Pointers

✦ References:

Programmer’s Guide to Fortran 90. Brainerd Goldberg and Adams

✦ What are Fortran pointers?

• A pointer variable can be though of as an alias for another variable.
• They are a descriptor listing the attributes of the objects (targets) that the pointer may point to, and the address, if any, of a target. They also encapsulate the lower and upper bounds of array dimensions, strides and other metadata.
• They have no associated storage until it is allocated or otherwise associated.
A pointer variable can be of any type
A pointer is a variable that has been given the **pointer** attribute.
A variable aliased or “pointed to” by a pointer must have the **target** attribute

For Example

```fortran
REAL, POINTER :: ptr
REAL, TARGET :: x

x = 4.7
ptr => x
print *, ptr

x = 8.3
print *, ptr
```
✧ A pointer variable can be of any type
✧ A pointer is a variable that has been given the \textit{pointer} attribute.
✧ A variable aliased or “pointed to” by a pointer must have the \textit{target} attribute
✧ For Example

\begin{verbatim}
REAL, POINTER :: ptr
REAL, TARGET :: x

x = 4.7
ptr => x
print *, ptr

x = 8.3
print *, ptr
\end{verbatim}

\begin{itemize}
\item A pointer variable can be of any type
\item A pointer is a variable that has been given the \textit{pointer} attribute.
\item A variable aliased or “pointed to” by a pointer must have the \textit{target} attribute
\item For Example
\end{itemize}
✧ A pointer variable can be of any type
✧ A pointer is a variable that has been given the \textit{pointer} attribute.
✧ A variable aliased or “pointed to” by a pointer must have the \textit{target} attribute
✧ For Example

```fortran
REAL, POINTER :: ptr
REAL, TARGET :: x

x = 4.7
ptr => x
print *, ptr
x = 8.3
print *, ptr
```

A pointer variable can be of any type. A pointer is a variable that has been given the \textit{pointer} attribute. A variable aliased or “pointed to” by a pointer must have the \textit{target} attribute. For Example:

```fortran
REAL, POINTER :: ptr
REAL, TARGET :: x

x = 4.7
ptr => x
print *, ptr
x = 8.3
print *, ptr
```
✦ A pointer variable can be of any type
✦ A pointer is a variable that has been given the **pointer** attribute.
✦ A variable aliased or “pointed to” by a pointer must have the **target** attribute
✦ For Example

```
REAL, POINTER :: ptr
REAL, TARGET :: x

x = 4.7
ptr => x
print *, ptr

x = 8.3
print *, ptr
```

```
x
ptr → 4.7
```

```
bliss 1 > 4.7
```

Tuesday, March 10, 2009
✧ A pointer variable can be of any type
✧ A pointer is a variable that has been given the **pointer** attribute.
✧ A variable aliased or “pointed to” by a pointer must have the **target** attribute
✧ For Example

```plaintext
REAL, POINTER :: ptr
REAL, TARGET :: x

x = 4.7
ptr => x
print *, ptr

x = 8.3
print *, ptr
```

A pointer variable can be of any type, a pointer is a variable that has been given the **pointer** attribute, a variable aliased or “pointed to” by a pointer must have the **target** attribute. For Example:

```plaintext
REAL, POINTER :: ptr
REAL, TARGET :: x

x = 4.7
ptr => x
print *, ptr

x = 8.3
print *, ptr
```
A pointer variable can be of any type
A pointer is a variable that has been given the `pointer` attribute.
A variable aliased or “pointed to” by a pointer must have the `target` attribute
For Example

```
REAL, POINTER :: ptr
REAL, TARGET :: x

x = 4.7
ptr => x
print *, ptr

x = 8.3
print *, ptr
```

```
bliss 1 > 4.7
bliss 2 > 8.3
```
There are two types of pointer assignment:

**Pointer assignment** (=>) transfers the status of one pointer to another.

**Ordinary assignment** (=) transfers values of the aliased targets in the usual way.

For Example

```fortran
REAL,POINTER :: ptr1,ptr2
REAL,TARGET :: x1,x2

x1 = 4.7
x2 = 8.3

ptr1 => x1
ptr2 => ptr1 ! pointer assignment
PRINT *,ptr2
ptr2 => x2
ptr1 = ptr2 ! ordinary assignment
PRINT *,ptr1
```

![Diagram](attachment:image.png)
There are two types of pointer assignment:

**Pointer assignment** (=>) transfers the status of one 
pointer to another

**Ordinary assignment** (=) transfers values of the 
aliased targets in the usual way

For Example

```
REAL,POINTER :: ptr1,ptr2
REAL,TARGET :: x1,x2

x1 = 4.7
x2 = 8.3

ptr1 => x1
ptr2 => ptr1  ! pointer assignment
PRINT *,ptr2
ptr2 => x2
ptr1 = ptr2  ! ordinary assignment
PRINT *,ptr1
```
✧ There are two types of pointer assignment:

**Pointer assignment** (=>) transfers the status of one pointer to another

**Ordinary assignment** (=) transfers values of the aliased targets in the usual way

✧ For Example

```fortran
REAL, POINTER :: ptr1, ptr2
REAL, TARGET :: x1, x2

x1 = 4.7
x2 = 8.3

ptr1 => x1
ptr2 => ptr1 ! pointer assignment
PRINT *, ptr2
ptr2 => x2
ptr1 = ptr2 ! ordinary assignment
PRINT *, ptr1
```

![Terminal output](Terminal-csh-40×12.png)
There are two types of pointer assignment:

**Pointer assignment** ($\rightarrow$) transfers the status of one pointer to another

**Ordinary assignment** ($=$) transfers values of the aliased targets in the usual way

For Example

```plaintext
REAL,POINTER :: ptr1,ptr2
REAL,TARGET :: x1,x2
x1 = 4.7
x2 = 8.3
ptr1 $\rightarrow$ x1
ptr2 $\rightarrow$ ptr1  ! pointer assignment
PRINT *,ptr2
ptr2 $\rightarrow$ x2
ptr1 = ptr2  ! ordinary assignment
PRINT *,ptr1
```

```plaintext
ptr1 $\rightarrow$ 4.7
ptr2 $\rightarrow$ 8.3
```

Terminal — csh — 40×12
There are two types of pointer assignment:

**Pointer assignment (=>)** transfers the status of one pointer to another.

**Ordinary assignment (=)** transfers values of the aliased targets in the usual way.

**For Example**

```fortran
REAL, POINTER :: ptr1, ptr2
REAL, TARGET :: x1, x2

x1 = 4.7
x2 = 8.3

ptr1 => x1
ptr2 => ptr1  ! pointer assignment
PRINT *, ptr2
ptr2 => x2
ptr1 = ptr2  ! ordinary assignment
PRINT *, ptr1
```

Terminal — csh — 40×12

```
bliss 1 > 4.7
```
There are two types of pointer assignment:

**Pointer assignment** (=>) transfers the status of one pointer to another

**Ordinary assignment** (=) transfers values of the aliased targets in the usual way

**For Example**

```fortran
REAL, POINTER :: ptr1, ptr2
REAL, TARGET :: x1, x2
x1 = 4.7
x2 = 8.3
ptr1 => x1
ptr2 => ptr1 ! pointer assignment
PRINT *, ptr2
ptr2 => x2
ptr1 = ptr2 ! ordinary assignment
PRINT *, ptr1
```

```
bliss 1 > 4.7
```

```
ptr1 → 4.7
ptr2 → 8.3
```

Tuesday, March 10, 2009
There are two types of pointer assignment:

**Pointer assignment** (=>) transfers the status of one pointer to another

**Ordinary assignment** (=) transfers values of the aliased targets in the usual way

**For Example**

```plaintext
REAL,POINTER :: ptr1,ptr2
REAL,TARGET :: x1,x2
x1 = 4.7
x2 = 8.3
ptr1 => x1
ptr2 => ptr1 ! pointer assignment
PRINT *,ptr2
ptr2 => x2
ptr1 = ptr2 ! ordinary assignment
PRINT *,ptr1
```

```plaintext
ptr1 8.3 ptr2 8.3
```

```
bliss 1 > 4.7
```

Tuesday, March 10, 2009
There are two types of pointer assignment:

**Pointer assignment** (=>) transfers the status of one pointer to another

**Ordinary assignment** (=) transfers values of the aliased targets in the usual way

For Example

```fortran
REAL,POINTER :: ptr1,ptr2
REAL,TARGET :: x1,x2
x1 = 4.7
x2 = 8.3
ptr1 => x1
ptr2 => ptr1 ! pointer assignment
PRINT *,ptr2
ptr2 => x2
ptr1 = ptr2 ! ordinary assignment
PRINT *,ptr1
```

```
alu 1 > 4.7
alu 2 > 8.3
```
A pointer can have three states:

1. **Null**. The pointer does not alias any other variable.
2. **Associated**. The pointer is a alias for another variable.
3. **Undefined**. The pointer in not null and not associated. Until a pointer is either nullified or associated it is undefined.
✦ The *allocate* statement applied to a pointer will create space and cause a pointer to refer to that space.
✦ The *deallocate* statement throws away the space pointed to by the argument and makes the argument *null*
✦ For example

```fortran
REAL, POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
DEALLOCATE (ptr)
```
✦ The `allocate` statement applied to a pointer will create space and cause a pointer to refer to that space.
✦ The `deallocate` statement throws away the space pointed to by the argument and makes the argument null
✦ For example

```plaintext
REAL, POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
DEALLOCATE (ptr)
```
The *allocate* statement applied to a pointer will create space and cause a pointer to refer to that space.

The *deallocate* statement throws away the space pointed to by the argument and makes the argument null.

For example

```fortran
REAL,POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
DEALLOCATE (ptr)
```
The *allocate* statement applied to a pointer will create space and cause a pointer to refer to that space.

The *deallocate* statement throws away the space pointed to by the argument and makes the argument null.

For example

```fortran
REAL, POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
DEALLOCATE (ptr)
```
The *nullify* statement causes a pointer variable to be in a state of not pointing to anything.

Nullifying a pointer can result in unreferenced storage. That is, storage which cannot be referenced by the program.

For example

```fortran
REAL, POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
NULLIFY (ptr)
```
✧ The **nullify** statement causes a pointer variable to be in a state of not pointing to anything.
✧ Nullifying a pointer can result in unreferenced storage. That is, storage which cannot be referenced by the program.
✧ For example

```fortran
REAL,POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
NULLIFY (ptr)
```
The *nullify* statement causes a pointer variable to be in a state of not pointing to anything.

Nullifying a pointer can result in unreferenced storage. That is, storage which cannot be referenced by the program.

For example:

```fortran
REAL,POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
NULLIFY (ptr)
```
✦ The **nullify** statement causes a pointer variable to be in a state of not pointing to anything.
✦ Nullifying a pointer can result in unreferenced storage. That is, storage which cannot be referenced by the program.
✦ For example

```
REAL, POINTER :: ptr
ALLOCATE (ptr)
ptr = 8.3
NULLIFY (ptr)
```
The **associated** intrinsic function is used to determine if a pointer variable is pointing to another object.

The **associated** intrinsic function returns true or false.

The pointer variable must be defined. That is, it must either be null or alias some data object.

For example

```plaintext
REAL,POINTER :: ptr
REAL,TARGET :: x

NULLIFY (ptr)
PRINT *,ASSOCIATED (ptr)
ptr => x
PRINT *,ASSOCIATED (ptr,x)
```
The **associated** intrinsic function is used to determine if a pointer variable is pointing to another object.

The **associated** intrinsic function returns true or false.

The pointer variable must be defined. That is, it must either be null or alias some data object.

For example

```fortran
REAL, POINTER :: ptr
REAL, TARGET :: x

NULLIFY (ptr)
PRINT *, ASSOCIATED (ptr)
ptr => x
PRINT *, ASSOCIATED (ptr, x)
```
The **associated** intrinsic function is used to determine if a pointer variable is pointing to another object.

The **associated** intrinsic function returns true or false.

The pointer variable must be defined. That is, it must either be null or alias some data object.

For example

```fortran
REAL,POINTER :: ptr
REAL,TARGET :: x
NULLIFY (ptr)
PRINT *,ASSOCIATED (ptr)
ptr => x
PRINT *,ASSOCIATED (ptr,x)
```

```
bliss 1 > F
```
The **associated** intrinsic function is used to determine if a pointer variable is pointing to another object.

The **associated** intrinsic function returns true or false.

The pointer variable must be defined. That is, it must either be null or alias some data object.

For example

```fortran
REAL,POINTER :: ptr
REAL,TARGET :: x
NULLIFY (ptr)
PRINT *,ASSOCIATED (ptr)
ptr => x
PRINT *,ASSOCIATED (ptr,x)
```

```
bliss 1 > F
```
The **associated** intrinsic function is used to determine if a pointer variable is pointing to another object.

The **associated** intrinsic function returns true or false.

The pointer variable must be defined. That is, it must either be null or alias some data object.

For example

```fortran
REAL, POINTER :: ptr
REAL, TARGET :: x
NULLIFY (ptr)
PRINT *, ASSOCIATED (ptr)
ptr => x
PRINT *, ASSOCIATED (ptr, x)
```

```
bliss 1 > F
bliss 2 > T
```
What are there good for?

✦ Pointers can be used to construct complicated data structures
  • Arrays of pointers
  • Linked list data structures
  • Tree data structures
Arrays of Pointers

- Suppose you have an array of things and the things are of different size.
- For example, consider a sparse matrix where the rows have different numbers of entries.
- We can define a derived data type with a pointer as its sole component, and define arrays of this data type.
- The storage for the rows can be allocated only as necessary.
- Array assignment will copy all components.

```
TYPE row
  REAL, POINTER :: r(:)
END TYPE row

TYPE (row), POINTER :: s(n), t(n)

DO i = 1,n
  ALLOCATE (t(i)%r(1:i))
END DO

s = t
```
Linked Lists

✧ **Linked lists** are a very useful data structure when the size of the data set is not initially known. They can grow to accompany any amount of data.

✧ Data can be put in order “on the fly”.

✧ A linked list is a list of **nodes**. Each node type contains some data and a pointer to the next node.

✧ The **list** type contains only a pointer to the first node of the list.

```fortran
TYPE node
   INTEGER :: value
   TYPE (node),POINTER :: next
END TYPE node

TYPE list
   TYPE (node),POINTER :: first
END TYPE list
```
Next we write the code to \textit{create a new linked list}

\begin{verbatim}
TYPE node
  INTEGER :: value
  TYPE (node),POINTER :: next
END TYPE node

TYPE list
  TYPE (node),POINTER :: first
END TYPE list
\end{verbatim}

\begin{verbatim}
PROGRAM main
  TYPE (list) :: lst
  lst = new ()
END PROGRAM main
\end{verbatim}

The call to function \texttt{new} does this:
\begin{verbatim}
FUNCTION new_list () RESULT (lst)
  TYPE (list) :: lst
  ALLOCATE (lst%first)
  NULLIFY (lst%first%next)
END FUNCTION new_list
\end{verbatim}
Next we write the code to **create a new linked list**

```fortran
TYPE node
  INTEGER :: value
  TYPE (node), POINTER :: next
END TYPE node

TYPE list
  TYPE (node), POINTER :: first
END TYPE list

PROGRAM main
  TYPE (list) :: lst
  lst = new ()
END PROGRAM main

FUNCTION new_list () RESULT (lst)
  TYPE (list) :: lst
  ALLOCATE (lst%first)
  NULLIFY (lst%first%next)
END FUNCTION new_list
```

The call to function new does this:

```
lst

first
```

Tuesday, March 10, 2009
Next we write the code to create a new linked list

```plaintext
PROGRAM main
  TYPE (list) :: lst
  lst = new ()
END PROGRAM main
```

The call to function new does this:

```plaintext
FUNCTION new_list () RESULT (lst)
  TYPE (list) :: lst
  ALLOCATE (lst%first)
  NULLIFY (lst%first%next)
END FUNCTION new_list
```
Next we write the code to **create a new linked list**

```fortran
TYPE node
  INTEGER :: value
  TYPE (node),POINTER :: next
END TYPE node

TYPE list
  TYPE (node),POINTER :: first
END TYPE list

PROGRAM main
  TYPE (list) :: lst
  lst = new ()
END PROGRAM main
```

The call to function `new` does this:

```fortran
FUNCTION new_list () RESULT (lst)
  TYPE (list) :: lst
  ALLOCATE (lst%first)
  NULLIFY (lst%first%next)
END FUNCTION new_list
```
Next we write the code to **add a node** to the linked list

```
SUBROUTINE insert (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  TYPE (node),POINTER :: ptr1,ptr2
  ! find location to put new number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) EXIT
    IF (number < ptr2%value) EXIT
    ptr1 => ptr2
    ptr2 => ptr2%next
  ENDDO
  ! insert new node
  ALLOCATE (ptr1%next)
  ptr1%next%value = number
  ptr1%next%next => ptr2
END SUBROUTINE insert
```

The first call to insert does this:

```
PROGRAM main
  lst = new ()
  CALL insert (lst,83)
  CALL insert (lst,14)
END PROGRAM main
```

The second call to insert does this:

```
PROGRAM main
  lst = new ()
  CALL insert (lst,83)
  CALL insert (lst,14)
END PROGRAM main
```
Next we write the code to **add a node** to the linked list.

```fortran
SUBROUTINE insert (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  TYPE (node),POINTER :: ptr1,ptr2

  ! find location to put new number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) EXIT
    IF (number < ptr2%value) EXIT
    ptr1 => ptr2
    ptr2 => ptr2%next
  ENDDO

  ! insert new node
  ALLOCATE (ptr1%next)
  ptr1%next%value = number
  ptr1%next%next => ptr2

END SUBROUTINE insert
```

The first call to insert does this:

![Diagram of linked list after first call to insert](image1)

The second call to insert does this:

![Diagram of linked list after second call to insert](image2)

```
PROGRAM main
  lst = new ()
  CALL insert (lst,83)
  CALL insert (lst,14)
END PROGRAM main
```
Next we write the code to add a node to the linked list.

```fortran
SUBROUTINE insert (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  TYPE (node),POINTER :: ptr1,pdr2
  ! find location to put new number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) EXIT
    IF (number < ptr2%value) EXIT
    ptr1 => ptr2
    ptr2 => ptr2%next
  ENDDO
  ! insert new node
  ALLOCATE (ptr1%next)
  ptr1%next%value = number
  ptr1%next%next => ptr2
END SUBROUTINE insert
```

The first call to insert does this:

![Diagram of linked list before first insertion](image1)

The second call to insert does this:

![Diagram of linked list after first insertion](image2)
Next we write the code to add a node to the linked list.

```fortran
SUBROUTINE insert (lst,number)
    TYPE (list) :: lst
    INTEGER :: number
    TYPE (node),POINTER :: ptr1,p指2
    ! find location to put new number
    ptr1 => lst%first
    ptr2 => ptr1%next
    DO
        IF (.NOT.ASSOCIATED (ptr2)) EXIT
        IF (number < ptr2%value) EXIT
        ptr1 => ptr2
        ptr2 => ptr2%next
    ENDDO
    ! insert new node
    ALLOCATE (ptr1%next)
    ptr1%next%value = number
    ptr1%next%next => ptr2
END SUBROUTINE insert
```

The first call to insert does this:

The second call to insert does this:

```fortran
PROGRAM main
    lst = new ()
    CALL insert (lst,83)
    CALL insert (lst,14)
END PROGRAM main
```

Tuesday, March 10, 2009
Next we write the code to **add a node** to the linked list.

```fortran
SUBROUTINE insert (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  TYPE (node),POINTER :: ptr1,ptr2
  ! find location to put new number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) EXIT
    IF (number < ptr2%value) EXIT
    ptr1 => ptr2
    ptr2 => ptr2%next
  ENDDO
  ! insert new node
  ALLOCATE (ptr1%next)
  ptr1%next%value = number
  ptr1%next%next => ptr2
END SUBROUTINE insert
```

The first call to insert does this:

![Diagram of the linked list after the first insertion](image1)

The second call to insert does this:

![Diagram of the linked list after the second insertion](image2)

PROGRAM main
```
lst = new ()
CALL insert (lst,83)
CALL insert (lst,14)
END PROGRAM main
```
Next we write the code to **add a node** to the linked list.

```fortran
SUBROUTINE insert (lst,number)
  TYPE (list) :: lst
  INTEGER :: number

  TYPE (node),POINTER :: ptr1,ptr2

  ! find location to put new number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) EXIT
    IF (number < ptr2%value) EXIT
    ptr1 => ptr2
    ptr2 => ptr2%next
  ENDDO

  ! insert new node
  ALLOCATE (ptr1%next)
  ptr1%next%value = number
  ptr1%next%next => ptr2
END SUBROUTINE insert
```

The first call to insert does this:

![Diagram of linked list after first insertion](image1)

The second call to insert does this:

![Diagram of linked list after second insertion](image2)

PROGRAM main

```fortran
lst = new ()
CALL insert (lst,83)
CALL insert (lst,14)
END PROGRAM main
```

**Tuesday, March 10, 2009**
Next we write the code to add a node to the linked list.

```fortran
SUBROUTINE insert (lst,number)
    TYPE (list) :: lst
    INTEGER :: number
    TYPE (node),POINTER :: ptr1,ptr2
    ! find location to put new number
    ptr1 => lst%first
    ptr2 => ptr1%next
    DO
        IF (.NOT.ASSOCIATED (ptr2)) EXIT
        IF (number < ptr2%value) EXIT
        ptr1 => ptr2
        ptr2 => ptr2%next
    ENDDO
    ! insert new node
    ALLOCATE (ptr1%next)
    ptr1%next%value = number
    ptr1%next%next => ptr2
END SUBROUTINE insert
```

The first call to insert does this:

![Diagram of the first call to insert](image1)

The second call to insert does this:

![Diagram of the second call to insert](image2)

Program main:

```fortran
PROGRAM main
    lst = new ()
    CALL insert (lst,83)
    CALL insert (lst,14)
END PROGRAM main
```

The first call to insert does this:

![Diagram of the first call to insert](image1)

The second call to insert does this:

![Diagram of the second call to insert](image2)

Tuesday, March 10, 2009
Next we write the code to **add a node** to the linked list.

```fortran
SUBROUTINE insert (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  TYPE (node),POINTER :: ptr1,ptr2
  ! find location to put new number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) EXIT
    IF (number < ptr2%value) EXIT
    ptr1 => ptr2
    ptr2 => ptr2%next
  ENDDO
  ! insert new node
  ALLOCATE (ptr1%next)
  ptr1%next%value = number
  ptr1%next%next => ptr2
END SUBROUTINE insert
```

The first call to insert does this:

![Diagram of linked list after the first call to insert](image1)

The second call to insert does this:

![Diagram of linked list after the second call to insert](image2)

PROGRAM main

lst = new ()
CALL insert (lst,83)
CALL insert (lst,14)

END PROGRAM main
Next we write the code to **add a node** to the linked list.

```plaintext
SUBROUTINE insert (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  TYPE (node), POINTER :: ptr1, ptr2
  ! find location to put new number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) EXIT
    IF (number < ptr2%value) EXIT
    ptr1 => ptr2
    ptr2 => ptr2%next
  ENDDO
  ! insert new node
  ALLOCATE (ptr1%next)
  ptr1%next%value = number
  ptr1%next%next => ptr2
END SUBROUTINE insert
```

The first call to insert does this:

```
lst          ptr1
     ↓                ↓
     83
```

The second call to insert does this:

```
lst          ptr1
     ↓                ↓
  14 83
```

**Tuesday, March 10, 2009**
Next we write the code to **delete a node** from the list.

```plaintext
SUBROUTINE delete (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  LOGICAL :: found
  TYPE (node),POINTER :: ptr1,ptr2

  ! find location to delete number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) THEN
      found = .FALSE.
      EXIT
    ELSE IF (number==ptr2%value) THEN
      found = .TRUE.
      EXIT
    ELSE
      ptr1 => ptr2
      ptr2 => ptr2%next
    ENDIF
  ENDDO

  ! delete node if found
  IF (found) THEN
    ptr1%next => ptr2%next
    DEALLOCATE (ptr2)
  ENDIF
END SUBROUTINE delete
```

The call to delete does this:

```
Tuesday, March 10, 2009
```
Next we write the code to **delete a node** from the list.

SUBROUTINE delete (lst,number)
  TYPE (list) :: lst
  INTEGER :: number

  LOGICAL :: found
  TYPE (node),POINTER :: ptr1,ptr2
  ! find location to delete number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) THEN
      found = .FALSE.
      EXIT
    ELSE IF (number==ptr2%value) THEN
      found = .TRUE.
      EXIT
    ELSE
      ptr1 => ptr2
      ptr2 => ptr2%next
    ENDIF
  ENDDO
  ! delete node if found
  IF (found) THEN
    ptr1%next => ptr2%next
    DEALLOCATE (ptr2)
  ENDIF
END SUBROUTINE delete

PROGRAM main
  lst = new ()
  CALL insert (lst,83)
  CALL insert (lst,14)
  CALL insert (lst,17)
  CALL delete (lst,17)
END PROGRAM main

The call to delete does this:

lst
  83
  14
  17
ptr1
ptr2
  83

Tuesday, March 10, 2009
Next we write the code to **delete a node** from the list.

```fortran
SUBROUTINE delete (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  LOGICAL :: found
  TYPE (node),POINTER :: ptr1,ptr2
  ! find location to delete number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) THEN
      found = .FALSE.
      EXIT
    ELSE IF (number==ptr2%value) THEN
      found = .TRUE.
      EXIT
    ELSE
      ptr1 => ptr2
      ptr2 => ptr2%next
    ENDIF
  ENDDO
  ! delete node if found
  IF (found) THEN
    ptr1%next => ptr2%next
    DEALLOCATE (ptr2)
  ENDIF
END SUBROUTINE delete
```

The call to delete does this:

```fortran
PROGRAM main
  lst = new ()
  CALL insert (lst,83)
  CALL insert (lst,14)
  CALL insert (lst,17)
  CALL delete (lst,17)
END PROGRAM main
```

Tuesday, March 10, 2009
Next we write the code to **delete a node** from the list.

**SUBROUTINE delete (lst,number)**

```fortran
SUBROUTINE delete (lst,number)
  TYPE (list) :: lst
  INTEGER :: number
  LOGICAL :: found
  TYPE (node),POINTER :: ptr1,ptr2
!
  ! find location to delete number
  ptr1 => lst%first
  ptr2 => ptr1%next
  DO
    IF (.NOT.ASSOCIATED (ptr2)) THEN
      found = .FALSE.
      EXIT
    ELSE IF (number==ptr2%value) THEN
      found = .TRUE.
      EXIT
    ELSE
      ptr1 => ptr2
      ptr2 => ptr2%next
    ENDIF
  ENDDO
!
  ! delete node if found
  IF (found) THEN
    ptr1%next => ptr2%next
    DEALLOCATE (ptr2)
  ENDIF
END SUBROUTINE delete
```

**PROGRAM main**

```fortran
PROGRAM main
  lst = new ()
  CALL insert (lst,83)
  CALL insert (lst,14)
  CALL insert (lst,17)
  CALL delete (lst,17)
END PROGRAM main
```

The call to delete does this:

- The list `lst` initially contains the nodes with values 83, 14, and 17.
- The call to `delete (lst,17)` searches for the node with value 17.
- When the node with value 17 is found, the next node in the list takes its place.
- The node with value 17 is then deallocated.

![Diagram of list deletion](diagram.png)
Next we write the code to **print** the linked list

```fortran
SUBROUTINE print_list (lst)

  TYPE (list) :: lst
  TYPE (node), POINTER :: ptr

  ptr => lst%first%next

  DO
    IF (.NOT.ASSOCIATED (ptr)) EXIT
      PRINT *, ptr%value
      ptr => ptr%next
    ENDIF
  ENDDO

END SUBROUTINE print_list
```
Binary Trees

✦ Storing data is linked list requires $n^2$ operations where $n$ is the number of pieces of data.
✦ Storing data in the binary tree only requires $n \log_2 n$ operations.
Next we write the code to create a new binary tree

```fortran
PROGRAM main
    NULLIFY (tree)
    CALL insert (tree, 83)
    CALL insert (tree, 14)
    CALL insert (tree, 17)
    CALL insert (tree, 91)
    CALL insert (tree, 11)
END PROGRAM main
```

RECURSIVE SUBROUTINE insert (tree, number)
TYPE (node) :: tree
INTEGER :: number

IF (.NOT.ASSOCIATED (tree)) THEN
    ALLOCATE (tree)
    tree%value = number
    NULLIFY (tree%left)
    NULLIFY (tree%right)
ELSE IF (number < tree%value) THEN
    CALL insert (tree%left, number)
ELSE
    CALL insert (tree%right, number)
ENDIF

END SUBROUTINE insert
```
Next we write the code to **create a new binary tree**

```fortran
TYPE node
   INTEGER :: value
   TYPE (node),POINTER :: left,right
END TYPE node
TYPE (node),POINTER :: tree

PROGRAM main
  NULLIFY (tree)
  CALL insert (tree,83)
  CALL insert (tree,14)
  CALL insert (tree,17)
  CALL insert (tree,91)
  CALL insert (tree,11)
END PROGRAM main
```

```fortran
RECURSIVE SUBROUTINE insert (tree,number)
  TYPE (node) :: tree
  INTEGER :: number
  IF (.NOT.ASSOCIATED (tree)) THEN
    ALLOCATE (tree)
    tree%value = number
    NULLIFY (tree%left)
    NULLIFY (tree%right)
  ELSE IF (number < tree%value) THEN
    CALL insert (tree%left,number)
  ELSE
    CALL insert (tree%right,number)
  ENDIF
END SUBROUTINE insert
```

Tuesday, March 10, 2009
Next we write the code to **create a new binary tree**.

```fortran
TYPE node
   INTEGER :: value
   TYPE (node),POINTER :: left,right
END TYPE node

TYPE (node),POINTER :: tree

PROGRAM main
   NULLIFY (tree)
   CALL insert (tree,83)
   CALL insert (tree,14)
   CALL insert (tree,17)
   CALL insert (tree,91)
   CALL insert (tree,11)
END PROGRAM main

RECURSIVE SUBROUTINE insert (tree,number)
   TYPE (node) :: tree
   INTEGER :: number

   IF (.NOT.ASSOCIATED (tree)) THEN
      ALLOCATE (tree)
      tree%value = number
      NULLIFY (tree%left)
      NULLIFY (tree%right)
   ELSE IF (number < tree%value) THEN
      CALL insert (tree%left,number)
   ELSE
      CALL insert (tree%right,number)
   ENDIF

END SUBROUTINE insert
```

Tuesday, March 10, 2009
Next we write the code to create a new binary tree.

```fortran
PROGRAM main
  NULLIFY (tree)
  CALL insert (tree, 83)
  CALL insert (tree, 14)
  CALL insert (tree, 17)
  CALL insert (tree, 91)
  CALL insert (tree, 11)
END PROGRAM main
```

```
RECURSIVE SUBROUTINE insert (tree, number)
  TYPE (node) :: tree
  INTEGER :: number

  IF (.NOT.ASSOCIATED (tree)) THEN
    ALLOCATE (tree)
    tree%value = number
    NULLIFY (tree%left)
    NULLIFY (tree%right)
  ELSE IF (number < tree%value) THEN
    CALL insert (tree%left, number)
  ELSE
    CALL insert (tree%right, number)
  ENDIF
END SUBROUTINE insert
```

<table>
<thead>
<tr>
<th>83</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
</tr>
</tbody>
</table>
|    | tree

Tuesday, March 10, 2009
Next we write the code to *create a new binary tree*

```fortran
PROGRAM main
   NULLIFY (tree)
   CALL insert (tree,83)
   CALL insert (tree,14)
   CALL insert (tree,17)
   CALL insert (tree,91)
   CALL insert (tree,11)
END PROGRAM main
```

```fortran
TYPE node
   INTEGER :: value
   TYPE (node),POINTER :: left,right
END TYPE node

TYPE (node),POINTER :: tree

RECURSIVE SUBROUTINE insert (tree,number)
   TYPE (node) :: tree
   INTEGER :: number

   IF (.NOT.ASSOCIATED (tree)) THEN
      ALLOCATE (tree)
      tree%value = number
      NULLIFY (tree%left)
      NULLIFY (tree%right)
   ELSE IF (number < tree%value) THEN
      CALL insert (tree%left,number)
   ELSE
      CALL insert (tree%right,number)
   ENDIF

END SUBROUTINE insert
```

```
83
  14
  17
```
Next we write the code to create a new binary tree

```fortran
PROGRAM main
 
NULLIFY (tree)
CALL insert (tree,83)
CALL insert (tree,14)
CALL insert (tree,17)
CALL insert (tree,91)
CALL insert (tree,11)
END PROGRAM main
```

```fortran
TYPE node
   INTEGER :: value
   TYPE (node),POINTER :: left,right
END TYPE node
TYPE (node),POINTER :: tree

RECURSIVE SUBROUTINE insert (tree,number)
   TYPE (node) :: tree
   INTEGER :: number
   
   IF (.NOT.ASSOCIATED (tree)) THEN
     ALLOCATE (tree)
     tree%value = number
     NULLIFY (tree%left)
     NULLIFY (tree%right)
   ELSE IF (number < tree%value) THEN
     CALL insert (tree%left,number)
   ELSE
     CALL insert (tree%right,number)
   ENDIF
   END SUBROUTINE insert
```

Tuesday, March 10, 2009
Next we write the code to create a new binary tree

```fortran
TYPE node
   INTEGER :: value
   TYPE (node), POINTER :: left, right
END TYPE node

TYPE (node), POINTER :: tree

PROGRAM main

   NULLIFY (tree)
   CALL insert (tree, 83)
   CALL insert (tree, 14)
   CALL insert (tree, 17)
   CALL insert (tree, 91)
   CALL insert (tree, 11)

END PROGRAM main

RECURSIVE SUBROUTINE insert (tree, number)
   TYPE (node) :: tree
   INTEGER :: number

   IF (.NOT.ASSOCIATED (tree)) THEN
      ALLOCATE (tree)
      tree%value = number
      NULLIFY (tree%left)
      NULLIFY (tree%right)
   ELSE IF (number < tree%value) THEN
      CALL insert (tree%left, number)
   ELSE
      CALL insert (tree%right, number)
   ENDIF

END SUBROUTINE insert
```

---

Tuesday, March 10, 2009