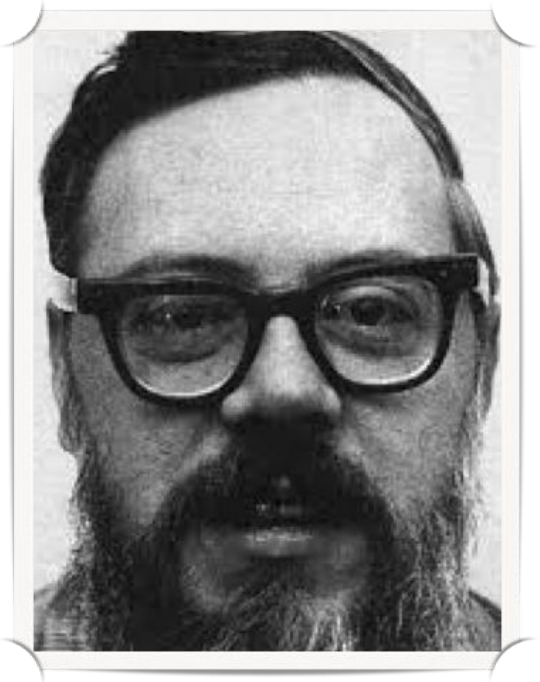
Introduction to C

CS 580U Fall 2017

History of C

- Created by Dennis Ritchie at Bell Laboratories in the early 1970s
- Developed for Unix, traditionally used for systems programming
- ANSI C - Standardized in 1989 by ANSI (American National Standards Institute)
- What C offers
  - Portability (kind of)
  - Performance (still today)



Why C?

- Direct relationship with Assembly
- Concise Syntax
- Portability
- Type Checking

Compilation: Translating High Level Languages

- Interpretation
  - interpreter = program that executes program statements generally one line/command at a time with limited preprocessing
    - easy to debug, make changes
      - languages: Ruby, Python, Java
- Compilation
  - translates statements into machine language but does not execute, allowing for optimization
    - change requires recompilation
      - languages: C, C++, Java

Compilation OR Interpretation

- Consider the following algorithm:
  - Get W from the keyboard.
    X=W+W
    Y=X+X
    Z=Y+Y
    Print Z to screen.
    - If interpreting, how many arithmetic operations occur?
    - If compiling, we can analyze the entire program and possibly reduce the number of operations. Can we simplify the above algorithm to use a single arithmetic operation?
      - Z = X*8

Compilation

- C is a compiled language
  - A program, run from the console, reads the code you write and converts it into machine language that can be read by the computer
  - A compiled language gets converted all in one go
    - The Good:
      - Optimization
      - Many errors are found during ‘compilation’, it’s much faster
    - The Bad:
      - requires more static programming techniques

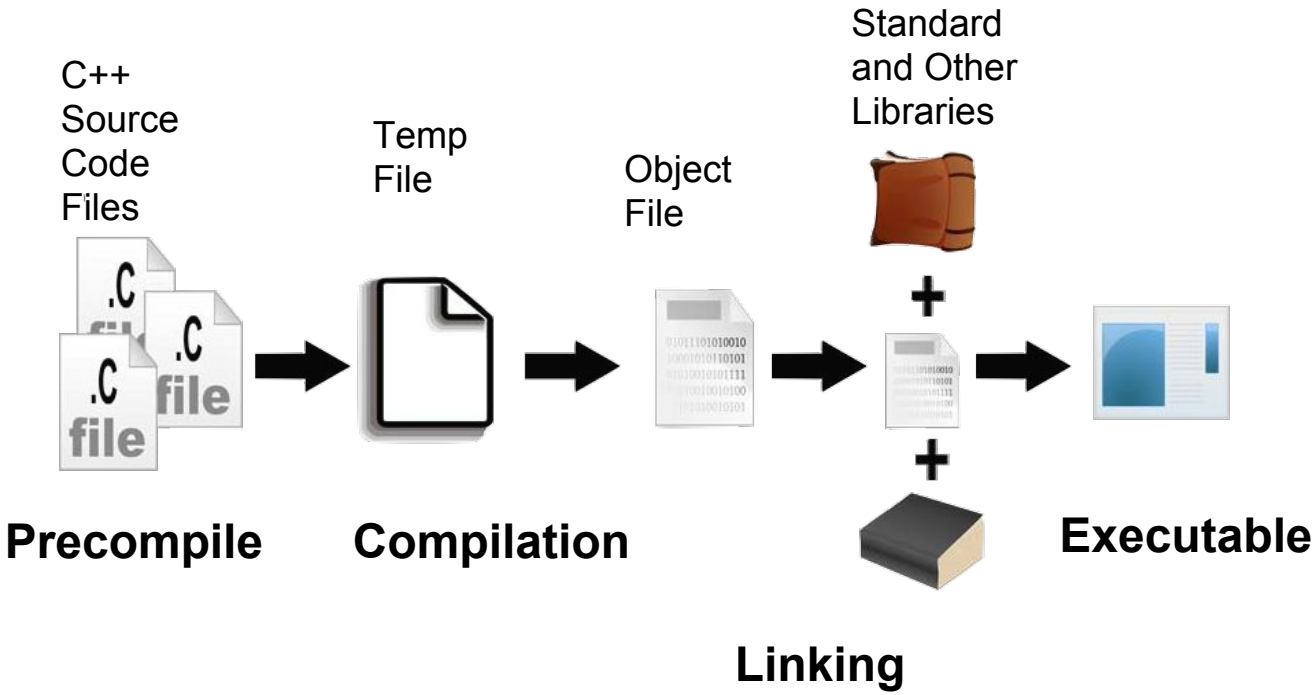
Compiler

- We will use the GNU compiler, gcc
  - gcc is a program, called from the command line, that compiles source code
- Syntax:
  - gcc <myfile>
    - produces an executable called a.out
- Example
  - gcc source.c
- Run your program with ./
  - ./a.out

