CS1100 - Lecture 8

Instructor : Jasine Babu

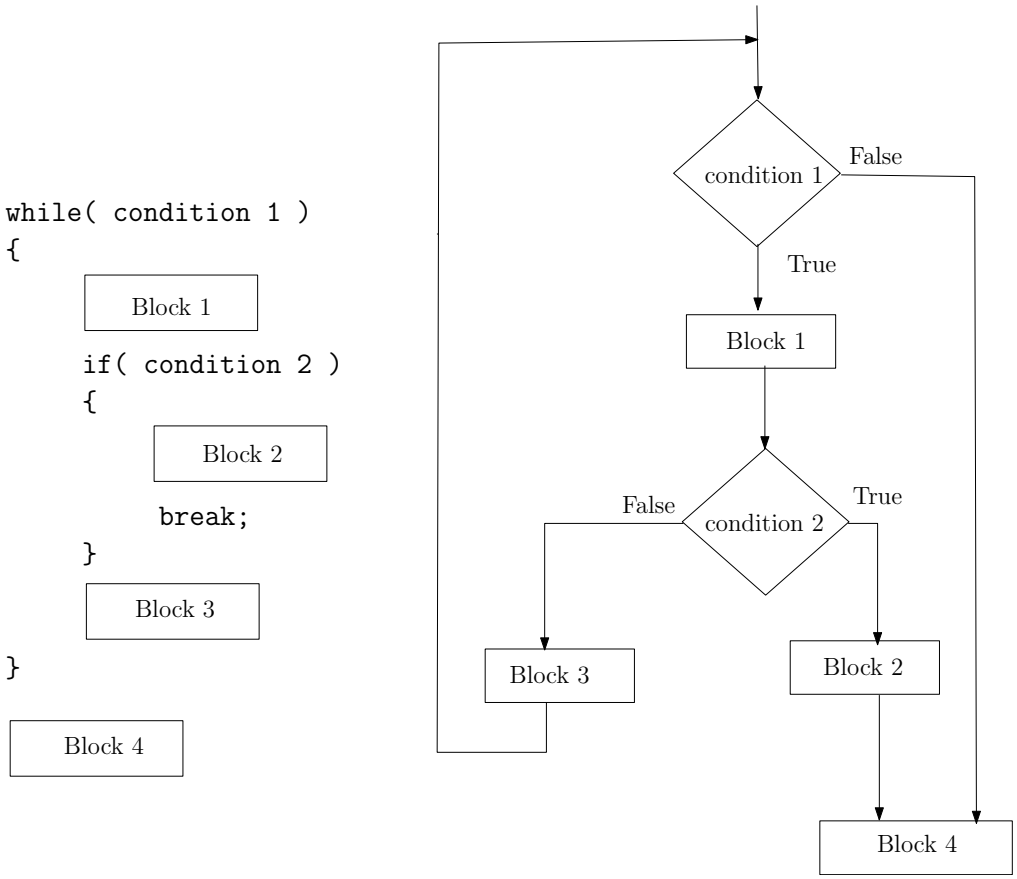
Teaching Assistants : Nikhila K N, Veena Prabhakaran

Break and Continue

`Break` and `continue` are control statements used to change the control flow inside a loop.

Break

A `break` statement is used to stop executing a loop. A `break` inside a nested loop exits from the inner most loop. The control flow associated with a `break` instruction inside a `while` loop is explained in the figure below.



Note that, when *condition2* is evaluated to *true* when the `if` statement inside the `while` loop is executed, the loop is immediately exited and the control goes to *Block 4*, eventhough *condition 1* is still true. This is the difference between a `while` loop without a `break` and a `while` loop with a `break`. Further, it is technically possible to have a `break`

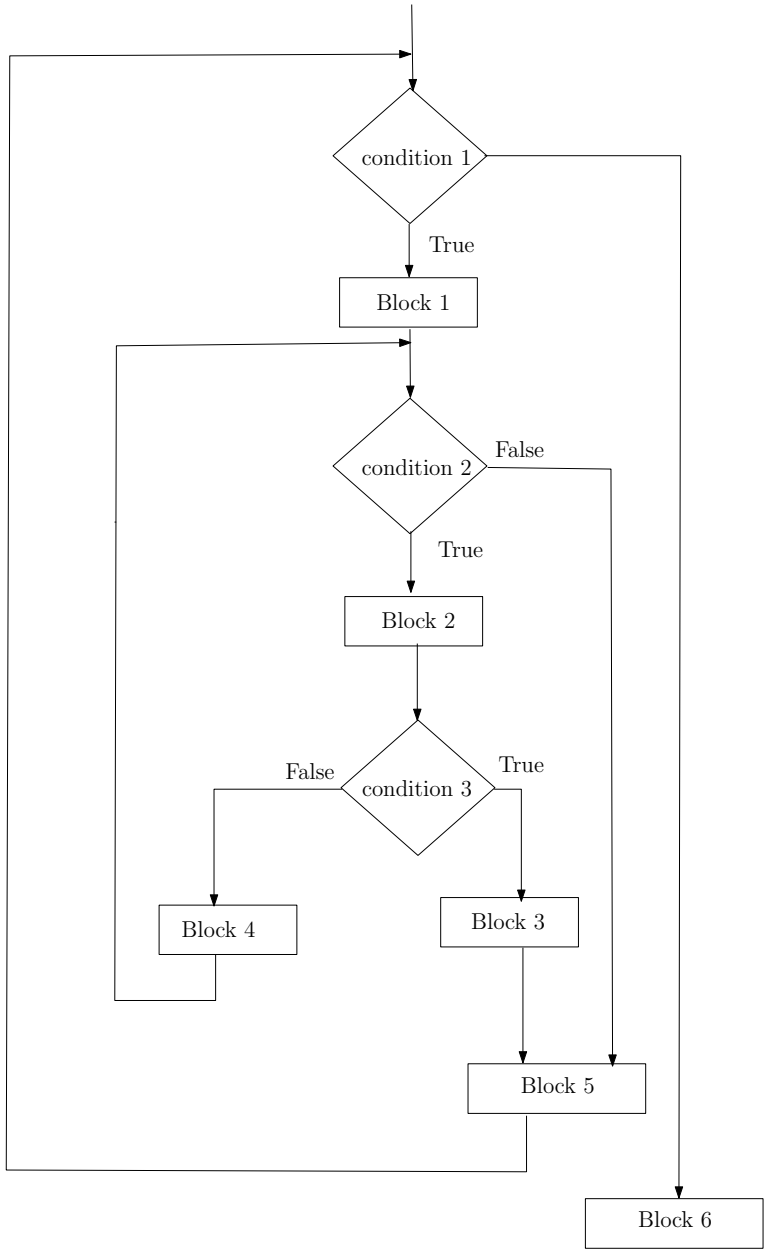
statement without a preceeding `if` condition. But often that is useless because, the loop is executed only once in such a case and the code inside the loop after the `break` (like `Block 3` in the figure) becomes a dead code which is never executed.

Remark: Note that, `break` instruction can be also used along with `switch` statements, which we will learn later.

break instruction in nested loops

Note that, if a `break` instruction is inside a nested loop, then executing the `break` instruction will only force exiting from the inner loop which is currently getting executed and not from an outer loop. This is explained in the figure below.

```
while(condition1)
{
    Block 1
    while(condition2)
    {
       Block 2
       if(condition 3)
       {
          Block 3
          break;
       }
       Block 4
    }
    Block 5
}
Block 6
```



Array Search

Consider the problem of searching for a number in an array of `n` numbers. Question no:4 of Exercise sheet 4 gives an outline of a program for solving this problem. However, this program does some unnecessary work because, even after finding the position of the searched number, it is again compared with all the remaining elements in the array. This can be avoided using `break`. After finding the searched number once, the execution of the loop can be stopped using a `break` instruction, so that no further comparisons are done with remaining elements in the array. The modified program is given below.

```
int main()
{
    int a[20], n, counter, num, posn;
    printf("enter n (<=20) \n");
    scanf("%d",&n);
    posn=-1;
    if(n>20)
    {
       printf("wrong input \n");
    }
    else
    {
        counter = 0;
        while (counter < n)
        {
             printf("enter the next element \n");
             scanf("%d",&a[counter]);
             counter = counter +1;
        }
        printf("enter the number to search \n");
        scanf("%d",&num);
        counter = 0;
        while (counter<n)
        {
           if(a[counter]==num)
           {
               posn=counter;
               break;
           }
           counter = counter + 1;
        }
        if (posn==-1)
           printf("searched number not found\n");
        else
           printf("searched number is the %dth item in the list\n", posn+1);
     }
     return 0;
}
```

Note that, if the searched number `num` matches with an element of the array in the instruction `if(a[counter]==num)`, the value of variable `posn` changes to the value of the variable `counter` and immediately break from the loop happens. If the `break` is never executed, it means a match never occured, and the value of the variable `posn` remains as -1 till the end.

Primality Checking

Now, we will consider the problem of checking whether a given number `n` is prime or not. Our method is the following. For each number `i` such that `1<i<n` check if `n` is divisible by `i`. If such a divisor is found, the number `n` is composite; otherwise, `n` is prime. This can be done in a loop.

Note that, if we find one divisor `i` of `n` as above, we can immediately exit from the loop using `break` without checking further. A program for primality checking using this method is given below.

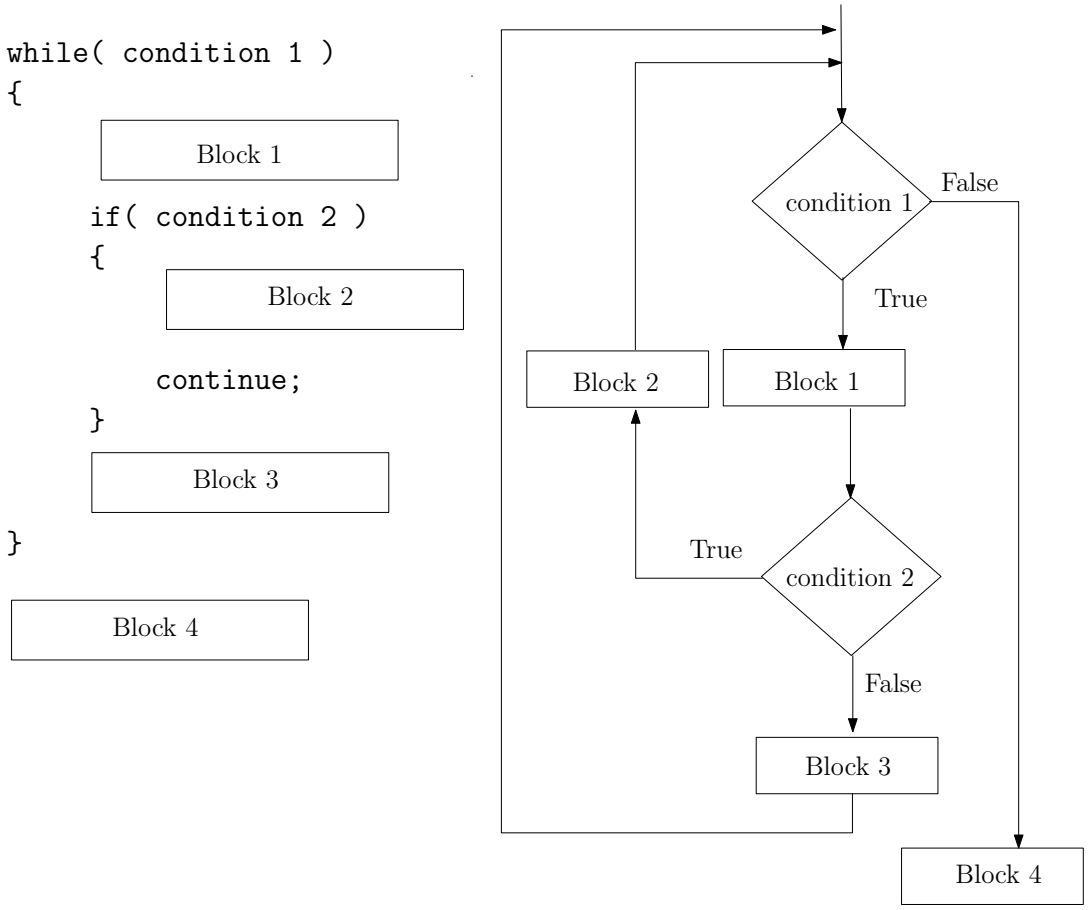
```
int main()
{
   int i, n;
   printf("enter a positive number to check for primality\n");
   scanf("%d", &n);
   if(n==1)
       printf("%d is not a prime number\n", n);
   else
   {
       i=2;
       while(i < n)
       {
          if (n%i == 0)  //we found a non-trivial divisor of n
          {
             printf("%d is not a prime\n", n);
             break;
          }
          i=i+1;
       }
       if (i==n)
          printf("Yes! %d is a prime!\n",n);
   }
  return(0);
}
```

Remark 1: This program can be optimised by checking values of `i` upto `n/2` only, because no nontrivial divisors of `n` are greater than `n/2`.

Remark 2: The above program can be further optimized by using the observation that for any composite number at least one of its divisors (other than 1) will be less than or equal to $\sqrt{n}$. However we will need to use libraries like `math.h` for this. Moreover, we need to be careful about the lack of precision associated with floating point computations.

Continue

A *continue* statement is used only inside a loop. When a *continue* instruction is executed inside a loop, it temporarily skips the rest of the instructions in the current iteration and program control goes to the begining of the loop and starts the next iteration of a loop. This is explained in the figure below.



Note that, as in the case of `break`, a `continue` instruction inside a nested loop affects the control flow of only the inner loop in which the `continue` instruction is placed.

Logical Operators

C supports logical operators such as &&(AND), ||(OR) and !(NOT).

If `expression1` and `expression2` are two valid expressions, then the expression `expression1 && expression2` has value 1 if both `expression1` and `expression2` have non-zero values; otherwise the value of `expression1 && expression2` is 0.

If `expression1` and `expression2` are two valid expressions, then the expression `expression1 || expression2` has value 0 if both `expression1` and `expression2` have value 0; otherwise the value of `expression1 && expression2` is 1.

If `expression1` is a valid expression, then the expression `!expression1` has value 1 if `expression1` has value 0; otherwise the value of `!expression1` is 0.

Apart from the semantics mentioned above, we also need to take care of some subtle facts while using these logical operators. This is explained in the section below.

Consider the nested `if` given below.

```
if(condition1)
{
    if(condition2)
    {
        BLOCK1;
    }
}
```

Here, `BLOCK1` will be executed when both `condition1` and `condition2` becomes true. This can also be implemented with a single `if` using the logical operator && as follows.

```
if(condition1 && condition2)
{
    BLOCK1;
}
```

Only if both conditions are *true Block 1* is executed. However, `condition 1` is evaluated first and if it is *false* `condition 2` is not evaluated.

Note that, this is not the same as the following.

```
if(condition2 && condition1)
{
    BLOCK1;
}
```

The above version has the same meaning as the following.

```
if(condition2)
{
    if(condition1)
    {
        BLOCK1;
    }
}
```

In this case, `condition 2` is evaluated first and if it is *false* `condition 1` is not evaluated.

A program to demonstrate the importance of ordering of conditions, when using && operator is given below.

```
  #include<stdio.h>
int main()
{
  int a[5], i;
  i=0;
  while(i<5)
  {
    a[i]=(i+1)*10;
    i=i+1;
  }
  printf("crash test 1!!\n");
  i=5000;
  if (i<5 && a[i]<50)
  {
     a[i]=a[i]*10;
     printf("%d",a[i]);
  }
  printf("crash test 1 passed!!\n");
  printf("crash test 2!!\n");
  i=5000;
  if ( a[i]<50 && i<5)
  {
     a[i]=a[i]*10;
     printf("%d",a[i]);
  }
  printf("crash test 2 passed!!\n");
  return(0);
}
```

This program outputs

```
crash test 1!!
crash test 1 passed!!
crash test 2!!
Segmentation fault (core dumped)
```

A programmer is unlikely to make a big mistake as above. However, something like the following is a very common mistake. Assume that, `a` is defined to be an array of size 5 in the following example.

```
i=0;
while((a[i] < 60)&& (i<5))
{
   printf("a[%d]=%d\n",i, a[i]);
```

```
    i=i+1;
}
```

This program is likely to crash, because the condition `a[5]<60` will be getting evaluated, though, the last element in the array is `a[4]`.

An `if` with conditions connected using `||` becomes *true* when either of them becomes *true*. As in the case of `&&`, the instructions `if(condition1 || condition2)` and `if(condition2 || condition1)` are not the same. In the first case, `condition1` is evaluated first and if it is *true*, `condition2` is not evaluated further. But in the second case, first `condition2` is evaluated and if it is *true*, `condition1` is not evaluated. In both the cases, the second condition is evaluated only when the first evaluates to *false*.

In general, a complex condition made up of smaller conditions connected by logical operators are evaluated only till the final answer of the complex condition is determined. Moreover, note that the && operator has higher preceedence than the || operator. The ! operator has higher preceedence than both || and && operators.