ALGONQUIN COLLEGE

MODULE 3 : USER-DEFINED FUNCTIONS AND LIBRARIES

Professor : Dave Houtman

Office: T323

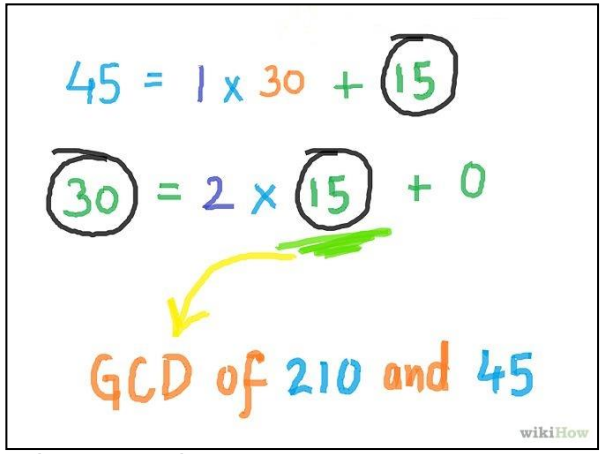
Office Hrs: Monday 15:30 – 16:00

Wednesday 15:30 – 16:00

Friday 15:30 – 16:00

Email: houtmad@algonquincollege.com

Example 09 :
The GCD algorithm



Program Description:

The *Greatest Common Divisor* algorithm can be used to reduce a ratio to its simplest fraction—for example 749/49 = 107/7. This algorithm, which is attributed to Euclid and is believed to be the oldest mathematical algorithm (written around 300 B.C.), says that in order to find the *greatest common divisor* (*gcd*) of two numbers—i.e. the largest integer number that evenly divides both values—execute the following algorithm:

```
                  ⎧ num > denom?  → gcd(num-denom, denom)
gcd(num, denom) = ⎨ denom > num?  → gcd(num, denom-num)
                  ⎩ denom == num? →  num
```

where **num** and **denom** are the numerator and denominator of the ratio respectively. Divide each number by **gcd** to get the simplest fraction.

Example 09: The GCD Algorithm

```
houtmad@ubuntu:~/examples$ Ex09
Enter the numerator: 91
Enter the denominator: 13
This is equal to the ratio: 7/1
houtmad@ubuntu:~/examples$ Ex09
Enter the numerator: 89
Enter the denominator: 17
This is equal to the ratio: 89/17
houtmad@ubuntu:~/examples$ Ex09
Enter the numerator: 749
Enter the denominator: 49
This is equal to the ratio: 107/7
houtmad@ubuntu:~/examples$
```

Example 09: The GCD Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

unsigned int getGCD (unsigned int num, unsigned int denom){
    if (num > denom)
        return getGCD(num - denom, denom);
    else if (denom > num)
        return getGCD(num, denom - num);
    else
        return num;
}
...
```

Example 09: The GCD Algorithm (con't)

```
...//con't

int main(void){
   unsigned int num, denom, gcd;

   printf("Enter the numerator: ");
   scanf("%u", &num);

   printf("Enter the denominator: ");
   scanf("%u", &denom);

   assert(denom != 0);

   gcd = getGCD(num, denom);

   num /= gcd;
   denom /= gcd;

   printf("This is equal to the ratio: %u/%u\n", num, denom);
}
```

3.01 Function definitions

- As with all programming projects, any program that consists of a long list of code becomes unwieldy unless it can be broken down into logical functional units i.e. code blocks, functions, and files.

- C allows you to create functions that effectively disentangle long stretches of code and package them together into related functional units

- A function is a block of code designed to perform a specific task; it may, of course, call on other functions in the execution of that task. `main()` is specified as the function that signals the start of the program, and it, in turn, generally calls on other functions to implement your program.

- For simple, one-line tasks, a macro definition will often suffice. But more complicated, multi-line operations should be packaged into their own functional units. Larger collections of related functions should be packaged into their own files.

3.01 Function definitions

```
unsigned int getGCD (unsigned int num, unsigned int denom){…}
```

- The rules for naming functions are the same as those for naming variables. i.e.
  - The first character can be an alphabetic character or underscore, however the latter should be avoided since functions used internally by the C compiler often begin with one or more underscores.
  - The second character can be alphanumeric or underscore, but other special characters like #,$,%,&, etc. are usually prohibited by the compiler
  - `gcc` allows input C source code files and output executable files to have names that being with a numeric digit, but functions cannot have a digit in the first character.

3.01 Function definitions

- Like Java, each C program is comprised of code contained in one or more source files. Unlike Java, C files are not related to any particular class, but rather consist of functions (or ***procedures***) and variables, which are roughly the equivalent of the methods and properties/fields (respectively) found in the world of object-oriented programming.

- As with methods in Java, each function takes zero, one, or even several arguments and returns a single value.

- There is no way to pass objects in C (much less create them), however C allows programmers to create packages of data consisting of multiple data types called ***structures***. Pointers to variables, arrays, strings, functions and structures can all be passed to and from functions, and so most of the functionality that comes from using objects (and their methods) in Java is available to you using pointers in C—but without the other common OOP features, like inheritance and public, private and protected access modifiers.

3.01 Function definitions

- Collectively, variable and function names are referred to as ***identifiers*** in C, just as in Java they are collectively referred to as ***members***.

```
unsigned int getGCD (unsigned int num, unsigned int denom){
    if (num > denom)
     return getGCD(num - denom, denom);
    else if (denom > num)
     return getGCD(num, denom - num);
    else
     return num;
}
```

3.01 Function definitions

- Function declarations in C look and work just like methods in Java (minus `private`, `public`, or `protected`). In place of access modifiers, C has something called a ***storage class***:

```
stor_class ret_type ftn_name (parameter list) {
    // your code goes here….
}
```

K&R 25

K&R 59

where:

stor_class—the storage class of the function (ignore this for now)

ret_type – data type returned by the function (or `void` if no return)

ftn_name – is the function name; it must be unique and have the same rules as variables

parameter list – the list of variables passed to the function, separated by commas when there is more than one argument

- When no return type is specified, C assumes a default type of `int`.

3.01 Function definitions

```
unsigned int getGCD (unsigned int num, unsigned int denom){
    if (num > denom)
        return getGCD(num - denom, denom);
    else if (denom > num)
        return getGCD(num, denom - num);
    else
        return num;
}
```

GCD is a special kind of function, called a ***recursive*** function, which has the feature that it calls itself. One crucial characteristic of a recursive function is that it must have an escape mechanism, otherwise the code calls itself forever, locking up the host program until it can be safety interrupted.

K&R 73

Recursive functions have one great virtue: because (on each recursion of the function) the code is already loaded in memory, it already 'lives' in the computer cache; recursive functions, by their very nature, are *fast*.

3.02 Using typedef

- C includes a shortcut method for aliasing data types, known as `typedef`. In Example09.c, we use `unsigned int` frequently. `typedef` can be used to shorten up data type declarations as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef unsigned int UINT;

UINT getGCD (UINT num, UINT denom){
    if (num > denom)
     return getGCD(num - denom, denom);
    else if (denom > num)
     return getGCD(num, denom - num);
    else
     return num;
}

int main(void){
    UINT num, denom, gcd    //…etc.
...
```

3.02 Using typedef

- In the above example, `typedef` allows all declarations of `unsigned int` to be replaced with `UINT` instead. As another example, say we had the declaration:

```
unsigned long int myLongInt;
```

You could create the following `typedef`:

```
typedef unsigned long int ULONG;
```

Then, whenever you want to declare `an unsigned long int`, use

```
ULONG   myLongInt;
```

K&R 119

K&R 181

3.02 Using typedef

K&R 119

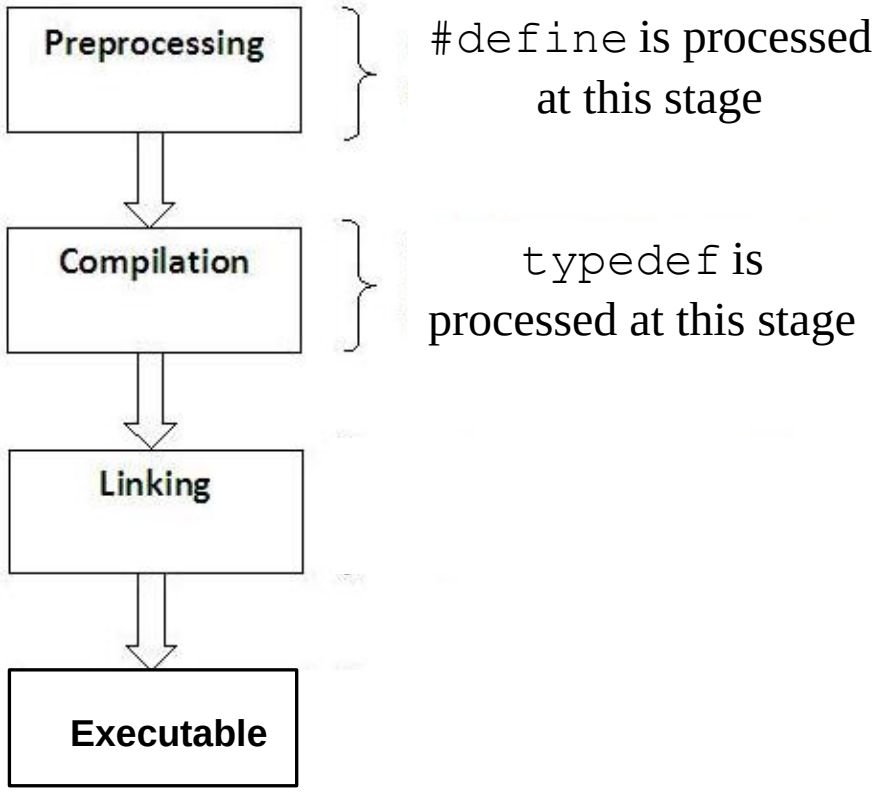
- Note that `typedef` and `#define` can sometimes be used to perform the same task. But the order of the assignment of the two is different (and potentially confusing):

```
typedef unsigned long int ULONG;
```

```
#define ULONG unsigned long int
```

3.02 Using typedef

- Note that `typedef` is a keyword in C; it is part of the core language. Therefore, it is dealt with during the actual compilation of your code, not during the preprocessor stage, as is the case with `#define`.



Review - Example 09: The GCD Algorithm

```
// Using typedef

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef unsigned int UINT;           // typedef added

UINT getGCD (UINT num, UINT denom)
{
   if (num > denom)
       return getGCD(num - denom, denom);
   else if (denom > num)
       return getGCD(num, denom - num);
   else
       return num;
}
```

Review - Example 09: The GCD Algorithm (con't)

```
int main(void){

    UINT num, denom, gcd;

    printf("Enter the numerator: ");
    scanf("%u", &num);

    printf("Enter the denominator: ");
    scanf("%u", &denom);

    assert(denom != 0);
    gcd = getGCD(num, denom);

    num /= gcd;
    denom /= gcd;

    printf("This is equal to "
       "the ratio: %u/%u\n", num, denom);
}
```

Questions

1. The gcd algorithm could potentially be implemented as a macro, e.g.

```
#define gcd(num, denom) (if (num > denom)   \
   return gcd(num-denom, denom);   \
else if (denom > num)   \
   return gcd(num, denom-num);   \
else \
   return num; \
)
```

   Explain why this *won't* actually work, even though the code is correct.

2. If you use the function header

```
main() {…}
```

   this works—it does not generate a compile time error—even without the standard `void` parameter or the `int` return type. Explain why.

   Note: avoid using this shortcut; `int main(void){…}` is safer.

Programming Challenges

1. Look at the GCD algorithm closely and make sure you understand how it works. Note that it is computationally expensive to calculate a ratio like `119235987 / 7`, since (initially) you need to subtract 7 repeatedly from the numerator (several million times) until it becomes a number less than or equal to 7. Write the code more efficiently using the modulus operator, so as to avoid all the unnecessary subtractions. (In fact, you can simplify the code so as to avoid the recursive nature of the algorithm altogether.)

2. Modify the GCD code so that it handles negative values for both the numerator and the denominator.

3. Write a function that takes, as parameters, the numerator and denominator of the reduced fraction (like `107/7`) and prints out the ratio in the form `"15 2/7"`. Test your code thoroughly so that it generates reasonable results. For example, you'd never say that `20004/4` is `"5001 0/4"`—your code shouldn't either.

Programming Challenges

4. Write a function that takes, as arguments, the numerator and denominator of two fractions, adds the factions together, and uses the gcd function to find the smallest fractional representation of the answer, (i.e. 1/3 + 1/6 = 1/2, not 9/18.)

5. Write a function that randomly returns a false value (i.e. a zero) 80% of the time and a true value (i.e. non-zero) the other 20% of the time.

6. An 18-speed bicycle has the following number of teeth on the front and rear sprockets:

```
frontSprocket = {23, 42, 52};
rearSprocket = {12, 16, 21, 26, 39, 42};
```

The ratio of the number of teeth on the front sprocket to the number of teeth on the rear sprocket creates the 18 gear ratios that the bicycle is capable of. Write a program that utilizes these two arrays and the GCD

Programming Challenges

algorithm to print out the gear ratios for each possible combination of sprockets, displaying the lowest possible ratios. Your output should say:

```
Front Sprocket: 1  Rear Sprocket 1:  Gear Ratio:   23/12
Front Sprocket: 2  Rear Sprocket 1:  Gear Ratio:   7/2
Front Sprocket: 3  Rear Sprocket 1:  Gear Ratio:   13/3
Front Sprocket: 1  Rear Sprocket 2:  Gear Ratio:   23/16
```

etc.

7. The factorial function is commonly cited in exercises/examples involving recursive functions. Mathematically, the factorial, expressed as

$$n!$$

is just equal to

$$n \times (n-1) \times (n-2) \times (n-3) \ldots \times 2 \times 1$$

Programming Challenges

The recursive function that performs the factorial operation is very simple:

```
long int factorial(long int N){
    (N > 0) ? return (N * factorial(N-1)):
return 1;
}
```

One complication that arises from the use of this function is that it blows up very quickly and triggers an overflow, even when used in common, everyday calculations. For example, the odds of winning the jackpot in Lotto 6/49 are expressed mathematically as in 1 in (49!)/(6!)(43!). The value of 49! is greater than $6.08e10^{62}$, vastly exceeding the storage capacity of a `long long int`; any attempt to calculate the odds of winning Lotto 6/49 using

```
factorial(49)/(factorial(6))*(factorial(43))
```

will trigger an error.

Programming Challenges

However the total number calculated by (49!)/(6!)(43!) is 'only' 13,983,816, and even the ratio 49!/43! is 'only' 10,068,347,520, because 49!/43! is just equal to

$$49 * 48 * 47 * 46 * 45 * 44$$

Write and test a function that takes in two values, M and N, and reliably calculates the ratio of two factorials N!/M! by modifying the traditional factorial function given at the start of this problem. When N > M, output the result of the solution directly; when M > N, print the output as '1/…' (where … is the value of M!/N!). If it is available, use the `long long int` data type for this exercise.

