Ex.

```
if (a == b)
    c = 3;
else
    c = 4;
cout << c;
```

Ex.

```
if (a = b)
    c = 3;
else
    c = 4;
cout << c;
```

THE FIRST EXAMPLE PRINTS 4, WHILE THE SECOND PRINTS 3.

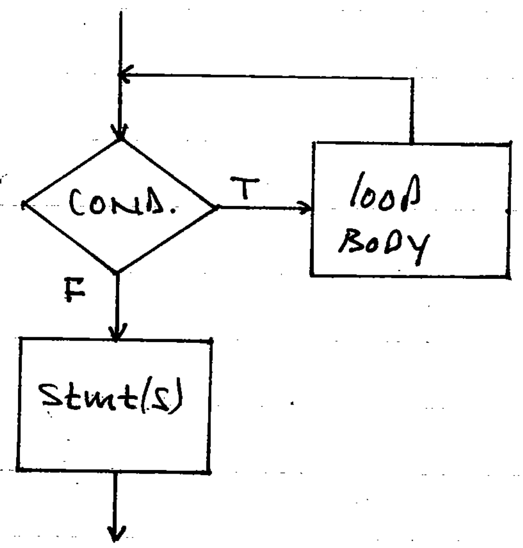
LOOPING STRUCTURES: while, do-while, for

```
while (CONDITION)
    stmt;
```

OR

```
while (CONDITION)
{
    stmt 1;
      :
    stmt k;
}
```

FLOW CHART



Ex.
```
int i, list[10]={2,4,6,8,10,12,14,16,18,20};

i=0
while(i<10)
{
    cout << list[i]<<' ';
    i = i+1;
}
cout << endl;
```

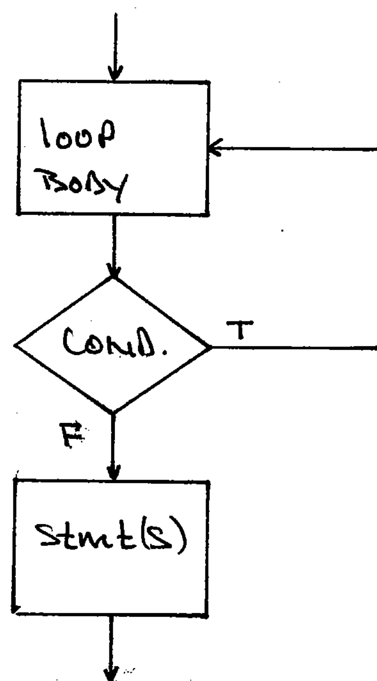
THE C/C++ WHILE LOOP IS ESSENTIALLY THE SAME AS THE WHILE LOOP IN OUR PSEUDO-CODE. C/C++ ALSO HAS A LOOP STRUCTURE SIMILAR TO (BUT NOT IDENTICAL TO) OUR PSEUDO-CODE REPEAT LOOP.

```
do
    stmt;
while(CONDITION);
```

OR

```
do
{
    stmt1;
      :
    stmtK;
} while(CONDITION);
```

FLOW CHART



EX.

```
int i, list[10]={2,4,6,8,10,12,14,16,18,20};

i = 0;
do
{
     cout << list[i] << ' ';
     i = i+1;
} while(i<10);
cout << endl;
```

do-while is similar to our repeat structure in that loop body executes at least once. It is different in that condition is a loop repitition condition, rather than a loop termination condition.

In both examples above we have a loop control variable (LCV) i, whose value determines the truth value of the loop repitition condition (LRC).

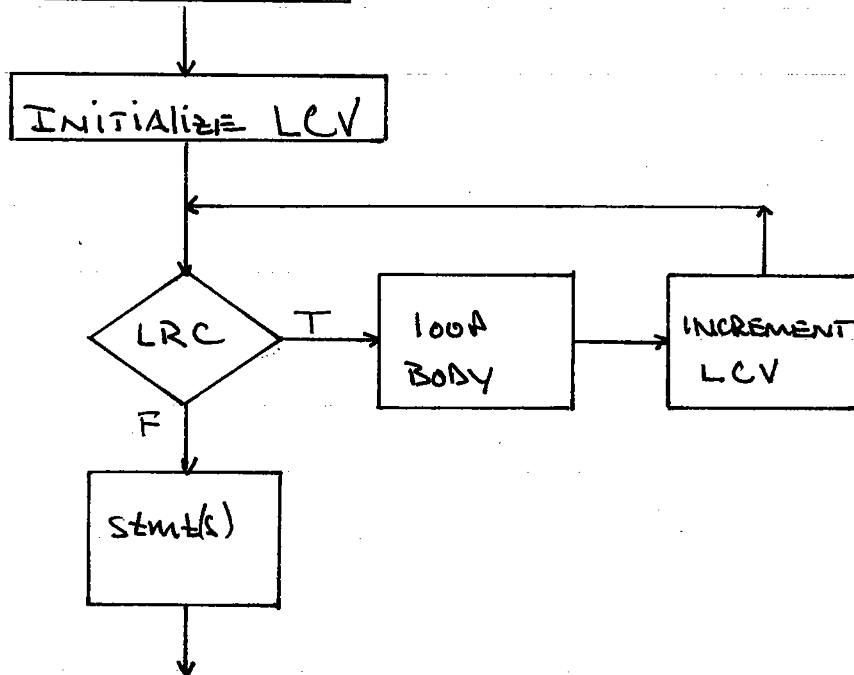
Both examples containing the following steps (not necessarily in this order.)