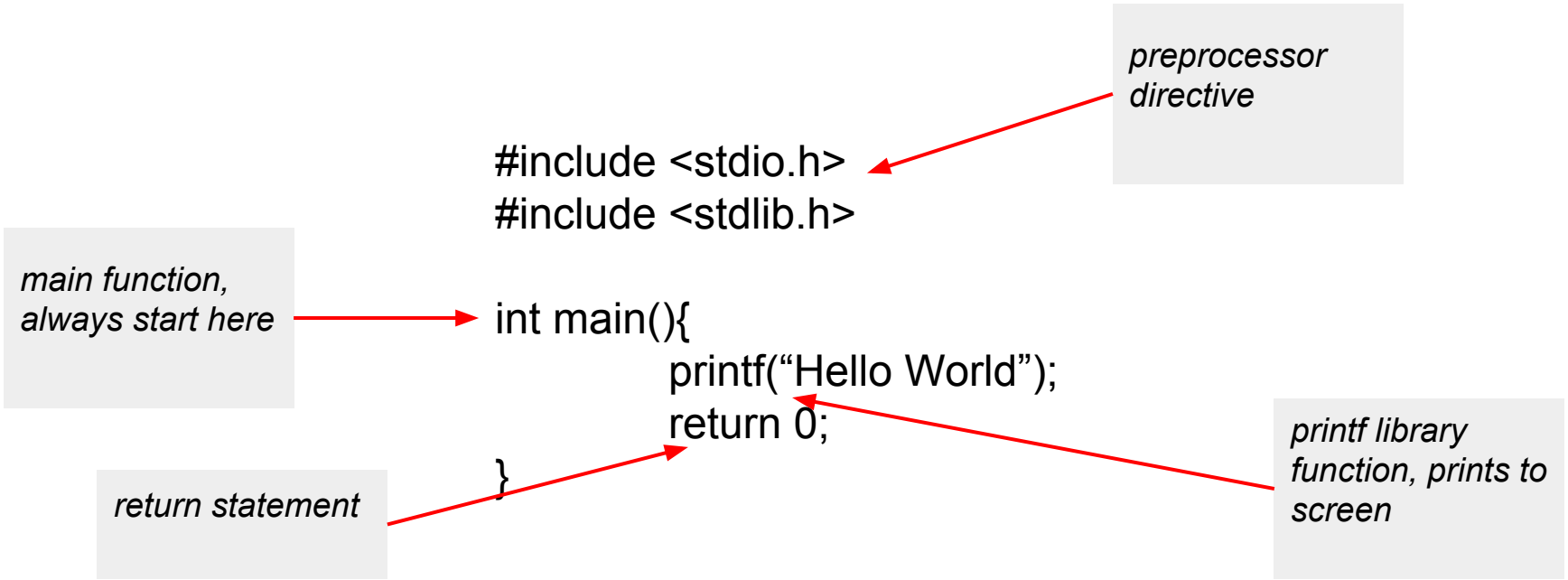
Libraries

- Pre-written libraries eliminate (some) code repetition
  - Essentially, you can load someone else code into your program and it’s not even cheating
- Abstract away some complexity
  - Such as memory mapped IO
- Simplifies code
  - Load libraries with #include <library>
    - #include <stdio.h>

An Overview of Compilation



Basic C program



C Notes

- C is…
  - whitespace insensitive
    - x = 2 + y is that same as x=2+y is the same as:
      x
      =2
      +y
  - Case sensitive
    - x does not equal X
    - ‘hello’ does not equal ‘HELLO’

Comments

- // -- Single line
  - //answer to the universe
    int answer = 42;
- /* */ -- Multiple line
  - /*
    Haikus are easy
    But sometimes they
    don’t make sense
    Refrigerator
    */
    char answer = ‘B’;

Formatted Printing

- You need to include system libraries that contain subroutines for standard input and output
  - including the "header file" called stdio.h which has the printf subroutine.
- printf takes two parameters
  - Format string
  - Variable list

printf()

- Printf sends formatted data to stdout
  - The standard output for your machine, usually the console
- A format string formats the output based on the data type with format specifiers
  - Each type has its own format specifier
  - Some characters have their own format specifier
    - Newline = \n
    - Tab = \t
    - “ = \“

Printf examples

- Print an integer
  - `printf(“%d”, 2);`
- Print a floating point number
  - `printf(“%f”, 2.5);`
- Print an integer and a string
  - `printf(“%d %s”, 42, “and bring a towel”);`
    - You can print multiple values in a single printf statement as long as their order matches the order of the format string

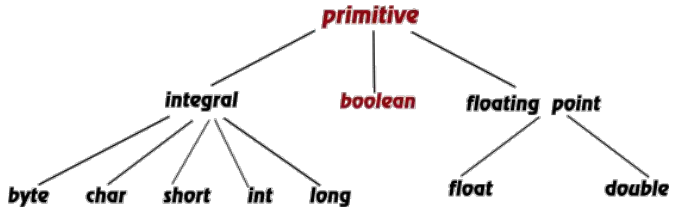
C Variables

- Variables names in C must follow the following rules:
  - Composed of letters, digits, and underscore
  - Begin with a letter or Underscore (not digits)
  - Case Sensitive
  - Must have a data type

Data Types

Each variable has a *type*, which tells the compiler how the data is to be interpreted (and how much space it needs, etc.)

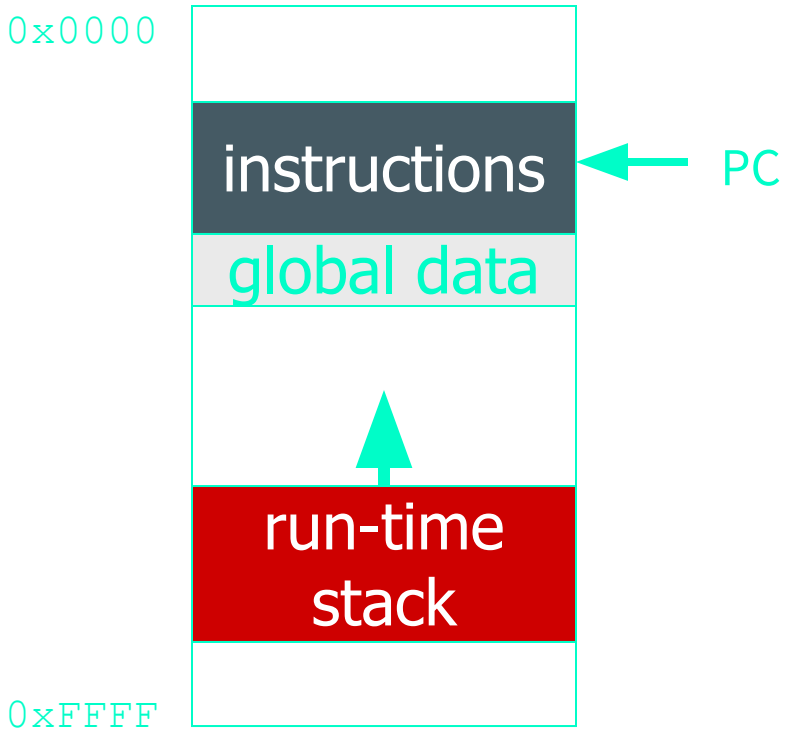
- C has (technically) unlimited Data types
- C has 6 primitive (built in) types



• Integers (whole numbers) ○ int ○ short ○ long ○ unsigned int	• Floating point (real numbers) ○ float ○ double	• Characters ○ char

Allocating Space for Variables

- What happens when you run a program?
  - OS allocates memory for your program
- All variable data is stored within this allocated segment



Declaration

- When you create a variable, you must declare it first.
  - Declaration prepares a memory location for the data
    - DOES NOT contain nothing.
      - Memory retains whatever garbage was previously in that memory location.
- You must declare a variable as a type
  - You can declare multiple variables of the same type on the same line
    - `int x, y, z;`

Initialization

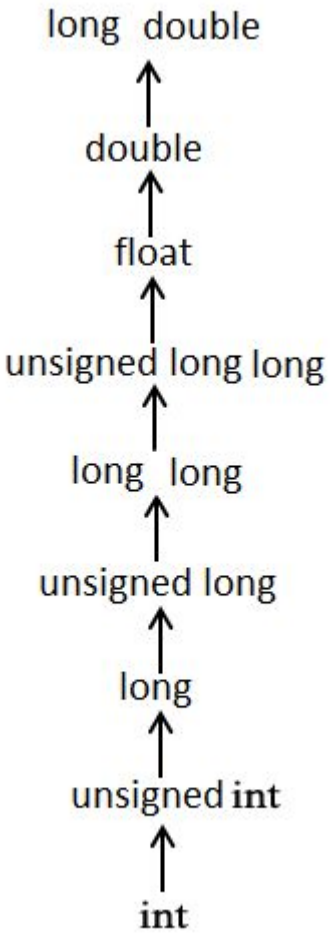
- Assigns a value to a variable
  - use ‘=’ to assign a value
    - Example: char letter = ‘z’;
      - use single quotes around letters
- Overrides existing value
  - There is always an existing value because declaring does not ‘clear’ memory
- Type of a variable cannot be changed
  - Even if assigned another type

Casting

- What if you you want to assign the value of a variable to another variable of a different type?
  - You must cast it.
- Casting forces the compiler to interpret the binary as a different type
  - `char c = ‘A’;`
    `int x = (int) c;`
    - Syntax <type> new_var = <type> old_var;

Logical vs Literal conversion

- Type promotion enables logical conversions for certain type conversions
  - implicit
    - Performed by the compiler to create equivalent values
  - explicit
    - The raw bit representation of the source is copied verbatim, and it is re-interpreted according to the destination type.



Problems with Casting

- This does not change the type of the old variable (or memory location)
- When casting from a larger type (like an int) to a smaller type (like a char), you may lose data.
  - int = 32 bits, char = 8 bits
    - C trusts you to know what you are doing
- Example: char c = (char) 256;
  - results in tab character

Arithmetic Expression

- If mixed types, smaller type is "promoted" to larger.
  - x + 4.3
    - if x is int, converted to double and result is double
- Integer division -- fraction is dropped.
  - x/3
    - if x is int and x=5, result is 1 (not 1.666666...)
- Modulo -- result is remainder.
  - x%3
    - if x is int and x=5, result is 2.

