Detail from C Propaganda Poster // Reddit user TheBall

BBM 101

Introduction to Programming I

Lecture #11 – C – Iterations, Functions, Multi-D Arrays

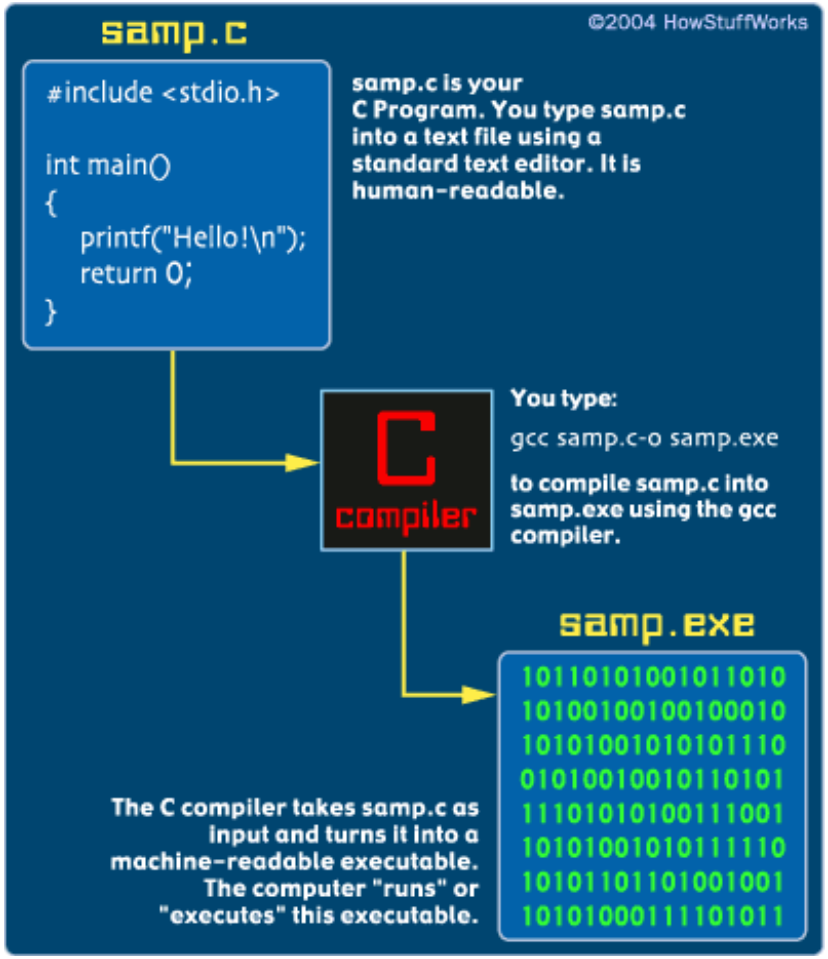
ONE WORLD: ONE LANGUAGE. C UNITES WORKERS

HACETTEPE UNIVERSITY

Aykut Erdem, Fuat Akal & Aydın Kaya // Fall 2018

Last time… **C for Python Programmers**

- General C Structure
- Variables
- **`printf` - `scanf`**
- If-Else Statements
- Arrays
- Constants



Today

- Another Flow Control Statement: **`switch`**/**`case`**
- Iteration Statements
- Multidimensional Arrays
- Functions

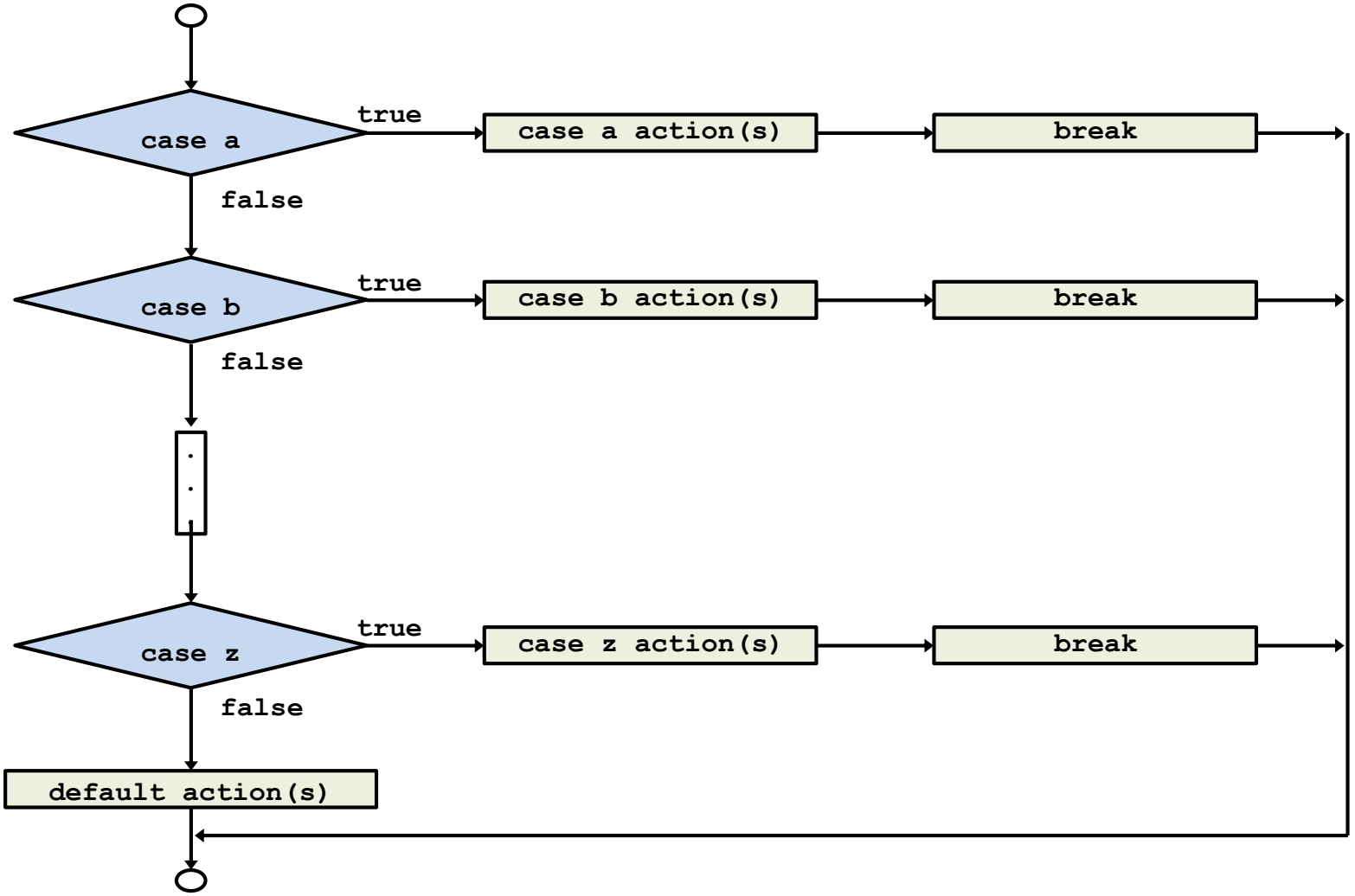
The **`switch`** Multiple-Selection Structure

