An array is a set of consecutive memory locations used to store data.

The number of elements in an array is called the dimension of the array.

A typical array declaration is:

```c
int data_list[3];
```

- To reference an element of an array, you use a number called the index—the number inside the square brackets (\[ \])
- start counting at 0. So, our three elements are numbered to 2.
#include <stdio.h>

float data[5]; /* data to average and total */
float total; /* the total of the data items */
float average; /* average of the items */
int main()
{
    data[0] = 34.0;
data[1] = 27.0;
data[2] = 45.0;
data[3] = 82.0;
data[4] = 22.0;
average = total / 5.0;
printf("Total %f Average %f\n", total, average);
return (0);
}
Strings are sequences of characters.

C does not have a built-in string type;
- Strings are created out of character arrays.
- One of the restrictions is that the special character \0 (NUL) is used to indicate the end of a string.

String constants consist of text enclosed in double quotes ("").

Strings are surrounded by double quotes (") and characters by single quotes ('). So "X" is a one-character string, while 'Y' is just a single character. (The string "X" takes up two bytes, one for the X and one for the end-of-string (\0). The character 'Y' takes up one byte.)
char name[4];
name = "Sam"; /* Illegal */

#include <string.h>
char name[4];
int main() {
    strcpy(name, "Sam"); /* Legal */
    return (0);
}
```c
#include <string.h>
#include <stdio.h>
char first[100]; /* first name */
char last[100]; /* last name */
char full_name[200]; /* full version of first and last name */
int main()
{
    strcpy(first, "Steve"); /* Initialize first name */
    strcpy(last, "Oualline"); /* Initialize last name */
    strcpy(full_name, first); /* full = "Steve" */
    /* Note: strcat not strcpy */
    strcat(full_name, " "); /* full = "Steve " */
    strcat(full_name, last); /* full = "Steve Oualline" */
    printf("The full name is %s\n", full_name);
    return (0);
}
```
The standard function `fgets` can be used to read a string from the keyboard.

```c
fgets(name, sizeof(name), stdin);
```

, where `name` identifies a string variable.
Modify and Add the code in order to print the string length which you typed.

```c
#include <stdio.h>
#include <string.h>
char first[100]; /* First name of person we are working with */
char last[100]; /* His last name */
/* First and last name of the person (computed) */
char full[200];
int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);
    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);
    strcpy(full, first);
    strcat(full, " ");
    strcat(full, last);
    printf("The name is %s\n", full);
    return (0);
}
```
The declaration for a two-dimensional array is:

```c
type variable[size1][size2]; /* Comment */
int matrix[2][4]; /* a typical matrix */
```

To access an element of the matrix, we use the notation:

```c
matrix[1][2] = 10;
```
Reading Numbers

- The function scanf works like printf, except that scanf reads numbers instead of writing them.
  - The function scanf is notorious for its poor end-of-line handling, which makes scanf useless for all but an expert.
- We use fgets to read a line of input and sscanf to convert the text into numbers.

```c
char line[100]; /* Line of keyboard input */
fgets(line, sizeof(line), stdin);
sscanf(line, format, &variable1, &variable2 . . .);
```

- If you forget to put & in front of each variable for sscanf, the result could be a "Segmentation violation core dumped" or "Illegal memory access" error.
What is the problem in this code?

```c
#include <stdio.h>

char line[100]; /* line of input data */
int height; /* the height of the triangle */
int width; /* the width of the triangle */
int area; /* area of the triangle (computed) */
int main()
{
    printf("Enter width height? ");
fgets(line, sizeof(line), stdin);
sscanf(line, "%d %d", &width, &height);
area = (width * height) / 2;
printf("The area is %d\n", area);
return (0);
}
```
C allows variables to be initialized in the declaration statement.

```c
int counter = 0; /* number cases counted so far */
```

Arrays can also be initialized in this manner. The element list must be enclosed in curly braces (```{}```)

```c
/* Product numbers for the parts we are making */
int product_codes[] = {10, 972, 45};

/* Product numbers for the parts we are making */
int product_codes[3] = {10, 972, 45};

product_codes[0] = 10;
product_codes[1] = 972;
product_codes[2] = 45;

int matrix[2][4] = { {1, 2, 3, 4}, {10, 20, 30, 40} };
Strings can be initialized in a similar manner.

```c
char name[] = {'S', 'a', 'm', '\0'};

char name[] = "Sam"; /* Illegal? */

char string[50] = "Sam";
```
Types of Integers

- Sometimes you need extra digits to store numbers larger than those allowed in a normal `int`
  - `long` int answer; /* the result of our calculations */
- If we are going to use small numbers and wish to reduce storage, we use the qualifier `short`.
  - `short` int year; /* Year including the 19xx part */
- `short <= int <= long`.
- `unsigned short int`
- `sizeof(type)`
  - Indicate the byte size of the specific type.
## Integer printf/sscanf Conversions

<table>
<thead>
<tr>
<th>%Conversion</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>%hd</td>
<td>(signed) short int</td>
</tr>
<tr>
<td>%d</td>
<td>(signed) int</td>
</tr>
<tr>
<td>%ld</td>
<td>(signed) long int</td>
</tr>
<tr>
<td>%hu</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned int</td>
</tr>
<tr>
<td>%lu</td>
<td>unsigned long int</td>
</tr>
</tbody>
</table>
Types of Floats

- **float** denotes normal precision (usually 4 bytes).
- **double** indicates double precision (usually 8 bytes).
- **long double** denotes extended precision.

**Float printf/sscanf Conversions**

<table>
<thead>
<tr>
<th>% Conversion</th>
<th>Uses</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f</td>
<td>float</td>
<td>printf only.</td>
</tr>
<tr>
<td>%lf</td>
<td>double</td>
<td>scanf only.</td>
</tr>
<tr>
<td>%Lf</td>
<td>long double</td>
<td>Not available on all compilers.</td>
</tr>
</tbody>
</table>
Sometimes you want to use a value that does not change.

The keyword `const` indicates a variable that never changes.

```c
const float PI = 3.1415927; /* The classic circle constant */
```

- If we tried to reset the value of PI to 3.0, we would generate an error message.
C not only provides you with a rich set of declarations, but also gives you a large number of special-purpose operators.

The programmer wants to increment (increase by 1) a variable.
- `total_entries = total_entries + 1;`
- `++total_entries;`

A similar operator, `--`, can be used for decrementing (decreasing by 1) a variable.
- `--number_left; /* is the same as */`
- `number_left = number_left - 1;`

```
total_entries += 2;
total_entries = total_entries + 2;
```
The two forms of the increment operator are called the prefix form (\texttt{++x}) and the postfix form (\texttt{x++}).

- \texttt{number = 5; result = number++;}
- \texttt{number = 5; result = ++number;}

\texttt{value = 1; result = (value++ * 5) + (value++ * 3);}
Programming Exercise (10 Minutes)

- Write a program that converts input kilometers of **WBC pitchers** per hour to miles per hour. \( (\text{miles} = \text{kilometer} \cdot 0.6213712) \)
  - Input: Pitcher Name (String), KMs/hours (Integer).
  - Output: Pitcher Name, Miles/hours (Integer).
Decision and Control Statements

Chapter 6.
The **if** statement allows us to put some decision-making into our programs.

```
if (condition) statement;
```

- If the condition is true (nonzero), the statement will be executed. If the condition is false (0), the statement will not be executed.

**Relational Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
</tbody>
</table>
Multiple statements may be grouped by putting them inside curly braces ({}).

```c
if (total_owed <= 0) {
    ++zero_count;
    printf("You owe nothing.\n");
}
```

**else Statement**

```c
if (condition)
    statement;
else
    statement;
```
What is the difference in these codes?

```c
if (count < 10) /* if #1 */
    if ((count % 4) == 2) /* if #2 */
        printf("Condition:White\n");
else
    printf("Condition:Tan\n");

if (count < 10) { /* if #1 */
    if ((count % 4) == 2) /* if #2 */
        printf("Condition:White\n");
    else
        printf("Condition:Tan\n");
}
The function `strcmp` compares two strings, and then returns zero if they are equal or nonzero if they are different.

```c
if (strcmp(string1, string2))
    printf("......");
else
    printf("......");
```

```c
/* Check to see if string1 == string2 */
if (strcmp(string1, string2) == 0)
    printf("Strings equal\n");
else
    printf("Strings not equal\n");
```
Loopy statements allow the program to repeat a section of code any number of times or until some condition occurs.

Loops are used to count the number of words in a document or to count the number of accounts that have past-due balances.
The **while** statement is used when the program needs to perform repetitive tasks.

```plaintext
while (condition)  
statement;
```

- The program will repeatedly execute the statement inside the **while** until the condition becomes false (0).
Fibonacci execution (10 Minute)

- \( fn = fn-1 + fn-2 \)
  - \( fn : \text{next\_number} \)
  - \( fn-1 : \text{current\_number} \)
  - \( fn-2 : \text{old\_number} \)
#include <stdio.h>

int old_number; /* previous Fibonacci number */
int current_number; /* current Fibonacci number */
int next_number; /* next number in the series */
int main()
{
    /* start things out */
    old_number = 1;
    current_number = 1;
    printf("1\n"); /* Print first number */
    while (current_number < 100) {
        printf("%d\n", current_number);
        next_number = current_number + old_number;
        old_number = current_number;
        current_number = next_number;
    }
    return (0);
}
Loops can be exited at any point through the use of a **break** statement.

- Suppose we want to add a series of numbers, but we don't know how many numbers are to be added together.
  
  ```
  while (1) {
    if (item == 0)
      break;
  }
  ```

- The only way to exit this loop is through a **break** statement.
- When we see the end of the list indicator (0), we use the statement:

  ```
  if (item == 0)
    break;
  ```
#include <stdio.h>

char line[100]; /* line of data for input */
int total; /* Running total of all numbers so far */
int item; /* next item to add to the list */
int main()
{
    total = 0;
    while (1) {
        printf("Enter # to add \n");
        printf(" or 0 to stop:");
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);
        if (item == 0)
            break;
        total += item;
        printf("Total: %d\n", total);
    }
    printf("Final total %d\n", total);
    return (0);
}
The `continue` statement is very similar to the `break` statement, except that instead of terminating the loop, `continue` starts re-executing the body of the loop from the top.

```c
while (1) {
    printf("Enter # to add\n");
    printf(" or 0 to stop:");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &item);
    if (item == 0)
        break;
    if (item < 0) {
        ++minus_items;
        continue;
    }
    total += item;
    printf("Total: %d\n", total);
}
```
The **for** statement allows the programmer to execute a block of code for a specified number of times.

```c
for (initial-statement; condition; iteration-statement)  
  body-statement;
```

```c
initial-statement;  
while (condition) {  
  body-statement;  
  iteration-statement;  
}
```
What does it give us the wrong answers?

```c
#include <stdio.h>
/
* This program produces a Celsius to Fahrenheit conversion
* chart for the numbers 0 to 100.
*/
/* The current Celsius temperature we are working with */
int celsius;
int main() {
    for (celsius = 0; celsius <= 100; ++celsius) {
        printf("Celsius:%d Fahrenheit:%d\n", celsius, (celsius * 9) / 5 + 32);
        return (0);
    }
```
#include <stdio.h>

char line[100]; /* line of input */
int seven_count; /* number of 7s in the data */
int data[5]; /* the data to count 3 and 7 in */
int three_count; /* the number of 3s in the data */
int index; /* index into the data */
int main() {
    seven_count = 0;
    three_count = 0;
    printf("Enter 5 numbers\n");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d %d %d %d", &data[1], &data[2], &data[3],
          &data[4], &data[5]);
    for (index = 1; index <= 5; ++index) {
        if (data[index] == 3) ++three_count;
        if (data[index] == 7) ++seven_count;
    }
    printf("Threes %d Sevens %d\n", three_count, seven_count);
    return (0);
}
The `switch` statement is similar to a chain of `if/else` statements.