Special operators: ++ and --

- Changes value of variable before (or after) its value is used in an expression.
- Symbol Operation Usage
  - ++ postincrement x++
  - -- postdecrement x--
  - ++ preincrement ++x
  - -- predecrement --x 3
    - Pre: Increment/decrement variable before using its value.
    - Post: Increment/decrement variable after using its value.

Boolean Expressions

Expressions that resolve to True or False

==	equal to	&&	AND
!=	not equal to	II	OR
<	less than		
>	greater than		
<=	less than or equal to		
>=	greater than or equal to		

In C:

- 0 == FALSE
- Anything other than 0 == TRUE

Arrays

- An array is a list of items that are all the same type
- It is a static entity
  - same size throughout program
  - size is determined at compile time

4	36	14	1	22

Declaration of Arrays

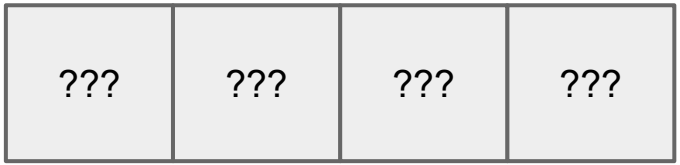
- When declaring arrays, specify
  - Variable Name
  - Type of array
  - Number of elements
    - `type array_name[ num_elements ];`
  - Examples:
    - `int c[ 10 ];`
    - `float my_array[ 3284 ];`

Definition of Arrays

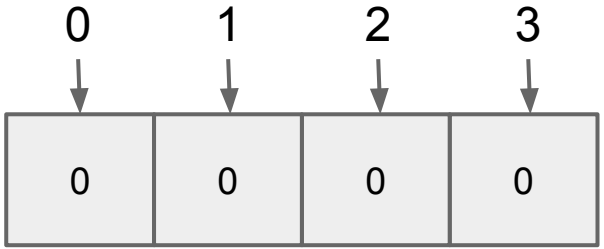
- Several ways to initialize
  - `int n[ 5 ] = { 1, 2, 3, 4, 5 };`
  - If not enough initializers, rightmost elements become 0
    - `int n[ 5 ] = { 0 }`
    - All elements 0, e.g. n = [0][0][0][0][0]
  - If too many a syntax error is produced syntax error

Refer to an element

- `int list[] =`

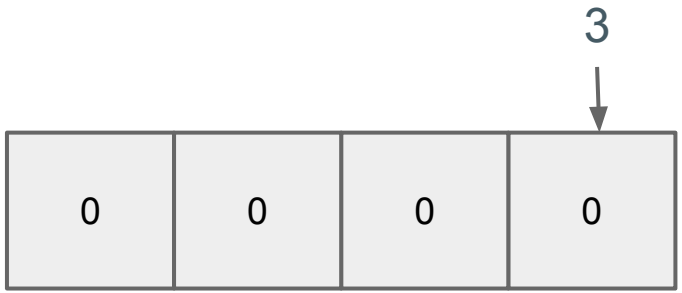


- The individual elements of an array are assigned unique subscripts. These subscripts are used to access the elements.
- Arrays are 0 indexed



Refer to an element

- Referencing an array element requires:
  - Array name
  - Position number
- Example:
  - `list[3]`



Array Size

- An array size is the total size of the array in bytes / the size of the type
- Determine the size of an array with sizeof
  - `sizeof(list)/sizeof(type)`
- **`sizeof`** is an operator that gives you the size in bytes of an element

Character Array

- Referred to as a string
  - string type is ‘char[ ]’
    - example: char hello[] = {‘H’,‘e’,‘l’,‘l’,‘o’}
    - For your convenience: char[] hello = “Hello”
- Where’s the end of a Character Array?
  - Strings always end in a null byte
    - NULL byte = ‘\0’ // also 0
  - char hello[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
  - Implicit when using shorthand
    - char hello[] = “Hello”; //’\0’ added

Classwork

Arrays

```
The size of an int: 4
The size of nums (int[]): 20
The number of ints in nums: 5
The first num is 10, the 2nd 12.
The size of a char: 1
The size of name (char[]): 11
The number of chars: 11
The size of secret_name (char[]): 13
The number of chars: 13
name="Spider-Man" and secret_name="Peter Parker"
```

Bounds Checking

- Bounds: the start and end of the array
- C arrays **have no bounds checking**
  - `n[5]` will return a value, but it's WRONG
- If size omitted, initializers determine the bounds
  - `int n[ ] = { 1, 2, 3, 4, 5 };`
  - 5 initializers, therefore 5 element array

Arrays as Parameters

- Arrays as parameters to functions don't need a size
  - Why not?
- For function foo that takes an array as a parameters the following is sufficient
  - ```
    int foo(int array[]){
        …
        return array[0];
    }
    ```
    ```

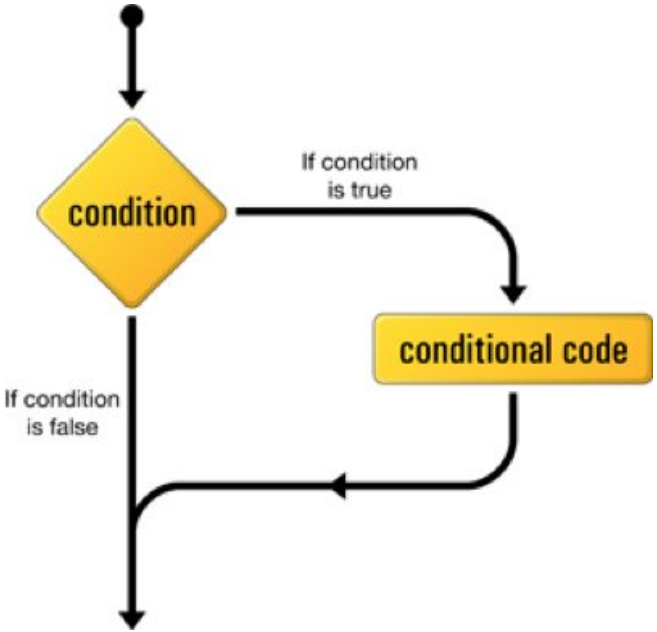
If Statement

- ‘if’ is a reserved word
- Syntax:

```
if(<boolean expr>){
        //code
}
```

- Example:

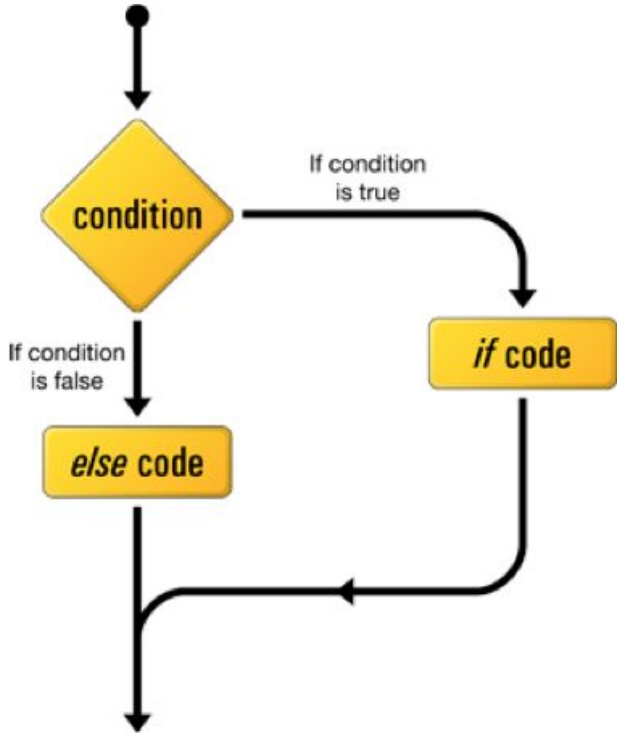
```
if( x >= 1)
```



else Statement

- ‘else’ is a reserved word
- Syntax:

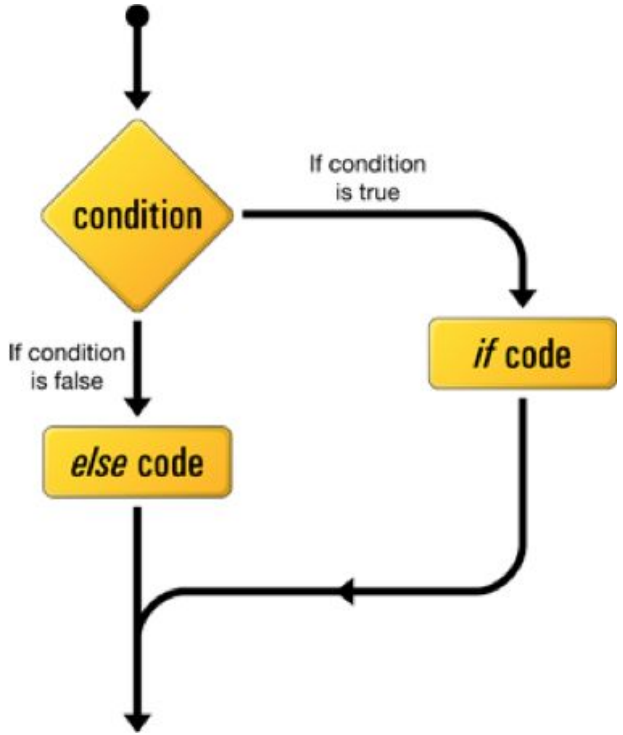
```
if(<boolean expr>){
        //code
}else{
    //code
}
```



else if Statement

- Syntax:

```
if(<boolean expr>){
    //code
} else if(<boolean expr>){
    //code
}
```



Common Error

- What happens if you write?

```
if(var = 0)
```

- Everyone makes this error. I still do.

Block Statements

- Several lines of code can be grouped together into a compound block statement
- A compound block is usually delimited by braces : { … } and can contain one or more standard C executable statements.
- Always indent nested blocks.
  - Indentation means nothing to the computer - only to a human reader!!!

Repetition Statement

- Repetition statements allow us to execute a statement multiple times

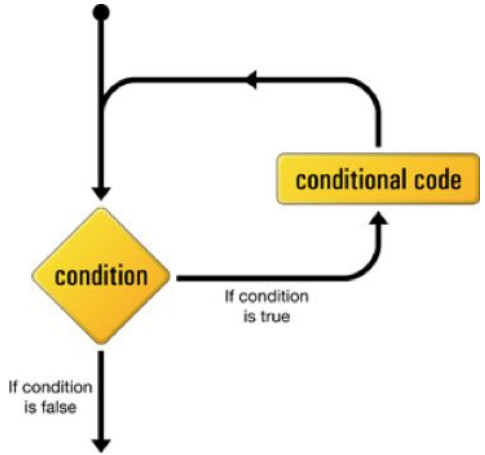
- Like conditional statements, they are controlled by boolean expressions

While Loop

- *while(<boolean expr>){*

  //do stuff

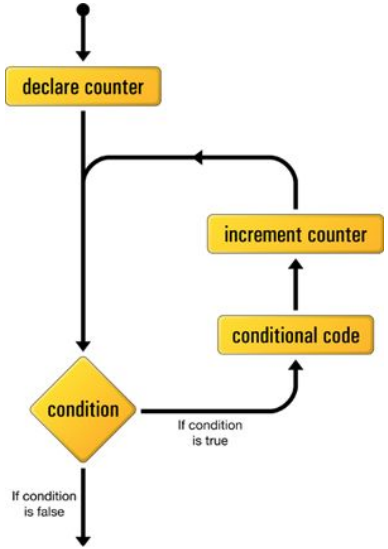
  }
- If the condition is true, the statement is executed
- The statement is executed repeatedly until the condition becomes false



For Loop

- A special kind while loop

```
int i;
for( i = 0;  i < 10;  i++ ){
    //do stuff
}
```



- *Syntax:*

    for(<define index>; <conditional>; <alter counter>)

Classwork

Prime Numbers