- **`switch`**
  - –Useful when a variable or expression is tested for all the values it can assume and different actions are taken
  - –Series of case labels and an optional default case

```
switch ( a_variable or expr ){
case value1:
actions
case value2 :
actions
...
default:
actions
}
```

  - –**`break;`** exits from structure

The **`switch`** Multiple-Selection Structure



A program to count letter (upper case) grades

```
#include <stdio.h>

int main() {
    /*Counting letter grades */
    char grade;
    int aCount = 0, bCount = 0, cCount = 0,dCount = 0, fCount = 0;
    int counter = 0;
    while(counter < 10){
        scanf("%c",&grade);
        printf("entered: %c \n",grade);
        switch ( grade ) {
            case 'A': ++aCount;break;
            case 'B': ++bCount;break;
            case 'C': ++cCount;break;
            case 'D': ++dCount;break;
            case 'F': ++fCount;break;
            default:          /* catch all other characters */
                printf( "Incorrect letter grade entered." );
                printf( " Enter a new grade.\n" );
                break;
        }
        counter++;
    }
    return 0;
}
```

A program to count letter (upper/lower case) grades

```
#include <stdio.h>

int main() {
    /*Counting letter grades */
    char grade;
    int aCount = 0, bCount = 0, cCount = 0, dCount = 0, fCount = 0;
    int counter = 0;
    while(counter < 10) {
        printf("Enter the letter grade.\n"  );
        scanf("%c",&grade);
        printf("entered: %c \n",grade);
        switch ( grade ) {
            case 'A':
            case 'a': ++aCount;break;
            case 'B':
            case 'b': ++bCount;break;
            case 'c':
            case 'C': ++cCount;break;
            case 'd':
            case 'D': ++dCount;break;
            case 'f':
            case 'F': ++fCount;break;
            default:            /* catch all other characters */
                printf( "Incorrect letter grade entered." );
                printf( " Enter a new grade.\n" );
                break;
        }
        counter++;
    }
    return 0;
}
```

Find the day count of month/year pair

```
#include <stdio.h>
int main()
{
  int month, year, days, leapyear;
  printf("Enter a month and a year:");
  scanf("%d %d", &month, &year);
  if(((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0))
    leapyear = 1;
  else
    leapyear = 0;
  switch(month){
      case 9 :
      case 4 :
      case 6 :
      case 11: days=30;break;
      case 2 :
          if(leapyear == 1)
            days = 29;
          else
            days = 28;
            break;
      default :
         days = 31;
   }
    printf("There are %d days in that month in that year.\n", days);
         return 0;
}
```

The Essentials of Repetition

- Loop
  - Group of instructions computer executes repeatedly while some condition remains **`true`**
- Counter-controlled repetition
  - Definite repetition: know how many times loop will execute
  - Control variable used to count repetitions
- Sentinel-controlled repetition
  - Indefinite repetition
  - Used when number of repetitions not known
  - Sentinel value indicates "end of data"

The **`while`** Repetition Structure

- Repetition structure
  - Programmer specifies an action to be repeated while some condition remains **`true`**
  - e.g.:

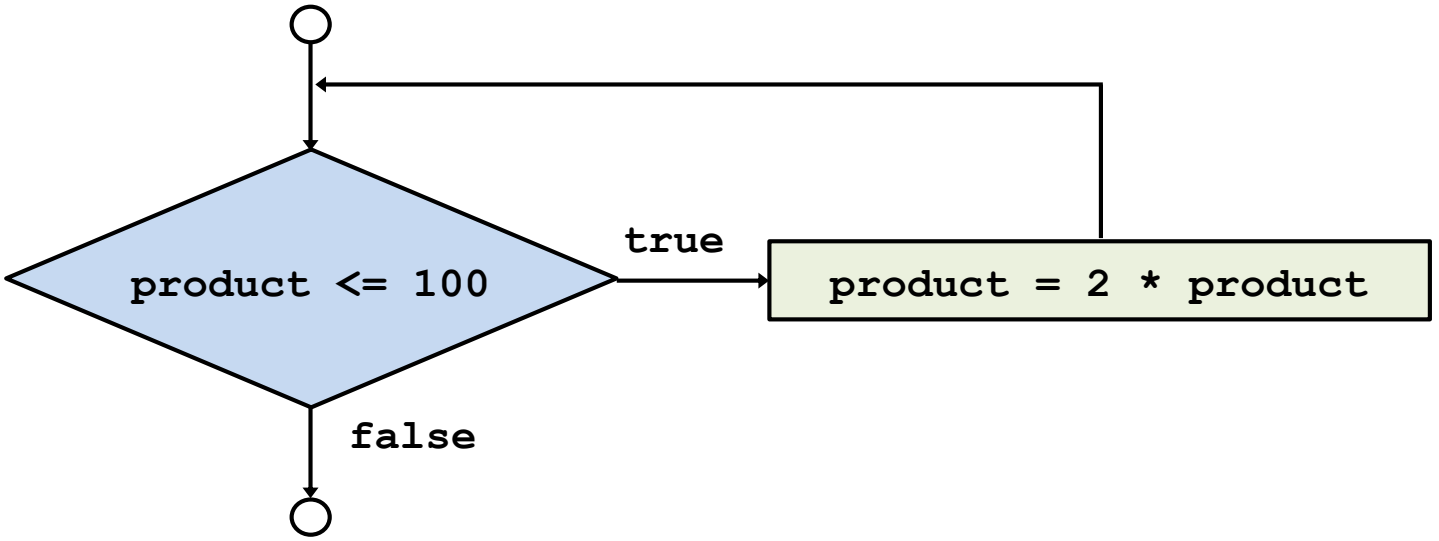
    While there are more items on my shopping list
    Purchase next item and cross it off my list

