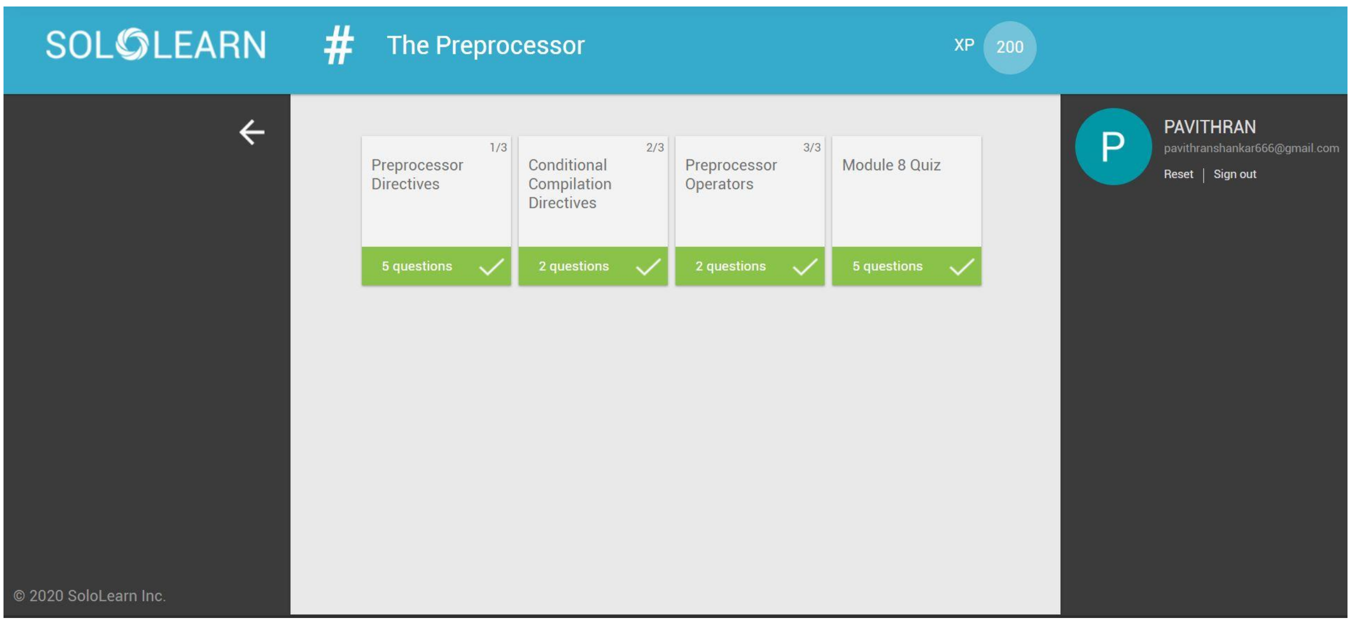
DAILY ASSESSMENT FORMAT

Date:	**20 JUNE 2020**	**Name:**	**PAVITHRAN S**
Course:	**C PROGRAMMING**	**USN:**	**4AL17EC068**
Topic:	**BASICS**	**Semester & Section:**	**6TH B**
Github Repository:	**Pavithran**		

FORENOON SESSION DETAILS

Image of session



Report – Report can be typed or hand written for up to two pages.

Functions in C

When the parameter types and names are included in a declaration, the declaration is called a **function prototype**.

For example, the **square** function prototype appears above main():

```
#include <stdio.h>

/* declaration */
int square (int num);

int main() {
int x, result;

x = 5;
result = square(x);
printf("%d squared is %d\n", x, result);

return 0;
}
```

Our **square** function returns an integer and takes one parameter of type int.

The last step is actually **defining** the function. Function definitions usually appear after the **main()** function.
The complete program below shows the **square** function declaration and definition:

```
#include <stdio.h>

/* declaration */
int square (int num);

int main() {
int x, result;

x = 5;
result = square(x);
printf("%d squared is %d\n", x, result);

return 0;
}

/* definition */
int square (int num) {
int y;

y = num * num;
```

```
return(y);
}
```

Static Variables

Static variables have a local scope but are not destroyed when a function is exited. Therefore, a static variable retains its value for the life of the program and can be accessed every time the function is re-entered.
A static variable is initialized when declared and requires the prefix **static**.
The following program uses a static variable:

```
#include <stdio.h>

void say_hello();

int main() {
int i;

for (i = 0; i < 5; i++) {
say_hello();
}

return 0;
}

void say_hello() {
static int num_calls = 1;

printf("Hello number %d\n", num_calls);
num_calls++;

}
```

Recursive Functions

An algorithm for solving a problem may be best implemented using a process called **recursion**. Consider the factorial of a number, which is commonly written as 5! = 5 * 4 * 3 * 2 * 1.
This calculation can also be thought of as repeatedly calculating num * (num -1) until num is 1.

A **recursive function** is one that calls itself and includes a base case, or exit condition, for ending the recursive calls. In the case of computing a factorial, the base case is **num** equal to 1.

For example:

```
#include <stdio.h>

//function declaration
int factorial(int num);

int main() {
int x = 5;

printf("The factorial of %d is %d\n", x, factorial(x));

return 0;
}

//function definition
int factorial(int num) {

if (num == 1) /* base case */
return (1);
else
return (num * factorial(num - 1));
}
```

Arrays in C

An **array** is a data structure that stores a collection of related values that are all the same type. Arrays are useful because they can represent related data with one descriptive name rather than using separate variables that each must be named uniquely.
For example, the array **test_scores[25]** can hold 25 test scores.
An array declaration includes the type of the values it stores, an identifier, and square brackets [ ] with a number that indicates the array size.
For example: `int test_scores[25]; /* An array size 25 */`
You can also initialize an array when it is declared, as in the following statement: `float prices[5] = {3.2, 6.55, 10.49, 1.25, 0.99};`
Note that initial values are separated by commas and placed inside curly braces { }.
An array can be partially initialized, as in: `float prices[5] = {3.2, 6.55};`

Accessing Array Elements

The contents of an array are called **elements** with each element accessible by an index number.
In C, index numbers start at **0**.
An array with 5 elements will have index numbers 0, 1, 2, 3, and 4. Consider an array x: `int x[5] =`

```
{20, 45, 16, 18, 22};
```

It can be thought of as:
0 => [20]
1 => [45]
2 => [16]
3 => [18]
4 => [22]

To access an array element, refer to its index number.
For example:

```
int x[5] = {20, 45, 16, 18, 22};
printf("The second element is %d\n", x[1]); /* 45 */
```

Using Loops with Arrays

Many algorithms require accessing every element of an array to check for data, store information, and other tasks. This can be done in a process called **traversing the array**, which is often implemented with a for loop because the loop control variable naturally corresponds to array indexes.
Consider the following program:

```
float purchases[3] = {10.99, 14.25, 90.50};
float total = 0;
int k;

/* total the purchases */
for (k = 0; k < 3; k++) {
total += purchases[k];
}

printf("Purchases total is %6.2f\n", total);
/* Output: Purchases total is 115.74 */
```

Two-Dimensional Arrays

A **two-dimensional array** is an array of arrays and can be thought of as a table. You can also think of a two-dimensional array as a grid for representing a chess board, city blocks, and much more.

A two-dimensional array declaration indicates the number of number rows and the number of columns.
For example: `int a[2][3]; /* A 2 x 3 array */`
Nested curly braces are used to initialize elements row by row, as in the following statement:

```
int a[2][3] = {
{3, 2, 6},
```

```
{4, 5, 20}
};
```
The same statement can also take the form: `int a[2][3] = { {3, 2, 6}, {4, 5, 20} };`

Accessing Two-Dimensional Arrays

To access an element of a two-dimensional array, both the row index and column index are required.
For example, the following statements display the value of an element and then assign a new value:
```
int a[2][3] = {
{3, 2, 6},
{4, 5, 20}
};
printf("Element 3 in row 2 is %d\n", a[1][2]); /* 20 */
a[1][2] = 25;
printf("Element 3 in row 2 is %d\n", a[1][2]); /* 25 */
```
Just as a **for** loop is used to iterate through a one-dimensional array, nested **for** loops are used to traverse a two-dimensional array:
```
int a[2][3] = {
{3, 2, 6},
{4, 5, 20}
};
int k, j;
/* display array contents */
for (k = 0; k < 2; k++) {
for (j = 0; j < 3; j++) {
printf(" %d", a[k][j]);
}
printf("\n");
}
```

Using Memory

C is designed to be a low-level language that can easily access memory locations and perform memory-related operations.
For instance, the **scanf()** function places the value entered by the user at the location, or **address**, of the variable. This is accomplished by using the & symbol.

For Example:
```
int num;
printf("Enter a number: ");
```

```
scanf("%d", &num);

printf("%d", num);
```

&num is the address of variable **num**.

A memory address is given as a **hexadecimal** number. **Hexadecimal**, or **hex**, is a base-16 number system that uses digits 0 through 9 and letters A through F (16 characters) to represent a group of four binary digits that can have a value from 0 to 15.
It's much easier to read a hex number that is 8 characters long for 32 bits of memory than to try to decipher 32 1s and 0s in binary.
The following program displays the memory addresses for variables **i** and **k**:

```
void test(int k);

int main() {
int i = 0;

printf("The address of i is %x\n", &i);
test(i);
printf("The address of i is %x\n", &i);
test(i);

return 0;
}

void test(int k) {
printf("The address of k is %x\n", &k);
```

What is a Pointer?

