**Keywords?**

keywords are words that have predefined uses and cannot be used for any other purpose in a C program.

Int, float, char, if, else....

Not allowed because computer will not know if its a keyword or a normal statement.

**Variables** -> reuse, single point change.

**Compound Assignments**.. += , -= etc..

**Comments:**

Part of every programming language.

Helps to identify program purpose and explain the code.

Valuable to us and others reading our code.

You insert comments to document programs and improve program readability.

they’re simply ignored.

**MultiLine Comment**.

/\* beginning of comment block \*/ end of a comment block

Anything written between these blocks will be ignored by the compiler when reading the program.

If we leave out one of the comment start or closing characters, the program will not compile.

If we write the characters in incorrect order. It will also not compile.

**Single Line Comment**

**//**  anything written after this symbol are ignored by the compiler for that line only.

# Printf

instructs the computer to perform an action, namely to display on the screen the string of characters enclosed in the quotation marks.

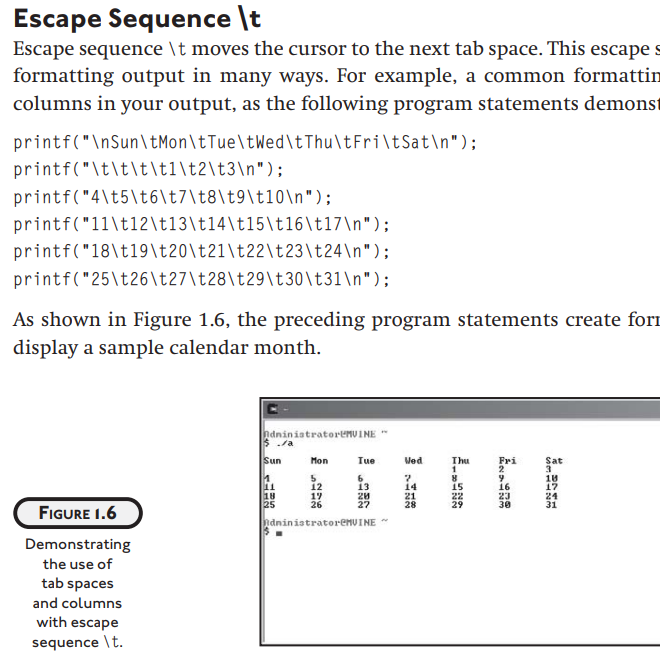
Can print values directly.

Can print variables.

Can print multiple variable/values in a single statement.

Can do calculations directly.

%.2f



**Scanf**

C, the scanf function is used to take input and store it in a variable. To do this, scanf needs the **address** of the variable, not just its value, because it needs to know *where* in memory to store the input.

**Why &variable?**

* **Memory Address**: The & (address-of) operator provides the memory address of the variable.
* **Storing Input**: scanf takes that address and places the user input directly into that memory location, updating the variable with the new value.

Make memory diagram and show a -> value and to get its address we add &before a.

Can take multiple inputs at a time.

**Getch**

reads single character input from user but doesn’t wait for us to press enter key, as soon as we press a key, it displays the output.

Doesnt show the pressed key on screen, Hide inputs like password, ATM pins. Etc..

***// Example for getch() in C***

***#include <stdio.h>***

***// Library where getch() is stored***

***#include <conio.h>***

***int main()***

***{***

***printf("%c", getch());***

***return 0;***

***}***

You can store the character input directly in a variable.

**char ch = getch();**

**Flow Of Control**

statements in a program execute one after the other in the order in which

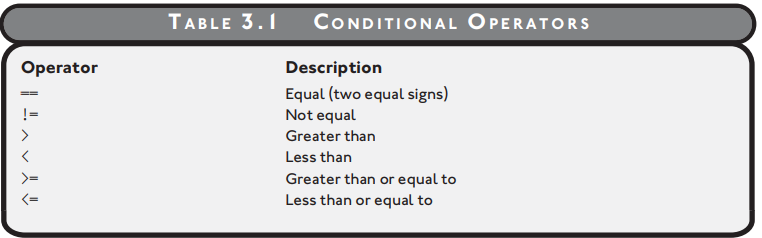
you write them. This is called sequential execution. As you’ll soon see, various C

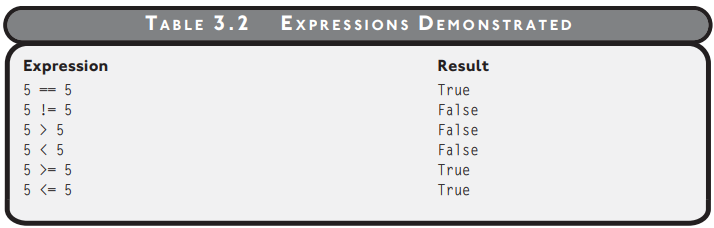
statements enable you to specify that the next statement to execute may be other than

the next one in sequence. This is called transfer of control.

# An algorithm is a step-by-step process for solving a problem

To write an algorithm we use PseudoCode and Flowchart





Take the following problem statement, for example.

Turn the air conditioning on when the temperature is greater than or equal to 80 degrees or else turn it off.

if temperature >= 80

Turn AC on

else

Turn AC off

end if

This is pseudocode not actual code, if u paste this in a C Program, it wont compile.

Identify the Inputs, processing, outputs in the pseudocode.

Allow a customer to deposit or withdraw money from a bank account, and if a user elects to withdraw funds, ensure that sufficient money exist

PseduoCode

if action == deposit

Deposit funds into account

else

if balance < withdraw amount

Insufficient funds for transaction

else

Withdraw monies

end if

end if

The first point of interest in the preceding pseudo code is that I have a nested condition inside a parent condition. This nested condition is said to belong to its parent condition, such that the nested condition will never be evaluated unless one of the parent conditional requirements is met. In this case, the action must not equal the deposit for the nested condition to be evaluated.

Also notice that for each algorithm implemented with pseudo code, I use a standard form of

indentation to improve the readability.

Take a look at the same pseudo code; this time without the use of indentation.

if action == deposit

Deposit funds into account

else

if balance < withdraw amount

Insufficient funds for transaction

else

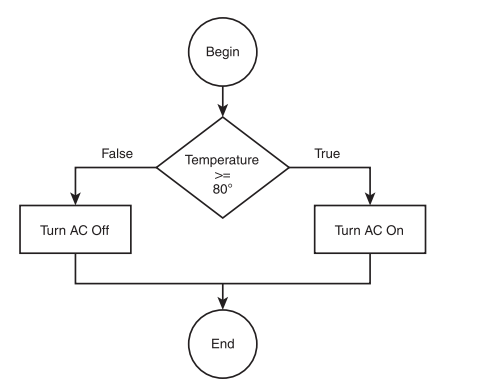
Withdraw monies

end if

end if

**Flowchart**

Turn the air conditioning on when the temperature is greater than or equal to 80 degrees or else turn it off.

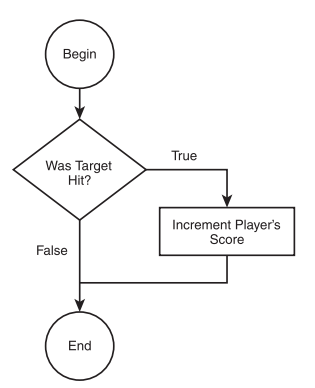
****

**Take a look at the following algorithm implemented in pseudo code.**

# if target hit == true

# Increase player’s score

# end if



if action == deposit

Deposit funds into account

else

if balance < withdraw amount

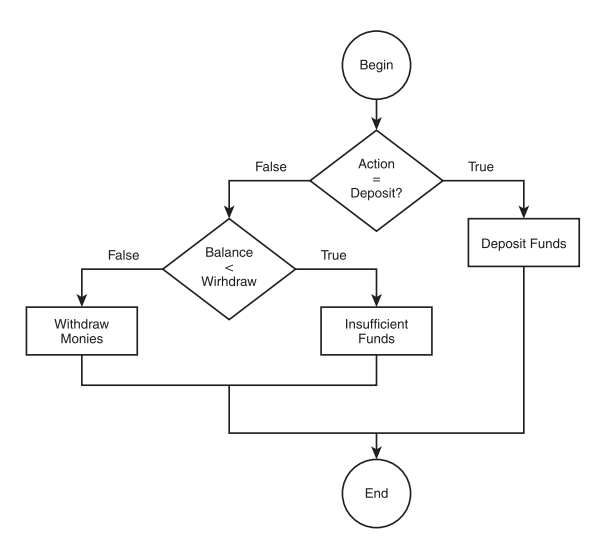
insufficient funds for transaction

else

Withdraw monies

end if

end if



# If

**If Statement: A Decision-Making Tool**

An **if statement** is a statement that allows you to execute a block of code only if a certain condition is true. It's like a decision-making process: if a specific condition is met, you take one action; otherwise, you take another.

if (condition)

{ // Code to be executed if the condition is true }

 **Condition:** This is an expression that evaluates to either true or false.

 **Code Block:** This is a block of code that will be executed if the condition is true.

if (iTemperature >= 80) //Turn AC on else //Turn AC off

The first statement is the condition, which checks for a true or false result in the expression (iTemperature >= 80). The expression must be enclosed in parentheses. If the expression’s result is true, the Turn AC on code is executed; if the expression’s result is false, the else part of the condition is executed. Also note that there is no end if statement in C.

If you process more than one statement inside your conditions, you must enclose the multiple

statements in braces, as shown next.

if (Temperature >= 80) {

//Turn AC on

printf("\nThe AC is on\n");

}

else {

//Turn AC off

printf("\nThe AC is off\n");

}

Code for AC ON/OFF IN C

#include <stdio.h>

main()

{

int iResponse = 0;

printf("\n\tAC Control Unit\n");

printf("\n1\tTurn the AC on\n");

printf("2\tTurn the AC off\n");

printf("\nEnter your selection: ");

scanf("%d", &iResponse);

if (iResponse == 1)

printf("\nAC is now on\n");

if (iResponse == 2)

printf("\nAC is now off\n");

}

Show code execution using debugger to show how lines are skipped.

Notice in my if structure that I’m comparing an integer variable to a number. This is acceptable—you can use variables in your if structures as long as you are comparing apples to apples and oranges to oranges. In other words, you can use a combination of variables and other data in your expressions as long as you’re comparing numbers to numbers and characters to characters.

Do the previous program using Character now!

EvenAndPositiveNumber.c

Convert the Temperate program into If Else.

# Nested If

if action == deposit

Deposit funds into account

else

if balance < withdraw amount

Insufficient funds for transaction

else

Withdraw monies

end if

end if

**C Algorithm**

if (action == deposit) {

//deposit funds into account

printf("\nFunds deposited\n");

}

else {

if (balance < withdraw)

//insufficient funds

else

//withdraw monies

}

C Program:

#include <stdio.h>

main()

{

int iSelection = 0;

float fTransAmount = 0.0;

float fBalance = 100.25;

printf("\n\tATM\n");

printf("\n1\tDeposit Funds\n");

printf("2\tWithdraw Funds\n");

printf("\nEnter your selection: ");

scanf("%d", &iSelection);

if (iSelection == 1) {

printf("\nEnter fund amount to deposit: ");

scanf("%f", &fTransAmount);

printf("\nYour new balance is: $%.2f\n", fBalance + fTransAmount);

} //end if

if (iSelection == 2) {

printf("\nEnter fund amount to withdraw: ");

scanf("%f", &fTransAmount);

if (fTransAmount > fBalance)

printf("\nInsufficient funds\n");

else

printf("\nYour new balance is $%.2f\n", fBalance - fTransAmount);

} //end if

} //end main function

# If Else

AgeChecker.c File

**Mini Project:**

**If Press X then exit program.**

getch(); // Pauses before exit return 0;

# Boolean Algebra

Boolean algebra is named after George Boole, a mathematician in the nineteenth century. Boole developed his own branch of logic containing the values true and false and the operators and, or, and not to manipulate the values. Even though Boole’s work was before the advent of computers, his research has become the foundation of today’s modern digital circuitry in computer architecture.

Bool datatype.

**And**

The and operator is used to build compound conditions. Each side of the condition must be

true for the entire condition to be true. Take the following expression, for example.

3 == 3 and 4 == 4

This compound condition contains two separate expressions or conditions, one on each side

of the and operator. The first condition evaluates to true and so does the second condition,

which generates a true result for the entire expression.

Here’s another compound condition that evaluates to false.

3==4 and 4==4

This compound condition evaluates to false because one side of the and operator does not

evaluate to true. Study Table 3.3 to get a better picture of possible outcomes with the and

operator.

Truth tables allow you to see all possible scenarios in an expression containing compound

conditions. The truth table in Table 3.3 shows two possible input values (x and y) for the and

operator. As you can see, there is only one possible combination for the and operator to generate a true result: when both sides of the condition are true.

**OR**

or Operator

The or operator is similar to the and operator in that it contains at least two separate expressions and is used to build a compound condition. The or operator, however, differs in that it

only requires one side of the compound condition to be true for the entire expression to be

true. Take the following compound condition, for example.

4 == 3 or 4 == 4

In the compound condition above, one side evaluates to false and the other to true, providing

a true result for the entire expression. To demonstrate all possible scenarios for the or operator, study the truth table in Table 3.4.

**NOT**

The last Boolean operator I discuss in this chapter is the not operator. The not operator is easily

understood at first, but can certainly be a bit confusing when programmed in compound

conditions.

T ABLE 3.3 T RUTH TABLE FOR THE AND OPERATOR

xy Result

true true true

true false false

false true false

false false false

Chapter 3 • Conditions 63

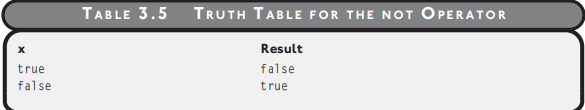
Essentially, the not operator generates the opposite value of whatever the current result is.

For example, the following expression uses the not operator in a compound condition.

not( 4 == 4 )

The inside expression, 4 == 4, evaluates to true, but the not operator forces the entire expression to result in false. In other words, the opposite of true is false.

**Take a look at Table 3.5 to evaluate the not operator further.**

****

**Notice that the not operator contains only one input variable (x) to build a compound**

**Order of operations**

To dictate order of operations, use parentheses to build clarification into your compound

conditions. For example, given x = 1, y = 2, and z = 3, study the following compound

condition.

z < y or z <= z and x < z

Without using parentheses to dictate order of operations, you must assume that the order of

operations for the compound condition flows from left to right. To see how this works, I’ve

broken down the problem in the following example:

problem-solving skills with Boolean algebra. Before you take that plunge, however, I must

1. First, the expression z < y or z <= z is executed, which results in false or true, and

results in the overall result of true.

2. Next, the expression true and x < z is executed, which results in true and true, and

results in the overall value of true.

But when I change the order of operations using parentheses, I get a different overall result

as shown next.

z < y or (z < x and x < z)

1. First, (z < x and x < z) is evaluated, which results in false and true, and results in the

overall value of false.

2. Next, the expression z < y or false is evaluated, which results in false or false, and

results in the overall value of false.

You should now see the consequence of using or not using parentheses to guide the order of

operations.

**Try to solve the following Boolean algebra problems, given**

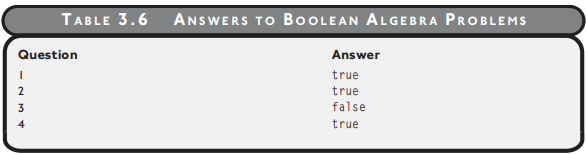
**x == 5, y == 3, and z == 4**

**1. x > 3 and z == 4**

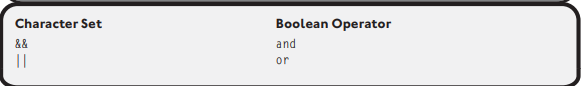
**2. y >= 3 or z > 4**

**3. NOT(x == 4 or y < z)**

**4. (z == 5 or x > 3) and (y == z or x < 10)**

****

**Compound If structure.**

****

**&& Operator**

The && operator implements the Boolean operator and; it uses two ampersands to evaluate a

Boolean expression from left to right. Both sides of the operator must evaluate to true before

the entire expression becomes true.

The following two code blocks demonstrate C’s && operator in use. The first block of code uses

the and operator (&&) in a compound if condition, which results in a true expression.

if ( 3 > 1 && 5 < 10 )

printf("The entire expression is true\n");

The next compound if condition results in false.

if ( 3 > 5 && 5 < 5 )

printf("The entire expression is false\n");

**|| Operator**

The || character set (or Boolean operator) uses two pipe characters to form a compound condition, which is also evaluated from left to right. If either side of the condition is true, the

whole expression results in true.

The following code block demonstrates a compound if condition using the || operator, which

results in a true expression.

if ( 3 > 5 || 5 <= 5 )

printf("The entire expression is true\n");

The next compound condition evaluates to false because neither side of the || operator evaluates to true.

if ( 3 > 5 || 6 < 5 )

printf("The entire expression is false\n");

**Invalid :**

if ( cResponse == 'A' || 'a' )

if ( cResponse == 'A' || == 'a' )

if ( cResponse || cResponse )

**Correct :**

if ( cResponse == 'A' || cResponse == 'a' )

**Checking for a Range of Values**

**Checking for a range of values is a common programming practice for input validation. You**

**can use compound conditions and relational operators to check for value ranges, as shown**

**in the following program:**

#include <stdio.h>

main()

{

int iResponse = 0;

printf("Enter a number from 1 to 10: ");

scanf("%d", &iResponse);

if ( iResponse < 1 || iResponse > 10 )

printf("\nNumber not in range\n");

else

printf("\nThank you\n");

}

# ! Not

#include <stdio.h>

#include <stdbool.h> // Allows us to use 'bool', 'true', and 'false'

int main() {

bool lightOn = true; // true means the light is on

if (lightOn) { // since lightOn is true, this condition is "true"

printf("The light is on.\n");

}

lightOn = false; // change lightOn to false

if (lightOn) { // since lightOn is now false, this condition is "false"

printf("This message won’t be printed.\n");

} else {

printf("The light is off.\n");

}

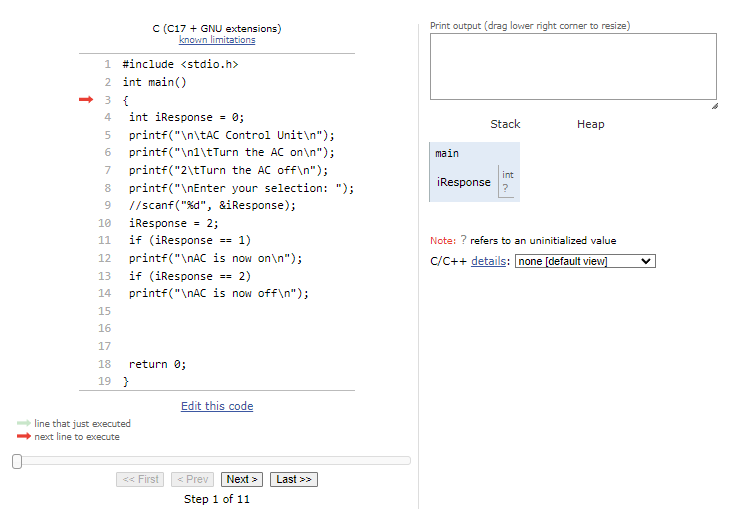
return 0;

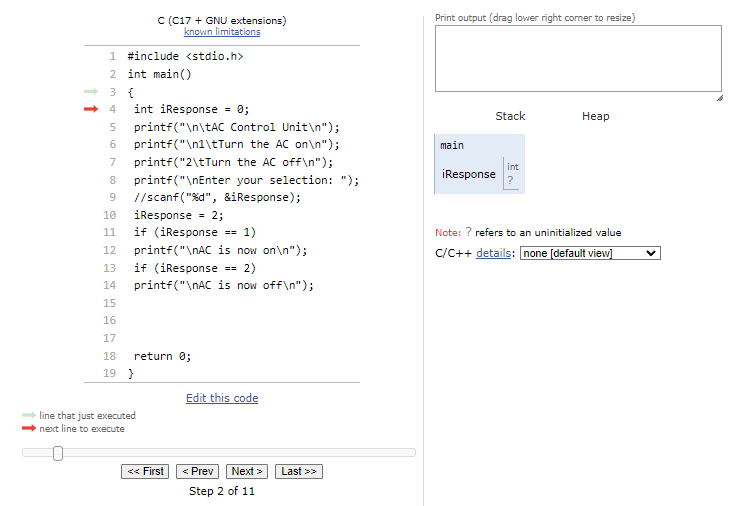
}

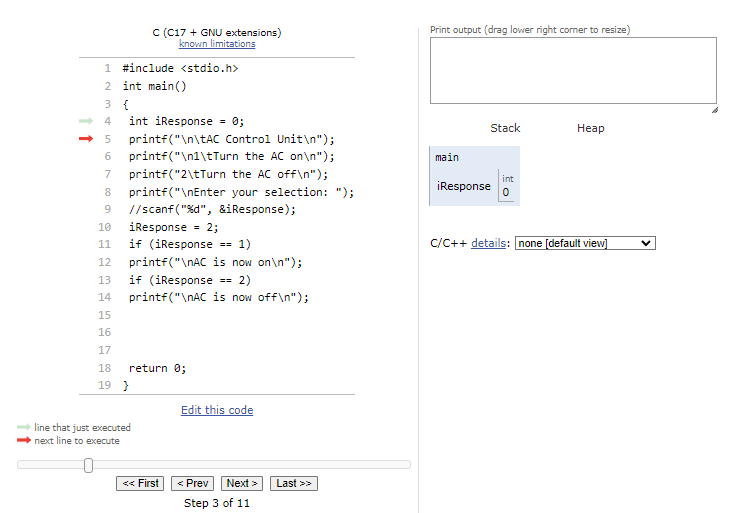
Common Errors:

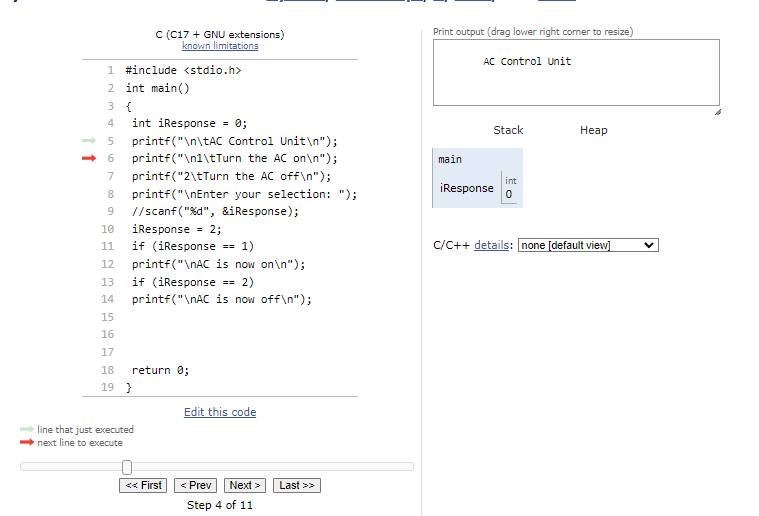
<https://womengovtcollegevisakha.ac.in/departments/C%20Programming%20for%20the%20Absolute%20Beginner,%20Second%20Edition%20(%20PDFDrive%20).pdf>  
page 37

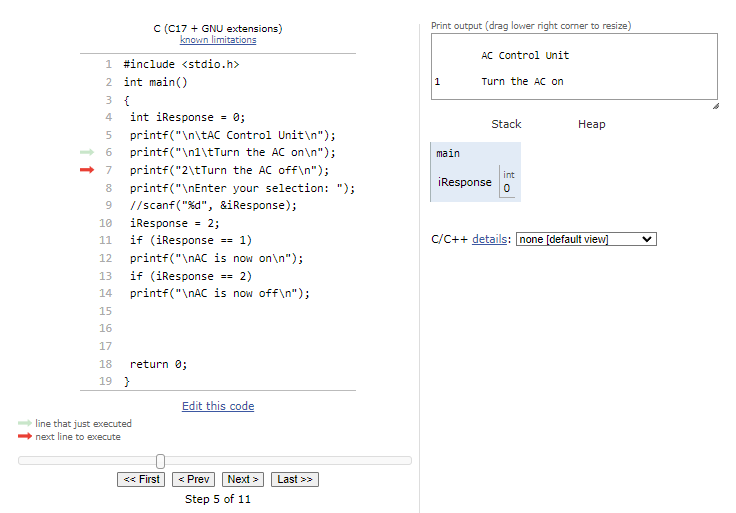
If execution

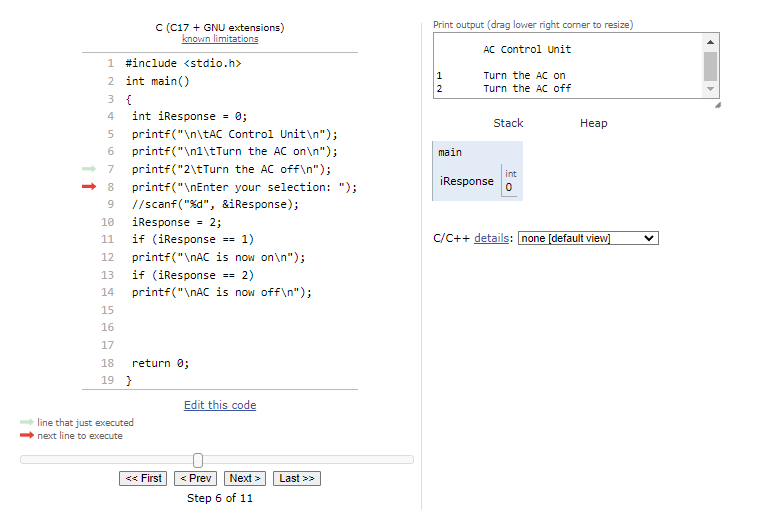


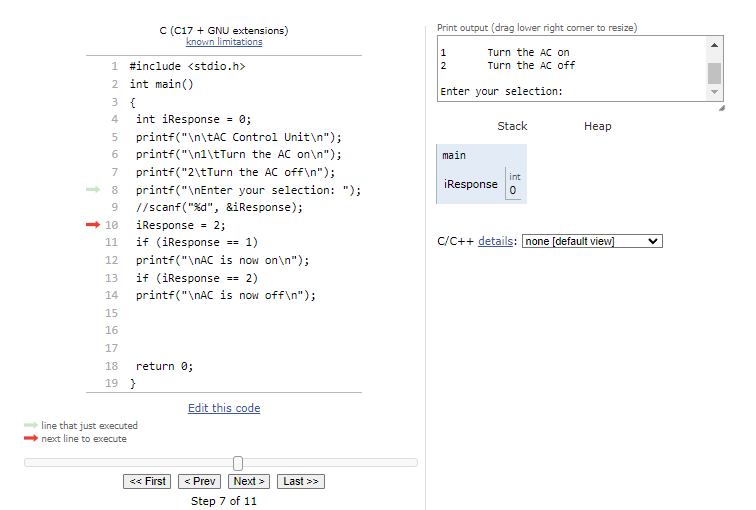


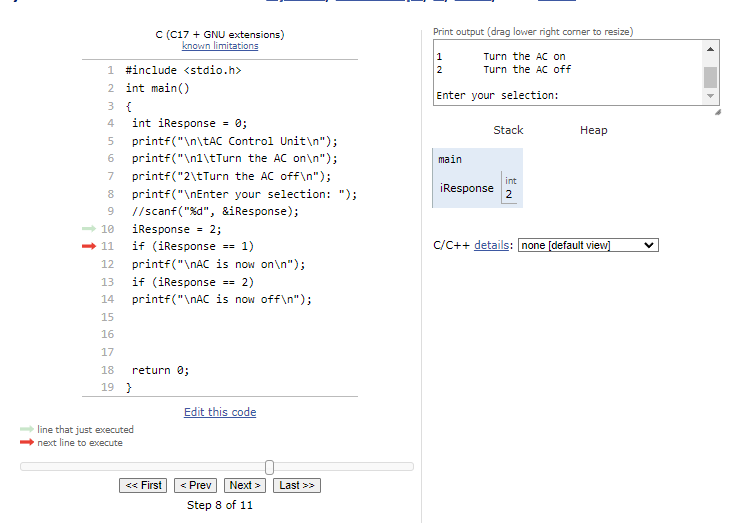


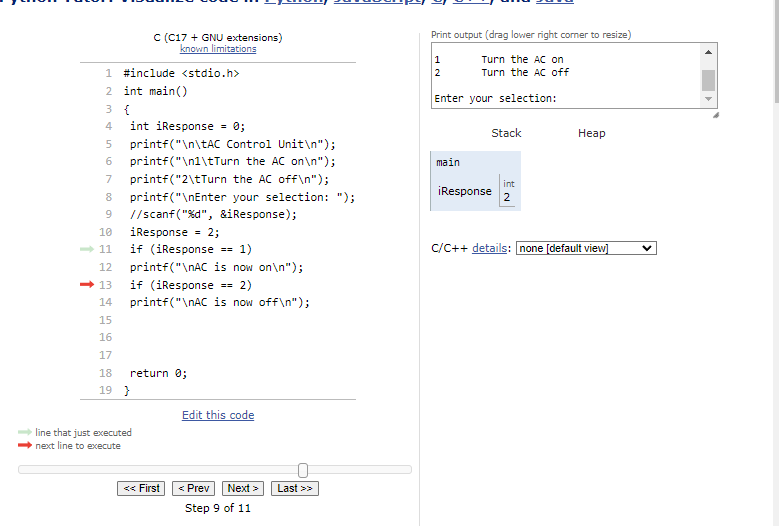


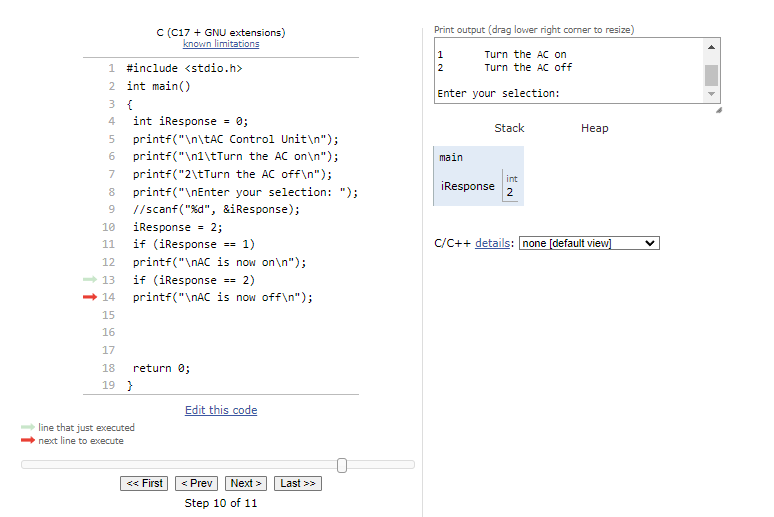


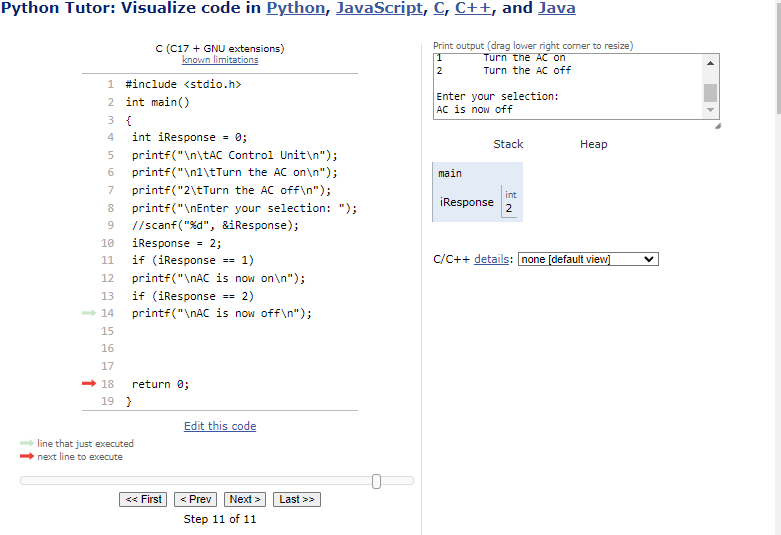


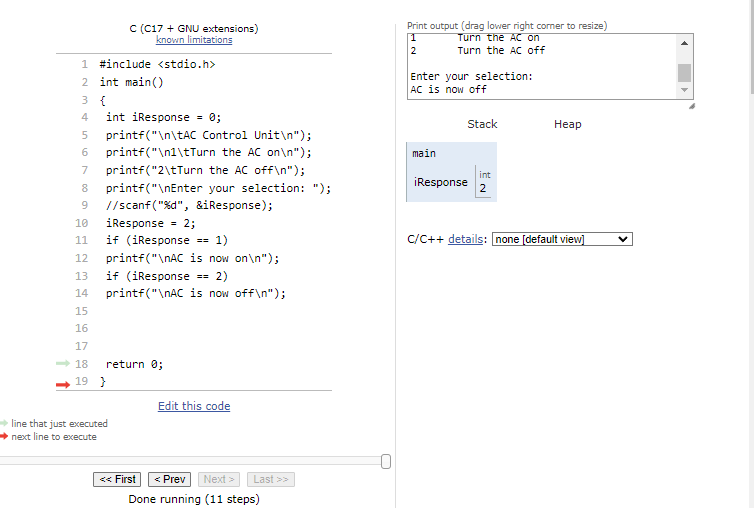












Test in debugger and see how that looks.