Basic Data Structures and Flow Control

Dr. G.H.J. Lanel

Lecture 2

Outline

Outline

Basic Data Structures

- Maple has many data structures that provide similar functionality, but certain data structures are better suited for certain types of operations.
- Therefore, when choosing which data structures to use, it is important to select a structure that performs well on the operations used in your code.
- Maple supports a variety of data structures such as tables, arrays, stacks, and queues.
- Two basic data structures commonly used in Maple are **sets** and **lists**.

Basic Data Structures

- Maple has many data structures that provide similar functionality, but certain data structures are better suited for certain types of operations.
- Therefore, when choosing which data structures to use, it is important to select a structure that performs well on the operations used in your code.
- Maple supports a variety of data structures such as tables, arrays, stacks, and queues.
- Two basic data structures commonly used in Maple are **sets** and **lists**.

Basic Data Structures

- Maple has many data structures that provide similar functionality, but certain data structures are better suited for certain types of operations.
- Therefore, when choosing which data structures to use, it is important to select a structure that performs well on the operations used in your code.
- Maple supports a variety of data structures such as tables, arrays, stacks, and queues.
- Two basic data structures commonly used in Maple are **sets** and **lists**.

Basic Data Structures

- Maple has many data structures that provide similar functionality, but certain data structures are better suited for certain types of operations.
- Therefore, when choosing which data structures to use, it is important to select a structure that performs well on the operations used in your code.
- Maple supports a variety of data structures such as tables, arrays, stacks, and queues.
- Two basic data structures commonly used in Maple are **sets** and **lists**.

Lists

- A list stores an ordered sequence of expressions.
- The ordering of the elements in a list is fixed when the list is created.
- Lists , in contrast to sets, will maintain duplicate elements.

Lists

- A list stores an ordered sequence of expressions.
- The ordering of the elements in a list is fixed when the list is created.
- Lists , in contrast to sets, will maintain duplicate elements.

Lists

- A list stores an ordered sequence of expressions.
- The ordering of the elements in a list is fixed when the list is created.
- Lists , in contrast to sets, will maintain duplicate elements.

Operations on a List

- The easiest way to create a list is to enclose a sequence of expressions in square brackets ([]).

```
> [x, y, y];
```

- The elements of a list can be any expressions, even other lists.

```
> M := [ [a,b], [1,2], [3, 4] ];
> L := [ [1] , [2, a] , [X, Y, Z] ];
> [ seq(x^j,j=1..3) ];
```

- The **op** command can be used to extract the sequence of elements in a list.

```
> op(L);
```

Operations on a List

- The easiest way to create a list is to enclose a sequence of expressions in square brackets ([]).

```
> [x, y, y];
```

- The elements of a list can be any expressions, even other lists.

```
> M := [ [a,b], [1,2], [3, 4] ];
> L := [ [1] , [2, a] , [X, Y, Z] ];
> [ seq(x^j,j=1..3) ];
```

- The **op** command can be used to extract the sequence of elements in a list.

```
> op(L);
```

Operations on a List

- The easiest way to create a list is to enclose a sequence of expressions in square brackets ([]).

```
> [x, y, y];
```

- The elements of a list can be any expressions, even other lists.

```
> M := [ [a,b], [1,2], [3, 4] ];
> L := [ [1] , [2, a] , [X, Y, Z] ];
> [ seq(x^j,j=1..3) ];
```

- The **op** command can be used to extract the sequence of elements in a list.

```
> op(L);
```

- Thus **op** can be used to create new lists based on existing lists.

  > NL := [op(L) , 0];

- The selection operation, [ ], can be used to read an element from a list.

  > L[2];

- You can also specify a range in the selection operation to extract a sub-list containing the elements that are **indexed** by that range.

  > K := [seq(i^2,i=1..10)];

  > K[3 .. 6];

- Assignment for a current element in a small list [number_of_elements $\leq$ 100] can be performed using assignment operator.

  > L[1] := -5;

- Thus **op** can be used to create new lists based on existing lists.

  > NL := [op(L) , 0];

- The selection operation, [ ], can be used to read an element from a list.

  > L[2];

- You can also specify a range in the selection operation to extract a sub-list containing the elements that are **indexed** by that range.

  > K := [seq(i^2,i=1..10)];

  > K[3 .. 6];

- Assignment for a current element in a small list [number_of_elements $\leq$ 100] can be performed using assignment operator.

  > L[1] := -5;

- Thus **op** can be used to create new lists based on existing lists.

  > NL := [op(L) , 0];

- The selection operation, [ ], can be used to read an element from a list.

  > L[2];

- You can also specify a range in the selection operation to extract a sub-list containing the elements that are **indexed** by that range.

  > K := [seq(i^2,i=1..10)];
  > K[3 .. 6];

- Assignment for a current element in a small list [number_of_elements $\leq$ 100] can be performed using assignment operator.

  > L[1] := -5;

- Thus **op** can be used to create new lists based on existing lists.

  > NL := [op(L) , 0];

- The selection operation, [ ], can be used to read an element from a list.

  > L[2];

- You can also specify a range in the selection operation to extract a sub-list containing the elements that are **indexed** by that range.

  > K := [seq(i^2,i=1..10)];
  > K[3 .. 6];

- Assignment for a current element in a small list [number_of_elements $\leq$ 100] can be performed using assignment operator.

  > L[1] := -5;

- Assignment to a large list is not permitted in Maple and will produce an error

```
> LL := [ seq( i, i=1..200 ) ]:
> LL[1] := -1;
```

- Therefore, if you need to make a copy of a list with one changed element, it is recommended that you use the subsop command instead.

```
> subsop( 1=-1 , LL);
```

- If you need to remove an element from the list then set the **index of the element** to **NULL**

```
> subsop( 1=NULL , LL);
```

- Assignment to a large list is not permitted in Maple and will produce an error

```
> LL := [ seq( i, i=1..200 ) ]:
> LL[1] := -1;
```

- Therefore, if you need to make a copy of a list with one changed element, it is recommended that you use the subsop command instead.

```
> subsop( 1=-1 , LL);
```

- If you need to remove an element from the list then set the **index of the element** to **NULL**

```
> subsop( 1=NULL , LL);
```

- Assignment to a large list is not permitted in Maple and will produce an error

```
> LL := [ seq( i, i=1..200 ) ]:
> LL[1] := -1;
```

- Therefore, if you need to make a copy of a list with one changed element, it is recommended that you use the subsop command instead.

```
> subsop( 1=-1 , LL);
```

- If you need to remove an element from the list then set the **index of the element** to **NULL**

```
> subsop( 1=NULL , LL);
```

Sets

- A set is an unordered sequence of unique expressions.
- When a set is created, Maple reorders the expressions to remove duplicate values and to make certain operations faster.

Sets

- A set is an unordered sequence of unique expressions.
- When a set is created, Maple reorders the expressions to remove duplicate values and to make certain operations faster.

Operations on Set

- The easiest way to create a set is to enclose a sequence of expressions in braces { }.

```
> {x, y, y};
```

- Similar to lists, the **op** command can be used to extract the sequence of elements in a set.

```
> S := { x,y,z} ;
> op(S);
```

Operations on Set

- The easiest way to create a set is to enclose a sequence of expressions in braces { }.

```
> {x, y, y};
```

- Similar to lists, the **op** command can be used to extract the sequence of elements in a set.

```
> S := { x,y,z} ;
> op(S);
```

- Maple provides operators for mathematical set manipulations: union, minus, intersect, and subset. These operators allow you to perform set arithmetic in Maple.

```
> T := { y,z,w } ;
> S union T ;
> S minus T ;
> S intersect T ;
> S subset T ;
```

- The selection operation, [ ], can be used to read an element from a set.

```
> A := {3,2,1} :
> A[1] ;
```

- Unlike lists, you cannot use the selection operation to create new sets.

```
> A[1] := 4 ;
```

- Maple provides operators for mathematical set manipulations: union, minus, intersect, and subset. These operators allow you to perform set arithmetic in Maple.

```
> T := { y,z,w } ;
> S union T ;
> S minus T ;
> S intersect T ;
> S subset T ;
```

- The selection operation, [ ], can be used to read an element from a set.

```
> A := {3,2,1} :
> A[1] ;
```

- Unlike lists, you cannot use the selection operation to create new sets.

```
> A[1] := 4 ;
```

- Maple provides operators for mathematical set manipulations: union, minus, intersect, and subset. These operators allow you to perform set arithmetic in Maple.

```
> T := { y,z,w } ;
> S union T ;
> S minus T ;
> S intersect T ;
> S subset T ;
```

- The selection operation, [ ], can be used to read an element from a set.

```
> A := {3,2,1} :
> A[1] ;
```

- Unlike lists, you cannot use the selection operation to create new sets.

```
> A[1] := 4 ;
```

- Therefore **subsop** command has to be used to assign elements for the list and remove elements from the list

```
> subsop(1=a, A);
> subsop(1=NULL, A);
```

- Like in lists **op** command has to be used for add elements to the set.

```
> T:={op(T), k};
```

- To test for set membership, use the **member** command or the in operator.

```
> member(x, S);
```

true

- To apply a function to the members of a set, use the **map** command.

```
> map( f, S );
```

{f(x),f(y),f(z)}

- Therefore **subsop** command has to be used to assign elements for the list and remove elements from the list

```
> subsop(1=a, A);
> subsop(1=NULL, A);
```

- Like in lists **op** command has to be used for add elements to the set.

```
> T:={op(T), k};
```

- To test for set membership, use the **member** command or the in operator.

```
> member(x, S);
```

true

- To apply a function to the members of a set, use the **map** command.

```
> map( f, S );
```

{f(x),f(y),f(z)}

- Therefore **subsop** command has to be used to assign elements for the list and remove elements from the list

```
> subsop(1=a, A);
> subsop(1=NULL, A);
```

- Like in lists **op** command has to be used for add elements to the set.

```
> T:={op(T), k};
```

- To test for set membership, use the **member** command or the in operator.

```
> member(x, S);
```

true

- To apply a function to the members of a set, use the **map** command.

```
> map( f, S );
```

{f(x),f(y),f(z)}

- Therefore **subsop** command has to be used to assign elements for the list and remove elements from the list

  > subsop(1=a, A);

  > subsop(1=NULL, A);

- Like in lists **op** command has to be used for add elements to the set.

  > T:={op(T), k};

- To test for set membership, use the **member** command or the in operator.

  > member(x, S);

  true

- To apply a function to the members of a set, use the **map** command.

  > map(f, S);

  {f(x),f(y),f(z)}

Other data structures in Maple

- Arrays
- Matrices
- Stacks
- Queues

Other data structures in Maple

- Arrays
- Matrices
- Stacks
- Queues

Other data structures in Maple

- Arrays
- Matrices
- Stacks
- Queues

Other data structures in Maple

- Arrays
- Matrices
- Stacks
- Queues

Outline

Selection (if)

- **if** statement has the syntax in which condition is a Boolean-valued expression. (that is, one which evaluates to one of the values true or false)
- syntax :

```
if condition then
    statseq
end if;
```

- **statseq** is a (possibly empty) sequence of Maple statements, often called the *body* of the **if** statement.
- The effect of an **if** statement is to divert the flow of control, under the right conditions, to the body of the statement.
- If the **condition** expression evaluates to **true**, the flow of control moves into the body of the **if** statement. Otherwise, if the **condition** expression evaluates to **false**.

Selection (if)

- **if** statement has the syntax in which condition is a Boolean-valued expression. (that is, one which evaluates to one of the values true or false)
- syntax :

```
if condition then
    statseq
end if;
```

- **statseq** is a (possibly empty) sequence of Maple statements, often called the *body* of the **if** statement.
- The effect of an **if** statement is to divert the flow of control, under the right conditions, to the body of the statement.
- If the **condition** expression evaluates to **true**, the flow of control moves into the body of the **if** statement. Otherwise, if the **condition** expression evaluates to **false**.

Selection (if)

- **if** statement has the syntax in which condition is a Boolean-valued expression. (that is, one which evaluates to one of the values true or false)
- syntax :

```
if condition then
    statseq
end if;
```

- **statseq** is a (possibly empty) sequence of Maple statements, often called the *body* of the **if** statement.
- The effect of an **if** statement is to divert the flow of control, under the right conditions, to the body of the statement.
- If the **condition** expression evaluates to **true**, the flow of control moves into the body of the **if** statement. Otherwise, if the **condition** expression evaluates to **false**.

Selection (if)

- **if** statement has the syntax in which condition is a Boolean-valued expression. (that is, one which evaluates to one of the values true or false)
- syntax :

```
if condition then
    statseq
end if;
```

- **statseq** is a (possibly empty) sequence of Maple statements, often called the *body* of the **if** statement.
- The effect of an **if** statement is to divert the flow of control, under the right conditions, to the body of the statement.
- If the **condition** expression evaluates to **true**, the flow of control moves into the body of the **if** statement. Otherwise, if the **condition** expression evaluates to **false**.