Pointers are very important in C programming because they allow you to easily work with memory locations.
They are fundamental to arrays, strings, and other data structures and algorithms.
A **pointer** is a variable that contains the **address** of another variable. In other words, it "points" to the location assigned to a variable and can indirectly access the variable.
Pointers are declared using the * symbol and take the form: `pointer_type *identifier`
pointer_type is the type of data the pointer will be pointing to. The actual pointer data type is a hexadecimal number, but when declaring a pointer, you must indicate what type of data it will be pointing to.
Asterisk * declares a pointer and should appear next to the identifier used for the pointer variable.
The following program demonstrates variables, pointers, and addresses:

```
int j = 63;
int *p = NULL;
```

```
p = &j;

printf("The address of j is %x\n", &j);
printf("p contains address %x\n", p);
printf("The value of j is %d\n", j);
printf("p is pointing to the value %d\n", *p);
```

Pointers in Expressions

Pointers can be used in **expressions** just as any variable. Arithmetic operators can be applied to whatever the pointer is pointing to.
For example:

```
int x = 5;
int y;
int *p = NULL;
p = &x;

y = *p + 2; /* y is assigned 7 */
y += *p; /* y is assigned 12 */
*p = y; /* x is assigned 12 */
(*p)++; /* x is incremented to 13 */

printf("p is pointing to the value %d\n", *p);
```

Pointers and Arrays

Pointers are especially useful with arrays. An array declaration reserves a block of contiguous memory addresses for its elements. With pointers, we can point to the first element and then use **address arithmetic** to traverse the array:
+ is used to move forward to a memory location
- is used to move backward to a memory location

Consider the following program:

```
int a[5] = {22, 33, 44, 55, 66};
int *ptr = NULL;
int i;

ptr = a;
for (i = 0; i < 5; i++) {
printf("%d ", *(ptr + i));
}
```

More Address Arithmetic

Address arithmetic can also be thought of as pointer arithmetic because the operations involve pointers.
Besides using + and – to refer to the next and previous memory locations, you can use the assignment operators to change the address the pointer contains.

For example:

```
int a[5] = {22, 33, 44, 55, 66};
int *ptr = NULL;

ptr = a; /* point to the first array element */
printf("%d %x\n", *ptr, ptr); /* 22 */
ptr++;
printf("%d %x\n", *ptr, ptr); /* 33 */
ptr += 3;
printf("%d %x\n", *ptr, ptr); /* 66 */
ptr--;
printf("%d %x\n", *ptr, ptr); /* 55 */
ptr -= 2;
printf("%d %x\n", *ptr, ptr); /* 33 */
```

Pointers and Functions

Pointers greatly expand the possibilities for functions. No longer are we limited to returning one value. With pointer parameters, your functions can alter actual data rather than a copy of data.
To change the actual values of variables, the calling statement passes addresses to pointer parameters in a function.
For example, the following program swaps two values:

```
void swap (int *num1, int *num2);

int main() {
int x = 25;
int y = 100;

printf("x is %d, y is %d\n", x, y);
swap(&x, &y);
printf("x is %d, y is %d\n", x, y);

return 0;
}

void swap (int *num1, int *num2) {
```

```
int temp;

temp = *num1;
*num1 = *num2;
*num2 = temp;
}
```

Functions with Array Parameters

An array cannot be passed by value to a function. However, an array name is a pointer, so just passing an array name to a function is passing a pointer to the array.

Consider the following program:

```
int add_up (int *a, int num_elements);

int main() {
int orders[5] = {100, 220, 37, 16, 98};

printf("Total orders is %d\n", add_up(orders, 5));

return 0;
}

int add_up (int *a, int num_elements) {
int total = 0;
int k;

for (k = 0; k < num_elements; k++) {
total += a[k];
}

return (total);
}
```

Functions that Return an Array

Just as a pointer to an array can be passed into a function, a pointer to an array can be returned, as in the following program:

```
int * get_evens();

int main() {
int *a;
int k;
```

```
a = get_evens(); /* get first 5 even numbers */
for (k = 0; k < 5; k++)
printf("%d\n", a[k]);

return 0;
}

int * get_evens() {
static int nums[5];
int k;
int even = 0;

for (k = 0; k < 5; k++) {
nums[k] = even += 2;
}

return (nums);
}
```

Strings

A **string** in C is an array of characters that ends with a **NULL character '\0'**.
A string declaration can be made in several ways, each with its own considerations.
For example: `char str_name[str_len] = "string";`
This creates a string named *str_name* of *str_len* characters and initializes it to the value "string".
When you provide a **string literal** to initialize the string, the compiler automatically adds a NULL character '\0' to the char array.
For this reason, you must declare the array size to be at least one character longer than the expected string length.
The statements below creates strings that include the NULL character. If the declaration does not include a char array size, then it will be calculated based on the length of the string in the initialization plus one for '\0':

```
char str1[6] = "hello";
char str2[ ] = "world"; /* size 6 */
```

String Input

Programs are often interactive, asking the user for input.
To retrieve a line of text or other string from the user, C provides the scanf(), **gets**(), and **fgets**() functions.
You can use scanf() to read input according to the format specifiers.