Of course, this function will still blow up for certain large values of M and N. One calculation suggests this will happen when the difference between N and M exceeds

$$\left[N * ln\left({}^{N}/_{M}\right) - ln(LLONG\_MAX)\right] / \left(ln(M)\right)$$

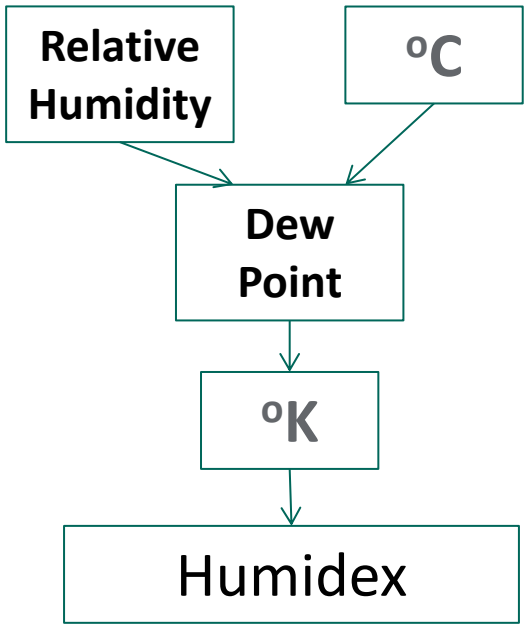
Programming Challenges

(for suitably large values of M and N, say each >20) where `LLONG_MAX` is the maximum value of a `long long int`, which is stored in `limits.c`, and `ln()` is the natural logarithm (available in `math.h`). Write your function to check that the values of M and N do not exceed this limit, and test the equation's effectiveness in preventing an overflow.

Example 10 : Calculating the Humidex

Program Description:

In this example, the Humidex is calculated using functions stored in a user-defined library. The value of the Humidex relies on a temperature of the dew point, T_{dew}, in ^{o}K (Kelvin, i.e. in Celsius degrees measured from absolute zero, -273.16 C). This in turn is dependent on the relative humidity and the temperature in ^{o}C. Since these operations are interrelated, they should logically be grouped together in a single library, in this case called `weatherEquations.c`.



Example 10: Calculating the Humidex

```
houtmad@ubuntu:~/examples$ Ex10

Enter the current air temperature in degrees Celsius:30
Enter the humidity as a percent (without the % sign, e.g. 87): 75
With Humidex added, the temperature feel like  42.4:

houtmad@ubuntu:~/examples$ Ex10

Enter the current air temperature in degrees Celsius:24
Enter the humidity as a percent (without the % sign, e.g. 87): 40
With Humidex added, the temperature feel like  25.1:

houtmad@ubuntu:~/examples$ Ex10

Enter the current air temperature in degrees Celsius:15
Enter the humidity as a percent (without the % sign, e.g. 87): 30
With Humidex added, the temperature feel like  15.0:

houtmad@ubuntu:~/examples$
```

Example 10: Calculating the Humidex

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "weatherEquations.c"

int main(void){

   unsigned int cTemp, relHumidity;

   printf("\nEnter the current air temperature "
      "in degrees Celsius:");
   scanf("%u", &cTemp);

   printf("Enter the humidity "
         "as a percent (without the %% sign, e.g. 87): ");
   scanf("%u", &relHumidity);
   assert(relHumidity > 0);

   printf("With Humidex added, the temperature feel like "
      "%5.1lf: \n\n", (cTemp + cHumidex(cTemp, relHumidity)));
}
```

Example 10: Calculating the Humidex

In the file **weatherEquations.c**:

```
#include <math.h>

#define ABS_ZERO (-273.16)
#define getDegreesK(cTemp) ((double)(cTemp) - ABS_ZERO)

double getcTdew(double cT, double RH){

   double gamma;
   const double b = 17.27, c = 237.3;
   gamma = (log(RH / 100.0) + ((b * cT) / (c + cT))) / b;
   return((c * gamma) / (1.0 - gamma));

}

double cHumidex(double cTemp, double relHumidity){

   double cTdew, kTdew, HMDX;
   cTdew = getcTdew(cTemp, relHumidity);
   kTdew = getDegreesK(cTdew);
   HMDX = (1394434913 * exp(-5417.753/kTdew)) - (50.0/9);
   return (HMDX > 0.00 ? HMDX : 0.00);

}
```

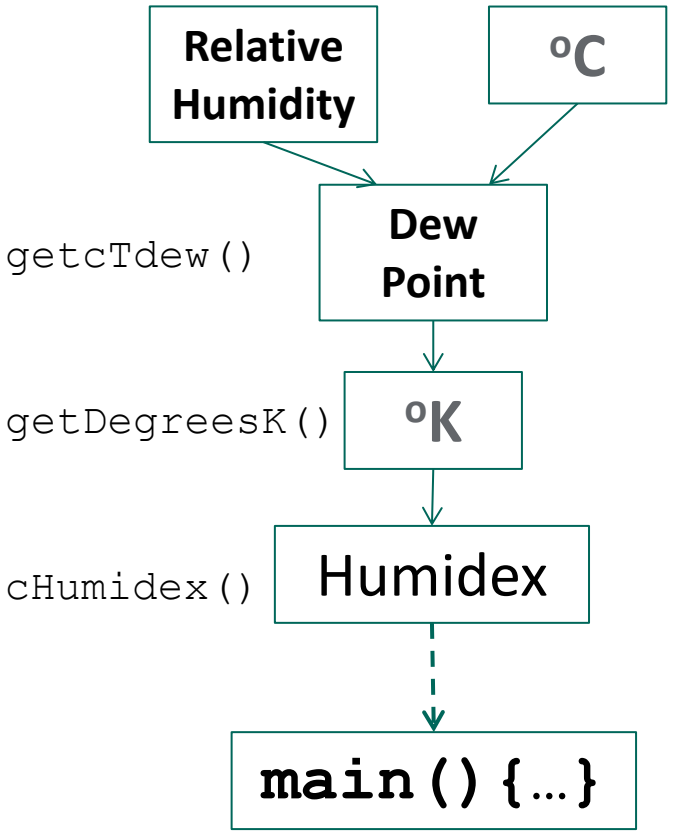
3.03 Order of declaration of functions

- Note that using `#include` to 'insert' a file into your main file is equivalent to inserting the code *in* that file directly into the main() file. Thus the above code is equivalent to

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

#define ABS_ZERO (-273.16)
#define getDegreesK(cTemp) ((double)(cTemp) - ABS_ZERO)

double getcTdew(double cT, double RH){
    ...
}
double cHumidex(double cTemp, double relHumidity){
    ...
}

int main(void){...}
```

3.03 Order of declaration of functions

- In general, any function ('A') that calls on another ('B') for its operation will need to have the function called upon declared first (B is declared *before* A, since A relies on B). So in Example 10, since both `getDegreesK()`, and `getcTdew()` are called from within `cHumidex()`, their declaration must precede `cHumidex()`. And, since `main()` calls `cHumidex()` its declaration must be made before it can be called.



3.03 Order of declaration of functions

```
// #includes go here

#define ABS_ZERO (-273.16)
#define getDegreesK(cTemp) ((double)(cTemp) - ABS_ZERO)

double getcTdew(double cT, double RH){
    ...
}

double cHumidex(double cTemp, double relHumidity){
    ...
    cTdew = getcTdew(cTemp, relHumidity);
    ...
}

int main(void){
    ...
    printf("The temperature with humidex "
    "added is %5.1lf: \n", (cTemp + cHumidex(cTemp, relHumidity)));
}
```

`cHumidex()` depends upon `getcTdew()` and `getDegreesK()` having been declared first

`main()` depends upon `cHumidex()` having been declared 1st

3.03 Order of declaration of functions

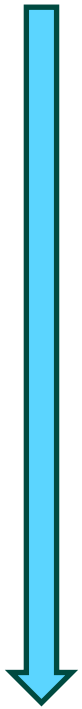
- So the order of the declarations is important; inverting the sequence of functions triggers compile time errors:

```
// #includes go here

int main(void){
...
   printf("The temperature with humidex added"
   " is %5.1lf: \n", (cTemp + cHumidex(cTemp, relHumidity)));
}

double cHumidex(double cTemp, double relHumidity){
   ...
   cTdew = getcTdew(cTemp, relHumidity);
   ...
}

double getcTdew(double cT, double RH){
    ...
}
```

Hasn't been declared yet

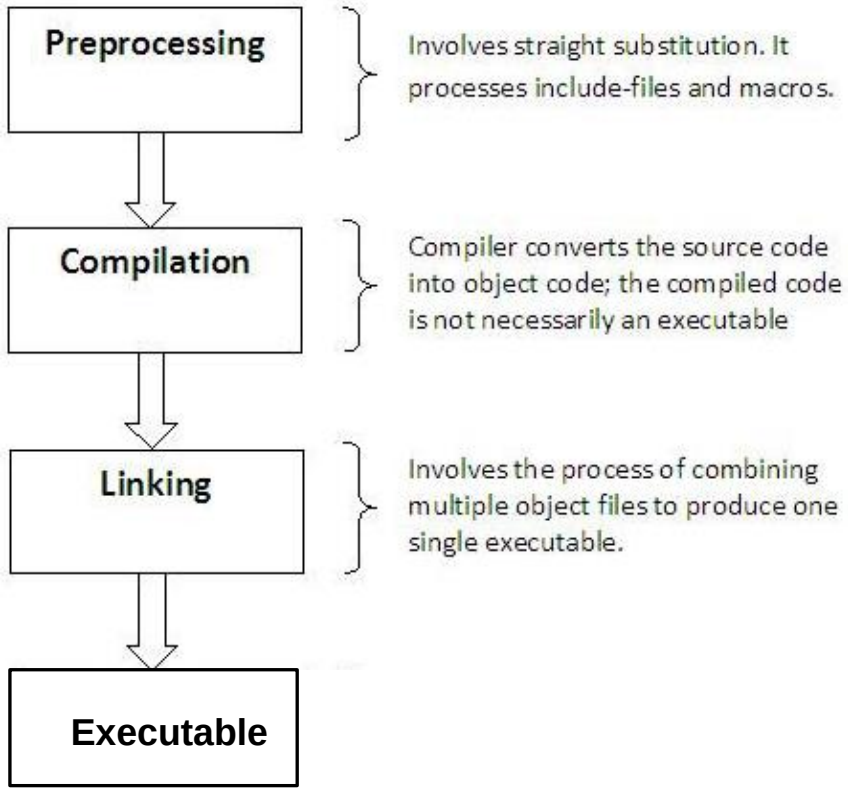
Hasn't been declared yet

Direction of compilation (single-pass)



3.03 Order of declaration of functions

- C uses a **single-pass compiler**; once preprocessing is complete, the compiler works its way through your code sequentially, line-by-line, from top to bottom—once. Each *identifier must be declared before it can be referenced in your code*. This is unlike Java, which uses a multi-pass operation during the conversion from source code to bytecode. If you use a function before it is declared, Java won't flag an error; it waits to find the definition during a subsequent pass though the code.



3.03 Order of declaration of functions – C vs. Java

- For example, in Java you can do this:

The `inputHours` method hasn't been declared yet, but Java doesn't signal an error; it waits for the method to be declared & defined.

Similarly, `hours` hasn't been declared at this point, but Java doesn't flag an error; it waits for the declaration, which could be at the bottom of the program.

```
public class Time {
 public static void main (String [] args) {
   Time runner1 = new Time();
   System.out.println("Enter the time for runner 1");
   runner1.inputHours();
       ...
 }
 public void inputHours() {
    Scanner keyboard= new Scanner (System.in);
    System.out.println("Enter the hours(0-23): ");
    hours = keyboard.nextInt();
 }

 private int hours;
    ...
}
```

3.03 Order of declaration of functions – C vs. Java

- You probably take this behaviour for granted. But in fact, in C and C++ (and most other languages prior to Java), failure to declare an identifier before its actual use in code would trigger a compile-time error. In C, everything must be declared up front, since the compiler won't be making any second passes through your code to look for 'late' declarations after their initial point of use.

- Effectively, this means that variables should be declared early on inside functions, and functions must be declared (in some form) before they are called on in `main`. So in general,

  1. `main()` will be the *last* function you declare in your code: all other functions referred to in `main()` must come first; and,

  2. all global variable declarations should be made at the top of a file, before any commands are executed that use these declarations.

3.03 Order of declaration of functions – C vs. Java

- Hence in Example09.c we have our `include` statements first:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
```

since these include all the functions we'll require later in the program. This was followed by the function definition itself:

```
unsigned int getGCD (unsigned int num, unsigned int denom){
   if (num > denom)
    return getGCD(num - denom, denom);
   else if (denom > num)
    return getGCD(num, denom - num);
   else
    return num;
}
```

which is then followed by the code that uses these functions:

```
int main(void){…}   //etc.
```

3.04 Forward declarations

- Defining multiple functions before they are actually used in code is somewhat inconvenient for the programmer. In general, we'd like to have `main()` appear near the top of program to improve readability, not buried at the end of a long list of functions.

- In addition to packaging code in files that we can include in our main() program, C provides two additional mechanisms to help alleviate this problem:

  1. use ***forward declarations***

  2. package .c code in external functions, compiled separately and accessible through a .h file; this often includes using forward declarations (see Examples 11 and 12)

3.04 Forward declarations

- When the number of identifiers associated with `main()` is relatively small, forward declarations can be used to unclutter code. When the identifier is a function, the forward declaration takes the form of a ***function prototype***. In this method, rather than *define* your functions completely prior to their use in `main()`, you merely *declare* them, essentially telling the compiler: "there's a function that I'll be using in this program; here's its general format. I'll take care of the details later."

- A function prototype is basically just a 'short-cut' form of the function header; it declares the data types that serve as the inputs and outputs of the actual function itself. It has this form:

```
return_type function_name (parameter list @types only);
```

(We'll ignore the storage class for now…)

K&R 27

3.04 Forward declarations

Despite the similarities, note that the format of a function prototype is not the same thing as the format of a function definition. The function definition has the form:

```
return_type  function_Name (parameter_list) {
        // your code goes here….
}
```

while the function prototype is much simpler:

```
return_type function_Name (parameter list @types only );
```

Note that the function *prototype* is a declaration only—a statement of intent about how the code is to be used. The function *definition* provides the actual code to be compiled and executed

3.04 Forward declarations

- For example, for Example09.c we could write a forward declaration of the **`getGCD()`** function as follows:

```
/* includes go here */

// forward declaration of the function
unsigned int getGCD (unsigned int, unsigned int); // prototype

int main(void){
   unsigned int num, denom, gcd;
   printf("Enter the numerator: ");
   // etc…
}

// function definition
unsigned int getGCD (unsigned int num, unsigned int denom){
   if (num > denom)
        return getGCD(num - denom, denom);
   // etc…
}
```

3.04 Forward declarations

- A function prototype is a shortcut form of the function heading itself, and includes only the information necessary for the compiler to check your code. In particular, note that

1. The function prototype ends with the ';' while the actual function definition does not,

2. The function prototype requires only data types; it does not need to include variable names in the parameter list,

3. There's nothing wrong with listing variable names inside your function prototype, e.g.

```
int myfunction (int varA, float varB);
```

but since the variable names **varA** and **varB** aren't used for anything (its only a declaration, not a definition), it doesn't serve any purpose to attach identifiers to your data types.

3.04 Forward declarations

4. If the function definition occurs before it is needed in code, there's really no need for a forward reference; it's technically redundant. For example, the following is impractical:

```
int myFunction(int, float);     //forward declaration
...

int myFunction(int a, float b){ //function definition
   // some commands go here...
}

int main(void){
   int c = myFunction(4, 2.72);  //call the function
   ...
}
```

While there's nothing wrong with this, the function prototype doesn't say anything that isn't covered by the actual function definition. However, as we shall see, there are situations that *require* a function prototype *and* a function definition, prior to the function call itself.

3.05 Order of declaration of functions

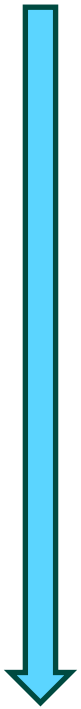
- In Example10.c, we can use forward declarations to tidy up the code and avoid the errors that would occur if we'd left the declarations to the end.

```
// #includes go here
double cHumidex(double, double);
double getcTdew(double, double);

int main(void){
...
   printf("The temperature with humidex added"
   " is %5.1lf: \n", (cTemp + cHumidex(cTemp, relHumidity)));
}

double cHumidex(double cTemp, double relHumidity){
   ...
   cTdew = getcTdew(cTemp, relHumidity);
   ...
}

double getcTdew(double cT, double RH){
   ...
}
```

cHumidex() is already declared

getcTdew() is already declared

Direction of compilation (single-pass)



3.05 Order of declaration of functions

- Forward declarations of functions are useful in two general contexts. First, as we've seen, in programs where a large number of functions are used in `main()` which don't really need to be included in a separate file:

```
// function prototypes
int helperFtnA(int);
int helperFtnB(int);
int helperFtnC(int);

// main()
int main(void){
   int a = helperFtnA(1);
   int b = helperFtnB(2);
   int c = helperFtnC(3);
}

// function definitions
int helperFtnA(int x){
   return (2*x)
}
// etc.
```

3.05 Order of declaration of functions

Second, when two functions each need to call on one another, one of them will always be defined first, while the other will need a forward declaration:

```
int second(int);

int first(int x) {
    if (x == 0)
        return 1;
    else
        return second(x-1);
}

int second(int x) {
    if (x == 0)
        return 0;
    else
        return first(x-1); // backward reference
}
```

second() is not defined until later, so it needs a forward reference to be called from inside **first()**

first() is already defined, so it doesn't require a forward declaration

Example taken from: http://en.wikipedia.org/wiki/Forward_declaration

3.05 Order of declaration of functions

One Wikipedia entry suggests that such mutually recursive usage is 'unwise' (http://en.wikipedia.org/wiki/Forward_declaration). However such usage is not at all uncommon in object-oriented code, in situations where each of two classes must contain a reference to the other.

Consider an example in which we wish to model students and their courses in an OOP language by creating classes for each. The `course` class will contain, along with appropriate methods and properties, an array of the `student` objects registered in that instance of the `course`. But each instantiated `student` object will need to contain an array of `courses` in which they are registered. Thus the `course` class needs to 'know' about `students`, and the `student` class needs to 'know' about `courses`. For languages that rely on a single-pass compiler, a forward reference to either the `student` class or the `course` class is necessary.

Since Java and C# use multi-pass compilers, no such forward references are required. In C++, the situation is unclear; whether a forward declaration will be required or not is somewhat compiler dependent. (The C++ standard does not required that a compiler be single-pass or multi-pass.)

3.06 The conditional if statement: using ?:

```
return (HMDX > 0.00 ? HMDX : 0.00);
```

- The conditional or ternary `if` is a single line equivalent to an `if…else` statement. The above is exactly the same as:

```
if (HMDX > 0.00)
   return (HMDX);
else
   return (0.00);
```

Thus the purpose of the above statement is to ensure that the value of the Humidex is always greater than 0.

- The conditional `if` is thus a very short and convenient replacement for the bulkier `if…else…` statement.

K&R 47

3.06 The `conditional if` statement: using ?:

```
return (HMDX > 0.00 ? HMDX : 0.00);
```

- The conditional `if` is short-circuited; if the first expression results in a non-zero number then only the expression between ? and : is executed and its value returned; the third expression (after the :) is ignored. (Note: other languages differ in that all three expressions may be evaluated regardless of the truth or falsity of the first argument, but only one of the two expressions is returned.

- As with all expressions that return a data type, the type of data returned by ternary if must be consistent with the value expected:

```
myDataType myResult =
(conditional_test ? retmyDataType1 : retmyDataType2)
```

In this example, if the type of `myResult` is an `int`, then `retmyDataType1` and `retmyDataType2` must also be `int`s also.

3.06 The `conditional if` statement: using ?:

- Here's another example of conditional `if` in action. Consider the following bug, which is still seen in most software:

```
There are 1 minutes remaining
```

This is easily corrected by writing the following code (assume `mins` is an integer variable containing the time remaining):

```
printf("There %s %d minute%s remaining\n",
    (mins==1?"is":"are"), mins, (mins==1?"":"s"));
```

Notice that when `mins` is equal to 1, in the second conditional `if`, the value returned is ""—an empty string. But then the grammar is correct:

```
There are 2 minutes remaining
There is 1 minute remaining
```

3.07 Summary – decision structures

In C, as in Java, decision structures will always be one of the following

1. `if (conditional_test) {...} // simple if`

2.
```
if (conditional_test) {...}

else {...}              //including else if
```

3.
```
switch(index){      // switch
   case x: …
   case y: …
   …
}
```

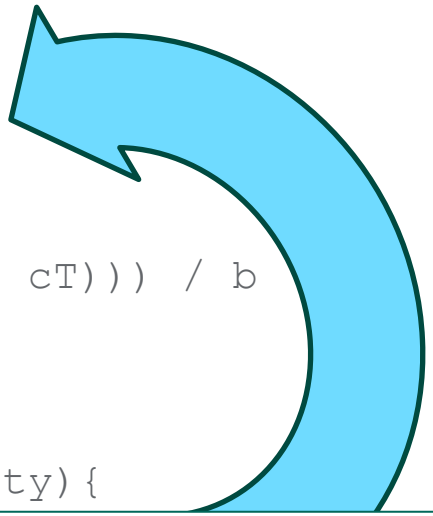
4. `?:` `//ternary or conditional if`

Review - Example 10: Calculating the Humidex

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

#define ABS_ZERO (-273.16)
#define getDegreesK(cTemp) ((double)(cTemp) - ABS_ZERO)

double cHumidex(double), getcTdew(double, double);

int main(void){

   unsigned int cTemp, relHumidity;
   double cTdew, kTdew;

   printf("Enter the current air temperature in degrees Celsius: ");
   scanf("%u", &cTemp);

   printf("Enter the humidity "
          "as a percent (without the %% sign, e.g. 87): ");
   scanf("%u", &relHumidity);
   assert(relHumidity > 0);
    ...
```

Review - Example 10: Calculating the Humidex (con't)

```
    ...
    printf("The temperature with humidex "
           "added is %5.1lf: \n", (cTemp + cHumidex(kTdew)));
}

double getcTdew(double cT, double RH){

    double gamma;
    const double b = 17.27, c = 237.3;
    gamma = (log(RH / 100.0) + ((b * cT) / (c + cT))) / b
    return((c * gamma) / (1.0 - gamma));

}

double cHumidex(double cTemp, double relHumidity){

    double cTdew, kTdew, HMDX;
    cTdew = getcTdew(cTemp, relHumidity);
    kTdew = getDegreesK(cTdew);
    HMDX = (1393310916 * exp(-5417.753/kTdew)) - (50.0/9);
    return (HMDX > 0.00 ? HMDX : 0.00);

}
```



You can swap these two functions without problems, since they've both been declared at the top of the program

Questions

1. Example10 contains a mixture of implicit and explicit data type conversions. Examine the code and determine the data type used in each operation and function call, along with the data type following the conversion.

2. Is the expression:

```
return (HMDX > 0.00 ? HMDX : 0.00);
```

   equivalent to:

```
return (HMDX? 0.00 : HMDX);
```

Programming Challenges

1. The equation for wind chill factor is:

$$T_{wc} = 13.12 + 0.6215T_a - 11.37V^{0.16} + 0.3965T_aV^{0.16}$$

where: T_a is the ambient temperature (without wind) and
V is the wind speed in km/hr.

Code this equation, and check the results to see if they agree with your experience.

Example 11 : The SimpleStats.c Library

Program Description:

Functions that perform related tasks should logically be grouped together in a library. In this example a small library of simple statistical functions is created and tested using calls from `main()`. The four functions are:

- A function to return the maximum value of a series of data entered
- A function to return the minimum value of a series of data entered
- A function to return the average of the series
- A 'reset' function, to clear all previous values from the library

This library will be used in future examples.

Example 11: The SimpleStats.c library

```
houtmad@ubuntu:~/examples$ Ex11
Enter the number of integers to be entered : 6
Enter an integer : 2
Enter an integer : 4
Enter an integer : 6
Enter an integer : 8
Enter an integer : 10
Enter an integer : 12
Max Value is : 12.00
Min Value is : 2.00
Average Value is : 7.00
houtmad@ubuntu:~/examples$ Ex11
Enter the number of integers to be entered : 5
Enter an integer : -14000
Enter an integer : -2864
Enter an integer : 0
Enter an integer : 2865
Enter an integer : 13999
Max Value is : 13999.00
Min Value is : -14000.00
Average Value is : 0.00
houtmad@ubuntu:~/examples$
```

Example 11: The SimpleStats.c library

- Our library will need to be able to signal an error condition back to the calling routine. We can't use `EXIT_FAILURE`, since this is just numerically equal to 1, which is a number we might reasonably generate in the normal course of operations. One possible solution is to use an extreme value to signal an error:

```
#include <float.h>
#include <assert.h>
#include <stdio.h>

#define ERROR FLT_MAX
...
```

Example 11: The SimpleStats.c library

- Additionally, our function will need certain variables to hold different parameters and values. These will need to be visible to all of the functions inside the file; hence they must be declared globally within the file (but outside of any particular function):

```
...
unsigned int numSamples = 0;
double maxNum;
double minNum;
double average=0;
...
```

Only then do we define our functions…

Example 11: The SimpleStats.c library

```
void setMaxValue(double newNum){
    if ((numSamples == 1) || (newNum > maxNum))
        maxNum = newNum;
}

void setMinValue(double newNum){
    if ((numSamples == 1) || (newNum < minNum))
        minNum = newNum;
}

void setAveValue(double newNum){
    average = (((numSamples-1) * average) + newNum)/numSamples;
}

void setValue(double newNum){
    assert(newNum != ERROR);
    numSamples++;
    setMaxValue(newNum);
    setMinValue(newNum);
    setAveValue(newNum);
}
...
```

Example 11: The SimpleStats.c library

- Each time a number is entered by the calling program, `setValue()` calls the three `set_()` functions shown above. They, in turn set the appropriate global values `maxNum`, `minNum`, and `average`. But just before these three functions are called, `setValue()` increments the counter of the number of samples, `numSamples`.

- For the reasons discussed in the previous section, the three initial `set_()` functions must be declared prior to their being called in `setValue()` (as shown in the previous slide.). Alternately, we could use forward declarations for the three functions near the start of the file:

```
void setMaxValue(double);
void setMinValue(double);
void setAveValue(double);
```

Example 11: The SimpleStats.c library

- We can add `get_()` functions to our library as well:

```
double getMaxValue(){
   return (numSamples ? maxNum: ERROR); // return error if no
}                                       // numbers entered

double getMinValue(){
   return (numSamples ? minNum : ERROR);
}

double getAveValue(){
   return (numSamples ? average : ERROR);
}

unsigned int getNumSamples(void){
  return (numSamples);
}

void reset(void){
   maxNum = minNum = average = numSamples = 0;
}
```

Example 11: The SimpleStats.c library

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "SimpleStats.c"

int main(void){
   int num = 0, ctr, numValues = 10;

   printf("Enter the number of integers to be entered : ");
   scanf("%d", &numValues);
   assert(numValues > 0);

   for (ctr = 0; ctr < numValues; ctr++){
      printf("Enter an integer : ");
      scanf("%d", &num);
      setValue((double)num);
   }

   printf("Max Value is : %3.2lf\n", getMaxValue());
   printf("Min Value is : %3.2lf\n", getMinValue());
   printf("Average Value is : %3.2lf\n", getAveValue());
   return EXIT_SUCCESS;
}
```

3.08 Storage classes – what is a storage class?

- There is, of course, just one problem: nothing prevents the user of the library from interacting directly with functions like `setMaxValue()`, `setMinValue()`, and `setAveValue()`, not to mention the variables, `maxNum`, `minNum`, `average` and `numSamples`. We'd like to be able to make these identifiers 'private'. That's where ***storage classes*** come in.

- In addition to having a type and range, variables also have a storage class. Storage classes control up to four features of a variable:

    a) The scope of the variable – *who* gets to see it
    b) The initial value assigned – *what* it is initialized to
    c) The life time of the variable – *when* it's value exists and when it disappears
    d) The location of the variable – *where* the information is stored

  …in other words, storage classes control the features of a variable that you normally associate with `public` and `private` access modifiers.

3.08 Storage classes

- The format for a variable declaration is, as before,

```
stor_class data_type var_name;
```

- There are only four storage classes:

```
auto        extern        static        register
```

- Storage classes can also be applied to functions. For functions, the format for a function prototype is

```
stor_class ret_type ftn_name (parameter list);
```

and for a function definition

```
stor_class ret_type ftn_name (parameter list) {
    ...
}
```

…but we'll start with variables first.

K&R 157

3.08 Storage classes – `register`

- The **`register`** storage class is the least-used and for most modern purposes (essentially) obsolete. It indicates that the compiler should use one of the CPU's registers to store a value. Registers allow for extremely efficient, high-speed processing, so anything that gets used a lot should be stored in a register. However, this request is optional, and most C compilers ignore it; it is taken, at best, as a directive to mean "this variable gets high usage, and should be treated specially."

- If you need to use a register, the best way to do so is to use the `asm` keyword. In gcc, see, for example, https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html or http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

- Different compilers use different methods to insert assembly code into C. The following, for example, is code used by an old MS C compiler:

```
for (j=0; j < MAX; j++){  //start asm code…
   _asm \
      push di
      push es
      mov ax, ds
      …
```

3.08 Storage classes – `auto`

- **`auto`** is the most commonly used storage class, and is the default—which is why you never see it. Whenever you write

```
float myVariable;
```

inside a function, you're really writing:

```
auto float myVariable;
```

- The `auto` storage class implies that a variable has local scope; it cannot be seen outside the current scope. *It exists only inside a block determined by { } from its point of declaration onward*, and vanishes when you exit that block.

3.08 Storage classes – auto

- For example:

```
int main (void){
    int A = 0;  \\ automatic storage class; A is visible
    printf("A = %d", A++);
    {              \\ new block means new scope
        int B = 0;
        printf("A = %d; B = %d;", A++, B++); \\A, B visible
    }
    printf("A = %d; B = %d;", A, B); \\ A visible, B not
}  // neither A nor B visible after closing brace
```

Scope of A

Scope of B

- What we call 'scope' is in fact a by-product of the way information is stored in memory. Local `auto`-declared variables are stored on the stack each time a new {} block is entered; when the block is exited, the variable disappears from the stack. This is the subject of a later discussion.

3.08 Storage classes – `auto`

- By default, `auto` storage class variables are set equal to whatever was stored in their memory location when the program began. When unsure, initialize your variables to 0, or you may wind up with a random number as the default value of the variable at startup.

- The other main storage classes, `static` and `extern`, set their variables to 0 at startup.

3.08 Storage classes – `static`

- In contrast to `auto`, when a locally-scoped variable is declared `static` it retains its value when a block of code is re-entered. For example:

```
void timesCalled(char c){
  static int ctr = 0;  // ctr=0 on first entry
  if (c=='i')
    printf("function called %d times\n", ++ctr);
  else if (c=='r')
     printf("resetting function to %d", (ctr=0));
}
```

Note that `static` changes the lifetime of the variable, but not its scope; it is still only visible inside the block in which it is declared

K&R 70 | K&R 171

3.08 Storage classes – `static`

- When a `static` variable is declared *external* to a block, the variable's scope becomes global within the file in which it is declared; it is accessible to every function after the point of declaration, but not outside of the file:

```
void f(void){
   …          // v is not available here
}

static int v; //static external variable declaration
              // global, but only in this file
              // and for the remainder of the file

void g(void){
   …          // v is visible to this function
              // and every function afterward
}
```

  Thus variables declared as `static` in a file essentially act as 'private' variables within that file.

- Note that `static` has a different meaning when used globally (in this context) than when it is used locally (see earlier example).

3.08 Storage classes – the scope of a variable

- Here's a real-world example* that uses a globally-declared `static` variable to demonstrate the operation of a pseudo-random number generator. Assume this information is in a separate file from `main()`:

```
#define    INITIAL_SEED  17
#define    MULTIPLIER    25173
#define    INCREMENT     13847
#define    MODULUS       65536

static unsigned int seed = INITIAL_SEED;

unsigned int random(void){
   seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
   return seed;
}
```

Each time this function is called, `seed` remains equal to its last determined value—and then changed within the function. The function `random()` effects a pseudo-random number generator.

*Most of the examples on storage class are taken from *A Book on C 4e*, I. Pohl and A. Kelly, Addison-Wesley, 1998, pg. 221-222

3.08 Storage classes – `extern`

- The `extern` storage class signals that a variable is "to be found elsewhere". This is the default when variables are declared outside a function block. For example in:

```
#include <stdio.h>
int a=1, b=2, c=3; // considered to exist extern-
                   // ally, even if not
                   // explicitly specified
int main (void){
   a++;
}
```

the variables `a`, `b`, and `c` are global to `main()` (and all other functions).

So variables declared *locally* in a function are considered to be `auto`, but when declared *globally* are `extern`. This also applies to files declared as forward declarations, outside `main()`.

K&R 185

3.08 Storage classes – extern

Consider the following example. In file1.c we have:

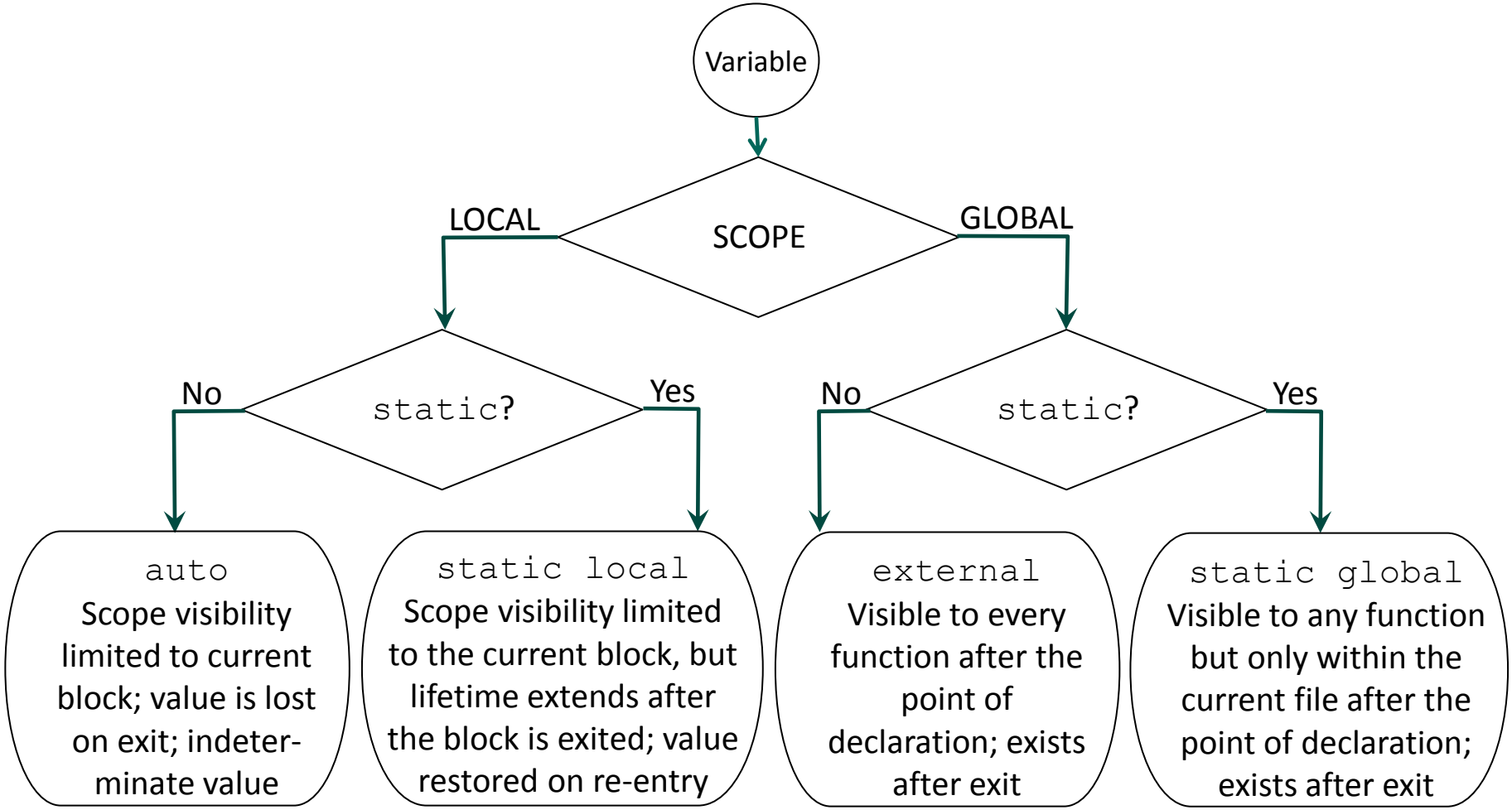
```
#include <stdio.h>

int A=1, B=2, C=3;  // implicitly external ie. global variables
int f(void);        // forward declaration; implicitly external

int main(void){
   printf("%d\n", f());    // Output: 12
   printf("%d %d %d", A + B + C);  // Output (4+2+3 =) 9
}
```

and in file2.c:

```
int f(void){
   extern int A;           // look for variable A elsewhere
   int B, C;               // local versions of B and C
   A = B = C = 4;          // set all values to 4
   return(A + B + C);      // return 12, as seen above
}
```

*Most of the examples on storage class are taken from *A Book on C 4e*, I. Pohl and A. Kelly, Addison-Wesley, 1998, pg. 218

3.08 Storage classes – Summary (variable declarations)



3.08 Storage classes – functions – using `static`

- Just as functions can be `extern`—essentially public, for use in every function, across separate files—they can also be made 'private' by using the word `static`. A function declared `static` is visible only inside a file; its scope is restricted to that file alone.

```
static int g(void);  // a forward declaration

void f(int a) {
   …      // g() is available here
          // because we referenced it above
          // but it is not available outside
          // this file
}

static int g(void) {
   …
}
```

*Most of the examples on storage class are taken from *A Book on C 4e*, I. Pohl and A. Kelly, Addison-Wesley, 1998, pg. 221-222

3.08 Storage classes – functions – using `extern`

- Note that `#include` and `extern` may have the same overall effect—they both make functions and variables located in external files accessible to `main()`. But they operate in very different ways:

  `#include` is a preprocessor directive that says: "copy the files here before continuing with compilation"

  `extern` is a command that says to the compiler: "look for this variable or function elsewhere". Note that functions declared with `extern` will generally be forward declarations, which must be defined elsewhere in a .c file, either in the main file itself, or elsewhere.

  As with `#define` and `const` or `typedef`, `#include` is a C preprocessor directive; `extern` is processed at compile time.

3.08 Storage classes – compiling `extern` functions

- When compiling `extern` variables and functions (whether implicitly or by actually using `extern`) using gcc, make sure you compile all the files that contain the C code being used. For example, using file1.c and file2.c from a few slides ago, you would compile this using:

```
gcc –o OutputFileName file1.c file2.c
```

If you `#include` one file in another file, then the contents of the first file are coped directly into that file *prior to* compilation. In this case there is no need to add extra files to the gcc parameter list. As long as the pre-compiler can find the `#include`d information, *you don't need to compile each file separately*.

But when one file contains protected information (i.e. declared static), that file should be compiled separately. Once you include one file inside another, all of the protections afforded to identifiers by the `static` storage class are lost.

3.09 Declarations versus definitions

```
void setMaxValue(double);
```

- A ***declaration***, like the forward declaration listed above, is a statement of intent about what data types a function will take as arguments, and what data type it will return. Declarations are used by the compiler to help it verify that your code will function correctly when it is actually employed during execution. However, no memory is allocated by this statement—it merely alerts the compiler of your intention to provide a complete definition at a later stage.

2.15 Declarations versus Definitions

```
void setMaxValue(double);
```

- A declaration represents a contract between the programmer and the program indicating how memory is *intended* to be used. A declaration keeps the programmer honest; if at some later point in the program the programmer attempts to use memory in an unorthodox way, the compiler will protest. Without a proper declaration, the compiler has no way of knowing whether memory is being used legitimately or not, and so has no way to warn the programmer about the dangers inherent in their code.

3.09 Declarations versus definitions

```
void setMaxValue(double newNum){
   if ((numSamples == 1) || (newNum > maxNum))
      maxNum = newNum;
}
```

- By contrast, a ***definition*** occurs when you specify the code itself. This results in space actually being allocated to store the executable code.

- For variables, the situation is somewhat simpler. When you say:

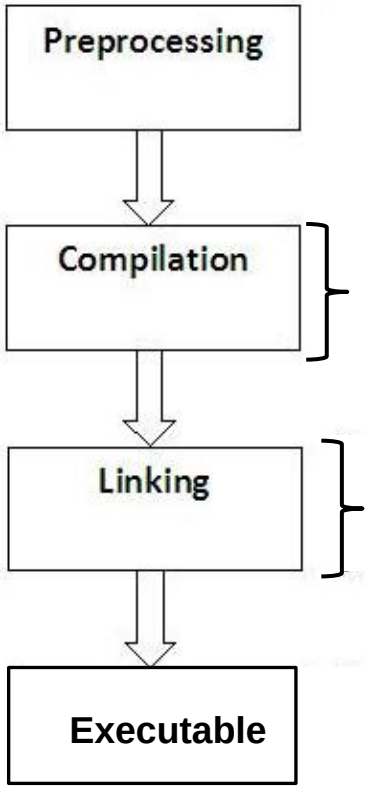
  ```
  int ctr = 0;
  ```

  you are both declaring and defining a variable: the intent of the variable is declared with the line `int ctr` and the assignment of a value (0)—complete with memory allocation—is made at the same time.

- Note that a function definition *includes* a declaration in its header

3.09 Declarations versus definitions

- The difference between a declaration and a definition is most obvious in the case in which an external declaration is made to a function whose code does not exist in the current file. The effect of `extern` is to tell the compiler to wait for the linker to find the missing code before generating the executable; until that time, an `extern` identifier is declared, but undefined. Only during linking are these unresolved connections to external code finally made, and the link to the executable code stored in memory finally completed



`extern` is processed at this stage…

…but its net effect is to delay final assembly of the program until the linker stage

K&R 187

Review - Example 11: The SimpleStats.c library

- We can now correct our library with the appropriate storage class modifiers. First, we can hide the values of `numSamples`, `maxNum`, `minNum` and the `average` from the calling program:

```
#include <float.h>
#include <assert.h>
#include <stdio.h>

#define ERROR FLT_MAX

static unsigned int numSamples = 0;
static double maxNum;
static double minNum;
static double average=0;
```

Review - Example 11: The SimpleStats.c library

- Next, we can make our `set_()` functions 'private':

```
static void setMaxValue(double newNum){
   if ((numSamples == 1) || (newNum > maxNum))
      maxNum = newNum;
}

static void setMinValue(double newNum){
   if ((numSamples == 1) || (newNum < minNum))
      minNum = newNum;
}

static void setAveValue(double newNum){
   average = (((numSamples-1) * average) + newNum)/numSamples;
}

void setValue(double newNum){
   assert(newNum != ERROR);
   numSamples++;
   setMaxValue(newNum);      // set_() ftns visible in this file
   setMinValue(newNum);
   setAveValue(newNum);
}
```

Review - Example 11: The SimpleStats.c library

- The `get_()` functions remain external; since they are outside any function, they are global and visible to any calling program.

```
void reset(void){
   maxNum = minNum = average = numSamples = 0;
}

double getMaxValue(){
   return (numSamples?maxNum:ERROR);
}

double getMinValue(){
   return (numSamples?minNum:ERROR);
}

double getAveValue(){
   return (numSamples?average:ERROR);
}

unsigned int getNumSamples(void){
  return (numSamples);
}
```

Review - Example 11: The SimpleStats.c library

- Finally, we can call the `SimpleStats.c` library from another file (Ex11.c)

```
// …include system files here
#include "SimpleStats.h"  // our user-defined header file
int main(void){
   int num = 0, ctr, numValues = 10;
   printf("Enter the number of integers to be entered : ");
   scanf("%d", &numValues);
   assert(numValues > 0);
   for (ctr = 0; ctr < numValues; ctr++){
      printf("Enter an integer : ");
      scanf("%d", &num);
      setValue((double)num);
   }
   printf("Max Value is : %3.2lf\n", getMaxValue());
   printf("Min Value is : %3.2lf\n", getMinValue());
   printf("Average Value is : %3.2lf\n", getAveValue());
   return EXIT_SUCCESS;
}
```

Review - Example 11: The SimpleStats.c library

- Here, we've used a separate .h file to include the following information:

```
extern void setValue(double);
extern double getMaxValue();
extern double getMinValue();
extern double getAveValue();
extern unsigned int getNumSamples(void);
```

We could, of course, insert these function prototypes directly into the top of Ex11.c where `#include SimpleStats.h` appears in the code.

- Note that although the above functions are 'visible' without an explicit declaration using extern, the gcc linker has trouble determining the return type of the functions without a proper forward declaration. Therefore, these function prototypes are essential if the program is to work correctly.

Review - Example 11: The SimpleStats.c library

- To compile this code, you must include the names of each .c file to be compiled:

```
gcc Example11.c SimpleStats.c –o Ex11
```

- System files work no differently. When you type

```
#include stdio.h
```

into your program, you are inserting a number of forward declarations that look like this:

```
extern struct _IO_FILE *stdin;  /* Standard input stream.  */
extern struct _IO_FILE *stdout; /* Standard output stream. */
extern struct _IO_FILE *stderr; /* Standard error output.  */
extern int fflush (FILE *__stream);
```

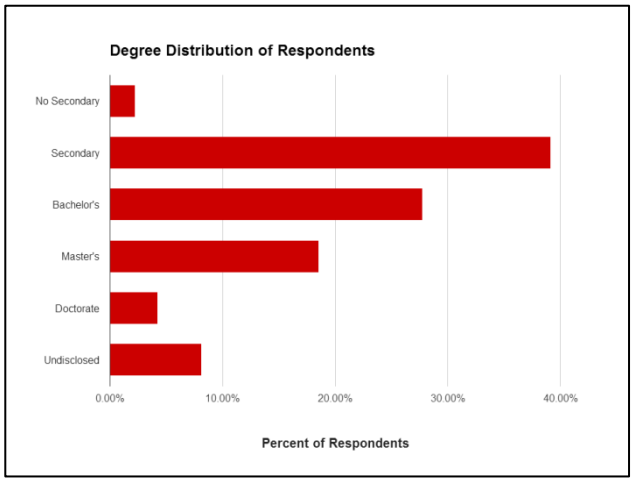
…which are just forward declarations to functions and variables in internally-used .c files.

Programming Challenges

1. The `ERROR` value used in Example 11 stored is used internally within the `SimpleStats.c` library. How would you make this value globally accessible to client functions in such a way that it cannot be altered by those clients?

2. Add a smoothing function to the `SimpleStats.c` file. This is just the running average calculation from lab 3 modified with appropriate storage class protections. You'll want to declare a static global array inside `SimpleStats.c` to hold the values entered, as well as appropriate 'helper functions' and variables (like `start` and `finish`), many of which will be `static`, and hence available only inside `SimpleStats.c` itself. Finally, you'll need to make your new smoothing function available to the outside world via an `extern` forward declaration in `SimpleStats.h`.

Example 12 : The Distribution Display Library



Program Description:

This library contains a set of functions that allow the user to output a distribution function to the screen, along with the data that generated the output. The number of input bins is variable up to a suitably-chosen maximum, and various helper functions are available to the client program to increment bins individually, set bin values directly, and reset the library prior to fresh output.

Note: only the vertical output is coded (like the one shown top right, and on the next slide); the more traditional horizontal distribution is left as a programming challenge.

Example 12: The distribution display library

```
houtmad@ubuntu:~/examples$ Ex12
Bin # 1 :        33
Bin # 2 :       480
Bin # 3 :      4407
Bin # 4 :     31303
Bin # 5 :    170154
Bin # 6 :    737557
Bin # 7 :   2629513
Bin # 8 :   7892724     **
Bin # 9 :  20150312     ******
Bin #10 :  44359731     **********
Bin #11 :  84658796     ************************
Bin #12 : 141127232     ****************************************
Bin #13 : 206243631     ************************************************************
Bin #14 : 265185457     ********************************************************************************
Bin #15 : 300566223     ************************************************************************************
Bin #16 : 300525426     ************************************************************************************
Bin #17 : 265182981     ********************************************************************************
Bin #18 : 206265867     ************************************************************
Bin #19 : 141110764     ****************************************
Bin #20 :  84672430     ************************
Bin #21 :  44347082     **********
Bin #22 :  20157877     ******
Bin #23 :   7891108     **
Bin #24 :   2629979
Bin #25 :    736858
Bin #26 :    169283
Bin #27 :     31384
Bin #28 :      4591
Bin #29 :       436
Bin #30 :        28
Bin #31 :         0
Bin #32 :         0
```

3.10 Using multiple files in a program

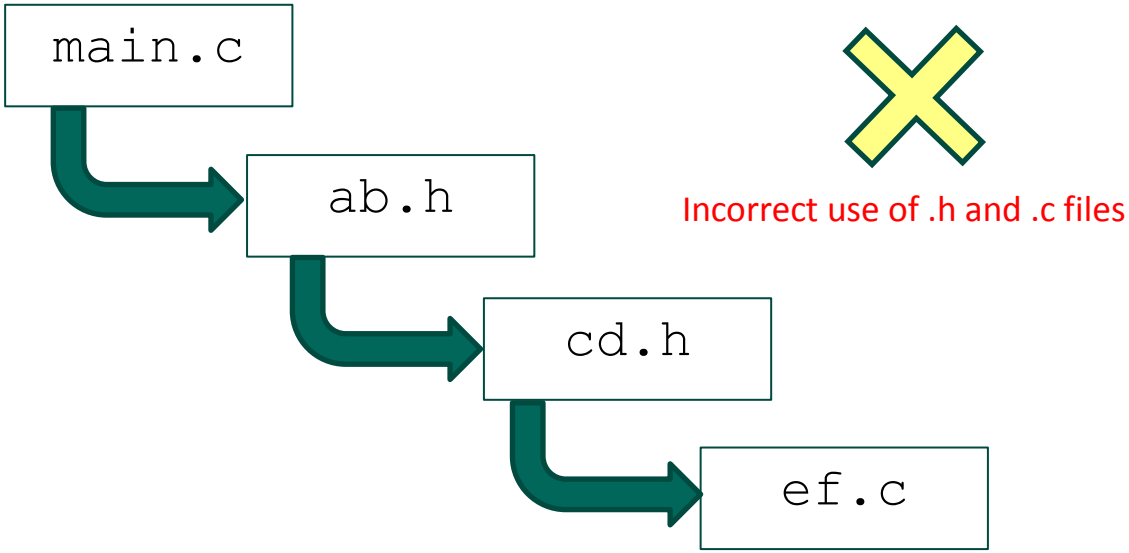
- Each C program typically consists of one or more ***source files*** (also sometimes called ***translation units***). Each source file contains some part of the entire C program. As with Java, some of these files will contain pre-written code libraries (like `stdio.h` and `limits.h`), while other files will contain your own hand-crafted code. In general there are two kinds of C files:
  - files ending in '.c', which store the actual C code, and
  - files ending in '.h', which are ***header files*** that contain references to files stored elsewhere, along with function prototypes and system defined constants. Thus `stdio.h` doesn't contain actual C code, but instead it contains a list of references for all of the functions and variables stored in the standard I/O library. The actual .c code is located elsewhere; the .h file merely contain handles to the code.

3.10 Using multiple files in a program

- The difference between .c and .h files is somewhat superficial, since the compiler won't complain if you confuse the contents of the two types of files or break any of the 'rules' listed below. However, in general:
  - `.h` files should act like a "table of contents" to help simplify access to other files. They may contain `#defines` and other CPP directives and defined constants, but in general they should NOT contain .c code;
  - By convention, aside from some CPP directives and .h references, `.c` files should contain mainly C code;
  - Code performing roughly the same function should be contained within the same `.c` file, not spread out over several modules;
  - `.h` files may contain `#includes` to `.c` files, and vice versa; each will be opened recursively inside the file in which it is found by the CPP.

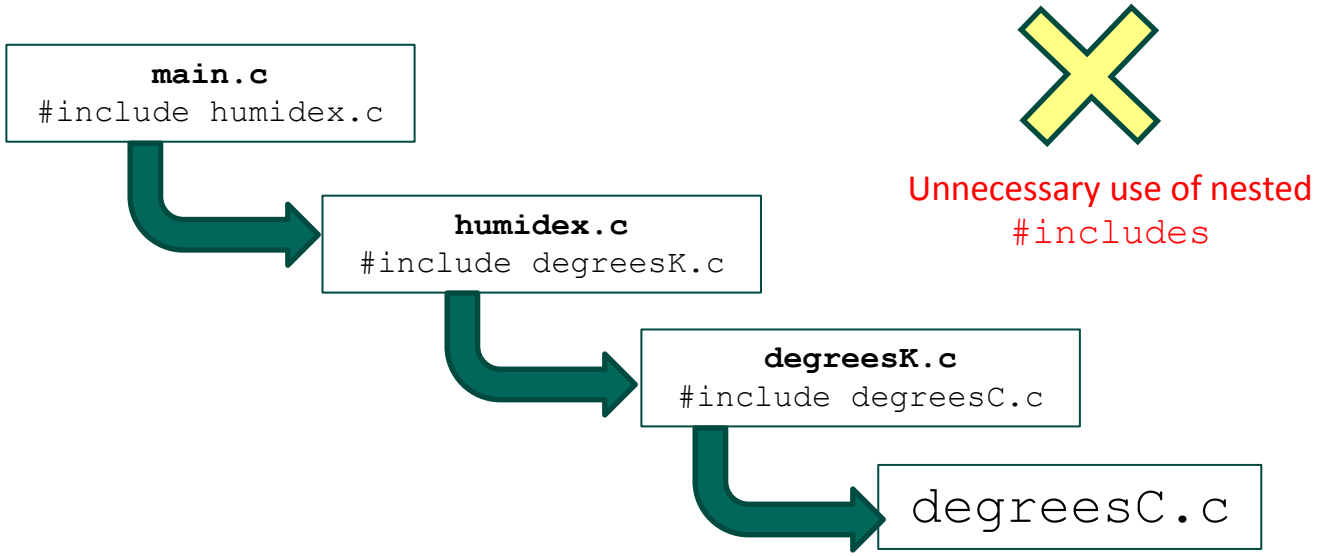
3.10 Using multiple files in a program

- Do NOT link your .h and .c files together like a 'string of pearls'. For example, the following is (most probably) wrong:



3.10 Using multiple files in a program

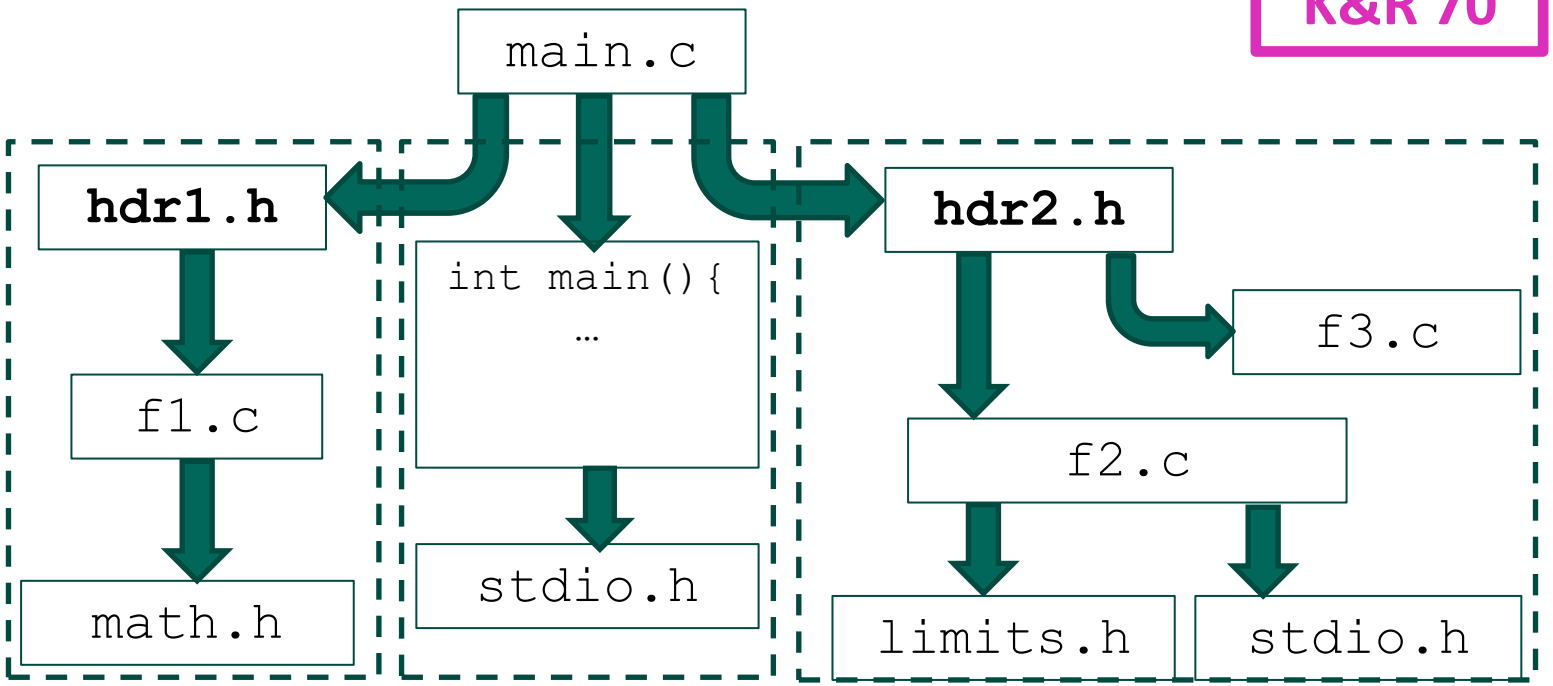
- For example, imagine the file that calculates Humidex includes reference to a file called `degreeK.c`, which in turn includes a file, `degreesC.c` that contains Celsius-related functions. This is overkill.



3.10 Using multiple files in a program

- The following is a more-typical structure of .h and .c files. Remember, .c Files should contain functions, variables, structures and constants that are related by a common purpose; each .h file typically acts as a 'table of contents' that organize other .c and .h files., and may contain constants and other (possibly external) declarations.

K&R 70



Example 12: The distribution display library

- The library declares an array of unsigned integers to hold the value stored in each bin. Since this number must be positive, an `unsigned int` has been `typedef`ed as `UINT`.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_BIN_NUMBER 64
#define SUCCESS 1
#define FAILURE 0
#define DEFAULT_CHAR '*'
#define BIN_NUM_OK ((binMax > 0) && (bNum <= binMax))

typedef unsigned int UINT;

static UINT bin[MAX_BIN_NUMBER] = {0};      // play it safe
static UNINT binMax = 0, charCtr, binCtr;
static char displayChar = DEFAULT_CHAR;
static float scaleFactor = 1.0;
```

Example 12: The distribution display library

- Since the number of usable bins can be any value up to the *maximum* size of the array, a variable `binMax` is used to set the *effective* bin size (the maximum value is 64). This will be the first thing the client sets before loading the bins, so we must make sure the value entered is correct:

```
UINT setMaxBinSize(UINT bMax){

   if ((bMax <= MAX_BIN_NUMBER) && (bMax > 0))
      return (binMax = bMax);
   else {
      printf("Error: Max bin size requested is 0 (or less), or"
      " it exceeds bin size available (=%u)\n", MAX_BIN_NUMBER);
      return FAILURE;
   }

}
```

Example 12: The distribution display library

- The library contains two functions that allow the client to change bin values. The first increments the value stored in a bin by 1; the second allows a value to be set directly.

```
UINT incrementBinValue(UINT bNum){
    if (BIN_NUM_OK && (bin[bNum-1] < UINT_MAX)) {
      ++(bin[bNum-1]);
      return SUCCESS;
   }
   else return FAILURE;
}

UINT setBinValue(UINT bNum, UINT bValue){
    if (BIN_NUM_OK && (bin[bNum-1] <= UINT_MAX)) {
       bin[bNum-1] = bValue;
       return SUCCESS;
   }
   else return FAILURE;
}
```

Example 12: The distribution display library

- In addition to these `set_` methods we can also get the value of any bin:

```
UINT getBinValue(UINT bNum){
   if (BIN_NUM_OK && (bNum > 0))
      return(bin[bNum]);
   else{
      printf("Bad bin number; exiting\n");
      exit(EXIT_FAILURE);
   }
}
```

Example 12: The distribution display library

- We may need to scale our distribution up or down, depending on the number of values. The following function allows the client to control the width of the output displayed by altering the value of the static variable `scaleFactor`: If successful, the value of `scaleFactor`, cast as an `unsigned int`, is returned.

```
UINT setScaleFactor(float sf){
   if (sf > 0){
      scaleFactor = sf;
      return (UINT)scaleFactor;
   }
   else
      return FAILURE;
}
```

Example 12: The distribution display library

- Finally, the distribution is displayed using two nested `for` loops. The outer loop gets the bin number and the value in each bin, while the inner loop displays a number of characters across the screen in proportion to the value stored in the current bin; the scale factor controls the actual width of the characters displayed on each line of the console window.

```
void displayVertDistribution(){
   for (binCtr = 0; binCtr < binMax; binCtr++){
      printf("Bin #%2u : %10u\t", (binCtr+1), bin[binCtr]);
      for(ctr = 0; ctr<(UINT)(scaleFactor * bin[binCtr]); ctr++)
         printf("%c", displayChar);
      printf("\n");  \\ advance to next line
   }
   printf("\n");
}
```

Example 12: The distribution display library

- The header file for this library contains the following declarations:

```
typedef unsigned int UINT;

/* See Distribution.c for definitions */

extern UINT setBinValue(UINT, UINT);
extern UINT incrementBinValue(UINT);
extern UINT getBinValue(UINT);
extern UINT setMaxBinSize(UINT);
extern void displayVertDistribution();
extern UINT setScaleFactor(float);
```

Example 12: The distribution display library

- We now proceed to test this library by calling it from `main()` in three separate examples. The first test is a simple one:

```
// #includes go here
#include "Distribution.h"

int main(void){
   unsigned int numBins=0, binNum, binValue, binValueOK;
   printf("Enter the number of bins : ");
   scanf("%u", &numBins);
   assert(setMaxBinSize(numBins));
   for (binNum = 1; binNum <= numBins; binNum++){
      do {
         printf("Enter bin %u value : ", binNum);
         scanf("%u", &binValue);
         binValueOK = setBinValue(binNum, binValue);
         if (!binValueOK) printf("Attempt to enter bad value"
               " in bin #%u; Please re-enter\n", binNum );
      } while (!binValueOK);
   }
   displayVertDistribution();
}
```

Example 11: The distribution display library

```
houtmad@ubuntu:~/examples$ Ex12
Enter the number of bins : 10
Enter bin 1 value : 4
Enter bin 2 value : 8
Enter bin 3 value : 3
Enter bin 4 value : 9
Enter bin 5 value : 11
Enter bin 6 value : 13
Enter bin 7 value : 8
Enter bin 8 value : 5
Enter bin 9 value : 2
Enter bin 10 value : 1
Bin # 1 :          4     ****
Bin # 2 :          8     ********
Bin # 3 :          3     ***
Bin # 4 :          9     *********
Bin # 5 :         11     ***********
Bin # 6 :         13     *************
Bin # 7 :          8     ********
Bin # 8 :          5     *****
Bin # 9 :          2     **
Bin #10 :          1     *

houtmad@ubuntu:~/examples$
```

Example 12: The distribution display library

- Our second test (Example12a.c) is somewhat more sophisticated. We wish to (1) count the total number of 1's found in a 32-bit random number, and (2) increment the bin number corresponding to that value, and then (3) output the distribution. We would expect that, on average, most random-valued ints will consist of numbers with sixteen 1's and sixteen 0's, but on rare occasions a random number will consist entirely of 1's, or perhaps have no 1's at all. We thus expect a bell-shaped distribution centered around 16 and quickly dropping off to 0.

  Before beginning, a review of binary operations is necessary.

3.11 Binary operations

- Binary operations are similar to Boolean operations. But while the latter treat a variable as a single entity—the value stored is non-zero or zero, hence true or false—binary operations act on all of the bits stored in a variable simultaneously.

K&R 45

Symbol	Operation
&	Bitwise AND – if both of the bits are both 1, the result is 1
|	Bitwise OR – if either of the bits are 1, the result is 1
^	Bitwise XOR – if the two bits are the same, the result is 0
~	Bitwise NOT – each bit value becomes its opposite, i.e. 1 → 0 and 0 → 1
<<	Shift the bits in the left operand to the left according to the value in the right operand
>>	Shift the bits in the left operand to the right according to the value in the right operand, and fill the left most bits with '0's

3.11 Binary operations

Bitwise AND (&)

- The bitwise AND, represented by &, functions just like its logical counterpart, &&, but on a bitwise level

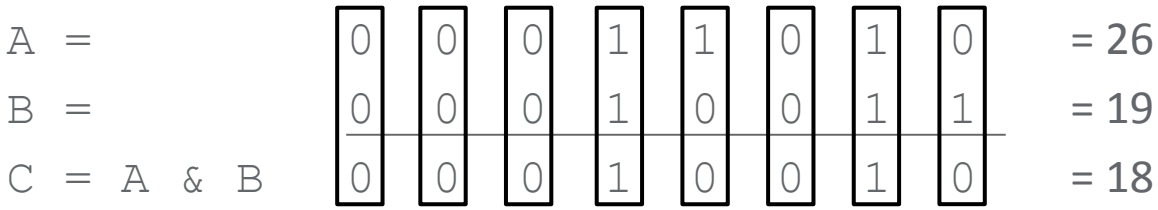
How does this work? Assuming the following byte-sized values:

A = 26, which is 00011010 in binary

B = 19, which is 00010011 in binary

For the bitwise AND operation (&), treat each 1 as true, each 0 as false.

Then



3.11 Binary operations

Bitwise OR (|)

- The bitwise OR, represented by |, functions just like its logical counterpart, ||, but on a bitwise level

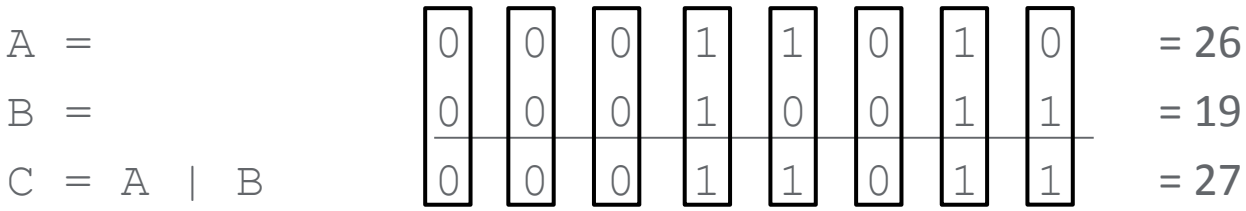
  Using the same values as before

  A = 26, which is 00011010 in binary

  B = 19, which is 00010011 in binary

  For the bitwise OR operation (|), treat each 1 as true, each 0 as false.

  Then



3.11 Binary operations

Bitwise XOR (^)

- The bitwise XOR, represented by ^, functions just like its logical counterpart, ^, but on a bitwise level

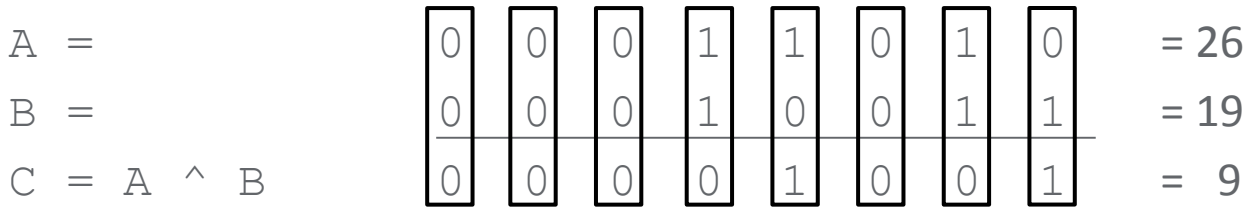
Again, using

`A` = 26, which is 00011010 in binary

`B` = 19, which is 00010011 in binary

For the bitwise XOR operation (^), treat each 1 as true, each 0 as false.

Then



3.11 Binary operations

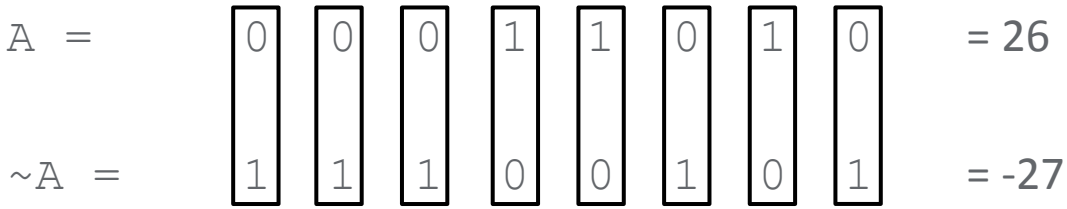
Bitwise NOT (~)

- The bitwise NOT, represented by ~, functions just like its logical counterpart, !, but on a bitwise level

Again, assume

A = 26, which is 00011010 in binary

For the bitwise NOT operation (~), change each 1 to a 0, each 0 to a 1:



Note: Don't confuse the bitwise NOT (~) with the logical NOT (!). The latter converts non-zero values to 0, and 0 to 1—something quite different from the bitwise NOT.

3.11 Binary operations

Bitwise Shifts (<<, >>)

- The bitwise shift operators have the effect of moving the bits stored in a variable left or right by the number of bits indicated.

A = 10, which is 00001010 in binary

B = 15, which is 00001111 in binary

The following example explains how these three operators are used:

```
A << 3;  //00001010 is shifted left 3 times: A =01010000 =80
A >> 2;  //Now 01010000 is shifted right twice: A =00010100 =20
```

3.11 Binary operations

- One possible way to calculate the number of 1's in an integer is to successively shift the bits to the right and, if the rightmost bit is 1, increment the total number of 1's. To do this, we need to create a ***bitmask***, a number that consists of 0's and 1's that masks out the bits we don't want, and leaves the ones we do want. To view only the rightmost bit, we'll need a bitmask of `0x01`, which is just the hexadecimal version of the 32-bit binary number

  `00000000000000000000000000000001`

  ANDing this value with any 32-bit number results in only the rightmost digit appearing in the output. So this code produces the desired result:

```
unsigned int bitCtr, num1s = 0;
for(bitCtr = 0; bitCtr < 32; bitCtr++){
   num1s += (0x1 & uI);
   uI >>= 1;
}
```

Example 12a: The distribution display library

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

#include "Distribution.h"
#define ASSUMED_SCREEN_WIDTH 80

const int szOfUInt = 8 * sizeof(unsigned int);

unsigned int getNumOf1s(unsigned int uI){
   unsigned int bitCtr, num1s = 0;
   for(bitCtr = 0; bitCtr < szOfUInt; bitCtr++){
      num1s += (0x1 & uI);
      uI >>= 1;
   }
   return(num1s);
}
...
```

Example 12a: The distribution display library

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

   unsigned int randCtr, binNum, binValueOK, randNum;

   assert(setMaxBinSize(32));

   srand(time(NULL));
   setScaleFactor((float)ASSUMED_SCREEN_WIDTH *
                           8/RAND_MAX);
   for (randCtr = 0; randCtr < RAND_MAX; randCtr++){
      randNum = ((double)rand()/(RAND_MAX)) * 0xFFFFFFFF;
      binNum = getNumOf1s(randNum);
      (void) incrementBinValue(binNum);
   }

   displayVertDistribution();

}
```

Example 12a: The distribution display library

```
houtmad@ubuntu:~/examples$ Ex12a
Bin # 1 :        29
Bin # 2 :       475
Bin # 3 :      4510
Bin # 4 :     31285
Bin # 5 :    169832
Bin # 6 :    736300
Bin # 7 :   2631445
Bin # 8 :   7890824     ##
Bin # 9 :  20163733     ######
Bin #10 :  44338271     #############
Bin #11 :  84689597     ##########################
Bin #12 : 141145823     ############################################
Bin #13 : 206262839     ##################################################################
Bin #14 : 265180847     ###########################################################################
Bin #15 : 300527838     ###################################################################################
Bin #16 : 300517238     ###################################################################################
Bin #17 : 265177402     ###########################################################################
Bin #18 : 206249727     ##################################################################
Bin #19 : 141128441     ############################################
Bin #20 :  84667396     ##########################
Bin #21 :  44351654     #############
Bin #22 :  20155737     ######
Bin #23 :   7888792     ##
Bin #24 :   2630969
Bin #25 :    736001
Bin #26 :    170053
Bin #27 :     31507
Bin #28 :      4576
Bin #29 :       469
Bin #30 :        34
Bin #31 :         0
Bin #32 :         1
```

Example 12b: The distribution display library

- Assume company XYZ wishes to display last year's monthly production of widgets, along with the minimum monthly output, the maximum output, and the average monthly output. In our third and final example (Example12b.c) we combine the `Distribution.c` library with the `SimpleStats.c` library. This will be compiled using

```
gcc Example12b.c SimpleStats.c Distribution.c -o Ex12b
```

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "Distribution.h"
#include "SimpleStats.h"

#define ASSUMED_SCREEN_WIDTH 80
```

Example 12b: The distribution display library

```
int main(void){

   unsigned int numBins=12, mthNum, binValue, binValueOK;
   assert(setMaxBinSize(numBins));  // Distribution.c function

   for (mthNum = 1; mthNum <= numBins; mthNum++){
      do {

        printf("Enter production output for month %u : ", mthNum);
        scanf("%u", &binValue);

        // set bin values -> Distribution.c function
        binValueOK = setBinValue(mthNum, binValue);

        if (!binValueOK) printf("Attempt to enter bad value"
                " for month #%u; Please re-enter\n", mthNum );
       else
           setValue((double)binValue); // SimpleStats.c function

      } while (!binValueOK);

   }
   ...
```

Example 12b: The distribution display library

```
    ...
    system("clear");  // programmatically clear the screen

    printf("\t\t\t\t2013 Monthly Production Values\n\n");

    // Distribution.c functions
    setScaleFactor((float)ASSUMED_SCREEN_WIDTH/6000);
    displayVertDistribution();
    printf("\n");

    // SimpleStats.c functions
    printf("Maximum monthly output"
       " : %u\n", (unsigned int)getMaxValue());
    printf("Minimum monthly output"
       " : %u\n", (unsigned int)getMinValue());
    printf("Average monthly output"
       " : %u\n", (unsigned int)getAveValue());

}
```

Example 12b: The distribution display library

```
houtmad@ubuntu:~/examples$ Ex12b
Enter production output for month 1 : 3772
Enter production output for month 2 : 3655
Enter production output for month 3 : 3648
Enter production output for month 4 : 3396
Enter production output for month 5 : 3100
Enter production output for month 6 : 2404
Enter production output for month 7 : 2609
Enter production output for month 8 : 2899
Enter production output for month 9 : 3555
Enter production output for month 10 : 4011
Enter production output for month 11 : 4196
Enter production output for month 12 : 3836
```

```
                              2013 Monthly Production Values

Bin # 1 :      3772     ###################################################
Bin # 2 :      3655     ##################################################
Bin # 3 :      3648     ##################################################
Bin # 4 :      3396     ################################################
Bin # 5 :      3100     ############################################
Bin # 6 :      2404     ##################################
Bin # 7 :      2609     #####################################
Bin # 8 :      2899     ########################################
Bin # 9 :      3555     ##################################################
Bin #10 :      4011     ########################################################
Bin #11 :      4196     ##########################################################
Bin #12 :      3836     ####################################################


Maximum monthly output : 4196
Minimum monthly output : 2404
Average monthly output : 3423
```

Questions

1. Write a function that takes an unsigned integer as a parameter, and returns a 'true' numerical value (i.e. non-zero) if the number is odd, a 'false' numerical value (i.e. 0) if even—using only binary operators.

2. If you look at the output from Example12a closely, you'll notice that the graph is slightly askew—there are more values closer to 0 1's than to all 1's. Find the error and correct the code.

Questions

3. Rather than shift our random valued `int` to the right each time and bitwise AND it with `0x01`, what would be the result if we had shifted `0x01` left instead, and performed the same operation, i.e

```
unsigned int bitCtr, num1s = 0, oneBitSet = 0x01;
for (bitCtr = 0; bitCtr < szOfUInt; bitCtr++){
    num1s += (oneBitSet & uI);
    oneBitSet <<= 1;
}
```

   What value results from this operation?

4. Estimate the number of calculations required to preform Example12a. There are 32 bit shifts over RAND_MAX number of values. Assume each of the above loops requires 20 assembly language instructions to complete, and each instruction takes 0.3 nsec on a 3-GHz computer. How long does it take this program to output its data?

Questions

5. An alternate—and faster—way to count the number of 1's in an integer data type does not rely on bit shifts at all:

```
int getNumOf1s(unsigned int uI) {
    for (unsigned int bitCtr = 0; uI; bitCtr++)
        uI &= uI - 1;
    return bitCtr;
}
```

Incorporate this into the code for Example12a.c. What effect does it have on the time required to produce the output?

More importantly, how does this work?

(This code was taken, with modifications, from http://stackoverflow.com/questions/109023/how-to-count-the-number-of-set-bits-in-a-32-bit-integer, which contains a spirited discussion on the topic of bit counting)

3.12 Summary of keywords

asm	**default**	**float**	**register**[1]	**switch**
auto	**do**	**for**	**return**	**typedef**
break	**double**	fortran[1]	**short**	union
case	**else**	goto[1]	**signed**	**unsigned**
char	entry[1]	**if**	**sizeof**	**void**
const	enum	**int**	**static**	volatile[2]
continue	**extern**	**long**	struct	**while**

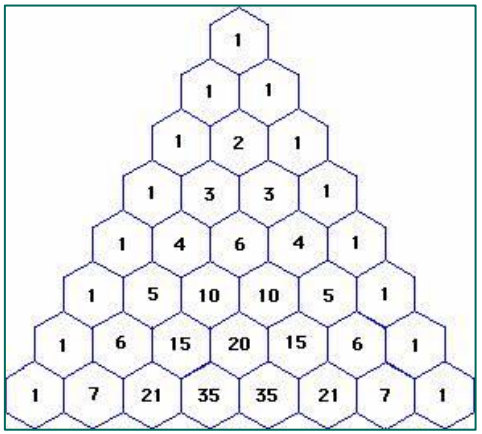
1. Essentially obsolete
2. A volatile object is one that can be changed in some unspecified way in hardware. For example:

   extern const volatile int real_time_clock

   would be a declaration for a variable called real_time_clock of int data type located elsewhere (extern), which should not be changed by software (const) , but which may be acted on by hardware (volatile).

Programming Challenges

The Binomial Coefficient



Program Description:

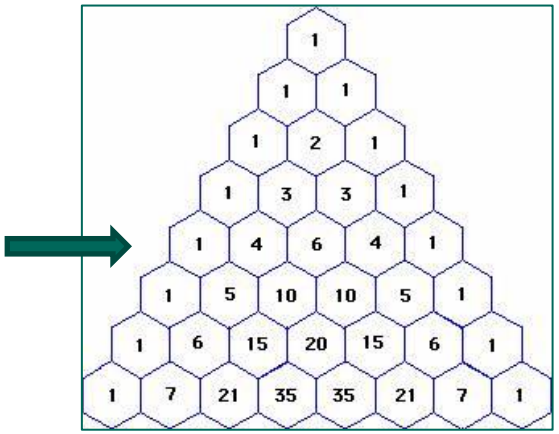
A mathematical operation commonly used in statistics and probability is the binomial coefficient, which has the form:

$$\frac{n!}{k!\,(n-k)!}$$

which we saw earlier on in the programming challenges at the end of Example 09. The results of this calculation are used to create Pascal's Triangle (shown top right), in which the value of any number is equal to the sum of the two numbers above it. It has several practical applications. For example, if you wish to calculate the coefficients of $(x+y)^a$ (for a = 0, 1, 2, 3…), i.e. $(x+y)^4$ then you simply look up the values in the $a+1^{th}$ row of Pascal's Triangle:

Programming Challenges

The Binomial Coefficient (con't)



For example,

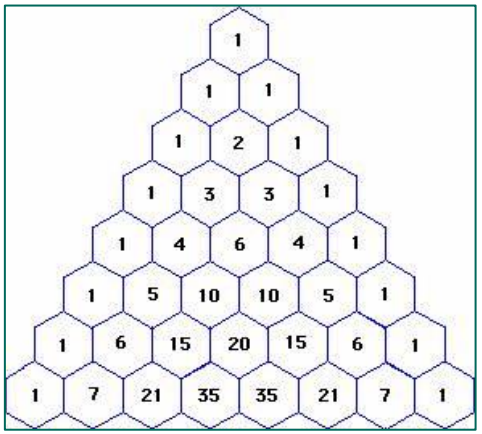
$$(x + y)^4 = \mathbf{1}x^4 + \mathbf{4}x^3y + \mathbf{6}\,x^2y^2 + \mathbf{4}xy^3 + \mathbf{1}y^4$$

The problems that exists with calculating *N!/M!* for large values of N and M (see programming challenges at the end of Example 09) are even worse in the calculation of the binomial coefficient—we have an extra term in the denominator to worry about.

In this programming challenge, write a function that creates two static arrays, each of size 100, and initialized to 1 in each location. The first array will store the values of the numbers left over from the ratio of *n!/(n-k)!*—essentially the same calculation as was done in the programming challenge following

Programming Challenges

The Binomial Coefficient (con't)



Example09.c. So if we wish to calculate

$$\frac{49!}{6!\,(49-6)!}$$

The first step is to store the value left over from the calculation for

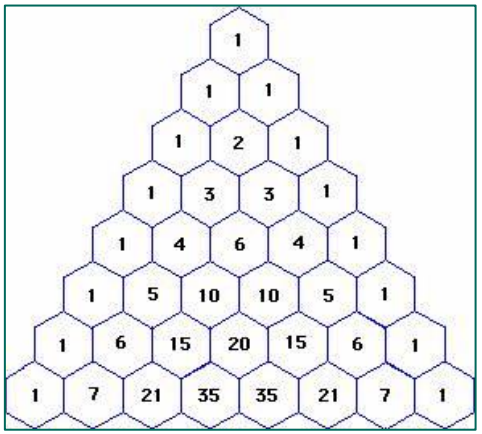
$$49!/_{43!}$$

in our first array:

...	1	1	1	49	48	47	46	45	44

Programming Challenges

The Binomial Coefficient (con't)



In our second array place the other numbers in the denominator of the calculation (the *k!* part):

…	1	1	1	6	5	4	3	2	1

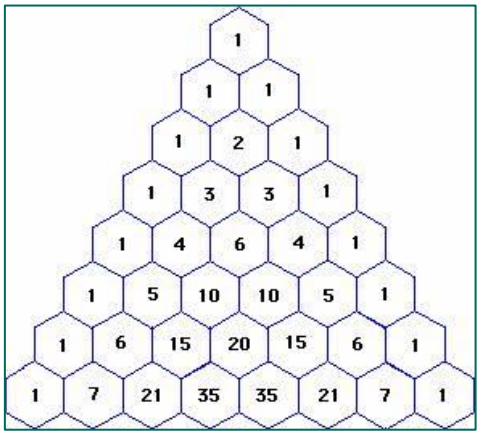
Using the GCD algorithm, divide those numbers in the denominator that evenly divide into those numbers in the numerator, and replace their values with the numbers produced by GCD. For example, given:

Numerator array:	…	1	1	1	49	48	47	46	45	44
Denominator array:	…	1	1	1	6	5	4	3	2	1

use GCD to cancel out the common factors. So the above two arrays

Programming Challenges

The Binomial Coefficient (con't)



become, successively:

… 1 1 1 49 8 47 46 45 44 (48)

… 1 1 1 1 5 4 3 2 1 (6)

… 1 1 1 49 8 47 46 9 44 (45)

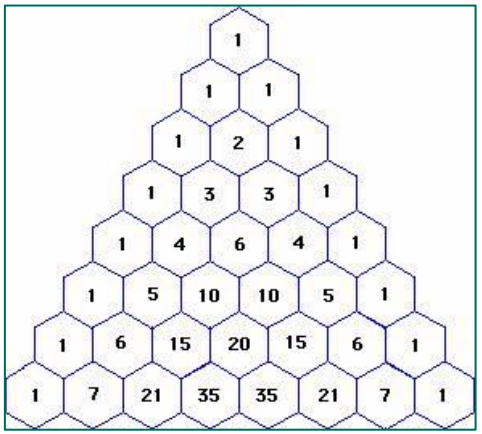
… 1 1 1 1 1 4 3 2 1 (5)

… 1 1 1 49 2 47 46 9 44 (8)

… 1 1 1 1 1 1 3 2 1 (4)

Programming Challenges

The Binomial Coefficient (con't)



etc.

…	1	1	1	49	2	47	46	3	44
…	1	1	1	1	1	1	1	2	1
…	1	1	1	49	2	47	46	3	22
…	1	1	1	1	1	1	1	1	1

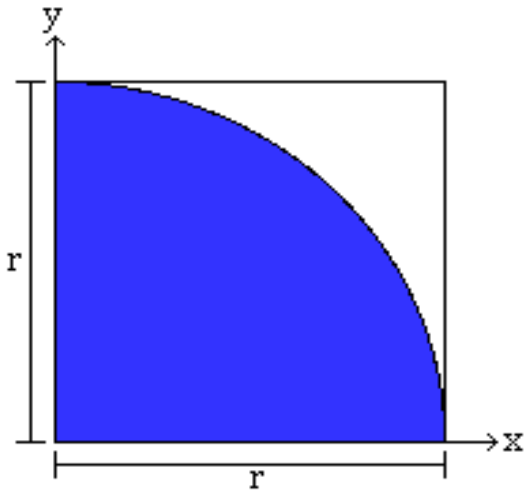
Finally, multiply all the numbers remaining in the numerator array, and divide them by all the numbers remaining in the denominator array. The result will be the solution to *n!/k!(n-k)!*

Note: gcc allows you to initialize an array to non-zero values using:

```
int ar[100] = { [0 ... 99] = 1 }; // non-standard C
```

Programming Challenges

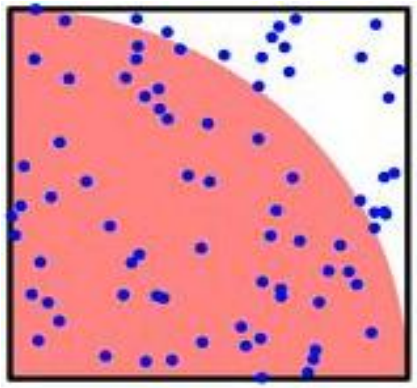
Calculating π using random numbers



Program Description:

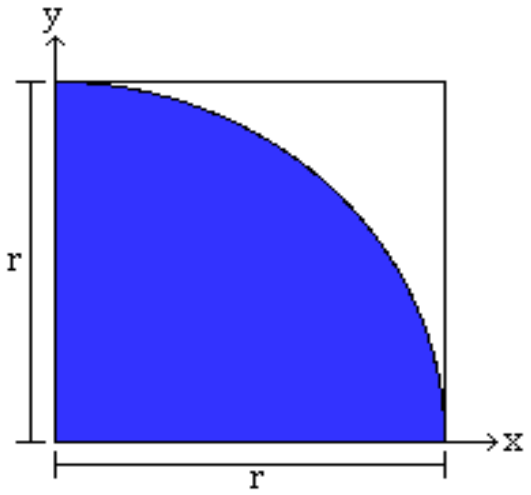
The area of a circle is given by πr^2, where r is the radius. The area of one quadrant of a circle is therefore one quarter of this area, $\pi r^2/4$ (shown above right in blue). Therefore, the ratio of the area of the quadrant to the area of the square it fits in is just $(\pi r^2/4)/r^2$, or $\pi/4$, about 78.3 %.

Imagine you randomly calculate an (x,y) coordinate inside a square 1 unit on each side. The probability (call it P_{2D}) that that coordinate falls *inside the quadrant* is just equal to the ratio of the two areas calculated above, i.e. $P_{2D} = \pi/4$. Rewriting this, we get $\pi = 4P_{2D}$. So if we count up the number of coordinates a distance less than $r = 1$ from (0,0) (i.e. inside the *quadrant*), divide that number by the total number of coordinates randomly selected inside the entire *square*, and multiply this by 4, we should get a value close to π.



Programming Challenges

Calculating π using random numbers (con't)

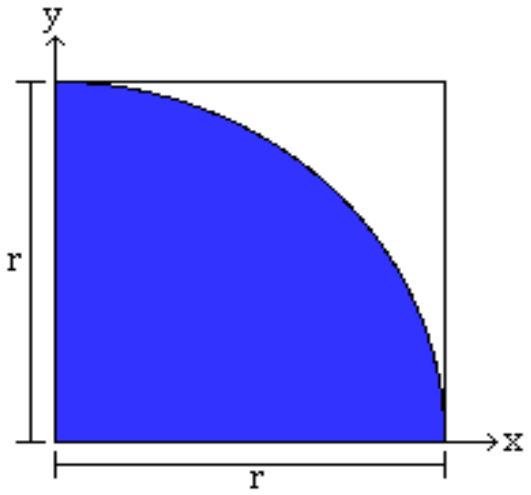


This program requires two functions. The first (call it `pythag2D()`) takes two `double` values and returns their distance from (0,0), i.e. $\sqrt{x^2 + y^2}$, as a `double`. *Each time the second function* (call it `prob()`) *is called from* `main()`, it randomly generates two double values (which must each be less than 1.0, since they must fall inside the square), passes this information to the first function, and calculates the ratio of the number of times the radius returned by `pythag2D()` is less that one divided by the total number of calls to the function; these two values, which are stored internally inside `prob()` are multiplied by 4, and returned to the calling function in `main()`.

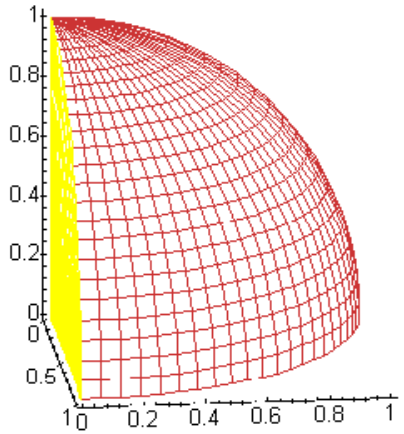
The `main()` function should call `prob()``RAND_MAX` times and give a running output of the value returned, which should approach $\pi$ as time goes on. (Remember that you can use the escape character `\r` to return the cursor back to the start of the current line; if you use `\n`, your screen will be filled with 4 billion numbers…)

Programming Challenges

Calculating π using random numbers (con't)



This concept can be generalized to a 3D-sphere. The volume of a sphere is $4\pi r^3/3$; therefore one octant ($1/8^{th}$ of the sphere) has a volume (1/8) * $4\pi r^3/3$, or $\pi r^3/6$. The probability that a randomly-generated 3D coordinate has a radial distance less than 1 unit from (0,0,0) is therefore given by $P_{3D} = \pi/6$.



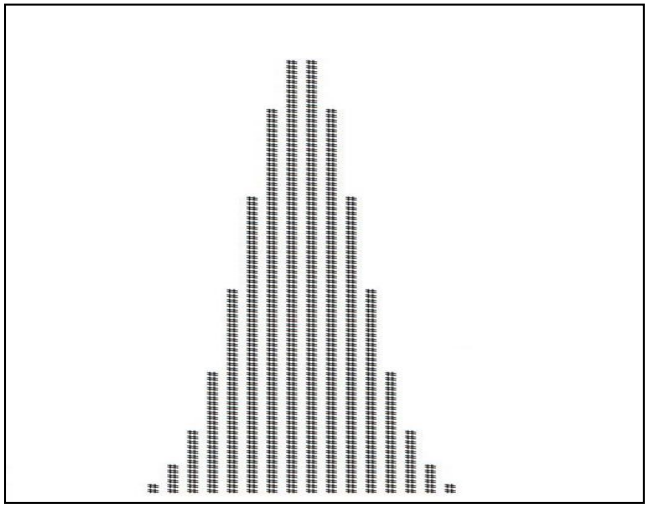
Using a function that calculates the 3D Pythagorean distance between the randomly chosen point and (0,0,0) (call it `pythag3D()`), along with the existing function `prob()`, show that π is approximately equal to $6P_{3D}$.

We can take this one step further. A 4D-hypersphere has a volume of $\pi^2 r^4/2$. Estimate the value of π by calculating the probability, P_{4D} ,that a randomly-generated coordinate will be less than 1 unit from the origin at (0,0,0,0).

Does one of these three methods converge on the value of π faster than the others?

Programming Challenges

The Horizontal Distribution Function (con't)

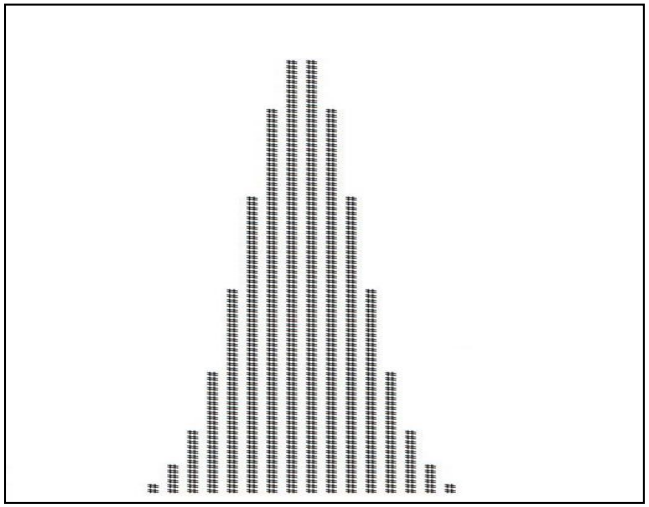


Program Description:

The code for the vertical distribution function was given in Example12; what's missing is the more traditional horizontal distribution function, like the one shown above. Conceptually, the process is straightforward: we wish to 'rotate' the output of the array and display the information horizontally.

Programming Challenges

The Horizontal Distribution Function (con't)



The way to approach this problem is to

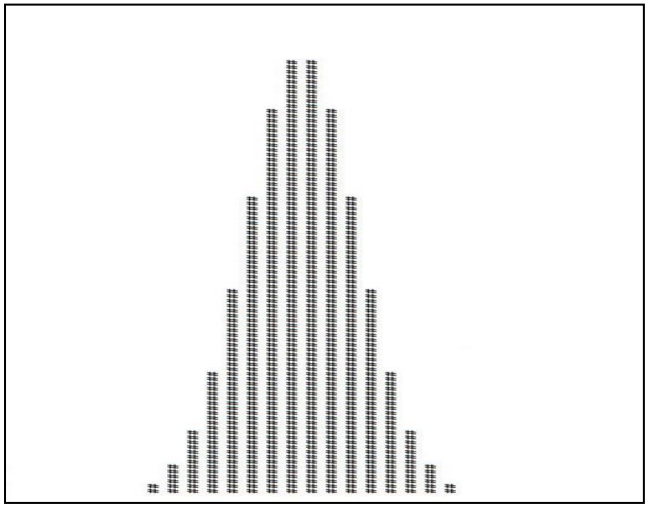
1) calculate the values of the output array ahead of time, i.e. use the line

```
for(ctr = 0; ctr<(UINT)(scaleFactor * bin[binCtr]); ctr++)
```

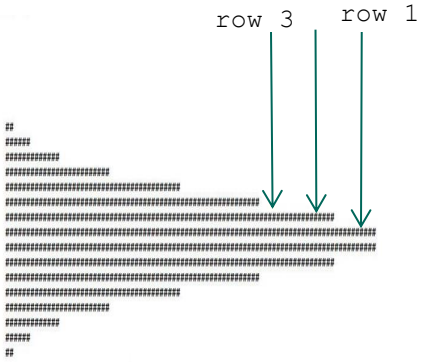
to populate the `bin[]` array with the numbers that will actually be needed (after the `scaleFactor` has been taken into account.) Note that a suitable value of `scaleFactor` needs to be found; in this case you can assume the height of the full screen display is 40 characters high.

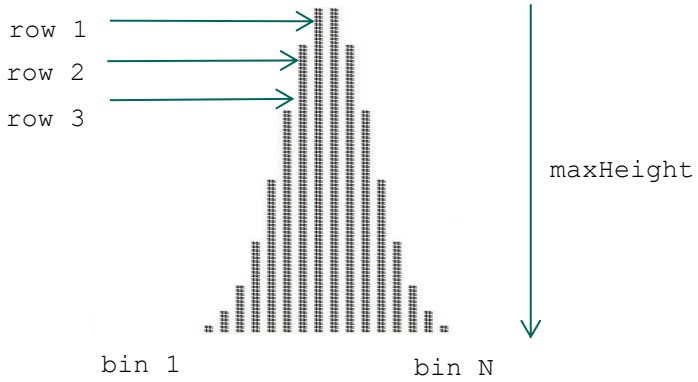
Programming Challenges

The Horizontal Distribution Function (con't)



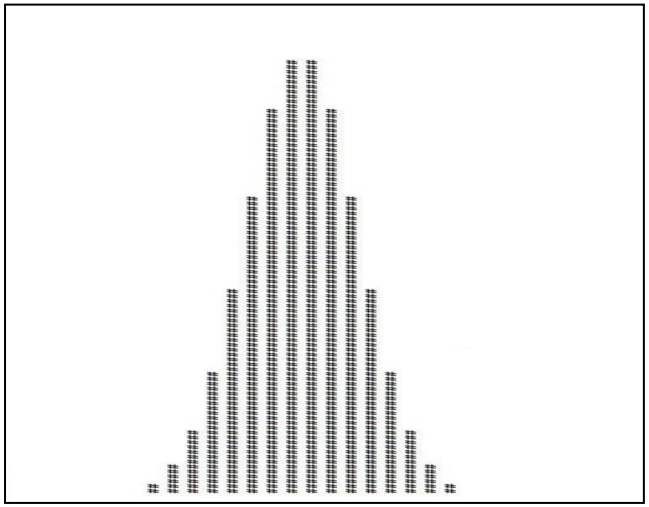
2) The largest valued number in the array will appear at the top of the screen—therefore your code will need to 'know' the maximum value of the array (which you can get from `SimpleStats.c`).





Programming Challenges

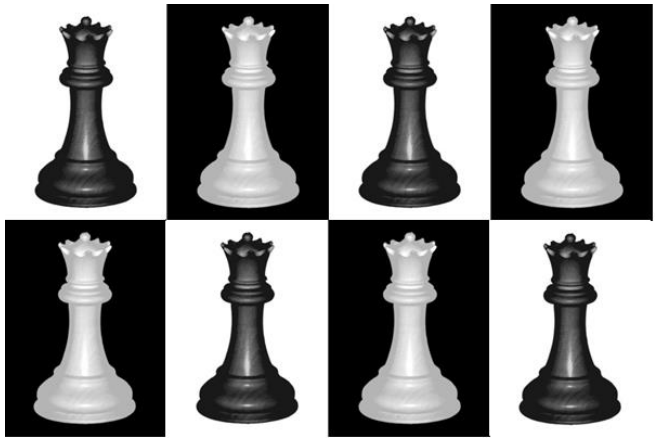
The Horizontal Distribution Function (con't)



Conceptually, this is easy. But getting your mind around the business of 'rotating' the output can be perplexing. In actual fact (once the correct values or entered into the array), the actual code required to output the horizontal distribution function amounts to a half-dozen lines of code.

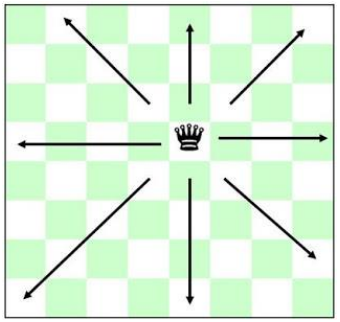
Programming Challenges

The Eight Queens Problem

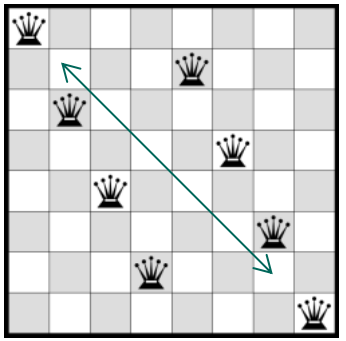


Program Description:

In chess, the queen can move both horizontally, vertically, and diagonally across the board, in any direction. In the 'Eight Queens Problem' one attempts to place eight queens on a standard 8 X 8 chess board in such a way that no queen can capture any other queen.

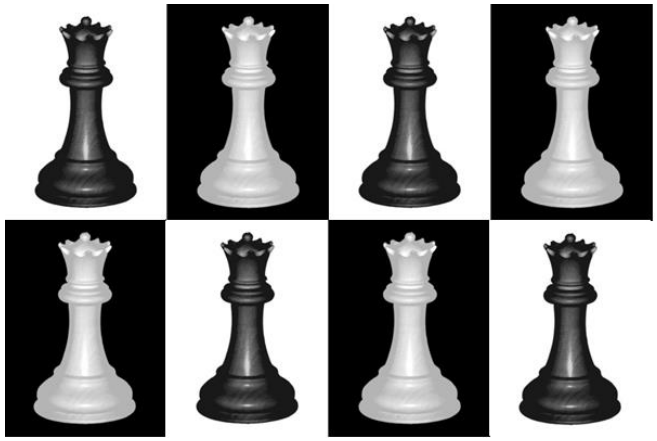


The chess board shown at right is *not* a solution to the eight queens problem, since the two queens at opposite corners of the board can take each other along the white diagonal squares (none of the other queens are in danger of capture, so this is *almost* a solution.)



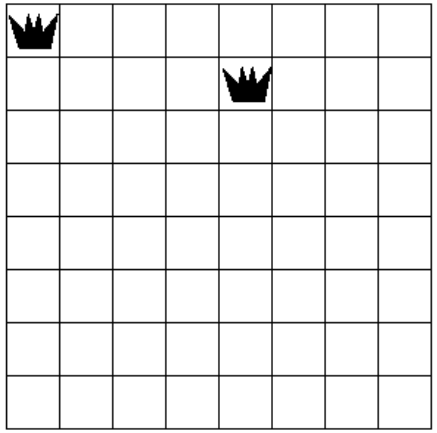
Programming Challenges

The Eight Queens Problem (con't)



There are twelve possible solutions to the eight queens problem; your program should find them all. With approximately 178 trillion possible ways to place eight queens on sixty-four squares, this is potentially a computationally-intensive problem; on a 3.2 GHz quad processor, it can take weeks to arrive at all twelve solutions. Therefore, careful planning is essential before you begin.

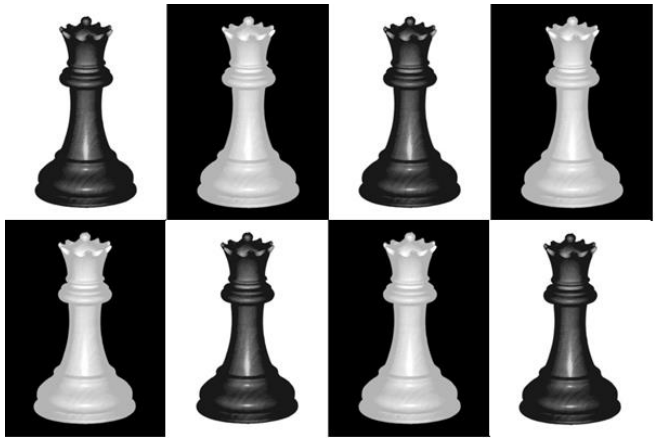
First, declare an array of 8 integers; this will allow you to store the location of each queen on the chessboard as a '1', as shown at right. (You could also use an array of 8 `unsigned chars`, but since C automatically converts `chars` to `ints` during any operation, its not worth the trouble.)



1	0	0	0	0	0	0	0
0	0	0	0	1	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

Programming Challenges

The Eight Queens Problem (con't)

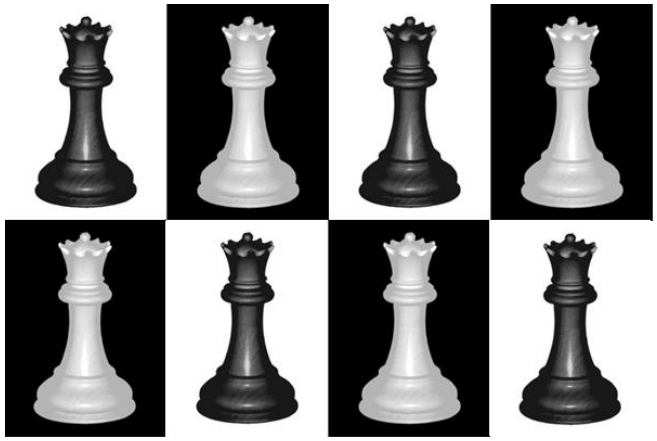


To simplify the number of computations required to complete this problem, consider the following:

1. Two queens cannot exist on the same row; this is not a possible solution. What does this tell you about the number of '1's in each element of the array? (Note: the answer to this questions limits the number of possible combinations to a mere sixteen million, greatly simplify the computational overhead.)
2. Two queens cannot exist on the same column, therefore if you perform a logical AND on each row with every succeeding row, the result should be '0' (i.e. false). If you AND any two `int`s in the array and get a non-zero number, then the two queens are on the same column, and can take one another. Again, this is not a possible solution, so right away, you can dispose of this combination.
3. Since our queens are abstracted in binary notation, you can use the shift operations ('>>' or '<<') to move each queen to a new position at the start of each loop, including simulating a move along the diagonal squares.

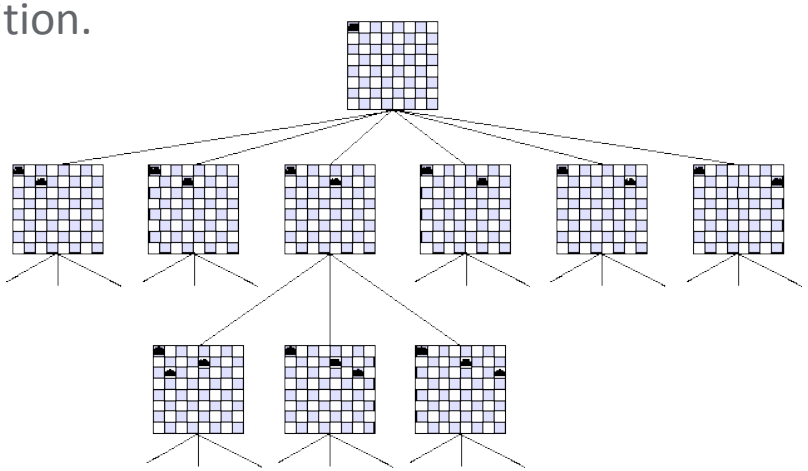
Programming Challenges

The Eight Queens Problem (con't)



Create a library the contains:

a. A static array of eight ints, to hold an entire chessboard
b. Functions that test to see if any two queens occupy the same row or column
c. Functions that 'move' the queens (1's) along the diagonals, and then check to see if any two queens (1's) occupy the same position.
d. A function that displays the state of the chess board, using 1's and 0's to indicate the location of the queens.



Your program should then loop through all possible combinations, discard those that result in two queens taking one another, and print out only the correct solutions.