Selection (if)

- **if** statement has the syntax in which condition is a Boolean-valued expression. (that is, one which evaluates to one of the values true or false)
- syntax :

```
if condition then
    statseq
end if;
```

- **statseq** is a (possibly empty) sequence of Maple statements, often called the *body* of the **if** statement.
- The effect of an **if** statement is to divert the flow of control, under the right conditions, to the body of the statement.
- If the **condition** expression evaluates to **true**, the flow of control moves into the body of the **if** statement. Otherwise, if the **condition** expression evaluates to **false**.

Example :

```
> x := 3:

> if x < 6 then
      print ( "HELLO" )
  end if;
                                  "HELLO"

> if x > 6 then
      print ( "How are you" )
  end if;
```

Example :

```
> x := 3:

> if x < 6 then
      print ( "HELLO" )
  end if;
                                  "HELLO"

> if x > 6 then
      print ( "How are you" )
  end if;
```

Example :

```
> x := 3:

> if x < 6 then
      print ( "HELLO" )
  end if;
```

"HELLO"

```
> if x > 6 then
      print ( "How are you" )
  end if;
```

Example :

```
> x := 3:

> if x < 6 then
      print ( "HELLO" )
  end if;
```

"HELLO"

```
> if x > 6 then
      print ( "How are you" )
  end if;
```

- More generally, an **if** statement has the syntax :

```
if condition then
    consequent
else
    alternative
end if;
```

- Here, **consequent** and **alternative** are statement sequences.

- If the **condition** expression evaluates to **true**, the **consequent** branch of the **if** statement is executed. Otherwise, the **alternative** branch is executed.

- More generally, an **if** statement has the syntax :

```
if condition then
    consequent
else
    alternative
end if;
```

- Here, **consequent** and **alternative** are statement sequences.

- If the **condition** expression evaluates to **true**, the **consequent** branch of the **if** statement is executed. Otherwise, the **alternative** branch is executed.

- More generally, an **if** statement has the syntax :

```
if condition then
    consequent
else
    alternative
end if;
```

- Here, **consequent** and **alternative** are statement sequences.

- If the **condition** expression evaluates to **true**, the **consequent** branch of the **if** statement is executed. Otherwise, the **alternative** branch is executed.

- The most general form of an **if** statement can have several conditions, corresponding consequences, and an optional alternative branch. This general form has the syntax:

```
if condition1 then
    consequent1
elif condition2 then
    consequent2
...
else
    alternative
end if;
```

- There can be any number of branches preceded by elif.
- order of the elif branches can affect the behaviour of the if statement.
- The branch introduced by else is optional.

- The most general form of an **if** statement can have several conditions, corresponding consequences, and an optional alternative branch. This general form has the syntax:

```
if condition1 then
    consequent1
elif condition2 then
    consequent2
...
else
    alternative
end if;
```

- There can be any number of branches preceded by **elif**.
- order of the elif branches can affect the behavior of the if statement.
- The branch introduced by **else** is optional.

- The most general form of an **if** statement can have several conditions, corresponding consequences, and an optional alternative branch. This general form has the syntax:

```
if condition1 then
    consequent1
elif condition2 then
    consequent2
...
else
    alternative
end if;
```

- There can be any number of branches preceded by elif.
- order of the elif branches can affect the behavior of the if statement.
- The branch introduced by else is optional.

- The most general form of an **if** statement can have several conditions, corresponding consequences, and an optional alternative branch. This general form has the syntax:

```
if condition1 then
    consequent1
elif condition2 then
    consequent2
...
else
    alternative
end if;
```

- There can be any number of branches preceded by **elif**.
- order of the **elif** branches can affect the behavior of the **if** statement.
- The branch introduced by else is optional.

- The most general form of an **if** statement can have several conditions, corresponding consequences, and an optional alternative branch. This general form has the syntax:

```
if condition1 then
    consequent1
elif condition2 then
    consequent2
...
else
    alternative
end if;
```

- There can be any number of branches preceded by **elif**.
- order of the **elif** branches can affect the behavior of the **if** statement.
- The branch introduced by else is optional.

- The most general form of an **if** statement can have several conditions, corresponding consequences, and an optional alternative branch. This general form has the syntax:

```
if condition1 then
    consequent1
elif condition2 then
    consequent2
...
else
    alternative
end if;
```

- There can be any number of branches preceded by **elif**.
- order of the **elif** branches can affect the behavior of the **if** statement.
- The branch introduced by else is optional.

if command

- In this form, **if** is always called with three arguments. The **if** operator has the following syntax:

  ‘if‘ (condition, consequent, alternative);

- The first argument **condition** is a Boolean-valued expression. The second argument is returned, if the first argument is **true**. The third argument is returned if the first argument is either **false**.

  > ‘if‘ (1 < 2, a, b);

  a

  > ‘if‘ (1 > 2, a, b);

  b

- The **if** command is much more limited than the **if** statement.
- The consequent and alternative must be single expressions, and there is nothing corresponding to the elif parts of the statement form.

if command

- In this form, **if** is always called with three arguments. The **if** operator has the following syntax:

  ‘if‘ (condition, consequent, alternative);

- The first argument **condition** is a Boolean-valued expression. The second argument is returned, if the first argument is **true**. The third argument is returned if the first argument is either **false**.

```
> ‘if‘ ( 1 < 2, a, b );
                                    a

> ‘if‘ ( 1 > 2, a, b );
                                    b
```

- The **if** command is much more limited than the **if** statement.
- The consequent and alternative must be single expressions, and there is nothing corresponding to the elif parts of the statement form.

if command

- In this form, **if** is always called with three arguments. The **if** operator has the following syntax:

  ‘if‘ (condition, consequent, alternative);

- The first argument **condition** is a Boolean-valued expression. The second argument is returned, if the first argument is **true**. The third argument is returned if the first argument is either **false**.

  > ‘if‘ (1 < 2, a, b);

  a

  > ‘if‘ (1 > 2, a, b);

  b

- The **if** command is much more limited than the **if** statement.
- The consequent and alternative must be single expressions, and there is nothing corresponding to the elif parts of the statement form.

if command

- In this form, **if** is always called with three arguments. The **if** operator has the following syntax:

  ‘if‘ (condition, consequent, alternative);

- The first argument **condition** is a Boolean-valued expression. The second argument is returned, if the first argument is **true**. The third argument is returned if the first argument is either **false**.

```
> ‘if‘ ( 1 < 2, a, b );
                                    a

> ‘if‘ ( 1 > 2, a, b );
                                    b
```

- The **if** command is much more limited than the **if** statement.
- The consequent and alternative must be single expressions, and there is nothing corresponding to the elif parts of the statement form.

Repetition Control (Loops)

- To cause a statement, or sequence of statements, to be run more than once, use a loop statement.
- Maple has a general and flexible loop statement. Two main loop statements in Maple are **for** and **while**

Repetition Control (Loops)

- To cause a statement, or sequence of statements, to be run more than once, use a loop statement.
- Maple has a general and flexible loop statement. Two main loop statements in Maple are **for** and **while**

for Loop

- Induction variable whose value changes at each iteration of the loop, is a particular kind of *loop* with the general form :

```
for var from start to finish by increment do
    statseq
end do;
```

- The **default** value for **start** is **1**, for **finish** is **infinity**, and for **increment** is **1**.

```
for i to 3 do
    print(i);
end do;
```

- The loop executes until i>3. In this case, when the loop terminates, the final value of i is 4.

for Loop

- Induction variable whose value changes at each iteration of the loop, is a particular kind of *loop* with the general form :

```
for var from start to finish by increment do
    statseq
end do;
```

- The **default** value for **start** is **1**, for **finish** is **infinity**, and for **increment** is **1**.

```
> for i to 3 do
      print( i )
  end do;
```

The loop executes until i>3. In this case, when the loop terminates, the final value of i is 4.

for Loop

- Induction variable whose value changes at each iteration of the loop, is a particular kind of *loop* with the general form :

```
for var from start to finish by increment do
    statseq
end do;
```

- The **default** value for **start** is **1**, for **finish** is **infinity**, and for **increment** is **1**.

```
> for i to 3 do
      print( i )
  end do;
```

- The loop executes until i>3. In this case, when the loop terminates, the final value of i is 4.

for Loop

- Induction variable whose value changes at each iteration of the loop, is a particular kind of *loop* with the general form :

```
for var from start to finish by increment do
    statseq
end do;
```

- The **default** value for **start** is **1**, for **finish** is **infinity**, and for **increment** is **1**.

```
> for i to 3 do
      print( i )
  end do;
```

- The loop executes until i>3. In this case, when the loop terminates, the final value of i is 4.

for Loop

- Induction variable whose value changes at each iteration of the loop, is a particular kind of *loop* with the general form :

```
for var from start to finish by increment do
    statseq
end do;
```

- The **default** value for **start** is **1**, for **finish** is **infinity**, and for **increment** is **1**.

```
> for i to 3 do
      print( i )
  end do;
```

- The loop executes until **i>3**. In this case, when the loop terminates, the final value of i is **4**.

Example :

```
> for i from 7 to 2 by -2 do
    print( i )
  end do;
```

```
7
5
3
```

- Loop control parameters (start, finish, and increment) do not need to be integers.

```
> for i from 0.2 to 0.7 by 0.25 do
    print( i )
  end do;
```

```
0.2
0.45
0.70
```

Example :

```
> for i from 7 to 2 by -2 do
      print( i )
  end do;
```

```
7
5
3
```

- Loop control parameters (start, finish, and increment) do not need to be integers.

```
> for i from 0.2 to 0.7 by 0.25 do
      print( i )
  end do;
```

```
0.2
0.45
0.70
```

Example :

```
> for i from 7 to 2 by -2 do
    print( i )
  end do;
```

```
7
5
3
```

- Loop control parameters (start, finish, and increment) do not need to be integers.

```
> for i from 0.2 to 0.7 by 0.25 do
    print( i )
  end do;
```

```
0.2
0.45
0.70
```

Example :

```
> for i from 7 to 2 by -2 do
    print( i )
  end do;
```

```
7
5
3
```

- Loop control parameters (start, finish, and increment) do not need to be integers.

```
> for i from 0.2 to 0.7 by 0.25 do
    print( i )
  end do;
```

```
0.2
0.45
0.70
```

Example :

```
> for i from 7 to 2 by -2 do
    print( i )
  end do;
```

```
7
5
3
```

- Loop control parameters (start, finish, and increment) do not need to be integers.

```
> for i from 0.2 to 0.7 by 0.25 do
    print( i )
  end do;
```

```
0.2
0.45
0.70
```

Example :

```
> for i from 7 to 2 by -2 do
    print( i )
  end do;
```

```
7
5
3
```

- Loop control parameters (start, finish, and increment) do not need to be integers.

```
> for i from 0.2 to 0.7 by 0.25 do
    print( i )
  end do;
```

```
0.2
0.45
0.70
```

- In addition to iterating over a numeric range, you can iterate over a range of characters.
- In this case, you must specify both the initial value start and the final value finish for the induction variable. Furthermore, the value of increment must be an integer.

- In addition to iterating over a numeric range, you can iterate over a range of characters.
- In this case, you must specify both the initial value start and the final value finish for the induction variable. Furthermore, the value of increment must be an integer.

```
> for i from "a" to "f" by 2 do
    print( i )
  end do;
```

```
"a"
"c"
"e"
```

- In addition to iterating over a numeric range, you can iterate over a range of characters.
- In this case, you must specify both the initial value start and the final value finish for the induction variable. Furthermore, the value of increment must be an integer.

```
> for i from "a" to "f" by 2 do
    print( i )
  end do;
```

```
"a"
"c"
"e"
```

- In addition to iterating over a numeric range, you can iterate over a range of characters.
- In this case, you must specify both the initial value start and the final value finish for the induction variable. Furthermore, the value of increment must be an integer.

```
> for i from "a" to "f" by 2 do
      print( i )
  end do;
```

```
"a"
"c"
"e"
```

- In addition to iterating over a numeric range, you can iterate over a range of characters.
- In this case, you must specify both the initial value start and the final value finish for the induction variable. Furthermore, the value of increment must be an integer.

```
> for i from "a" to "f" by 2 do
      print( i )
  end do;
```

```
"a"
"c"
"e"
```

while Loop

- One simple kind of terminating loop is the *while loop*.

```
while condition do
    statseq
end do;
```

- The loop header of a **while** loop involves only a single termination condition introduced by the keyword **while**.
- The loop repeats the statement sequence *statseq* until the Boolean-valued expression *condition* does not hold.

while Loop

- One simple kind of terminating loop is the *while loop*.

```
while condition do
    statseq
end do;
```

- The loop header of a **while** loop involves only a single termination condition introduced by the keyword **while**.
- The loop repeats the statement sequence **statseq** until the Boolean-valued expression **condition** does not hold.

while Loop

- One simple kind of terminating loop is the *while loop*.

```
while condition do
    statseq
end do;
```

- The loop header of a **while** loop involves only a single termination condition introduced by the keyword **while**.
- The loop repeats the statement sequence **statseq** until the Boolean-valued expression **condition** does not hold.

while Loop

- One simple kind of terminating loop is the *while loop*.

```
while condition do
    statseq
end do;
```

- The loop header of a **while** loop involves only a single termination condition introduced by the keyword **while**.
- The loop repeats the statement sequence **statseq** until the Boolean-valued expression **condition** does not hold.

End!