For example:

```
char first_name[25];
int age;
printf("Enter your first name and age: \n");
scanf("%s %d", first_name, &age);
```

String Output

String output is handled with the **fputs**(), **puts**(), and **printf**() functions.
The **fputs()** requires the name of the string and a pointer to where you want to print the string. To print to the screen, use **stdout** which refers to the **standard output**.
For example:

```
#include <stdio.h>
int main()
{
char city[40];
printf("Enter your favorite city: ");
gets(city);
// Note: for safety, use
// fgets(city, 40, stdin);

fputs(city, stdout);
printf(" is a fun city.");

return 0;
}
```

The sprintf and sscanf Functions

A formatted string can be created with the **sprintf()** function. This is useful for building a string from other data types.

For example:

```
#include <stdio.h>
int main()
{
char info[100];
char dept[ ] = "HR";
int emp = 75;
sprintf(info, "The %s dept has %d employees.", dept, emp);
printf("%s\n", info);

return 0;
```

}Try It Yourself

The string.h Library

The string.h library contains numerous string functions.
The statement **#include <string.h>** at the top of your program gives you access to the following:
strlen(str) Returns the length of the string stored in **str**, not including the NULL character.
strcat(str1, str2) Appends (concatenates) **str2** to the end of **str1** and returns a pointer to **str1**.
strcpy(str1, str2) Copies **str2** to **str1**. This function is useful for assigning a string a new value.

The program below demonstrates string.h functions:

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

int main()
{
char s1[ ] = "The grey fox";
char s2[ ] = " jumped.";

strcat(s1, s2);
printf("%s\n", s1);
printf("Length of s1 is %d\n", strlen(s1));
strcpy(s1, s2);
printf("s1 is now %s \n", s1);

return 0;
}
```

Converting a String to a Number

Converting a string of number characters to a numeric value is a common task in C programming and is often used to prevent a run-time error.
Reading a string is less error-prone than expecting a numeric value, only to have the user accidentally type an "o" rather than a "0" (zero).

The **stdio.h** library contains the following functions for converting a string to a number:
int atoi(str) Stands for ASCII to integer. Converts **str** to the equivalent int value. 0 is returned if the first character is not a number or no numbers are encountered.
double atof(str) Stands for ASCII to float. Converts **str** to the equivalent double value. 0.0 is returned if the first character is not a number or no numbers are encountered.
long int atol(str) Stands for ASCII to long int. Converts **str** to the equivalent long integer value. 0 is returned if the first character is not a number or no numbers are encountered.

The following program demonstrates **atoi**.

```
#include <stdio.h>
int main()
{
char input[10];
int num;

printf("Enter a number: ");
gets(input);
num = atoi(input);

return 0;
}
```

Array of Strings

A two-dimensional array can be used to store related strings.
Consider the following statement which declares an array with 3 elements, each holding 15 characters:

```
char trip[3][15] = {
"suitcase",
"passport",
"ticket"
};
```

Although the string lengths vary, it is necessary to declare a size large enough to hold the longest string. Additionally, it can be very cumbersome to access the elements. Referring to trip[0] for "suitcase" is error-prone. Instead, you must think of the element at **[0][0]** as **'s'**, the element at **[2][3]** as **'k'**, and so on.

An easier, more intuitive way to deal with a collection of related strings is with an array of pointers, as in the following program:

```
char *trip[ ] = {
"suitcase",
"passport",
"ticket"
};

printf("Please bring the following:\n");
for (int i = 0; i < 3; i++) {
printf("%s\n", trip[ i ]);
}
```

Function Pointers

Since pointers can point to an address in any memory location, they can also point to the start of executable code.
Pointers to functions, or **function pointers**, point to executable code for a function in memory. Function pointers can be stored in an array or passed as arguments to other functions.

A function pointer **declaration** uses the * just as you would with any pointer:`return_type (*func_name)(parameters)`
The parentheses around (*func_name) are important. Without them, the compiler will think the function is returning a pointer.

After declaring the function pointer, you must assign it to a function. The following short program declares a function, declares a function pointer, assigns the function pointer to the function, and then calls the function through the pointer:

```
#include <stdio.h>
void say_hello(int num_times); /* function */

int main() {
void (*funptr)(int); /* function pointer */
funptr = say_hello; /* pointer assignment */
funptr(3); /* function call */

return 0;
}

void say_hello(int num_times) {
int k;
for (k = 0; k < num_times; k++)
printf("Hello\n");
}
```

Array of Function Pointers

An array of function pointers can replace a **switch** or an **if** statement for choosing an action, as in the following program:

```
#include <stdio.h>

int add(int num1, int num2);
int subtract(int num1, int num2);
int multiply(int num1, int num2);
int divide(int num1, int num2);

int main()
{
```

```
int x, y, choice, result;
int (*op[4])(int, int);

op[0] = add;
op[1] = subtract;
op[2] = multiply;
op[3] = divide;
printf("Enter two integers: ");
scanf("%d%d", &x, &y);
printf("Enter 0 to add, 1 to subtract, 2 to multiply, or 3 to divide: ");
scanf("%d", &choice);
result = op[choice](x, y);
printf("%d", result);

return 0;
}

int add(int x, int y) {
return(x + y);
}

int subtract(int x, int y) {
return(x - y);
}

int multiply(int x, int y) {
return(x * y);
}

int divide(int x, int y) {
if (y != 0)
return (x / y);
else
return 0;
}
```

Functions Using void Pointers

Void pointers are often used for function declarations.
For example: `void * square (const void *);`
Using a void * return type allows for any return type. Similarly, parameters that are void * accept any argument type. If you want to use the data passed in by the parameter without changing it, you declare it **const**.

You can leave out the parameter name to further insulate the declaration from its implementation. Declaring a function this way allows the definition to be customized as needed without having to change the declaration.

Consider the following program:

```
#include <stdio.h>

void* square (const void* num);

int main() {
int x, sq_int;
x = 6;
sq_int = square(&x);
printf("%d squared is %d\n", x, sq_int);

return 0;
}

void* square (const void *num) {
int result;
result = (*(int *)num) * (*(int *)num);
return result;
}
```

Function Pointers as Arguments

Another way to use a function pointer is to pass it as an argument to another function.
A function pointer used as an argument is sometimes referred to as a **callback function** because the receiving function "calls it back".
The **qsort()** function in the **stdlib.h** header file uses this technique.

Quicksort is a widely used algorithm for sorting an array. To implement the sort in your program, you need only include the **stdlib.h** file and then write a compare function that matches the declaration used in **qsort**:`void qsort(void *base, size_t num, size_t width, int (*compare)(const void *, const void *))`
To breakdown the **qsort** declaration:
void *base A void pointer to the array.
size_t num The number of elements in the array.
size_t width The size of an element.
int (*compare (const void *, const void *) A function pointer which has two arguments and returns 0 when the arguments have the same value, <0 when arg1 comes before arg2, and >0 when arg1 comes after arg2.

The actual implementation of the compare function is up to you. It doesn't even need to have the name "compare". You have the opportunity to designate a sort from high to low or low to high, or if an array contains structure elements, you can compare member values.

The following program sorts an array of **ints** from low to high using **qsort**:

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

int compare (const void *, const void *);

int main() {
int arr[5] = {52, 23, 56, 19, 4};
int num, width, i;

num = sizeof(arr)/sizeof(arr[0]);
width = sizeof(arr[0]);
qsort((void *)arr, num, width, compare);
for (i = 0; i < 5; i++)
printf("%d ", arr[ i ]);

return 0;
}

int compare (const void *elem1, const void *elem2) {
if ((*(int *)elem1) == (*(int *)elem2))
return 0;
else if ((*(int *)elem1) < (*(int *)elem2))
return -1;
else
return 1;
}
```

Declarations Using Structures

To **declare variables** of a structure data type, you use the keyword **struct** followed by the struct tag, and then the variable name.
For example, the statements below declares a structure data type and then uses the **student** struct to declare variables **s1** and **s2**:

```
struct student {
int age;
int grade;
char name[40];
};
```

```
/* declare two variables */
struct student s1;
struct student s2;
```

Accessing Structure Members

You access the members of a struct variable by using the **. (dot operator)** between the variable name and the member name.
For example, to **assign** a value to the **age** member of the **s1** struct variable, use a statement like:

```
s1.age = 19;
```

You can also assign one structure to another of the same type:

```
struct student s1 = {19, 9, "Jason"};
struct student s2;
//....
s2 = s1;
```

Using typedef

The **typedef** keyword creates a type definition that simplifies code and makes a program easier to read.
typedef is commonly used with structures because it eliminates the need to use the keyword **struct** when declaring variables.
For example:

```
typedef struct {
int id;
char title[40];
float hours;
} course;

course cs1;
course cs2;
```

Structures with Structures

The members of a structure may also be structures.
For example, consider the following statements:

```
typedef struct {
int x;
int y;
} point;
```

```
typedef struct {
float radius;
point center;
} circle;
```

Pointers to Structures

Just like pointers to variables, pointers to structures can also be defined.

struct myStruct *struct_ptr;
defines a pointer to the *myStruct* structure.

struct_ptr = &struct_var;
stores the address of the structure variable *struct_var* in the pointer *struct_ptr*.

struct_ptr -> struct_mem;
accesses the value of the structure member *struct_mem*.

For example:
```
struct student{
char name[50];
int number;
int age;
};

// Struct pointer as a function parameter
void showStudentData(struct student *st) {
printf("\nStudent:\n");
printf("Name: %s\n", st->name);
printf("Number: %d\n", st->number);
printf("Age: %d\n", st->age);
}

struct student st1 = {"Krishna", 5, 21};
showStudentData(&st1);
```

Structures as Function Parameters

A function can have structure parameters that accept arguments **by value** when a copy of the structure variable is all that is needed.
For a function to change the actual values in a struct variable, pointer **parameters** are required.