  - **`while`** loop repeated until condition becomes **`false`**

The **`while`** Repetition Structure

- Example:

```
int product = 2;
while ( product <= 100 )
   product = 2 * product;
```



Example: Counter-Controlled Repetition

- A class of 10 students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz

- The algorithm

Set total to zero
Set grade counter to one
While grade counter is less than or equal to 10
Input the next grade
Add the grade into the total
Add one to the grade counter
Set the class average to the total divided by ten
Print the class average

Example: Counter-Controlled Repetition

```c
/* Class average program with counter-controlled repetition */
#include <stdio.h>

int main()
{
   int counter, grade, total, average;

   /* initialization phase */
   total = 0;
   counter = 1;

   /* processing phase */
   while ( counter <= 10 ) {
         printf( "Enter grade: " );
         scanf( "%d", &grade );
         total = total + grade;
         counter = counter + 1;
    }

   /* termination phase */
   average = total / 10.0;
   printf( "Class average is %d\n", average );

   return 0;   /* indicate program ended successfully */
}
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

A Similar Problem

- Problem becomes:

  Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.

  - Unknown number of students
  - How will the program know to end?

- Use sentinel value
  - Also called signal value, dummy value, or flag value
  - Indicates “end of data entry.”
  - Loop ends when user inputs the sentinel value
  - Sentinel value chosen so it cannot be confused with a regular input (such as `-1` in this case)

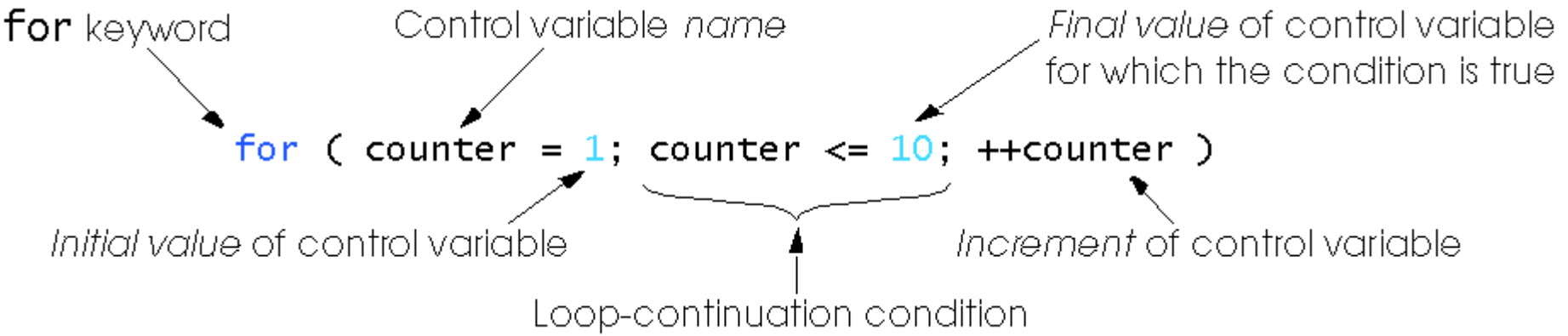
Example on Sentinel Value on **`while`** Loop

```
/* Class average program with sentinel-controlled repetition */
#include <stdio.h>
int main()
{
   float average;
   int counter, grade, total;

   /* initialization phase */
   total = 0;
   counter = 0;

   /* processing phase */
   printf( "Enter grade, -1 to end: " );
  scanf( "%d", &grade );
  while ( grade != -1 )
  {
         total = total + grade;
         counter = counter + 1;
         printf( "Enter grade, -1 to end: " );
         scanf( "%d", &grade );
  }
   /* termination phase */
  if( counter != 0 ) {
      average = ( float ) total / counter;
      printf( "Class average is %.2f", average ); }
  else
      printf( "No grades were entered\n" );
  return 0;   /* indicate program ended successfully */
}
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