The `switch` statement evaluates the value of an expression and branches to one of the case labels.

Duplicate labels are not allowed.

The `default` label can be put anywhere in the `switch`. No two `case` labels can have the same value.

```java
switch (expression) {
    case constant1 :
        statement
        . . .
        break ;

    case constant2 :
        statement
        . . .
        /* Fall through */

    default:
        statement
        . . .
        break ;
}
```
Example: Why do you need to use the switch Statement?

```c
if (operator == '+') {
    result += value;
}
else if (operator == '-') {
    result -= value;
}
else if (operator == '*') {
    result *= value;
}
else if (operator == '/') {
    if (value == 0) {
        printf("Error: Divide by zero\n");
        printf(" operation ignored\n");
    } else
    result /= value;
}
else {
    printf("Unknown operator %c\n", operator);
}
```
What is the result and the unintended problem in this code?

code = 0;
/* a not so good example of programming */
switch (control) {
    case 0:
        printf("Reset\n");
        break;
    case 1:
        printf("Initializing\n");
        break;
    case 2:
        printf("Working\n");
}
Exercise 1.
- Write a program that converts numbers to words. For example, 895 results in "eight nine five."

Exercise 2.
- Write a program to average $n$ numbers.

Exercise 3.
- Write a program to draw a pyramidal figure using a star (*).
Variable Scope and Functions

- **Scope and Class**
  - All variables have two attributes: scope and class.
  - The scope of a variable
    - The area of the program in which the variable is valid.
      - A *global variable* is valid everywhere (hence the name global), so its scope is the whole program.
      - A *local variable* has a scope that is limited to the *block* in which it is declared and cannot be accessed outside that block.
      - A *block* is a section of code enclosed in curly braces (`{}`).
    - You can declare a local variable with the same name as a global variable.
Local and Global Variables

```plaintext
int global;                   /* a global variable */
main()
{
    int local;                /* a local variable */
    global = 1;               /* global can be used here */
    local = 2;                /* so can local */
    {
        int very_local       /* beginning a new block */
        /* this is local to the block */
        very_local = global + local;
    }
    /* We just closed the block */
    /* very_local can not be used */
}
```
int total;  /*total number of entries*/
int count;  /*count of total entries*/

main()
{
    total = 0;
    count = 0;  /*set global counter*/

    [ /*local code*/
    int count;  /*a local counter*/
    count=0;
    while (1) {
        if (count > 10)
            break;
        total += count;
        ++count;
    }
    ++count;
    return (0);
}
Variable Scope and Functions

Class

- The class of a variable may be either permanent or temporary.
  - Global variables are always permanent
    - They are created and initialized before the program starts and remain until it terminates.
  - Temporary variables are allocated from a section of memory called the stack at the beginning of the block.
    - If you try to allocate too many temporary variables, you will get a "Stack overflow" error.
  - The size of the stack depends on the system and compiler you are using. On many
    - UNIX systems, the program is automatically allocated the largest possible stack.
- Local variables are temporary unless they are declared static.
- static has an entirely different meaning when used with global variables. It indicates that a variable is local to the current file.
#include <stdio.h>

int main() {
    int counter; /* loop counter */
    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1; /* A temporary variable */
        static int permanent = 1; /* A permanent variable */
        printf("Temporary %d Permanent %d\n", temporary, permanent);
        ++temporary;
        ++permanent;
    }
    return (0);
}

<table>
<thead>
<tr>
<th>Declared</th>
<th>Scope</th>
<th>Class</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outside all blocks</td>
<td>Global</td>
<td>Permanent</td>
<td>Once</td>
</tr>
<tr>
<td><strong>static</strong> outside all blocks</td>
<td>Global</td>
<td>Permanent</td>
<td>Once</td>
</tr>
<tr>
<td>Inside a block</td>
<td>Local</td>
<td>Temporary</td>
<td>Each time block is entered</td>
</tr>
<tr>
<td><strong>static</strong> inside a block</td>
<td>Local</td>
<td>Permanent</td>
<td>Once</td>
</tr>
</tbody>
</table>