CS1100 - Lecture 3

Instructor : Jasine Babu

Teaching Assistants : Nikhila K N, Veena Prabhakaran

1 Introduction

Control flow of a program is the way the program counter changes during the program execution. In previous classes, we have learned how the control flow happens sequentially and a few cases of how control flow happens conditionally. Sometimes we can use a *flow chart* to show the control flow of a program.

1.1 Conditional Construct

if is a *conditional construct*, that tests a condition, then changes the control flow based on whether the condition is true or false. Different forms of the if statement with their corresponding flow chart showing the control flow are shown below.

1.1.1 Form 1

```
if( condition )
{
    Block 1
}
Instruction 1
```

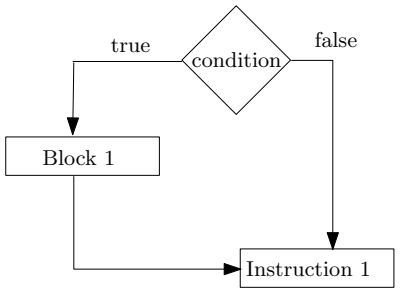

Figure 1: i

In this form when the program control reaches the *if instruction*, the condition is evaluated and if the condition evaluates to *true*, then instructions in Block 1 will be executed. On the other hand, if the condition evaluates to *false*, then the execution of instructions in Block 1 will be skipped.

1.1.2 Form 2

```
if( condition )
{
    Block 1
}
else
{
    Block 2
}
```

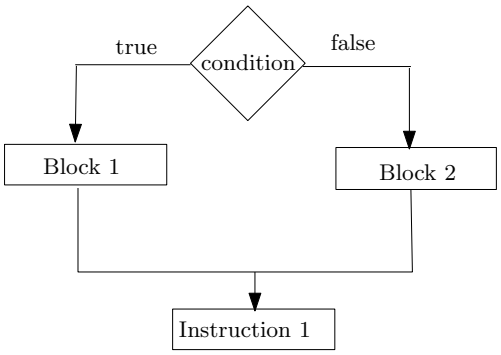

Figure 2: ii

In this form when the program control reaches the *if instruction*, the condition is evaluated and if the condition evaluates to *true*, then instructions in Block 1 will be executed and the instructions in Block 2 will be skipped. On the other hand, if the condition evaluates to *false*, then the execution of instructions in Block 1 will be skipped and the instructions in Block 2 will be executed.

1.1.3 Form 3

```
if( condition 1 )
{
    Block 1
}
else if( condition 2)
{
    Block 2
}
else if( condition 3)
{
    Block 3
}
else
{
    Block 4
}
Instruction 1
      .
      .
      .
```