The `for` Repetition Structure

- Format when using **`for`** loops



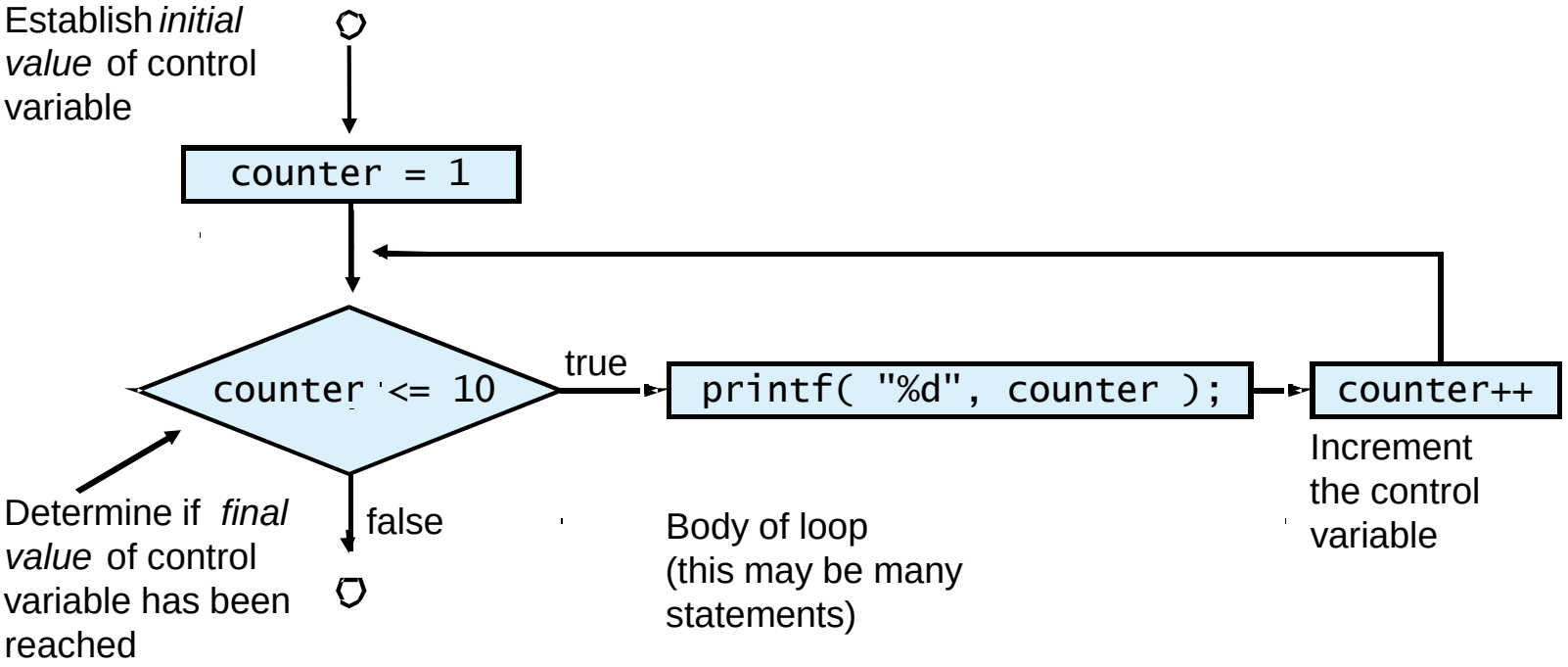
Example:

```
for(counter = 1; counter <= 10; counter++ )
   printf( "%d\n", counter );
```

– Prints the integers from one to ten

No semicolon (;) after for statement

The `for` Flowchart



The `for` Repetition Structure

- For loops can usually be rewritten as while loops:

```
initialization;
while ( loopContinuationTest ) {
  statement;
  increment;
}
```

- Initialization and increment
  - Can be comma-separated lists

```
for (i = 0, j = 0;  j + i <= 10; j++, i++)
   printf( "%d\n", j + i );
```

  - Initialization, loop-continuation, and increment can contain arithmetic expressions. If **x** equals **2** and **y** equals **10**

```
for ( j = x; j <= 4 * x * y; j += y / x )
```

equals to

```
for ( j = 2; j <= 80; j += 5 )
```

Provided that the values of x and y remains constant

Example: Print the sum of all numbers from 2 to 100

```
/*Summation with for */
#include <stdio.h>

int main()
{
   int sum = 0, number;
   for ( number = 2; number <= 100; number += 1 )
           sum += number;
   printf( "Sum is %d\n", sum );
   return 0;
}
```

