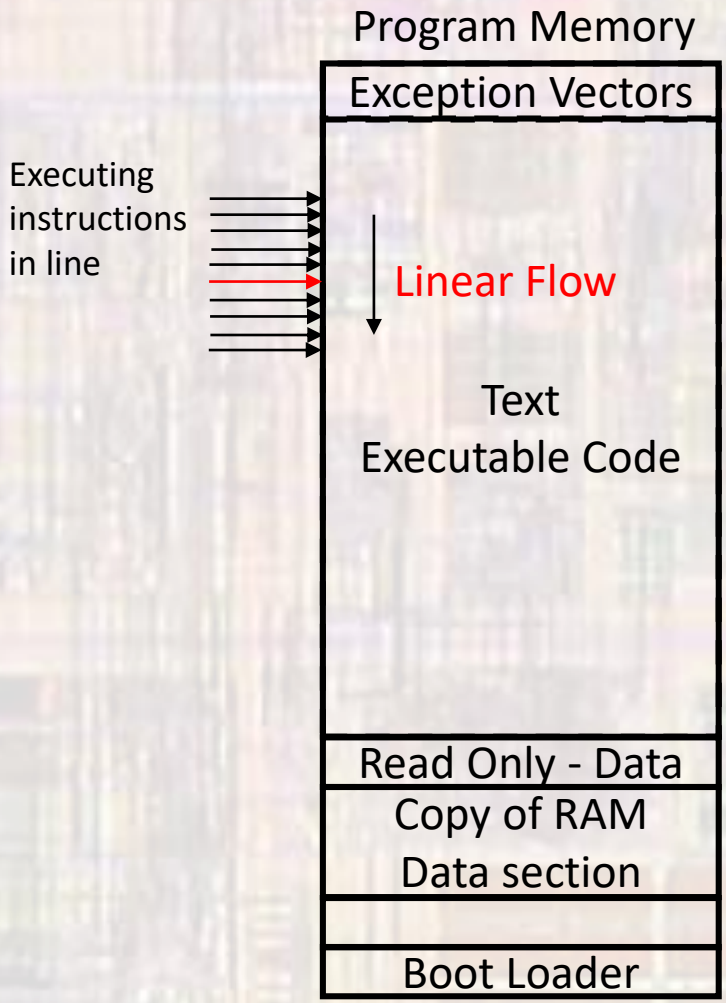
```
0x2000FFD8  0x01 0x00 0x00 0x00 0x61 0x37 0x00 0x00
0x2000FFE0  0x01 0x00 0x00 0x00 0x02 0x00 0x61 0x00
0x2000FFE8  0x02 0x00 0x03 0x03 0x77 0xBE 0x8F 0x3F
0x2000FFF0  0x61 0x00 0x00 0x00 0x77 0xBE 0x8F 0x3F
0x2000FFF8  0x61 0x61 0x01 0x20 0x3F 0x33 0x00 0x00
0x20010000  -- -- -- -- -- -- -- --
```

2000 FFD8	01	2000 FFEC	77
	00		BE
	00		8F
	00		3F
2000 FFDC	61	2000 FFF0	61
2000 FFE0	01	2000 FFF4	77
	00		BE
	00		8F
	00		3F
2000 FFE4	02	2000 FFF8	61
	00	2000 FFF9	61
2000 FFE6	61		
2000 FFE8	02		
	00		
2000 FFEA	03		
2000 FFEB	03		

C Review II

- Linear Flow
  - Series of instructions
  - No decisions → no change in program flow

```
foo = a + b;
soo = foo++;
a = foo + soo;
```



C Review II

- Conditional Flow
  - Series of instructions
  - Decisions → change in program flow

```
if (a==b){
   foo = a + b;
   soo = foo++;
} else {
   a = foo + soo;
} // end if
```