```
int i, j;
for (i=1; i<100; i++) {
        int prime=1;
        for (j=i-1; j>1; j--){
                if (i % j == 0){
                        prime=0;
                        break;
                }
        }
        if(prime) printf("%d\t", i);
}
return 0;
```

What are Functions?

- Function Definition
  - A component or module in a program
    - A subroutine
  - Kinds of functions
    - Library (printf, pow)
    - User defined
- Analogy:
  - Give ingredients for cookies to baker
  - Baker returns cookies
  - Eat cookies without caring how they were made

Declaration of a Function

- Function Definitions require:
  - return type
  - name (Use camelCase)
  - parameters
- Example:
  - `int foo(int var)`
  - `double pow(`
    `double base,`
    `double exponent);`

<returnType> <functionName> <parameter1> <parameter2> …

Return Values

- Can return any single valid type:
  - ```
    return_type functionName(type parameter)
        //do stuff
        return value;
    }
    ```
- Can also return nothing: ‘void’
  - ```
    void functionName(type parameter)
        //do stuff
        return; //this isn’t required for a void function
    }
    ```

Return Values

- Can return a value from anywhere
  - ```
    int functionName(int p){
        if(p == 0){
            return 5;
        }else{
            return 10;
        }
    }
    ```
- Once you return a value, the function stops executing
    ```

Functions need to be declared

- Functions take up memory, and anything that takes up memory requires declaration
  - The compiler must know how much memory it will need to pass values around
- You can declare and define a function simultaneously
  - ```
    int add_ints(int x, int y){
        return x + y;
    }
    ```
    ```

Functions need to be declared before used

- Just like variables, you can declare and define a function separately
  - ```
    int add_ints(int x, int y);

    //brilliant code...

    int add_ints(int x, int y){
        return x + y;
    }
    ```
    ```

Classwork

Functions and Scope

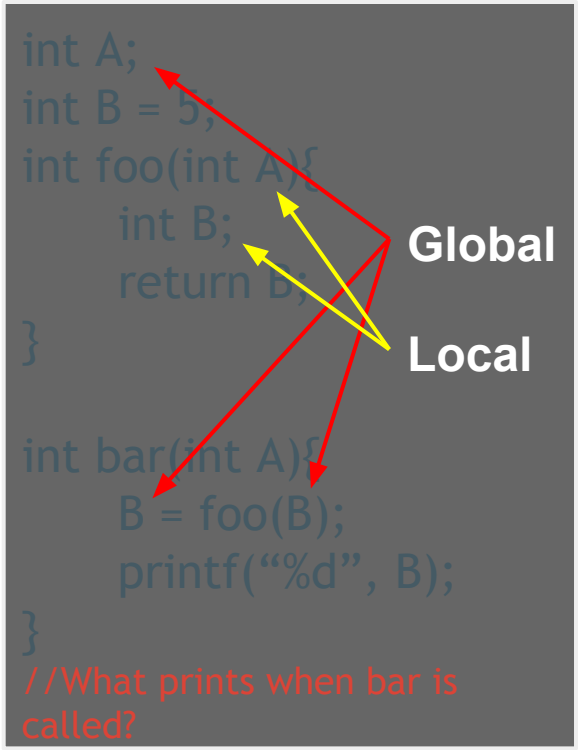
```
#include <stdio.h>
void printPrimes(int limit){
	int i, j;
	for (i=1; i<limit; i++) {
		int prime=1;
		for (j=2; j<i; j++){
			if (i % j == 0){
				prime=0;
				break;
			}
		}
		if(prime) printf("%d\n", i);
	}
}
void inc(int num){
	printf("%d\n", num++);
}
int main(){
	int x = 50, i;
	printPrimes(x);
	for(i = 0; i < 3; i++){
		inc(x);
	}
	return 0;
}
```

Parameters are copies

- In C, all parameters to functions are always copies of the value passed in
  - ‘Pass by Value’
- Any changes made to function parameters only exists for the scope of the function

Arguments and Scope

- Variables are only valid within their scope and nested scopes
  - Scope can be defined by the braces, everything between the braces is a scope
- Each function has its own scope
  - Can create a nested scope with { }
- Global Scope is accessible anywhere



Local Variables Disappear with scope

- Variables created within a scope disappear when the program leaves the scope
- Example:
  - ```
    void foo(){
        int x = 2;
    }
    ```


After returning from foo(), ‘x’ no longer exists
    ```