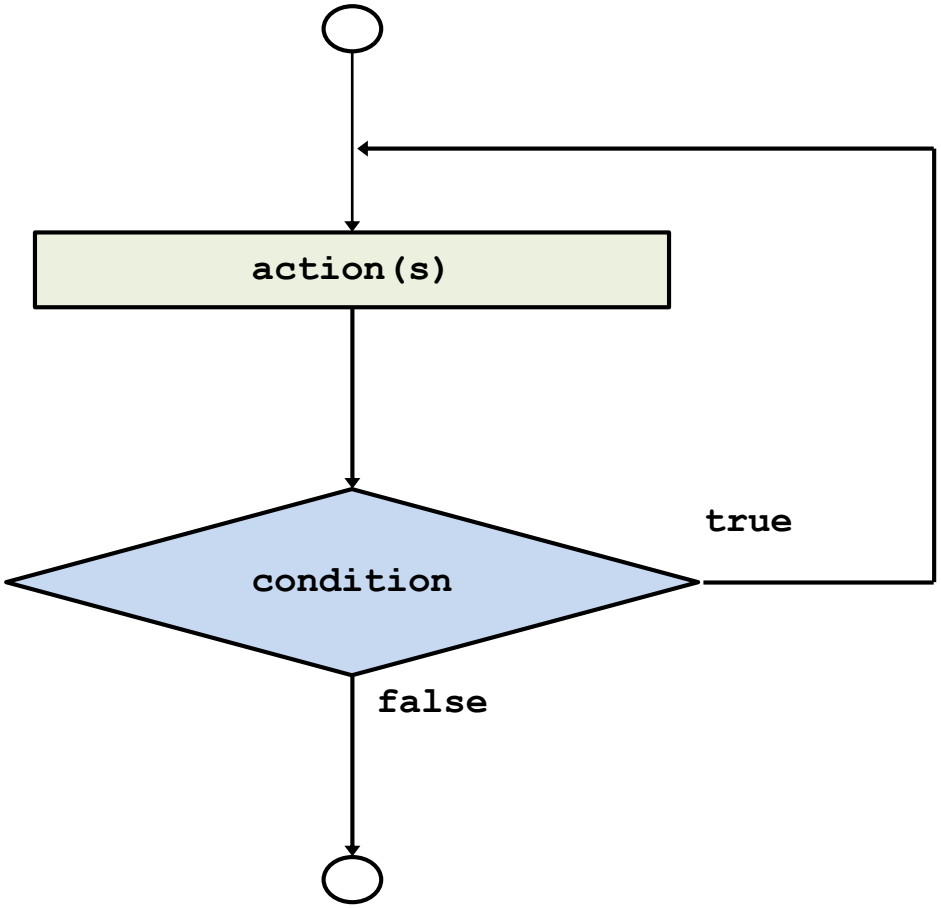
```
Sum is 2550
```

The **`do/while`** Repetition Structure

- The **`do`**/**`while`** repetition structure
  - Similar to the **`while`** structure
  - Condition for repetition tested after the body of the loop is performed
    - All actions are performed at least once
  - Format:

```
do {
   statement;
} while ( condition );
```

The **do/while** Repetition Structure



Prints the integers from one to ten

```c
/*Using the do/while repetition structure */

#include <stdio.h>
int main()
{
   int counter = 1;

   do {
        printf( "%d  ", counter );
        counter = counter + 1;
   } while ( counter <= 10 );

   return 0;
}
```

```
1  2  3  4  5  6  7  8  9  10
```

Nested Loops

- When a loop body includes another loop construct this is called a *nested loop*.

- In a nested loop structure the inner loop is executed from the beginning every time the body of the outer loop is executed.

```
value = 0;
for (i=1; i<=10; i=i+1)
    for (j=1; j<=5; j=j+1)
                value = value + 1;
```

- How many times the inner loop is executed?

Printing a triangle

- Write a program to draw a triangle like the following: (input: the number of lines)

```
*
**
***
****
*****
```

We can use a nested for-loop:

```
for (i=1; i<=num_lines; ++i)
{
   for (j=1; j<=i; ++j)
      printf("*");
        printf("\n");
}
```

Nesting **while** and **for**

```
int main()
{
   int num, count, total = 0;

   printf("Enter a value or a negative number to end: " );
   scanf("%d", &num );

   while( num >= 0 ) {
        for (count = 1; count <= num; count++)
            total = total + count;
        printf("%d %d",num, total);
        printf( "Enter a value or a negative number to end:");
        scanf( "%d", &num );
        total = 0;
   }
   return 0;
}
```

This program reads numbers until the user enters a negative number. For each number read, it prints the number and the summation of all values between 1 and the given number.

Example: Nesting **`while`** and **`switch`**

```
#include <stdio.h>
int main()
{
   char grade;
   int aCount = 0, bCount = 0, cCount =0,
       dCount = 0, eCount = 0 ;
   printf( "Enter the letter grades. Enter X to exit. \n" );

  while((grade = getchar()) != 'X')
  {
     switch ( grade ) {
         case 'A': ++aCount;  break;
         case 'B': ++bCount;  break;
         case 'C': ++cCount;  break;
         case 'D': ++dCount;  break;
         case 'F': ++fCount;  break;
         default:             /* catch all other characters */

          printf( "Incorrect letter grade entered." );
         printf( "Enter a new grade.\n" );
         break;
      }
   }
```

Reads a character from the standard input

`break` statement

- **`break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do`**…**`while`** or **`switch`** statement
  - Program execution continues with the first statement after the structure
  - Common uses of the **`break`** statement
    - Escape early from a loop
    - Skip the remainder of a **`switch`** statement

Example

```
#include <stdio.h>

int main()
{
   int x;

   for(x = 1; x <= 10 ; x++)
    {
      if( x == 5) {
             break;
      printf("%d ", x);
    }

    printf("\nBroke out of the loop at x=%d ", x);
    return 0;
}
```

```
1 2 3 4
Broke out of loop at x == 5
```

`continue` statement

- **`continue`**
  - Skips the remaining statements in the body of a **`while, for`** or **`do`**…**`while`** statement
    - Proceeds with the next iteration of the loop
  - **`while`** and **`do`**…**`while`**
    - Loop-continuation test is evaluated immediately after the `continue` statement is executed
  - **`for`**
    - Increment expression is executed, then the loop-continuation test is evaluated