For example:

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

typedef struct {
int id;
char title[40];
float hours;
} course;

void update_course(course *class);
void display_course(course class);

int main() {
course cs2;
update_course(&cs2);
display_course(cs2);
return 0;
}

void update_course(course *class) {
strcpy(class->title, "C++ Fundamentals");
class->id = 111;
class->hours = 12.30;
}

void display_course(course class) {
printf("%d\t%s\t%3.2f\n", class.id, class.title, class.hours);
}
```

Array of Structures

An array can store elements of any data type, including structures.
After declaring an array of structures, an element is accessible with the index number.
The dot operator is then used to access members of the element, as in the program:

```
#include <stdio.h>

typedef struct {
int h;
int w;
int l;
} box;
```

```
int main() {
box boxes[3] = {{2, 6, 8}, {4, 6, 6}, {2, 6, 9}};
int k, volume;

for (k = 0; k < 3; k++) {
volume = boxes[k].h*boxes[k].w*boxes[k].l;
printf("box %d volume %d\n", k, volume);
}
return 0;
```

Unions

A **union** allows to store different data types in the same memory location.
It is like a structure because it has members. However, a union variable uses the same memory location for all its member's and only one member at a time can occupy the memory location.

A union **declaration** uses the keyword **union**, a **union tag**, and curly braces { } with a list of **members**.

Union members can be of any data type, including basic types, strings, arrays, pointers, and structures.
For example:

```
union val {
int int_num;
float fl_num;
char str[20];
};
```

Accessing Union Members

You access the members of a union variable by using the **. dot operator** between the variable name and the member name.
When assignment is performed, the union memory location will be used for that member until another member assignment is performed.

Trying to access a member that isn't occupying the memory location gives **unexpected results**.

The following program demonstrates accessing union members:

```
union val {
int int_num;
float fl_num;
```

```
char str[20];
};

union val test;

test.int_num = 123;
test.fl_num = 98.76;
strcpy(test.str, "hello");

printf("%d\n", test.int_num);
printf("%f\n", test.fl_num);
printf("%s\n", test.str);
```

Structures With Unions

Unions are often used within structures because a structure can have a member to keep track of which union member stores a value.
For example, in the following program, a vehicle struct uses either a vehicle identification number (VIN) or an assigned id, but not both:

```
typedef struct {
char make[20];
int model_year;
int id_type; /* 0 for id_num, 1 for VIN */
union {
int id_num;
char VIN[20];
} id;
} vehicle;

vehicle car1;
strcpy(car1.make, "Ford");
car1.model_year = 2017;
car1.id_type = 0;
car1.id.id_num = 123098;
```

Pointers to Unions

A pointer to a union points to the memory location allocated to the union.
A **union pointer** is declared by using the keyword **union** and the union tag along with * and the pointer name.
For example, consider the following statements:

```
union val {
int int_num;
```

```
float fl_num;
char str[20];
};

union val info;
union val *ptr = NULL;
ptr = &info;
ptr->int_num = 10;
printf("info.int_num is %d", info.int_num);
```

unions as Function Parameters

A function can have union parameters that accept arguments **by value** when a copy of the union variable is all that is needed.

For a function to change the actual value in a union memory location, pointer parameters are required.

For example:

```
union id {
int id_num;
char name[20];
};

void set_id(union id *item) {
item->id_num = 42;
}

void show_id(union id item) {
printf("ID is %d", item.id_num);
}
```

rray of Unions

An array can store elements of any data type, including **unions**.
With unions, it is important to keep in mind that only one member of the union can store data for each array element.

After declaring an array of unions, an element is **accessible with the index number**. The dot operator is then used to access members of the union, as in the program:

```
union val {
int int_num;
float fl_num;
char str[20];
```

```
};

union val nums[10];
int k;

for (k = 0; k < 10; k++) {
nums[k].int_num = k;
}

for (k = 0; k < 10; k++) {
printf("%d ", nums[k].int_num);
}
```
Try It Yourself

An array is a data structure that stores collection values that are all **the same type**. Arrays of unions allow storing values of **different types**.
For example:

```
union type {
int i_val;
float f_val;
char ch_val;
};
union type arr[3];
arr[0].i_val = 42;
arr[1].f_val = 3.14;
arr[2].ch_val = 'x';
```

Memory Management

Understanding memory is an important aspect of C programming. When you declare a variable using a basic data type, C automatically allocates space for the variable in an area of memory called the **stack**.

An int variable, for example, is typically allocated 4 bytes when declared. We know this by using the sizeof operator:

```
int x;
printf("%d", sizeof(x)); /* output: 4 */
```

Memory Management Functions

The **stdlib.h** library includes memory management functions.
The statement **#include <stdlib.h>** at the top of your program gives you access to the following:

malloc(*bytes*) Returns a pointer to a contiguous block of memory that is of size *bytes*.

calloc(*num_items, item_size*) Returns a pointer to a contiguous block of memory that has *num_items* items, each of size *item_size* bytes. Typically used for arrays, structures, and other derived data types. The allocated memory is initialized to 0.

realloc(*ptr, bytes*) Resizes the memory pointed to by ptr to size bytes. The newly allocated memory is not initialized.

free(*ptr*) Releases the block of memory pointed to by ptr.

The malloc Function

The **malloc()** function allocates a specified number of **contiguous bytes** in memory.

For example:

```
#include <stdlib.h>

int *ptr;
/* a block of 10 ints */
ptr = malloc(10 * sizeof(*ptr));

if (ptr != NULL) {
*(ptr + 2) = 50; /* assign 50 to third int */
}
```

The malloc Function

The allocated memory is **contiguous** and can be treated as an **array**. Instead of using brackets [ ] to refer to elements, pointer arithmetic is used to traverse the array. You are advised to use + to refer to array elements. Using ++ or += changes the address stored by the pointer.

If the allocation is unsuccessful, **NULL** is returned. Because of this, you should include code to check for a NULL pointer.

The free Function

The **free()** function is a memory management function that is called to **release memory**. By freeing memory, you make more available for use later in your program.

For example:

```
int* ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL)
```

```
*(ptr + 2) = 50; /* assign 50 to third int */
printf("%d\n", *(ptr + 2));

free(ptr);
```

The calloc Function

The **calloc()** function allocates memory based on the size of a specific item, such as a structure.
The program below uses **calloc** to allocate memory for a structure and **malloc** to allocate memory for the string within the structure:

```
typedef struct {
int num;
char *info;
} record;

record *recs;
int num_recs = 2;
int k;
char str[ ] = "This is information";

recs = calloc(num_recs, sizeof(record));
if (recs != NULL) {
for (k = 0; k < num_recs; k++) {
(recs+k)->num = k;
(recs+k)->info = malloc(sizeof(str));
strcpy((recs+k)->info, str);
}
}
```

Allocating Memory for Strings

When allocating memory for a string pointer, you may want to use string length rather than the sizeof operator for calculating bytes.
Consider the following program:

```
char str20[20];
char *str = NULL;

strcpy(str20, "12345");
str = malloc(strlen(str20) + 1);
strcpy(str, str20);
printf("%s", str);
```

Dynamic Arrays

Many algorithms implement a **dynamic** array because this allows the number of elements to grow as needed.
Because elements are not allocated all at once, dynamic arrays typically use a structure to keep track of current array size, current capacity, and a pointer to the elements, as in the following program.

```
typedef struct {
int *elements;
int size;
int cap;
} dyn_array;

dyn_array arr;

/* initialize array */
arr.size = 0;
arr.elements = calloc(1, sizeof(*arr.elements) );
arr.cap = 1; /* room for 1 element */
```

Accessing Files

An external file can be opened, read from, and written to in a C program. For these operations, C includes the **FILE** type for defining a file stream. The **file stream** keeps track of where reading and writing last occurred.

The **stdio.h** library includes file handling functions:
FILE Typedef for defining a file pointer.

fopen(filename, mode) Returns a FILE pointer to file *filename* which is opened using *mode*. If a file cannot be opened, NULL is returned.
Mode options are:
- **r** open for reading (file must exist)
- **w** open for writing (file need not exist)
- **a** open for append (file need not exist)
- **r+** open for reading and writing from beginning
- **w+** open for reading and writing, overwriting file
- **a+** open for reading and writing, appending to file

fclose(fp) Closes file opened with FILE fp, returning 0 if close was successful. **EOF** (end of file) is returned if there is an error in closing.

The following program opens a file for writing and then closes it:

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

```
int main() {
FILE *fptr;

fptr = fopen("myfile.txt", "w");
if (fptr == NULL) {
printf("Error opening file.");
return -1;
}
fclose(fptr);
return 0;
}
```

Reading from a File

The **stdio.h** library also includes functions for reading from an open file. A file can be read one character at a time or an entire string can be read into a character **buffer**, which is typically a char array used for temporary storage.

fgetc(fp) Returns the next character from the file pointed to by *fp*. If the end of the file has been reached, then **EOF** is returned.

fgets(buff, n, fp) Reads n-1 characters from the file pointed to by *fp* and stores the string in buff. A NULL character '\0' is appended as the last character in *buff*. If fgets encounters a newline character or the end of file before n-1 characters is reached, then only the characters up to that point are stored in buff.

fscanf(fp, conversion_specifiers, vars) Reads characters from the file pointed to by *fp* and assigns input to a list of variable pointers *vars* using *conversion_specifiers*. As with scanf, fscanf stops reading a string when a space or newline is encountered.

The following program demonstrates reading from a file:

```
#include <stdio.h>

int main() {
FILE *fptr;
int c, stock;
char buffer[200], item[10];
float price;

/* myfile.txt: Inventory\n100 Widget 0.29\nEnd of List */

fptr = fopen("myfile.txt", "r");
```

```
fgets(buffer, 20, fptr); /* read a line */
printf("%s\n", buffer);

fscanf(fptr, "%d%s%f", &stock, item, &price); /* read data */
printf("%d %s %4.2f\n", stock, item, price);

while ((c = getc(fptr)) != EOF) /* read the rest of the file */
printf("%c", c);

fclose(fptr);
return 0;
}
```

Writing to a File

The **stdio.h** library also includes functions for writing to a file. When writing to a file, newline characters '\n' must be explicitly added.

fputc(char, fp) Writes character *char* to the file pointed to by *fp*.

fputs(str, fp) Writes string *str* to the file pointed to by *fp*.

fprintf(fp, str, vars) Prints string *str* to the file pointed to by *fp*. *str* can optionally include format specifiers and a list of variables vars.

The following program demonstrates writing to a file:

```
FILE *fptr;
char filename[50];
printf("Enter the filename of the file to create: ");
gets(filename);
fptr = fopen(filename, "w");

/* write to file */
fprintf(fptr, "Inventory\n");
fprintf(fptr, "%d %s %f\n", 100, "Widget", 0.29);
fputs("End of List", fptr);
```

Binary File I/O

Writing only characters and strings to a file can become tedious when you have an array or structure. To write entire blocks of memory to a file, there are the following binary functions:

Binary file mode options for the fopen() function are:
- **rb** open for reading (file must exist)
- **wb** open for writing (file need not exist)
- **ab** open for append (file need not exist)
- **rb+** open for reading and writing from beginning
- **wb+** open for reading and writing, overwriting file
- **ab+** open for reading and writing, appending to file

fwrite(ptr, item_size, num_items, fp) Writes *num_items* items of *item_size* size from pointer *ptr* to the file pointed to by file pointer *fp*.

fread(ptr, item_size, num_items, fp) Reads *num_items* items of *item_size* size from the file pointed to by file pointer *fp* into memory pointed to by *ptr*.

fclose(fp) Closes file opened with file *fp*, returning 0 if close was successful. **EOF** is returned if there is an error in closing.

Binary File I/O

The following program demonstrates writing to and reading from binary files:

```
FILE *fptr;
int arr[10];
int x[10];
int k;

/* generate array of numbers */
for (k = 0; k < 10; k++)
arr[k] = k;

/* write array to file */
fptr = fopen("datafile.bin", "wb");
fwrite(arr, sizeof(arr[0]), sizeof(arr)/sizeof(arr[0]), fptr);
fclose(fptr);

/* read array from file */
fptr = fopen("datafile.bin", "rb");
fread(x, sizeof(arr[0]), sizeof(arr)/sizeof(arr[0]), fptr);
fclose(fptr);

/* print array */
for (k = 0; k < 10; k++)
printf("%d", x[k]);
```

Controlling the File Pointer

There are functions in stdio.h for controlling the location of the file pointer in a binary file:
ftell(fp) Returns a long int value corresponding to the *fp* file pointer position in number of bytes from the start of the file.

fseek(fp, num_bytes, from_pos) Moves the *fp* file pointer position by *num_bytes* bytes relative to position *from_pos*, which can be one of the following constants:
- **SEEK_SET** start of file
- **SEEK_CUR** current position
- **SEEK_END** end of file

The following program reads a record from a file of structures:

```
typedef struct {
int id;
char name[20];
} item;

int main() {
FILE *fptr;
item first, second, secondf;

/* create records */
first.id = 10276;
strcpy(first.name, "Widget");
second.id = 11786;
strcpy(second.name, "Gadget");

/* write records to a file */
fptr = fopen("info.dat", "wb");
fwrite(&first, 1, sizeof(first), fptr);
fwrite(&second, 1, sizeof(second), fptr);
fclose(fptr);

/* file contains 2 records of type item */
fptr = fopen("info.dat", "rb");

/* seek second record */
fseek(fptr, 1*sizeof(item), SEEK_SET);
fread(&secondf, 1, sizeof(item), fptr);
printf("%d %s\n", secondf.id, secondf.name);
fclose(fptr);
return 0;
}
```

EDOM and ERANGE Error Codes

Some of the mathematical functions in the **math.h** library set **errno** to the defined macro value **EDOM** when a domain is out of range.
Similarly, the **ERANGE** macro value is used when there is a range error.
For example:

```
float k = -5;
float num = 1000;
float result;

errno = 0;
result = sqrt(k);
if (errno == 0)
printf("%f ", result);
else if (errno == EDOM)
fprintf(stderr, "%s\n", strerror(errno));

errno = 0;
result = exp(num);
if (errno == 0)
printf("%f ", result);
else if (errno == ERANGE)
fprintf(stderr, "%s\n", strerror(errno));
```

The feof and ferror Functions

In addition to checking for a NULL file pointer and using errno, the **feof()** and **ferror()** functions can be used for determining file I/O errors:
feof(fp) Returns a nonzero value if the end of stream has been reached, 0 otherwise. feof also sets EOF.
ferror(fp) Returns a nonzero value if there is an error, 0 for no error.

The following program incorporates several exception handling techniques:

```
FILE *fptr;
int c;

errno = 0;

fptr = fopen("myfile.txt", "r");
if (fptr == NULL) {
fprintf(stderr, "Error opening file. %s\n", strerror(errno));
exit(EXIT_FAILURE);
```

```
}

while ((c = getc(fptr)) != EOF) /* read the rest of the file */
printf("%c", c);

if (ferror(fptr)) {
printf("I/O error reading file.");
exit(EXIT_FAILURE);
}
else if (feof(fptr)) {
printf("End of file reached.");
}
```

CERTIFICATE: