Array
- An array is a set of consecutive memory locations used to store data.
- int data_list[3];

String
- Strings are sequences of characters.
- String constants consist of text enclosed in double quotes ("").
- Reading Strings
  - The standard function fgets can be used to read a string from the keyboard.

Reading Numbers
- We use fgets to read a line of input and sscanf to convert the text into numbers.

Types of Integers/Floats
- Long, short, sizeof
Decision and Control Statements

- The **if** statement allows us to put some decision-making into our programs.
- The **while** statement is used when the program needs to perform repetitive tasks.
- Loops can be exited at any point through the use of a **break** statement.
- The **continue** starts re-executing the body of the loop from the top.
- The **for** statement allows the programmer to execute a block of code for a specified number of times.
- The **switch** statement evaluates the value of an expression and branches to one of the case labels.
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 (hiding).
Local and Global Variables

```c
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*/
}
```
```c
int total;        /* total number of entries */
int count;        /* count of total entries */

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

    count = 0;
    while (1) {
        if (count > 10)
            break;
        total += count;
        ++count;
    }

    ++count;
    return (0);
}
```
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**.
- The **static** has an entirely different meaning when used with global variables. It indicates that a variable is local to the current file.
```c
#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);
}
```
Functions allow us to group commonly used code into a compact unit that can be used repeatedly.

**main**
- A special function called at the beginning of the program.
- All other functions are directly or indirectly called from main.

Each function should begin with a comment block containing the following:

- **Name**
  - Name of the function

- **Description**
  - Description of what the function does

- **Parameters**
  - Description of each of the parameters to the function

- **Returns**
  - Description of the return value of the function
Functions

- \textit{float triangle (float width, float height)}
  - \textit{float} is the function type. The two parameters are width and height. They are of type \textit{float} also.

- C uses a form of parameter passing called \textit{"Call by value"}.
  - \textit{triangle(1.3, 8.3)};
  - C copies the value of the parameters into the function's parameters (width and height) and then starts executing the function's code.

- \textbf{Exercise: complete a code which print an area using a \textit{“triangle” function}.}
  ```c
  float triangle (float width, float height)
  {
    float area; /* Area of the triangle */
    area = width * height / 2.0;
    return (area);
  }
  ```
A function can have any number of parameters, including none.

- `int next_index();`
- `int next_index(void);`
  - `void` is also used to indicate that a function does not return a value.
A bit is the smallest unit of information. Normally, it is represented by the values 1 and 0.
The `printf` format for hexadecimal is `%x`; for octal the format is `%o`.

```c
int number = 0xAF;
printf("Number is %x %d %o\n", number, number, number);
```

Hexadecimal and Binary

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>
Bit Operators

- Bit operators allow the programmer to work on individual bits.
- The bit operators treat each bit as independent.
- Bitwise Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>~</td>
<td>Complement</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift right</td>
</tr>
</tbody>
</table>
Bit Operators

- **The and Operator (*)&
  - The *and* operator compares two bits. If they both are one, the result is one.

- **Bitwise or (|)**
  - The *inclusive or* operator (also known as just the *or* operator) compares its two operands and if one or the other bit is a one, the result is a one.

- **The Bitwise Exclusive or (^)**
  - The *exclusive or* (also known as *xor*) operator results in a one when either of its two operands is a one, but not both.

- **The Ones Complement Operator (Not) (~)**
  - The *not* operator (also called the invert operator, or bit flip) is a unary operator that returns the inverse of its operand.

- **The Left- and Right-Shift Operators (<<, >>)**
  - The left-shift operator moves the data to the left a specified number of bits. Any bits that are shifted out the left side disappear. New bits coming in from the right are zeros.
  - The right shift does the same thing in the other direction.
C provides the programmer with a rich set of data types.

Through the use of **structures, unions, and enumerated types**, the programmer can extend the language with new types.
In previous chapters, we have used arrays for storing a group of similar data types. However we can have a mixed bag – **structure**.

- In a structure, each element or **field** is named and has its own data type.
- To access the field of structure

```c
struct [structure-name] {
    field-type field-name; /* comment */
    field-type field-name; /* comment */
    . . . .  
}

[variable-name];
```

```c
type printer_cable_bin; /* $12.95 is the new price */
```

printer_cable_bin.cost = 1295; /* $12.95 is the new price */
Declarations of Structure

```c
struct bin {
    char name[30]; /* name of the part */
    int quantity; /* how many are in the bin */
    int cost; /* The cost of a single part (in cents) */
};

printer_cable_bin;
/* where we put the print cables */
```

```c
struct {
    char name[30]; /* name of the part */
    int quantity; /* how many are in the bin */
    int cost; /* The cost of a single part (in cents) */
};

printer_cable_bin;
```

```c
struct bin{
    char name[30]; /* name of the part */
    int quantity; /* how many are in the bin */
    int cost; /* The cost of a single part (in cents) */
};
```
- A **union** is similar to a structure; however, it defines a single location that can be given many different field names:
- You might think of a **structure** as a large box divided up into several different compartments, each with its own name. A **union** is a box, not divided at all, with several different labels placed on the single compartment inside.
In a union, all fields occupy the same space, so only one may be active at a time.

- **If you put something in i_value, assigning something to f_value wipes out the old value of i_value.**

Unions are frequently used in the area of communications.

- **For overlapping of a variety of messages.**
C allows the programmer to define her own variable types through the **typedef** statement.

- **type-declaration** is the same as a variable declaration except that a type name is used instead of a variable name.

```c
typedef type-declaration;
```

- **typedef int count;**
  - defines a new type count that is the same as an integer.
  - “**count flag;**” is equivalent to “**int flag;**”
```c
struct complex_struct {
    double real;
    double imag;
};

typedef struct complex_struct complex;

typedef struct complex_struct {
    double real;
    double imag;
} complex_struct;

typedef int group[10]; /* Create a new type "group" */

group totals; /* Use the new type for a variable */
for (i = 0; i < 10; i++)
    totals[i] = 0;
```
The enumerated data type is designed for variables that contain only a limited set of values. These values are referenced by name (tag).

```
enum enum-name { tag-1, tag-2, ...} variable-name

typedef int week_day;
/* define the type for week_days */
const int SUNDAY = 0;
const int MONDAY = 1;
const int TUESDAY = 2;
const int WEDNESDAY = 3;
const int THURSDAY = 4;
const int FRIDAY = 5;
const int SATURDAY = 6;
/* now to use it */
week_day today = TUESDAY;
```

generate enum week_day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
/* now use it */
enum week_day today = TUESDAY;
Arrays of Structures

- Structures and arrays can be combined.

```c
struct time {
    int hour; /* hour (24 hour clock) */
    int minute; /* 0-59 */
    int second; /* 0-59 */
};

const int MAX_LAPS = 4; /* we will have only 4 laps */

/* the time of day for each lap*/
struct time lap[MAX_LAPS];
```
Simple Pointers

A thing

A pointer

0x1000
thing_ptr
Syntax of Pointers

- **Declarations**
  - `int thing; /* define a thing */`
  - `int *thing_ptr; /* define a pointer to a thing */`

- **Pointer Operators**
  - The operator `&` (ampersand) returns the *address of a thing* which is a pointer.
  - The operator `*` (asterisk) returns the *object to which a pointer points*.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*</code></td>
<td>Dereference (given a pointer, get the thing referenced)</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>Address of (given a thing, point to it)</td>
</tr>
</tbody>
</table>
**Pointer Operator Syntax**

<table>
<thead>
<tr>
<th>C Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>thing</code></td>
<td>Simple thing (variable)</td>
</tr>
<tr>
<td><code>&amp;thing</code></td>
<td>Pointer to variable <code>thing</code></td>
</tr>
<tr>
<td><code>thing_ptr</code></td>
<td>Pointer to an integer (may or may not be specific integer <code>thing</code>)</td>
</tr>
<tr>
<td><code>*thing_ptr</code></td>
<td>Integer</td>
</tr>
</tbody>
</table>

```c
int thing; /* Declare an integer (a thing) */
thing = 4;

int *thing_ptr; /* Declare a pointer to a thing */
thing_ptr = &thing; /* Point to the thing */
*thing_ptr = 5; /* Set "thing" to 5 */
```
**Pointer Operations**

A. `thing_ptr = &thing;

Assigns `thing`'s address to `thing_ptr`.

B. `other = *thing_ptr;`

Assigns to `other` the value at the address `thing_ptr` carries.

C. `*thing_ptr = 6;`

Assigns to a value to what `thing_ptr` points to.
Memory Access using Variables

<table>
<thead>
<tr>
<th>0x1003</th>
<th>0x1002</th>
<th>0x1001</th>
<th>0x1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‘A’ (=65, 0x41)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...  char c
short int b

int a;
short int b
c = ‘A’;

int a  “Little/Big-endian Problem”

start of stack

Memory (specially, stack)
Memory Access using Pointers

```
int a;
char c;
char * pointer_of_c = &c;
*pointer_of_c = 'A';
```
int thing;
Int* thing_ptr;
*thing  
- illegal. It asks C to get the object pointed to by the variable thing. Because thing is not a pointer, this operation is invalid.

&thing_ptr  
- legal, but strange. thing_ptr is a pointer. The & (address of operator) gets a pointer to the object (in this case thing_ptr). The result is a pointer to a pointer.
Examples of Pointer Operations

```c
int something;
int *first_ptr; /* one pointer */
int *second_ptr; /* another pointer */

something = 1; /* give the thing a value */

first_ptr = &something;
second_ptr = first_ptr;

int thing_var; /* define a variable for thing */
int *thing_ptr; /* define a pointer to thing */

thing_var = 2; /* assigning a value to thing */
printf("Thing %d\n", thing_var);
thing_ptr = &thing_var; /* make the pointer point to thing */
*thing_ptr = 3; /* thing_ptr points to thing_var so */
printf("Thing %d\n", thing_var);
printf("Thing %d\n", *thing_ptr);
```
C passes parameters using "call by value."

- the parameters go only one way into the function.
- The only result of a function is a single return value.
#include <stdio.h>
void inc_count(int count)
{
    count++;
}
int main()
{
    int count = 0; /* number of times through */
    printf("Before function call- count:%d\n", count);
    inc_count(count);
    printf("After function call- count:%d\n", count);
    return (0);
}
Pointers as Function Arguments

- Pointers can be used to get around this restriction.

```c
#include <stdio.h>
void inc_count(int * count_ptr)
{
    (*count_ptr)++;
}
int main()
{
    int count = 0; /* number of times through */
    printf(“Before function call- count:%d\n”, count);
    inc_count(&count);
    printf(“After function call- count:%d\n”, count);
    return (0);
}
```
There is a special pointer called **NULL**.

- It points to nothing. (The actual numeric value is 0.)
- The standard include file, `locale.h`, defines the constant NULL.
*array_ptr is the same as array[0], *(array_ptr+1) is the same as array[1], *(array_ptr+2) is the same as array[2], and so on.

```c
char array[5];
char *array_ptr = &array[0];
```
Exercise

```c
#include <stdio.h>
#define ARRAY_SIZE 10
char array[ARRAY_SIZE + 1] = "0123456789";
int main()
{
    int index;
    printf("\narray[0] (array+0) array[0]\n");
    for(index=0;index < ARRAY_SIZE;++index)
        printf("%-10p %-10p 0x%x\n", 
                &array[index],(array+index),array[index]);
    return 0;
}
```
C provides a shorthand for dealing with arrays. Rather than writing:

- `array_ptr = &array[0];`
- `array_ptr = array;`
```c
#include <stdio.h>
int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int *array_ptr;
int main()
{
    array_ptr = array;
    while (*((unsigned char*)array_ptr) != 0)
        ++array_ptr;
    printf("Number of elements before zero %d\n", array_ptr - array);
    return (0);
}
```
Problem

- Suppose we are given a string of the form "Last/First." We want to split this into two strings, one containing the first name and one containing the last name.
- We need a function to find the slash in the name. The standard function `strchr` performs this job for us. **However we use the pointer.**
Implement this function.

- `char *my_strchr(char * string_ptr, char find)`
  - It checks character `find` from `char * string_ptr`.
  - If `string_ptr` contains the character in it, function returns the pointer for the substring started from the pointer.
  - Unless, function returns `NULL`.
int main()
{
    char line[80]; /* The input line */
    char *first_ptr; /* pointer to the first name */
    char *last_ptr; /* pointer to the last name */
    fgets(line, sizeof(line), stdin);
    /* Get rid of trailing newline */
    line[strlen(line)-1] = '\0';
    last_ptr = line; /* last name is at beginning of line */
    first_ptr = my_strchr(line, '/'); /* Find slash */
    /* Check for an error */
    if (first_ptr == NULL) {
        fprintf(stderr,"Error: Unable to find slash in %s\n", line);
        exit (8);
    }
    *first_ptr = '\0'; /* Zero out the slash */
    ++first_ptr; /* Move to first character of name */
    printf("First:%s Last:%s\n", first_ptr, last_ptr);
    return (0);
}