Example

```
#include <stdio.h>

int main()
{
   int x;

   for(x = 1; x <= 10 ; x++)
    {
      if( x == 5) {
         continue;
      printf("%d ", x);
    }

    printf("\nUsed continue to skip printing the value 5");
    return 0;
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Reminder: Arrays

Python supports many types that combine the basic atomic types into a group: tuples, lists, strings, dictionaries, sets.
C's support is much more rudimentary: The *only* composite type is the **array**

Similar to Python's list except that an array in C cannot grow or shrink — its size is fixed at the time of creation.

Example:

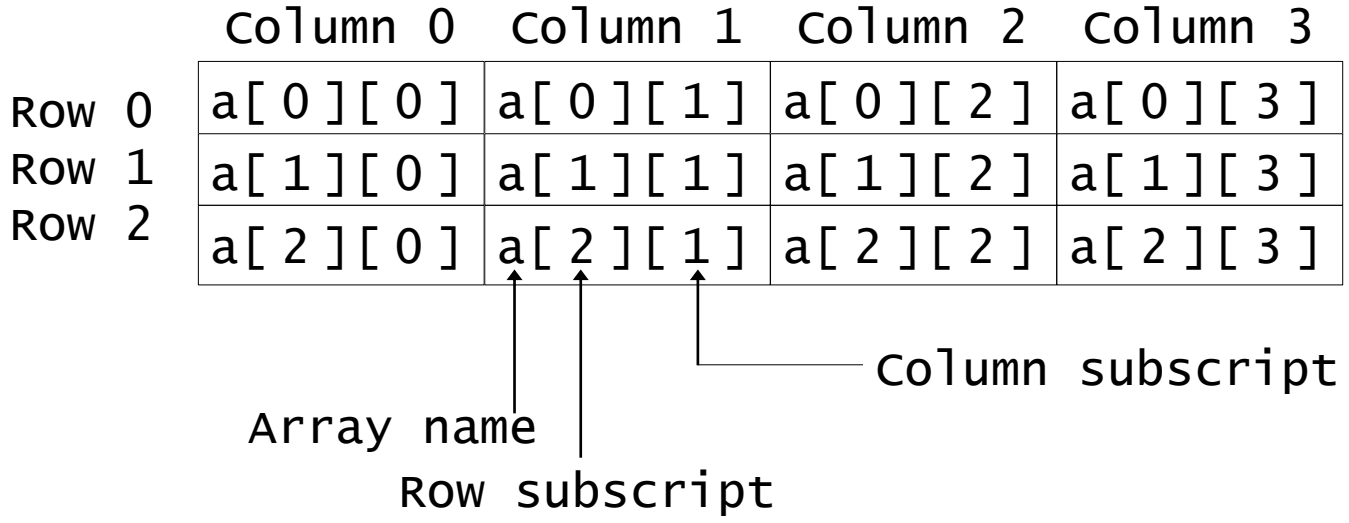
```
double pops[50];
pops[0] = 897934;
pops[1] = pops[0] + 11804445;
```

Another way to make an array, if you know all the elements upfront, is:

```
char vowels[6] = {'a', 'e', 'i', 'o', 'u', 'y'};
```

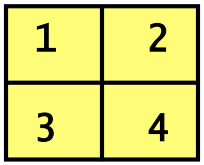
Multi-Dimensional Arrays

- Multiple subscripted arrays
  - Tables with rows and columns (**m** by **n** array)
  - Like matrices: specify row, then column

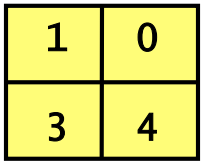


Multi-Dimensional Arrays

- Initialization
  - `int b[2][2] = {{1,2},{3,4}};`
  - Initializers grouped by row in braces
  - If not enough, unspecified elements set to zero




```
int b[2][2] = {{1},{3,4}};
```



- Referencing elements
  - Specify row, then column

```
printf( "%d", b[0][1] );
```

Example:Multi-Dimensional Array

```
#include <stdio.h>
int main()
{
        int i,j;
        /* initialize array1, array2, array3 */
        int array1[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
        int array2[2][3] = { 1, 2, 3, 4, 5 };
        int array3[2][3] = { { 1, 2 }, { 4 } };

        printf( "Values in array1 by row are:\n" );
        for ( i = 0; i <= 1; i++ ) {  /* loop through rows */
                for ( j = 0; j <= 2; j++ )
                     printf( "%d ", array1[ i ][ j ] ); /* output column values */
                printf( "\n" );
        }
        printf( "Values in array2 by row are:\n" );
        for ( i = 0; i <= 1; i++ ) {                    /* loop through rows */
                for ( j = 0; j <= 2; j++ )
                    printf( "%d ", array2[ i ][ j ] );  /* output column values */
                printf( "\n" );
        }

        printf( "Values in array3 by row are:\n" );
        for ( i = 0; i <= 1; i++ ) {                    /*
                for ( j = 0; j <= 2; j++ )
                    printf( "%d ", array3[ i ][ j ] );
                printf( "\n" );
        }
        return 0;
}
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Another Example

- Assume that, out program takes a square matrix as input (NxN). This matrix contains only 1's and 0's. Let's define this matrix directly on our code for now (or use **`scanf`**).

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

- Try to find a special pattern. X symbol, defined as 3x3 matrix.

1	**(0 or 1)**	**1**
(0 or 1)	**1**	**(0 or 1)**
1	**(0 or 1)**	**1**

- If a pattern has found, print the center points (row and column) of the matched pattern.

Input representation as figure:						
1	0	1	0	0	0	0
0	**1**	0	1	0	1	0
1	0	1	0	**1**	0	0
0	0	0	1	0	1	0
0	1	0	1	**1**	0	1
0	0	**1**	1	0	**1**	0
0	1	0	1	1	1	1

Output:

Pattern found at 1,1
Pattern found at 2,4
Pattern found at 4,4
Pattern found at 5,2
Pattern found at 5,5

Solution

```
#include <stdio.h>

int main()
{
    // prepare the matrix
    int m[7][7] = {{1,0,1,0,0,0,0},
                   {0,1,0,1,0,1,0},
                   {1,0,1,0,1,0,0},
                   {0,0,0,1,0,1,0},
                   {0,1,0,1,1,0,1},
                   {0,0,1,1,0,1,0},
                   {0,1,0,1,1,1,1}
                   };
    int N = 7;
    int i,j;
    for(i = 1; i < N-1; i++){
        for(j = 1; j < N-1; j++)
        {
             if((m[i][j] == 1) && (m[i-1][j-1] == 1) &&
                (m[i+1][j-1] == 1) && (m[i-1][j+1] == 1) &&
                (m[i+1][j+1] == 1))
                     printf("Pattern found at %d,%d\n",i,j);
        }
    }
    return 0;
}
```

```
Pattern found at 1,1
Pattern found at 2,4
Pattern found at 4,4
Pattern found at 5,2
Pattern found at 5,5
```

Functions

- We have already used main function and some of the library functions:
  - **`main`** is a function that must exist in every C program.
  - **`printf`**, **`scanf`** are library functions which we have already used in our programs.

- We need to do two things with functions:
  - create functions
  - call functions (Function invocation)

Function Definition

- A function definition has the following form:

```
return_type  function_name  (parameter-declarations)
 {
     variable-declarations

     function-statements
 }
```

return_type - specifies the type of the function and corresponds to the type of value returned by the function

- **void** – indicates that the function returns nothing.
- if not specified, of type **int**

function_name – name of the function being defined (any valid identifier)

parameter-declarations – specify the types and names of the parameters (a.k.a. formal parameters) of the function, separated by commas.

Example: Function returning a value

- Let's define a function to compute the cube of a number:

```
int cube ( int num ) {
     int result;

     result = num * num * num;
     return result;
}
```

- This function can be called as:

```
int n = cube(5);
```

Example: void Function

```
void prn_message(void) /* function definition */
{
      printf("A message for you:   ");
      printf("Have a nice day!\n");
}


int main (void)
{
     prn_message ( );      /* function invocation */
     return 0;
}
```

Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - **`#include <math.h>`**

- Format for calling functions
  - **`FunctionName(`** *argument* **`);`**
    - If multiple arguments, use comma-separated list
  - **`y = sqrt( 900.0 );`**
    - Calls function **`sqrt`**, which returns the square root of its argument
  - Arguments may be any r-value (constants, variables, or expressions)

Math Library Functions

Function Header	Description
`int abs(int num)`	Returns the absolute value of an integer element.
`double fabs(double num)`	Returns the absolute value of a double precision element.
`double pow(double x,double y)`	Returns x raised to the power of y.
`int rand(void)`	returns a random number
`double sin(double angle)`	Returns the sine of an angle; the angle should be in Radian.
`double cos(double angle)`	Returns the cosine of an angle; the angle should be in Radian.
`double sqrt(double num)`	Returns the the square root of a double

Math Library Functions

- Calculate the square root of $(x1 - x2)^2 + (y1 - y2)^2$

```
a = x1 – x2;
b = y1 – y2;
c = pow(a,2) + pow(b, 2);
d = sqrt(d);
```

Variable Declarations within Function Definitions

- Variables declared local to a function supersede any identically named variables outside the function (remember shadowing in python)

```
int lcm(int m, int n) {
    int i;
    ...
}

int gcd(int m, int n) {
    int i;
    ...
}
```

The **`return`** statement

- When a return statement is executed, the execution of the function is terminated and the program control is immediately passed back to the calling environment.

- If an expression follows the keyword return, the value of the expression is returned to the calling environment as well.

- A return statement can be one of the following two forms:

```
return;
return expression;
```

Examples

```
return;

return 1.5;

return result;

return a+b*c;

return x < y ? x : y;
```

It's like ternary assignments in Python!

Example

```
int IsLeapYear(int year)
{
   return ( ((year % 4 == 0) && (year % 100 != 0))
            || (year % 400 == 0) );
}
```

- This function may be called as:

```
if (IsLeapYear(2005))
   printf("29 days in February.\n");
else
   printf("28 days in February.\n");
```

Example

```
#include  <stdio.h>
int min(int a, int b)
{
   if (a < b)
      return a;
   else
      return b;
}

int main (void)
{
   int j, k, m;

   printf("Input two integers:    ");
   scanf("%d %d", &j, &k);
   m = min(j,k);
   printf("\nThe minimum is %d.\n", m);
   return 0;
}
```

```
Input two integers:  5  6
The minimum is 5.

Input two integers:  11  3
The mininum is 3.
```

Parameters

- A function can have zero or more parameters.
- In declaration header:

```
int f (int x, double y, char c);
```

the *formal parameter list* (parameter variables and their types are declared here)

- In function calling:

```
value = f(age, 100*score, initial);
```

actual parameter list (cannot tell what their type are from here)

Rules for Parameter Lists

- The number of parameters in the actual and formal parameter lists must be *consistent*
- Parameter association is *positional:* the first *actual* parameter matches the first *formal* parameter, the second matches the second, and so on
- *Actual* parameters and *formal* parameters must be of compatible *data types*
- *Actual* parameters may be a variable, constant, any expression matching the type of the corresponding formal parameter

Invocation and Call-by-Value

- Each argument is evaluated, and its value is used locally in place of the corresponding formal parameter.

- If a variable is passed to a function, the stored value of that variable in the calling environment will not be changed.

- In C, all calls are call-by-value unless specified otherwise.

Function Call

- The type of a function-call expression is the same as the type function being called, and its value is the value returned by the function.

- Function calls can be embedded in other function calls.
  - **e.g.**

```
t = cubesum(i);
j = cubesum(t);
```

is equivalent to

```
j = cubesum(cubesum(i));
```

Example

```
#include  <stdio.h>

int compute_sum (int n)
{
   int  sum;

   sum = 0;

   for (  ; n > 0; --n)
      sum += n;
   printf("%d\n", n);
   return sum;
}
```

```
int main (void)
{
   int n, sum;

   n = 3;

   printf("%d\n", n);
   sum=compute_sum(n);
   printf("%d\n",n);
   printf("%d\n", sum);
   return 0;
}
```

```
3
0
3
6
```

Example

```
/* Finding the maximum of three integers */
#include <stdio.h>
/* Function maximum definition */
int maximum( int x, int y, int z )
{
   int max = x;
   if ( y > max )
      max = y;
   if ( z > max )
      max = z;

   return max;
     }
int main()
{
   int a, b, c;

   printf( "Enter three integers: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Maximum is: %d\n", maximum( a, b, c ) );

   return 0;
}
```

```
Enter three integers: 22 85 17
Maximum is: 85
```

Function Call

- ANSI-C does not set the arguments evaluation order in function calls!

```
#include <stdio.h>

void f(int a, int b, double c){
    printf("%d \n", a);printf("%d \n", b);printf("%f \n", c);
}

int main(void)
{
    int i = 0;
    int x = 7; float a = 2.25;

    f(x=5, x-7, a); // ---?-----
    printf("\n\n");
    f(x=6, x-7, a); // ---?-----
    printf("\n\n");

    //ambiguous, beware
    printf("%d %d\n", i, i++); // ---?-----
    printf("%d %d\n", i, ++i); // ---?-----
    return 0;
}
```

```
5
0
2.250000


6
-2
2.250000


1 0
2 2
```

Function Prototypes

- General form for a function prototype declaration:

```
return_type  function_name(parameter-type-list)
```

- Used to validate functions
  - Prototype only needed if function definition comes after use in program

- The function with the prototype

```
int maximum( int, int, int );
```

  - Takes in 3 **int**s
  - Returns an **int**

Alternative styles for function definition order

```c
#include <stdio.h>

int max(int,int);
int min(int,int);

int main(void)
{
  min(x,y);
  max(u,v);
  ...
}
int max (int a, int b)
{
  ...
}
int min (int a, int b)
{
 ...
}
```

```c
#include <stdio.h>
int max (int a, int b)
{
  ...
}

int min (int a, int b)
{
  ...
}

int main(void)
{
  ...
  min(x,y);
  max(u,v);
  ...
}
```

Block Structure

- A block is a sequence of variable declarations and statements enclosed within braces.
- Block structure and the scope of a variable

```
int factorial(int n)
  {
     if (n<0) return -1;
     else if (n==0) return 1;
     else
     {
        int i, result=1;
        for (i=1;i<=n; i++) result *= i;
        return result;
     }
   }
```

External Variables

- Local variables can only be accessed in the function in which they are defined.

- If a variable is defined outside any function at the same level as function definitions, it is available to all the functions defined below in the same source file

  → external variable

- **Global variables:** external variables defined before any function definition
  - Their scope will be the whole program

Example

```
#include   <stdio.h>
void print_message (int k);   /*function prototype */

int main (void)
{
     int n;

     printf("There is a message for you.\n");
     printf("How many times do you want to see it?  ");
     scanf("%d", &n);
     print_message(n);
     return 0;
}

void print_message (int k)  /* function definition */
{
     int i;

     printf("\nHere is the message.\n");
     for (i=0; i < k; ++i)
          printf("Have a nice day!\n");
}
```

Example

```
/* An example demonstrating local variables */
#include  <stdio.h>

void func1 (void);

int main (void)
{
      int i = 5;
      printf("%d \n", i);
      func1( );
      printf("%d \n",i);
      return 0;
}

void func1 (void)
{
       int i = 5;
       printf("%d\n", i);
       i++;
        printf("%d\n", i);
}
```

```
5
5
6
5
```

Example: Transforming rectangular coordinates to polar coordinates

```
#include <math.h>
#include <stdio.h>
#define PI 3.1415927

float r, theta;
void polar (float x, float y)

int main(void){
  float x, y;
  scanf("%f %f", &x, &y);
  polar(x,y);
  printf("r = %f, theta = %f\n", r, theta);
  return 0;
}

void polar(float x, float y)
{
   if (x==0 && y==0) r = theta = 0;
   else {
      r = sqrt(x*x + y*y);
      theta = atan2(y,x); }
}
```

Static Variables

- A variable is said to be static if it is allocated storage at the beginning of the program execution and the storage remains allocated until the program execution terminates.
- They doesn't disappear when their scope ends.
- External variables are always static
- Within a block, a variable can be specified to be static by using the keyword static before its type declaration:

  `static` type variable-name;

- Variable declared static can be initialized only with constant expressions (if not, its default value is zero)

Example

```
#include <stdio.h>
void incr(void);

int main(void)
{
   int i;
   void incr(void);

   for (i=0; i<3; i++)
        incr();
   return 0;
}

void incr(void)
{
   static int static_i=0;
   printf("static_i = %d\n", static_i++);
}
```

```
static_i = 0
static_i = 1
static_i = 2
```

Example

```
#include <stdio.h>

put_stars(int n)
{
  static int old_n;
  int i;
  for (i=0;i<old_n;i++)
       printf(" ");
  for (i=0;i<n;i++)
       printf("*");
  printf("\n");
  old_n += n;
}

int main(void)
{
   put_stars(3); put_stars(2); put_stars(3);
   return 0;
}
```

```
***
  **
    ***
```

Correct the errors in the following program segments

```
int g (void) {
    printf (“Inside function g\n”);

    int h(void) {
        printf(“Inside function h\n”);
     }
}
```

```
int sum(int x, int y) {
    int result;
    result = x + y;
}
```

Correct the errors in the following program segments

```
void f (float a); {
     float a;
     printf (“%f”, a); }
```

```
void product (void) {
     int a, b, c, result;
     printf(“Enter 3 integers: ”);
     scanf(“%d %d %d”, &a, &b, &c);
     result = a * b * c;
     printf(“Result is %d\n”, result);
     return result;
}
```

Exercises

- Define a function to calculate
  $(x^2 + y^2 + z^2)^{1/2}$
  and use it to calculate

  $a = 1/(u^2+v^2+w^2)^{1/2}$, $b = (u^4 + v^4 + w^4)^{1/2}$,
  $g = (4u^2+9v^2+25w^2)^{1/2}$, $h = (3u^2)^{1/2}(12v^2)^{1/2}(27w^2)^{1/2}$

Exercises

- Analyze the output of the following program

```
#include <stdio.h>
int i=0;

void f(void)
{
  int i;
  i = 1;
}
void g(void)
{
  i=2;
}
void h(int i)
{
  i=3;
}
```

```
int main(void)
{
  {
    int i=4;
    printf("%d\n", i);
  }
  printf("%d\n", i);
  f();
  printf("%d\n", i);
  g();
  printf("%d\n", i);
  h(i);
  printf("%d\n", i);
  return 0;
}
```

Exercises

- Write a program that that reads in the side of a square and then prints a hollow square. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 4, it should print:

```
****
*  *
*  *
****
```