1.) initialize LCV
2.) test LRC
3.) increment LCV

C/C++ PROVIDES A THIRD LOOPING STRUCTURE WHICH PLACE (1), (2), (3) IN THE SAME LOCATION.

```
for (INITIALIZE LCV; TEST LRC; INCREMENT LCV)
{
      stmt 1;
        :
      stmt K;
}
```

FLOW CHART



EX.
```
int i, list[ ] = {2,4,6,8,10,12,14,16,18,20};

for (i=0; i<10; i=i+1)
      cout << list[i] << ' ';
cout << endl;
```

C/C++ HAS SEVERAL SHORTCUTS WHICH ARE OFTEN USED TO INCREMENT THE LCV.

COMPOUND ASSIGNMENT OPERATORS

```
a = a + b;      a += b;
a = a - b;      a -= b;
a = a * b;      a *= b;
a = a / b;      a /= b;
```

EX.
```
for (i = 0; i < 10; i += 1)
    ....
```

AUTO INCREMENT/DECREMENT OPERATORS

```
a += 1;     a++;     ++a;
a -= 1;     a--;     --a;
```

EX.
```
for (i = 0; i < 10; i++)
    ....
```

IN FACT THIS IS HOW THE C++ LANGUAGE GOT ITS NAME.

WE NOW HAVE THE TOOLS NEEDED TO IMPLEMENT MOST OF THE ALGORITHMS STUDIED BACK IN CHAPTERS 2 AND 3.

RECALL SELECTION SORT

— SEE EXAMPLE : ALGORITHM —

EX.

WRITE A C++ PROGRAM TO PERFORM SELECTION SORT.

— SEE EXAMPLE : SELECTIONSORT.CPP —

EXCERCISE

WRITE A C++ PROGRAM TO PERFORM BUBBLE SORT (CHAPTER 3, PROBLEM 13, P. 110). USE SELECTION SORT EXAMPLE AS MODEL.

BUBBLE SORT

1.) GET VALUES FOR n, $a_1, \ldots, a_n$ FROM USER
2.) SET TOP TO n.
3.) WHILE TOP > 1 DO 4 – 9
4.) SET j TO 2
5.) WHILE $j \le$ TOP DO 6 – 8
6.) IF $a_j < a_{j-1}$
7.) EXCHANGE $a_j \leftrightarrow a_{j-1}$
8.) SET j TO $j+1$
9.) SET TOP TO TOP – 1
10.) PRINT LIST $a_1, \ldots, a_n$
11.) STOP.

INPUT: $n \geq 1$, $a_1, \ldots, a_n$

OUTPUT: MODIFIED LIST IN INCREASING ORDER.

1.) SET TOP TO n
2.) REPEAT UNTIL TOP < 2
3.) FIND INDEX imax OF MAX IN UNSORTED PART
4.) EXCHANGE a_{imax} WITH a_{TOP}
5.) SET TOP TO (TOP − 1)
6.) END LOOP
7.) STOP

WE REFINE STEP 3 BY RECALLING THE FIND-LARGEST ALGORITHM.

3.1) SET max TO a_1
3.2) SET imax TO 1
3.3) SET j TO 2
3.4) REPEAT UNTIL j > TOP
3.5) IF a_j > max
3.6) SET max TO a_j
3.7) SET imax TO j
3.8) SET j TO j+1
3.9) END LOOP

TO REFINE STEP 4 WE MUST INTRODUCE A NEW VARIABLE.

4.1) SET TEMP TO a_{imax}
4.2) SET a_{imax} TO a_{TOP}
4.3) SET a_{TOP} TO TEMP.

```
////////////////////////////////////////////////////////////////////////
//      File: SelectionSort.cpp
//      Input:  list length n, and array list.
//      Output: list in increasing order.
////////////////////////////////////////////////////////////////////////

#include<iostream>
using namespace std;

#define MAX_LIST_LEN  100

int main(void)
{
        int    n,                          // list length
               j,                          // loop control
               imax,                       // index of max
               top;                        // unsorted marker
        double max,                        // maximum value
               temp,                       // temporary storage
               list[MAX_LIST_LEN];         // array to be sorted

        // Get values for n and array list.
        cout << "Enter the list length "
             << "(must be less than or equal to "
             << MAX_LIST_LEN << "): ";
        cin  >> n;
        cout << "Enter " << n << " numbers:\n";
        for(j=0; j<n; j++) { cin >> list[j]; }

        // Perform selection sort on array list.
        for(top=n-1; top>0; top--)
        {
                // Find imax, index of max element in unsorted section
                max = list[0];
                imax = 0;
                for(j=1; j<=top; j++)
                {
                        if( list[j] > max )
                        {
                                max = list[j];
                                imax = j;
                        }
                }

                // Exchange list[imax] with list[top].
                temp = list[imax];
                list[imax] = list[top];
                list[top] = temp;
        }

        // Print out sorted list.
        for(j=0; j<n; j++)
                cout << list[j] << ' ';
        cout << endl;

        return(0);
}
```

FUNCTIONS

MANAGING COMPLEXITY IN LARGE PROGRAMS CAN BE QUITE DIFFICULT. PROGRAMMERS OFTEN USE A TECHNIQUE CALLED DIVIDE AND CONQUER.

SUPPOSE WE WISH TO PERFORM A COMPLEX TASK T. WE DIVIDE T INTO A SET OF SMALLER, MORE MANAGEABLE SUBTASKS, SAY A, B, C, D. WE THEN PERFORM A, B, C, D SEPARATELY. THESE MAY NEED TO BE DIVIDED STILL FURTHER INTO SMALLER SUBTASKS E, F, G, H, I.

A STRUCTURE CHART IS OFTEN USED TO DEPICT THE RELATIONSHIPS BETWEEN SUBTASKS: