|  |  |  |  |
| --- | --- | --- | --- |
| Date: | **18-06-2020** | Name: | **Varun G Shetty** |
| Course: | **C programming** | USN: | **4AL17EC093** |
| Topic: | * Basic concepts * Conditionals | Semester & Section: | **6th & ‘B’** |
| GitHub Repository: | **Varunshetty4** |  |  |

**BASIC CONCEPTS:**

**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.

**#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.

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.

**Data Types**: C supports the following 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.
* C has a built-in **sizeof** operator that gives the memory requirements for a particular 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.

**Variable**: A variable is a name for an area in memory. The name of a variable (also called the identifier) must begin with either a letter or an underscore and can be composed of letters, digits, and the underscore character. Variables must also be declared as a data type before they are used. The value for a declared variable is changed with an **assignment statement**.

|  |  |
| --- | --- |
| int my\_var;  my\_var = 42; | Can be written as:  int my\_var = 42; |

You can declare multiple variables on a single line by separating them with a **comma**. Also, notice the use of format specifiers for float (%f) and char (%c) output. The C programming language is **case-sensitive**, so **my\_Variable** and **my\_variable** are two different identifiers.

**Constants**: A **constant**stores a value that cannot be changed from its initial assignment. To distinguish constants from variables, a common practice is to use uppercase identifiers. One way to define a constant is by using the **const**keyword in a variable declaration.

**const** double PI = 3.14;

Constants must be initialized with a value when declared. Another way to define a constant is with the **#define** pre-processor directive. The **#define** directive uses macros for defining constant values.

**#define PI 3.14**

Before compilation, the preprocessor replaces every macro identifier in the code with its corresponding value from the directive. In this case, every occurrence of **PI**is replaced with **3.14**. The final code sent to the compiler will already have the constant values in place. The difference between **const**and **#define** is that the former uses memory for storage and the latter does not. **Do NOT put a semicolon character at the end of #define statements**.

**Input:** C supports a number of ways for taking user input. **getchar()** Returns the value of the next **single character** input. The **gets()** function is used to read input as an **ordered sequence of characters,** also called a **string.**

Syntax: **char a = getchar();** **gets**(a);

**scanf()** scans input that matches format specifiers. Syntax: **scanf**("%d", &a); 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. **scanf()** stops reading as soon as it encounters a space, so text such as "Hello World" is two separate inputs for **scanf()**.

**Output**: We have already seen the printf(). **putchar()** Outputs a single character. The **puts()** function is used to display output as a **string**.

Syntax: **putchar**(a); **puts**(a);

**Formatted Input:** The scanf() function is used to assign input to variables. A call to this function scans input according to format specifiers that convert input as necessary.  
If input can't be converted, then the assignment isn't made.  
The scanf() statement waits for input and then makes assignments. The **&** must be used to access the variable addresses. **The & isn't needed for a string because a string name acts as a pointer.**

**Format specifiers** begin with a percent sign **%** and are used to assign values to corresponding arguments after the control string. **Blanks, tabs, and newlines are ignored**. A format specifier can include several options along with a conversion character.

Syntax: **%[\*][max\_field]conversion character**

The optional **\*** **will skip the input field**. The optional **max\_width** gives the **maximum number of characters to assign to an input field**. The **conversion character converts the argument**, if necessary, to the indicatedtype:

**d-** decimal  
**c-** character  
**s-** string  
**f-** float

**e**- scientific notation

**x-** hexadecimal

**Formatting Output**: The **printf**function was introduced in your very first Hello World program. A call to this function requires a **format string** which can include escape sequences for outputting special characters and format specifiers that are replaced by values.

Escape sequences begin with a **backslash \**:

**\n-** new line

**\t-** horizontal tab

**\\-** backslash  
**\b-** backspace  
**\'-** single quote

**\"-** double quote

Format specifiers begin with a**percent sign %** and are replaced by corresponding arguments after the format string. A format specifier can include several options along with a conversion character:

Syntax: %[-][width].[precision]conversion character

The **optional -** specifies **left alignment** of the data in the string. The optional **width**gives the **minimum number of characters** for the data. The **period .** separates the **width**from the **precision**. The optional **precision**gives **the number of decimal places for numeric data**. If **s** is used as the conversion character, then precision determines **the number of characters to print**. The **conversion character** **converts the argument.** **To print the % symbol, use %% in the format string.**

**OPERATORS:**

**Arithmetic Operators:** C supports arithmetic operators + (addition), - (subtraction), \* (multiplication), / (division), and % (modulus division). C has **two division operators: / and %.** 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. **Modulus division** cannot be performed on floats or doubles.

Operator Precedence: C evaluates a numeric expression based on **operator precedence**. The **+ and – are equal in precedence, as are \*, /, and %.** The **\*, /, and % are performed first in order from left to right and then + and -, also in order from left to right**. You can change the order of operations by using parentheses ( ) to indicate which operations are to be performed first.

**Type Conversion:** When a numeric expression contains operands of different data types, they are automatically converted as necessary in a process called **type conversion**. When you want to force the result of an expression to a different type you can perform explicit type conversion by **type casting**, as in the statements:

|  |  |
| --- | --- |
| float average;  int total = 23;  int count = 4;  average = **(float)** total / count; | /\* average = 5.75 \*/ |

Without the type casting, average will be assigned 5. **Explicit type conversion**, even when the compiler may do automatic type conversion, is considered good programming style.

**Assignment Operators**: almost same as used in python. The short methods also works the same.

**Increment and Decrement:** Adding 1 to a variable can be done with the increment operator ++. Similarly, the decrement operator -- is used to subtract 1 from a variable. The **prefix(++i)**form increments/decrements the variable and then uses it in the assignment statement. The **postfix(i++)**form uses the value of the variable first, before incrementing/decrementing it.

**CONDITIONALS:**

**Relational Operators:** There are six relational operators that can be used to form a Boolean expression, which returns true or false:

< less than

<= less than or equal to

> greater than

>= greater than or equal to

== equal to

!= not equal to

**The if statement:** Executes the statement when expression is true**. if is also known as a decision structure.**

|  |
| --- |
| If (expression)  Statements; |

An expression that evaluates to a non-zero value is considered true.

**The if-else statement:** The if statement can include an optional else clause that executes statements when an expression is false.

|  |
| --- |
| **if** (expession)  Statement1;  **else** Statement2; |

Another way to form an if-else statement is by using the **?:** operator in a **conditional expression**. The **?:** operator can have only one statement associated with the**if** and the **else**.

**y = (x >= 5) ? 5 : x; // y=expression?statement1:statement2**

**Nested 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.

|  |  |
| --- | --- |
| if (profit > 1000)  if (clients > 15)  bonus = 100;  else  bonus = 25; | 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.

**if else-if statement:** When a decision among three or more actions is needed, the if-else if statement can be used. There can be multiple else if clauses and the last else clause is optional.

|  |
| --- |
| **if** (score >= 90)  printf("%s", "Top 10%\n");  **else if** (score >= 80)  printf("%s", "Top 20%\n");  **else if** (score > 75)  printf("%s", "You passed.\n");  **else** printf("%s", "You did not pass.\n"); |

**switch statement:** The **switch**statement branches program control by matching the result of an expression with a constant **case** value. The **switch**statement often provides a more elegant solution to **if-else if** and **nested if**statements.

|  |
| --- |
| **switch** (expression){  **case** val1:  statements break; **case** val2:  statements break; **default**: statements } |

There can be multiple **cases**with unique labels. The optional **default**case is executed when no other matches are made. A **break**statement is needed in each case to branch to the end of the **switch**statement. Without the **break**statement, program execution falls through to the next **case**statement. This can be useful when the same statement is needed for several cases. Consider the following **switch**statement:

|  |
| --- |
| switch (num) {  case 1:  case 2:  case 3:  printf("One, Two, or Three.\n");  break;  case 4:  case 5:  case 6:  printf("Four, Five, or Six.\n");  break; default: printf("Greater than Six.\n");  } |