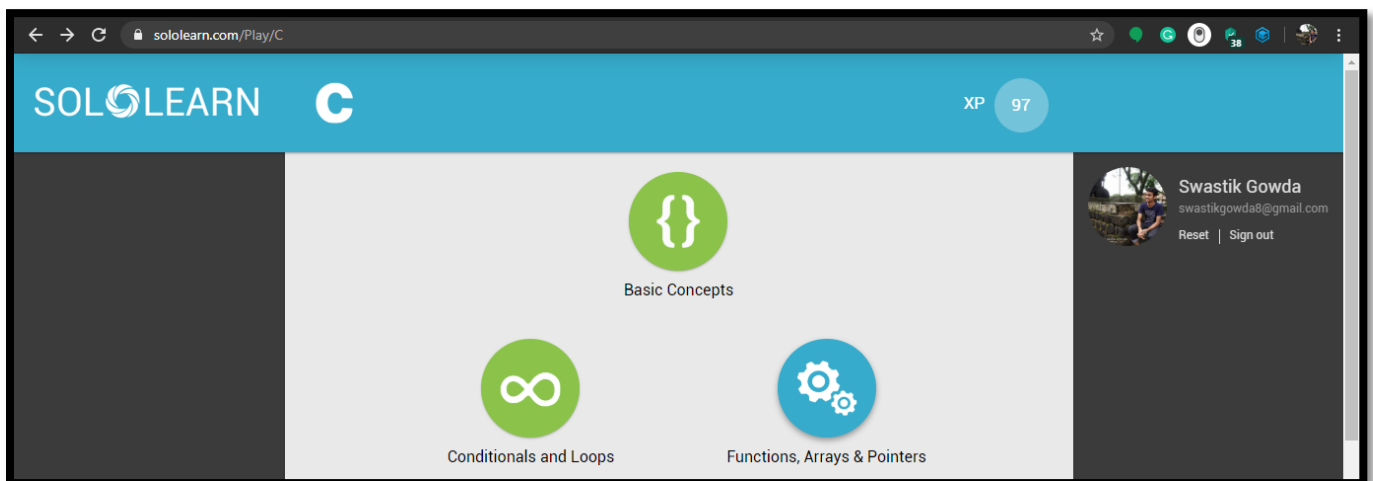
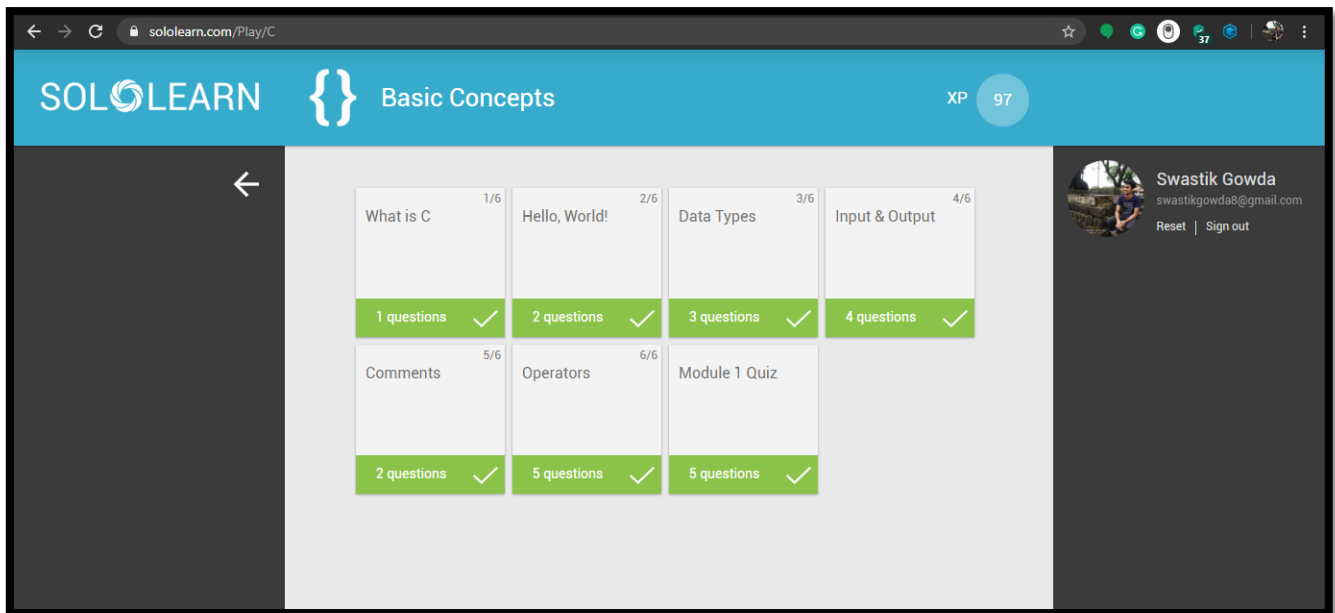


DAILY ASSESSMENT

Date:	18-June-2020	Name:	Swastik R Gowda
Course:	Solo-learn (C Programming)	USN:	4AL17EC091
Topic:	<ul style="list-style-type: none"> ❖ Basic Concept ❖ Conditionals & Loops ❖ Functions, Array & Pointers ❖ Strings & Function Pointers 	Semester & Section:	6 th Sem 'B' Sec
Github Repository:	swastik-gowda		

SESSION DETAILS

Image of session



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

Introduction :

- ❖ C is a general-purpose programming language that has been around for nearly 50 years.
- ❖ C has been used to write everything from operating systems (including Windows and many others) to complex programs like the Python interpreter, Git, Oracle database, and more.
- ❖ The versatility of C is by design. It is a low-level language that relates closely to the way machines work while still being easy to learn.

"Hello World!" program:

```
#include <stdio.h>
int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

- ❖ **#include <stdio.h>**: The function used for generating output is defined in ***stdio.h***. In order to use the ***printf*** function, we need to first include the required file, also called a header file.
- ❖ **Int main ()**: The ***main ()*** function is the entry point to a program. Curly brackets ***{ }*** indicate the beginning and end of a function (also called a code block). The statements inside the brackets determine what the function does when executed.
- ❖ **printf :** Here, we pass the text "Hello World!" to it. The ***\n*** escape sequence outputs a newline character. Escape sequences always begin with a backslash ******. The semicolon; indicates the end of the statement. Each statement must end with a semicolon.
- ❖ **return 0**: This statement terminates the ***main()*** function and returns the value 0 to the calling process. The number 0 generally means that our program has successfully executed. Any other number indicates that the program has failed.

Basic Data Types:

- ❖ **Int**: integer, a whole number.
- ❖ **Float**: floating point, a number with a fractional part.
- ❖ **Double**: double-precision floating point value.
- ❖ **Char**: single character.

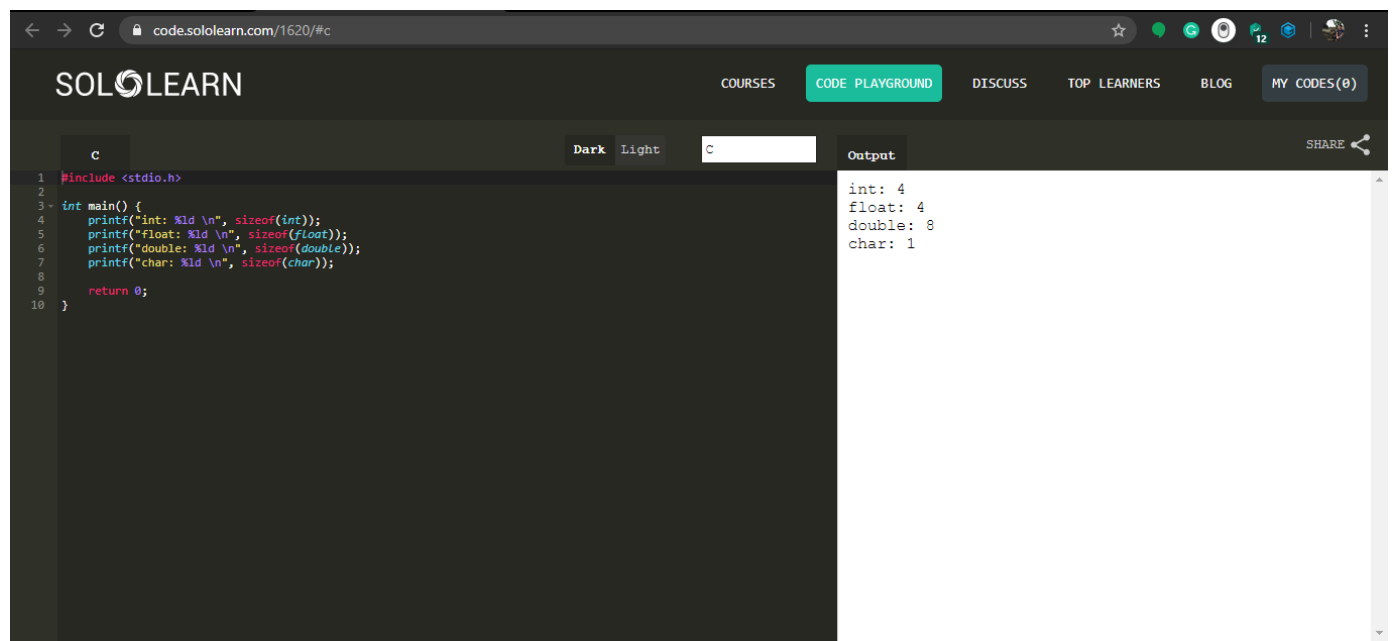
The amount of storage required for each of these types varies by platform.

C has a built-in ***sizeof*** operator that gives the memory requirements for a particular data type.

sizeof operator:

```
#include <stdio.h>
int main()
{
    printf("int: %ld \n", sizeof(int));
    printf("float: %ld \n", sizeof(float));
    printf("double: %ld \n", sizeof(double));
    printf("char: %ld \n", sizeof(char));

    return 0;
}
```



The screenshot shows the Sololearn Code Playground interface. The top navigation bar includes links for COURSES, CODE PLAYGROUND (highlighted), DISCUSS, TOP LEARNERS, BLOG, and MY CODES(0). The code editor is set to C language and Dark theme. The code being executed is the same as shown in the previous block. The output window on the right displays the results of the printf statements.

```
int: 4
float: 4
double: 8
char: 1
```

The program output displays the corresponding size in bytes for each data type. The **printf** statements in this program have two arguments. The first is the output string with a format specifier (**%ld**), while the next argument returns the sizeof value. In the final output, the **%ld** (for long decimal) is replaced by the value in the second argument.

Inputs and output:

Input:

C supports a number of ways for taking user input.

- ❖ **getchar ()**: *getchar ()* Returns the value of the next single character input.

For example:

```
#include <stdio.h>
int main()
{
    char a = getchar();
    printf("You entered: %c", a);
    return 0;
}
```

The input is stored in the variable a.

- ❖ **gets ()**: The *gets ()* is used to read input as an ordered sequence of characters, also called a string. A string is stored in a char array.

For example:

```
#include <stdio.h>
int main()
{
    char a[100];
    gets(a);
    printf("You entered: %s", a);
    return 0;
}
```

Here we stored the input in an array of 100 characters.

- ❖ **scanf()**: Scans input that matches format specifiers.

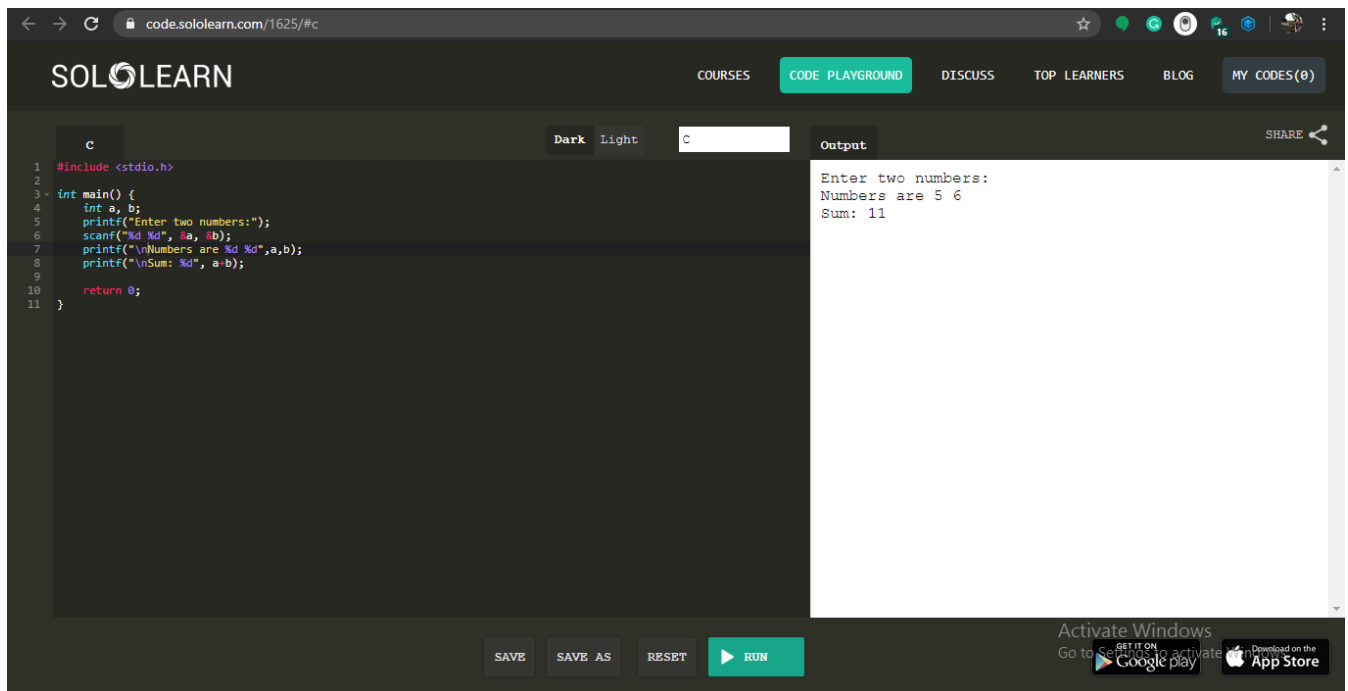
For example:

```
#include <stdio.h>
int main()
{
    int a;
    scanf("%d", &a);
    printf("You entered: %d", a);
    return 0;
}
```

The **&** sign before the variable name is the address operator. It gives the address, or location in memory, of a variable. This is needed because **scanf** places an input value at a variable address

As another example, let's prompt for two integer inputs and output their sum:

```
#include <stdio.h>
int main()
{
    int a, b;
    printf("Enter two numbers:");
    scanf("%d %d", &a, &b);
    printf("\nNumbers are %d %d", a,b);
    printf("\nSum: %d", a+b);
    return 0;
}
```



The screenshot shows the Sololearn Code Playground interface. The browser address bar displays 'code.sololearn.com/1625/#c'. The page header includes the Sololearn logo and navigation links: COURSES, CODE PLAYGROUND (highlighted), DISCUSS, TOP LEARNERS, BLOG, and MY CODES(0). The interface is split into two main sections: a code editor on the left and an output window on the right. The code editor is set to 'Dark' theme and contains the following C code:

```
1 #include <stdio.h>
2
3 int main() {
4     int a, b;
5     printf("Enter two numbers:");
6     scanf("%d %d", &a, &b);
7     printf("\nNumbers are %d %d", a,b);
8     printf("\nSum: %d", a+b);
9
10    return 0;
11 }
```

The output window on the right displays the program's execution results:

```
Enter two numbers:
Numbers are 5 6
Sum: 11
```

At the bottom of the interface, there are buttons for 'SAVE', 'SAVE AS', 'RESET', and 'RUN' (highlighted with a play icon). An 'Activate Windows' watermark is visible in the bottom right corner, along with links to 'Google play' and 'App Store'.

Output:

We have already used the ***printf ()*** function to generate output in the previous lessons. In this lesson, we cover several other functions that can be used for output.

❖ **putchar ()**: Outputs a single character.

For example:

```
#include <stdio.h>
int main()
{
    char a = getchar();
    printf("You entered: ");
    putchar(a);
    return 0;
}
```

The input is stored in the variable a.

❖ **puts ()**: The ***puts()*** function is used to display output as a string. A string is stored in a char array.

For example:

```
#include <stdio.h>

int main()
{
    char a[100];
    gets(a);
    printf("You entered: ");
    puts(a);
    return 0;
}
```

Here we stored the input in an array of 100 characters.

Operators:

Arithmetic Operators

- ❖ C supports arithmetic operators + (addition), - (subtraction), * (multiplication), / (division), and % (modulus division).
- ❖ Operators are often used to form a numeric expression such as $10 + 5$, which in this case contains two operands and the addition operator.
- ❖ Numeric expressions are often used in assignment statements.

Division

- ❖ C has two division operators: / and %.
- ❖ The division / operator performs differently depending on the data types of the operands. When both operands are int data types, integer division, also called truncated division, removes any remainder to result in an integer. When one or both operands are real numbers (float or double), the result is a real number.
- ❖ The % operator returns only the remainder of integer division. It is useful for many algorithms, including retrieving digits from a number. Modulus division cannot be performed on floats or doubles.
- ❖ The following example demonstrates division:

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

```
int main()
```

```
{
```

```
    int i1 = 10;
```

```
    int i2 = 3;
```

```
    int quotient, remainder;
```

```
    float f1 = 4.2;
```

```
    float f2 = 2.5;
```

```
    float result;
```

```
    quotient = i1 / i2;
```

```
    remainder = i1 % i2;
```

```
    result = f1 / f2;
```

```
    printf("%d \n", quotient);
```

```
    printf("%d \n", remainder);
```

```
    printf("%f \n", result);
```

```
    return 0;
```

```
}
```

Output

```
3
1
1.680000
```

If Statements

An **if** statement can include another **if** statement to form a nested statement. Nesting an **if** allows a decision to be based on further requirements.

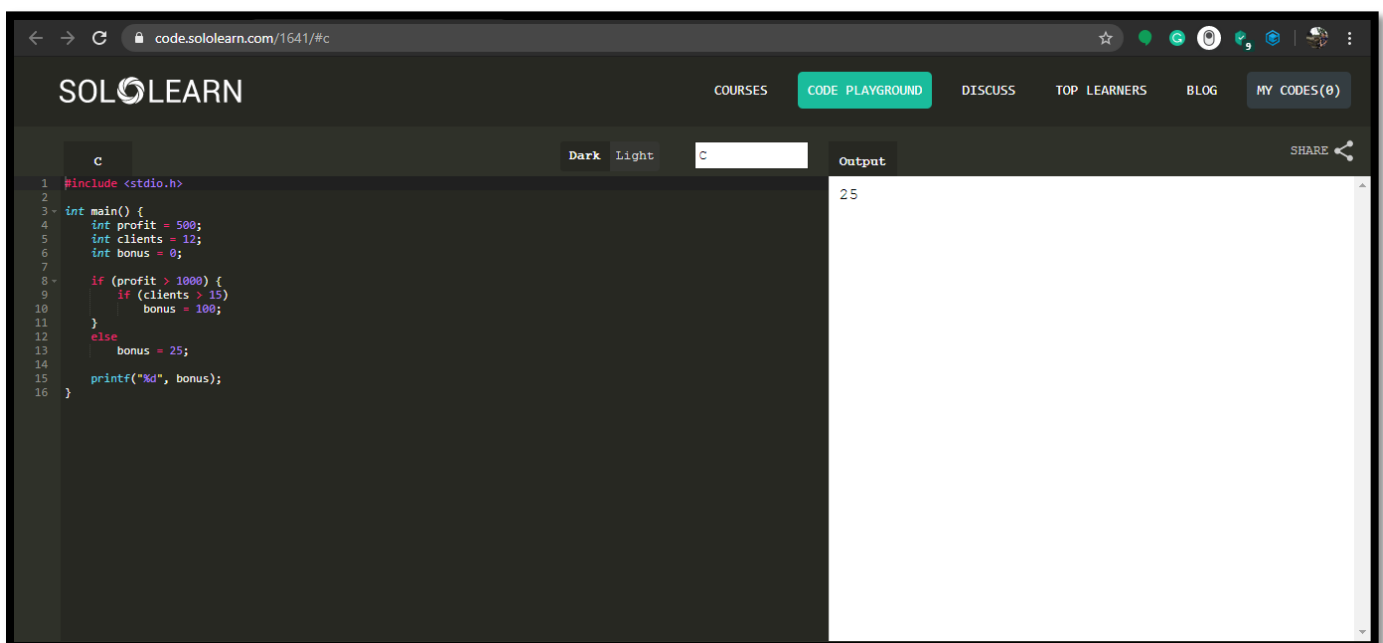
Consider the following statement:

```
if (profit > 1000)
    if (clients > 15)
        bonus = 100;
else
    bonus = 25;
```

Appropriately indenting nested statements will help clarify the meaning to a reader. However, be sure to understand that an **else** clause is associated with the closest **if** unless curly braces **{ }** are used to change the association.

For example:

```
if (profit > 1000)
{
    if (clients > 15)
        bonus = 100;
}
else
    bonus = 25;
```



The screenshot shows the Sololearn Code Playground interface. The browser address bar displays 'code.sololearn.com/1641/#c'. The page header includes the Sololearn logo and navigation links: COURSES, CODE PLAYGROUND (highlighted), DISCUSS, TOP LEARNERS, BLOG, and MY CODES(0). The code editor is set to 'c' language and 'Dark' theme. The code is as follows:

```
1 #include <stdio.h>
2
3 int main() {
4     int profit = 500;
5     int clients = 12;
6     int bonus = 0;
7
8     if (profit > 1000) {
9         if (clients > 15)
10            bonus = 100;
11     }
12     else
13         bonus = 25;
14     printf("%d", bonus);
15 }
16 }
```

The 'Output' panel on the right shows the result '25'. A 'SHARE' button with a share icon is located at the top right of the code editor.

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 */`

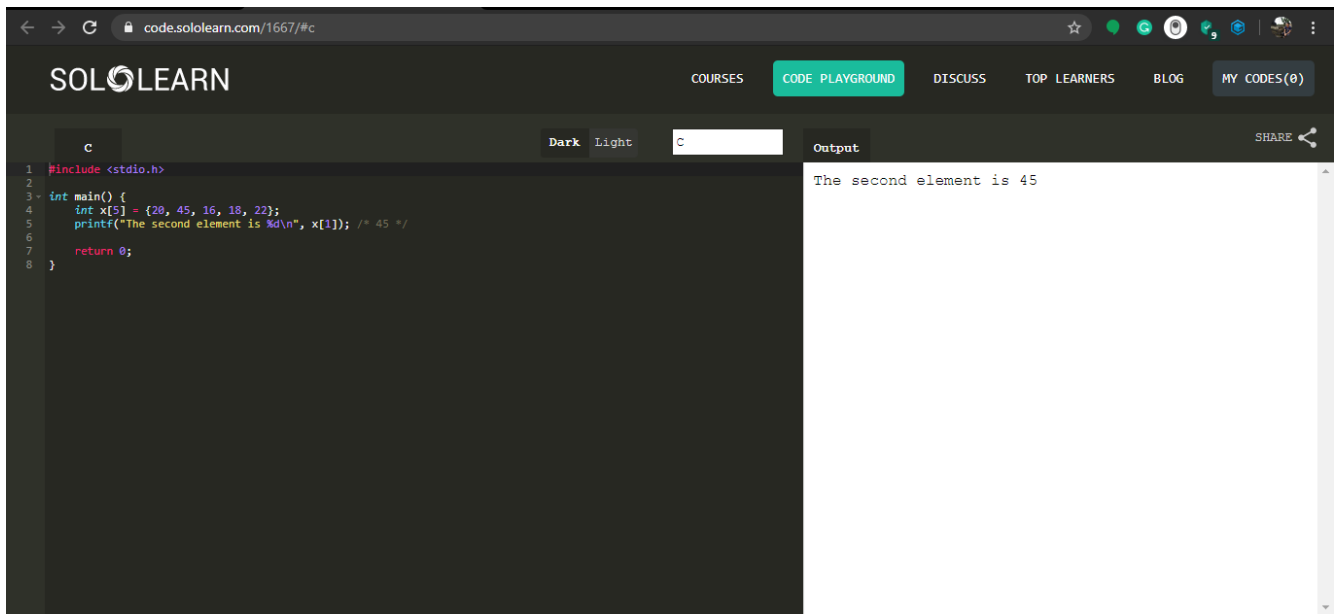
You can also initialize an array when it is declared, as in the following statement:

```
float prices[5] = {3.2, 6.55, 10.49, 1.25, 0.99};
```

An array can be partially initialized, as in:

```
float prices[5] = {3.2, 6.55};
```

Missing values are set to 0.



The screenshot shows a web-based code editor interface. At the top, there's a navigation bar with 'SOLOLEARN' and links for 'COURSES', 'CODE PLAYGROUND', 'DISCUSS', 'TOP LEARNERS', 'BLOG', and 'MY CODES(0)'. Below this, the editor is split into two panes. The left pane, titled 'c', contains the following C code:

```
1 #include <stdio.h>
2
3 int main() {
4     int x[5] = {20, 45, 16, 18, 22};
5     printf("The second element is %d\n", x[1]); /* 45 */
6
7     return 0;
8 }
```

The right pane, titled 'Output', displays the result of the program execution: 'The second element is 45'.

Pointers

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);
```

The address of j is ff3652cc
p contains address ff3652cc
The value of j is 63
p is pointing to the value 63

There are several things to notice about this program:

- Pointers should be initialized to ***NULL*** until they are assigned a valid location.
- Pointers can be assigned the address of a variable using the ***ampersand &*** sign.
- To see what a pointer is pointing to, use the * again, as in ****p***. In this case the * is called the ***indirection*** or ***dereference operator***. The process is called ***dereferencing***.

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 create 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':

To safely and conveniently operate with strings, you can use the *Standard Library* string functions shown below. Don't forget to include `<string.h>`.

strlen() - get length of a string

strcat() - merge two strings

strcpy() - copy one string to another

strlwr() - convert string to lower case

strupr() - convert string to upper case

strrev() - reverse string

strcmp() - compare two strings

strncat(str1, str2, n) - Appends (concatenates) first **n** characters of **str2** to the end of **str1** and returns pointer to **str1**.

strncpy(str1, str2, n) - Copies the first **n** characters of **str2** to **str1**.

strcmp(str1, str2) - Returns 0 when **str1** is equal to **str2**, less than 0 when **str1** < **str2**, and greater than 0 when **str1** > **str2**.

strncmp(str1, str2, n) - Returns 0 when the first **n** characters of **str1** is equal to the first **n** characters of **str2**, less than 0 when **str1** < **str2**, and greater than 0 when **str1** > **str2**.

strchr(str1, c) - Returns a pointer to the first occurrence of char **c** in **str1**, or NULL if character not found.

strrchr(str1, c) - Searches **str1** in reverse and returns a pointer to the position of char **c** in **str1**, or NULL if character not found.

strstr(str1, str2) - Returns a pointer to the first occurrence of **str2** in **str1**, or NULL if **str2** not found.

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");
}
```

A function name points to the start of executable code, just as an array name points to its first element. Therefore, although statements such as **funptr = &say_hello** and **(*funptr)(3)** are correct, it isn't necessary to include the address operator & and the indirection operator * in the function assignment and function call.



Basic Concepts



Conditionals and Loops



Functions, Arrays & Pointers



Strings & Function Pointers



Swastik R Gowda
swastikgowda8@gmail.com

[Reset](#) | [Sign out](#)

Activate Windows