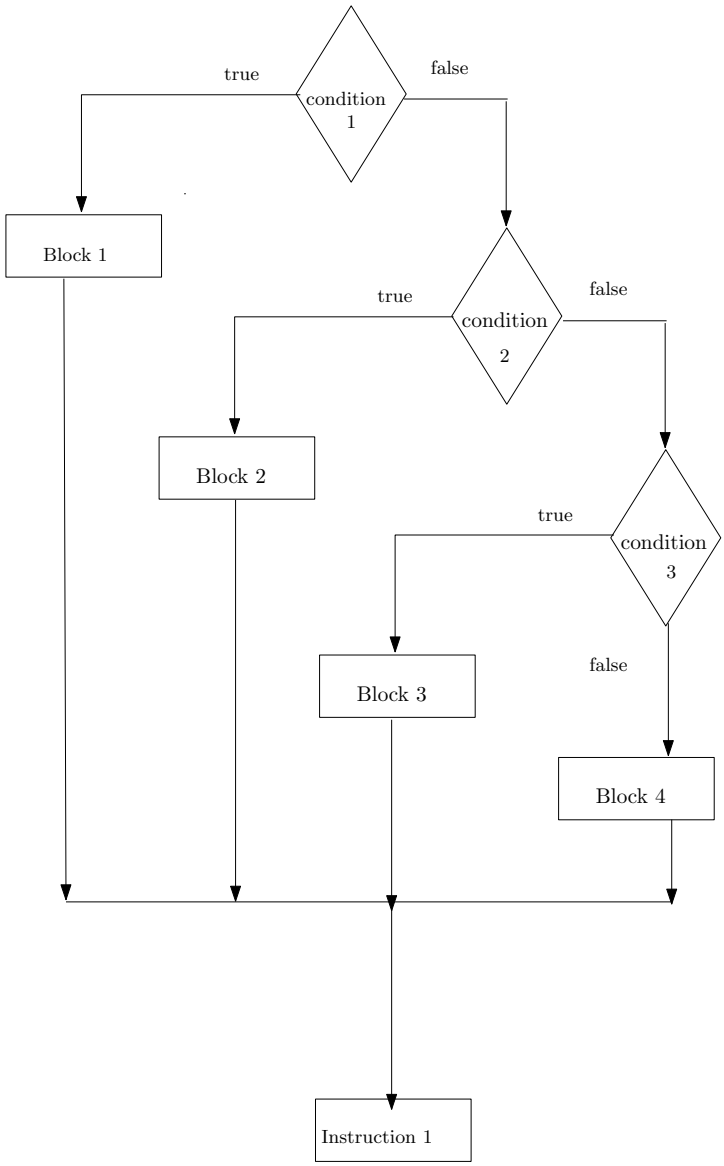

Figure 3: iii

When the control reaches the first *if* instruction, condition 1 is evaluated and if it evaluates to *true*, Block 1 will be executed and Block 2 to Block 4 are skipped. If condition 1 evaluates to *false*, Block 1 is skipped and condition 2 is evaluated. If condition 2 evaluates to *true* at this point, Block 2 will be executed and Block 3 and Block 4 are skipped. If condition 2 evaluates to *false*, Block 2 is skipped and condition 3 is evaluated. If condition 3 evaluates to *true* at this point, Block 3 will be executed and Block 4 will be skipped. If condition 3 also evaluates to *false*, then Block 3 is skipped and Block 4 is executed.

In general, there may be more conditions, not just 3.

The same logic can be implemented using nested *if* statements as follows.

```
if( condition 1 )
{
    Block 1
}
else
{
    if( condition 2)
    {
        Block 2
     }
    else
    {
        if( condition 3 )
        {
            Block 3

        }
        else
        {
            Block 4
        }
    }
}
Instruction 1
.
.
.
```

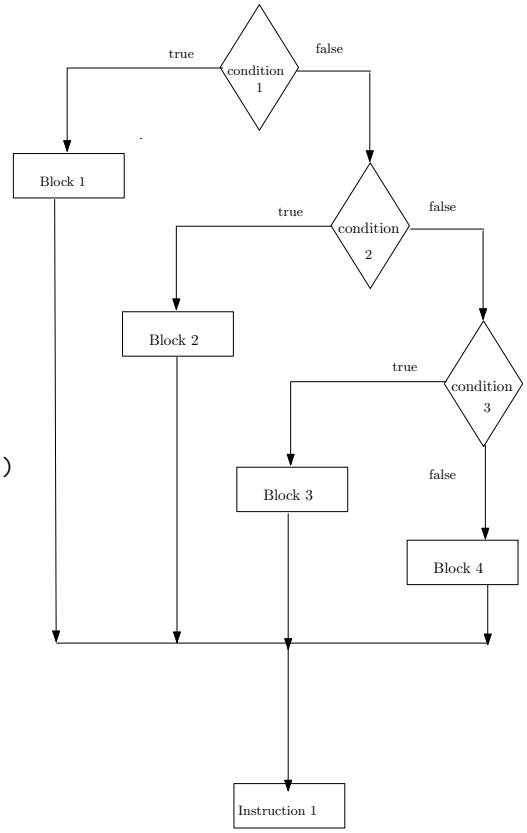

Figure 4: iii

However, Form 3 is preffered over the above nested form, to avoid deep nesting.

The usage of combination of different forms of *if* is illustarted in the example below.

```
 /* Program to decide if three input numbers are
    ascending, descending, neither or both */
#include<stdio.h>
int main()
{
  int a, b,c;
  printf("Give three numbers \n");
  scanf("%d%d%d",&a, &b, &c);
  if(a<b)
  {
     if (b<=c)
     {
        printf("ascending \n");
     }
     else
     {
       printf("neither ascending or descending \n");
     }
  }
  else if(a>b)
  {
     if (b>=c)
     {
        printf("descending \n");
     }
     else
     {
       printf("neither ascending or descending \n");
     }
  }
  else
  {
     if (b==c)
     {
        printf("both ascending and descending \n");
     }
     else if (b<c)
     {
       printf("ascending \n");
     }
     else
     {
       printf("descending \n");
     }
  }
  return 0;
}
```

1.2 Iterative Constructs

For performing a particular task several times, repeating the instructions may not work always. For example, consider a task of finding the sum of n numbers, where the value of n is a part of the user input (which is unknown while writing the program). In this situation, the number of times the instructions to update *sum* has to be repeated is unknown to the programmer. Moreover, using only the instructions we have learned so far, even finding the sum of 1000 numbers would need at least 1000 instructions, which is difficult.

while is one of the simplest iterative constructs. It is used for repeatedly executing a set of instructions based on the result of evaluation of a condition.

A figure showing the syntax and a flow chart showing the control flow associated with a while instruction is given below.

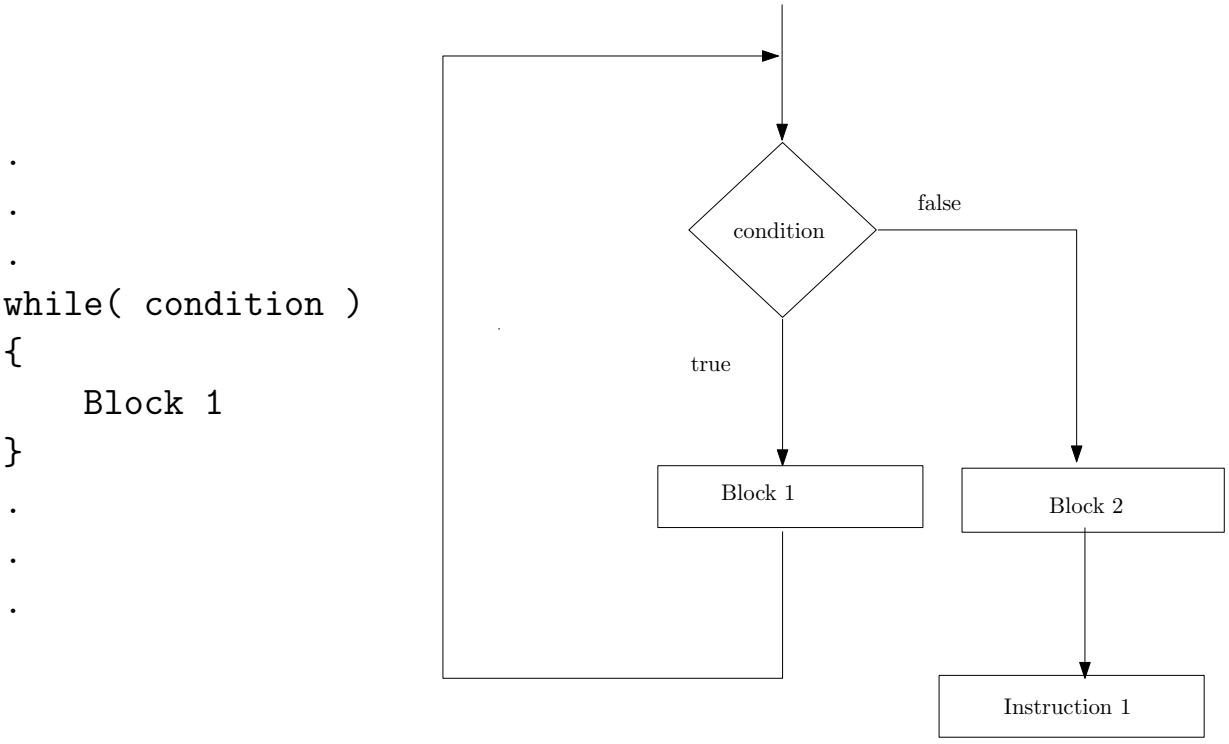

Figure 5: i

If the control reaches the *while* instruction, the condition is evaluated and if the condition is *true*, the instructions in Block 1 are executed and control comes back to the *while* instruction again for checking the condition. If the condition evaluates to *false* when the *while* instruction is executed, then control skips Block 1 and jumps to the instruction after the } after Block 1.

The program for finding sum of four numbers by using a *while* loop is shown below.

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

When the statement `while (i < 4)` is executed, the current values of variable `i` and 4 are taken to the ALU and compared. If the condition evaluates to *true* , then the instructions between the opening bracket { after the *while* instruction and its pairing closing bracket } are to be executed and then the control returns to the *while* instruction again to check the condition. This process of checking the condition and executing the block between the opening bracket { after the *while* instruction and its pairing closing bracket } is repeated, until the condition becomes *false* at the time of executing the *while* instruction. When the condition evaluates to *false* when the *while* instruction is executed, the program control jumps to the statement after the }.

The memory state diagrams during the execution of the above program for the inputs 10,15,18,20 is shown below.

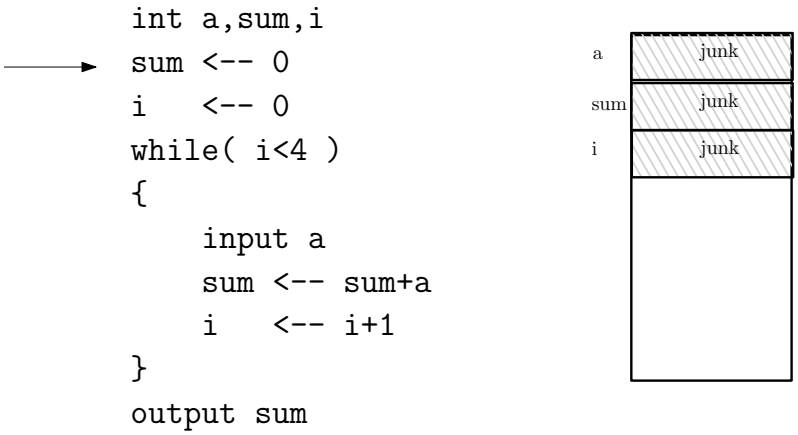

Figure 6: i

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

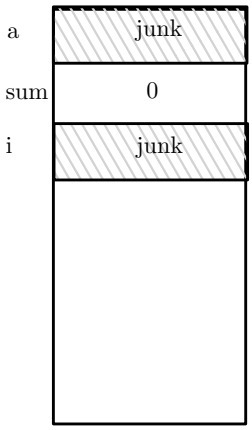

Figure 7: ii

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

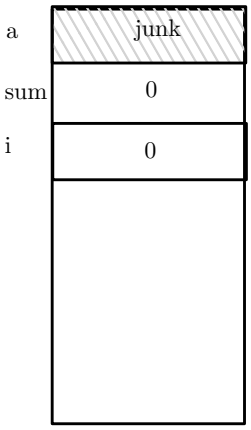

Figure 8: iii

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

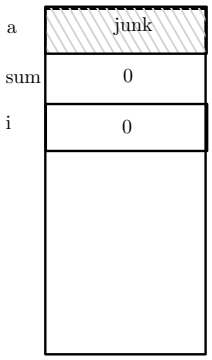

Figure 9: iv

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

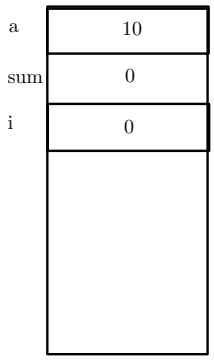

Figure 10: v

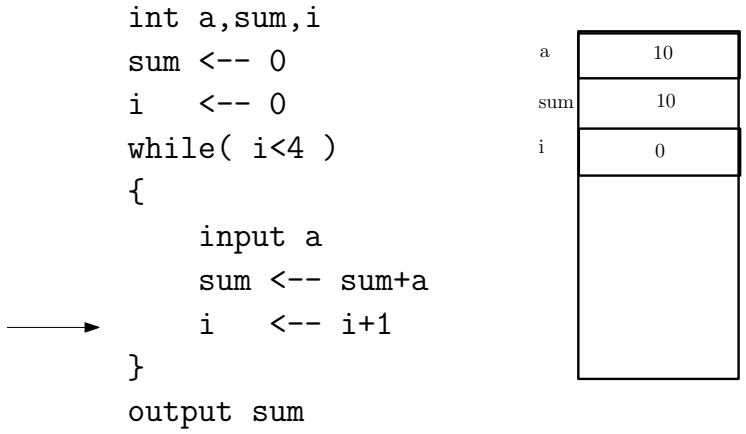

Figure 11: vi

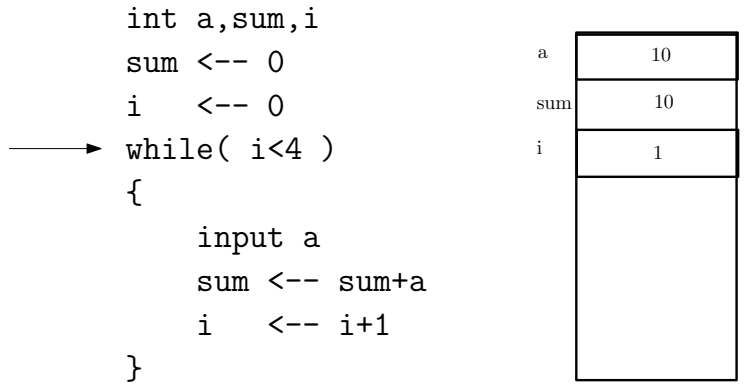

Figure 12: vii

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

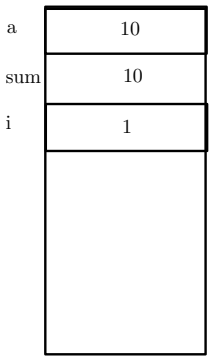

Figure 13: viii

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

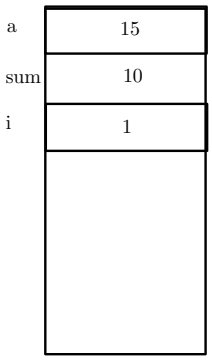

Figure 14: ix

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

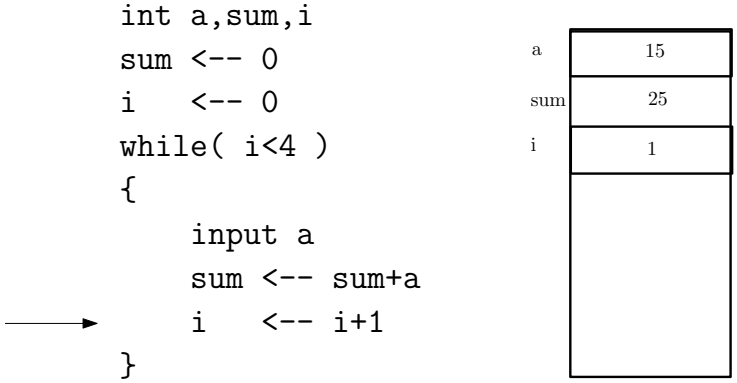

Figure 15: x

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

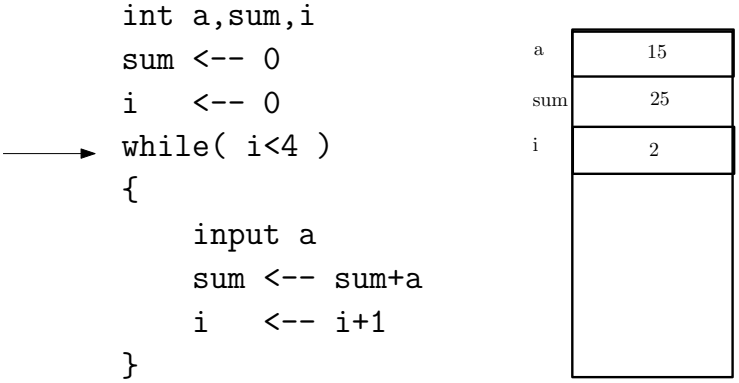

Figure 16: xi

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

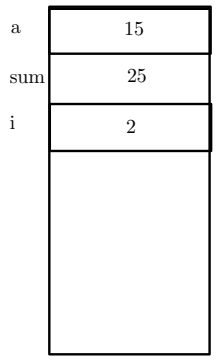

Figure 17: xii

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

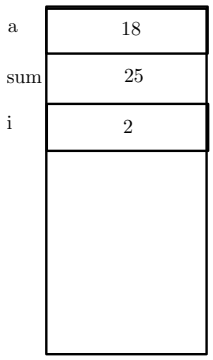

Figure 18: xiii

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

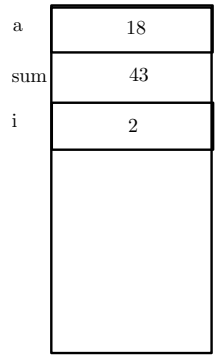

Figure 19: xiv

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

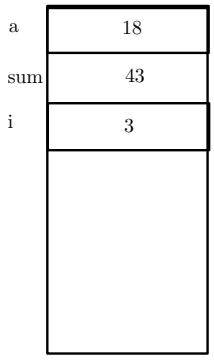

Figure 20: xv

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

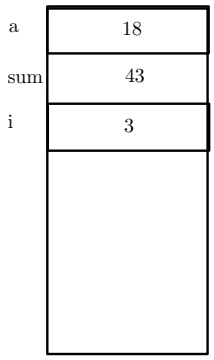

Figure 21: xvi

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

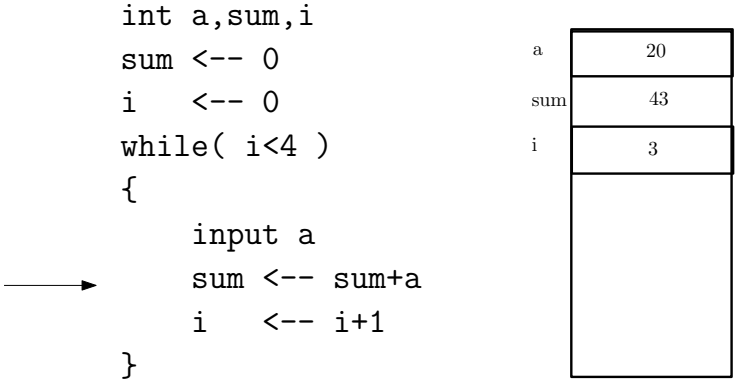

Figure 22: xvii

```
int a,sum,i
sum <-- 0
i   <-- 0
while( i<4 )
{
    input a
    sum <-- sum+a
    i   <-- i+1
}
output sum
```

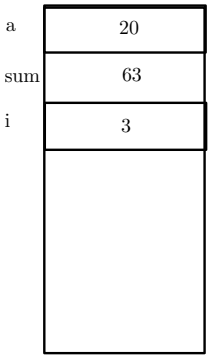

Figure 23: xviii

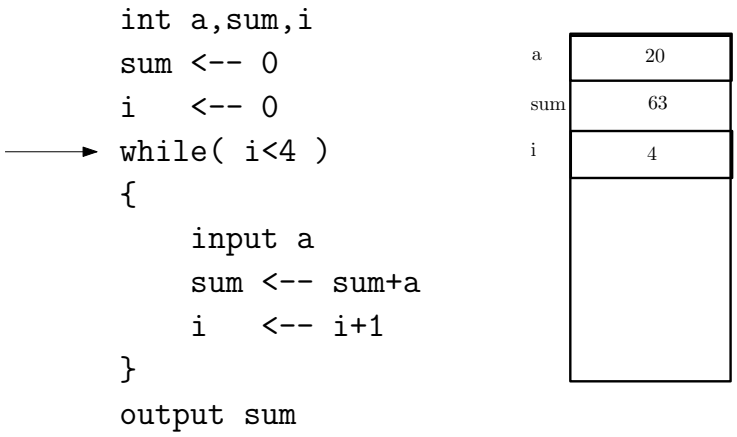

Figure 24: xix

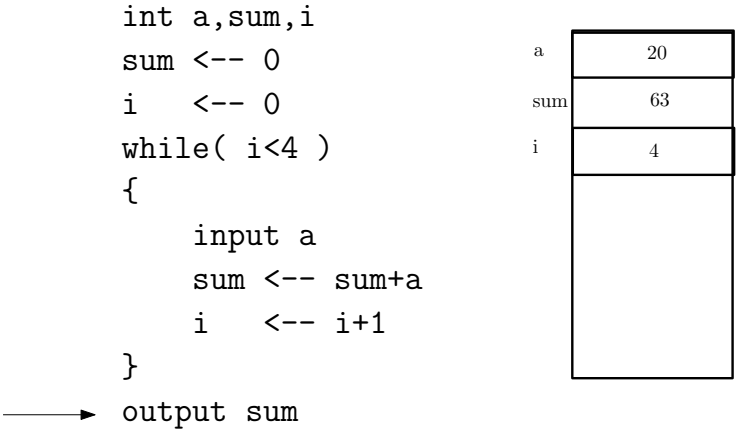

Figure 25: xx

In C language, condition evaluation is done in a slightly different way. If the result of a comparision is true, then the evaluation of the condition gets the value 1 and if the result of a comparision is false, then the evaluation of the condition gets the value 0. Moreover, instead of a condition, even an arithmetic expression can occur. In this case, if the expression evaluates to non-zero value, it is considered equivalent to a true condition and if the expression evaluates to 0, it is considered equivalent to false condition.

Another point to note is that in C, the symbol `==` is used for equality checking, whereas the symbol `=` is reserved for assignment instructions (i.e., for assigning values to variables).