Data Structures

In this topic

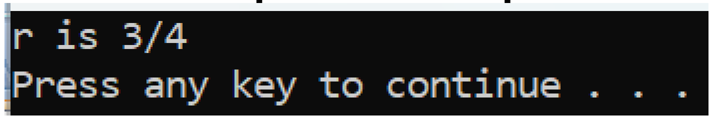
- C/C++ structures
- Member variables
- Objects and data encapsulation
- Operator overloading
- Linked Lists
- Data Structures

C/C++ Structures

- Consider the following C++ program and its required output

```cpp
#include <iostream>
using namespace std;
int main()
{
    //This program is designed to work with RationalNumber data type values
    //A rational number value is a number of the form a/b where a and b
    //are integer numbers with the condition that b is not equal to 0

    //Given a rational number r = a/b, then a is called the numerator of
    //the rational number and b is called the denominator of the rational number.

    RationalNumber r;

    //Set the value of r to 3/4  (We don't know how yet!)

    //Print the value of r (We don't know how yet!)

    system("Pause");
    return 0;
}
```

Required Output



```
r is 3/4
Press any key to continue . . .
```

C/C++ Structures

- Obviously, the program has a syntax error because there is no **`RationalNumber`** data type in C++ programming language
- This is where the programmer defined data types come to play a role
- C/C++ programming languages allow us to create our own data types and use these data types in our programs in the same way we use the C++ primitive data types such as **`int, float, double, bool, etc`**

C/C++ Structures

- In C/C++ programming languages, **structs** (**read as structures**) help us to create our own data types by grouping related data into one named unit called a **struct**
- The name of the struct then becomes a data type
- Consider for example rational number
- A rational number is made up of two related data; namely the numerator and the denominator of the rational number
- Thus structs help us to group the numerator and the denominator values into a struct and then use the name of the struct as a data type to represent rational number values

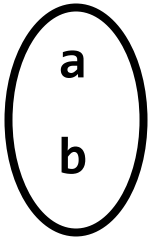
struct Declaration

- The syntax to declare (create) a struct is as follows

```
struct name_of_struct
{
        data type variable_name;
        data_type variable_name;
        ⋮
};
```

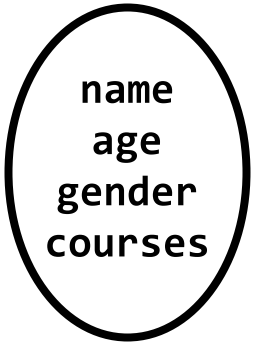
- For example, in order to create a data type for rational number values, we would proceed as follows

```
struct RationalNumber
{
        int a;  //The numerator value
        int b;  //The denominator value
};
```



- As anther example, in order to create a data type for student values, we could proceed as follows

```
struct Student
{
        string name;
        int age;
        char gender;
        vector<string> courses;
};
```



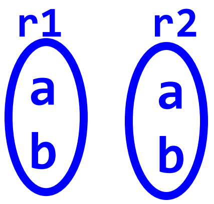
Member Variables

- The related data inside a struct are known as the **`member variables`** of the struct
- A struct can have as many member variables of the same or different data types as we wish
- The data type of the member variables can be any of the C++ primitive data types such as int, float, double, char, bool, etc or a programmer defined data type (a struct) that is already declared before using it
- Often we put the struct declaration at the top of our program just below the include directives and namespaces
- This way we can use the struct as a data type anywhere in our main program or in our functions

Member Variables

- We use a ***struct*** as a data type in the same we use the C++ primitive data types
- For example, we may declare variables to store rational numbers as follows

```
    RationalNumber r1, r2;
```



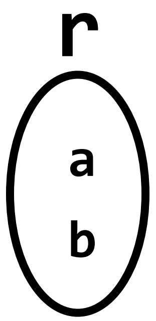
- Now each of the variables **r1** and **r2** has two member variables namely **a** and **b** that represent respectively the numerator and the denominator of the rational number

Member Variables

- Given a struct variable declaration such as

```
RationalNumber r;
```

- We access the member variables of **r** using a **dot** operator
- Thus
  - **r.a** accesses the numerator of **r** and
  - **r.b** accesses the denominator of **r**

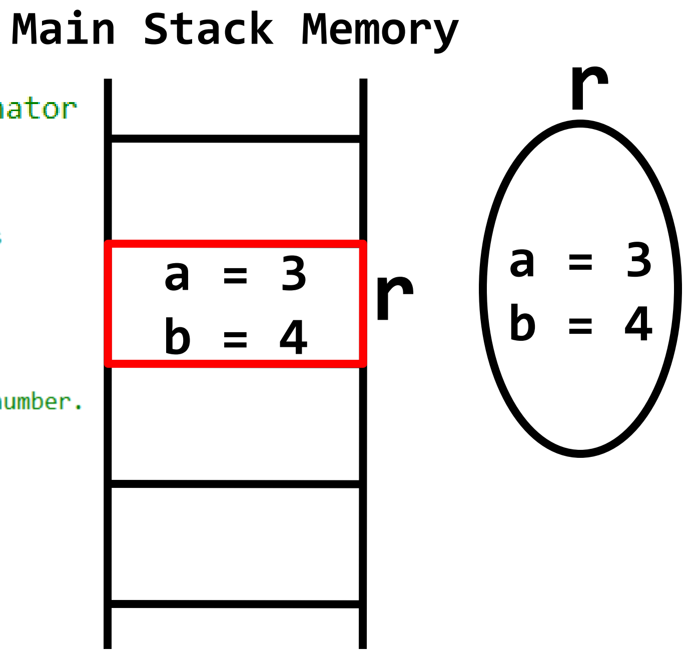


- Thus in order to assign the variable r the rational number value ¾, we proceed as follows
  - `r.a = 3;`
  - `r.b = 4;`

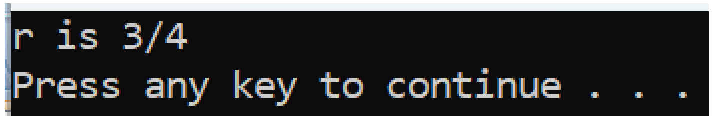
Example

- We may now therefore present a working program of the example shown earlier as follows. A memory diagram is also shown for clarity purposes

```
#include <iostream>
using namespace std;
struct RationalNumber
{
    int a, b;       //a is the numerator and b is the denominator
};
int main()
{
    //This program is designed to work with RationalNumber data type values
    //A rational number value is a number of the form a/b where a and b
    //are integer numbers with the condition that b is not equal to 0

    //Given a rational number r = a/b, then a is called the numerator of
    //the rational number and b is called the denominator of the rational number.

    RationalNumber r;

    //Set the value of r to 3/4
    r.a = 3;
    r.b = 4;

    //Print the value of r
    cout << "r is " << r.a << "/" << r.b << endl;

    system("Pause");
    return 0;
}
```



Output


```
r is 3/4
Press any key to continue . . .
```

Copying and Assigning Objects

- We can use the assignment operator with rational number data types in exactly the same way we use it with C++ primitive data types
- For example, given a rational number r1 already initialized, the statement

```
r2 = r1;
```

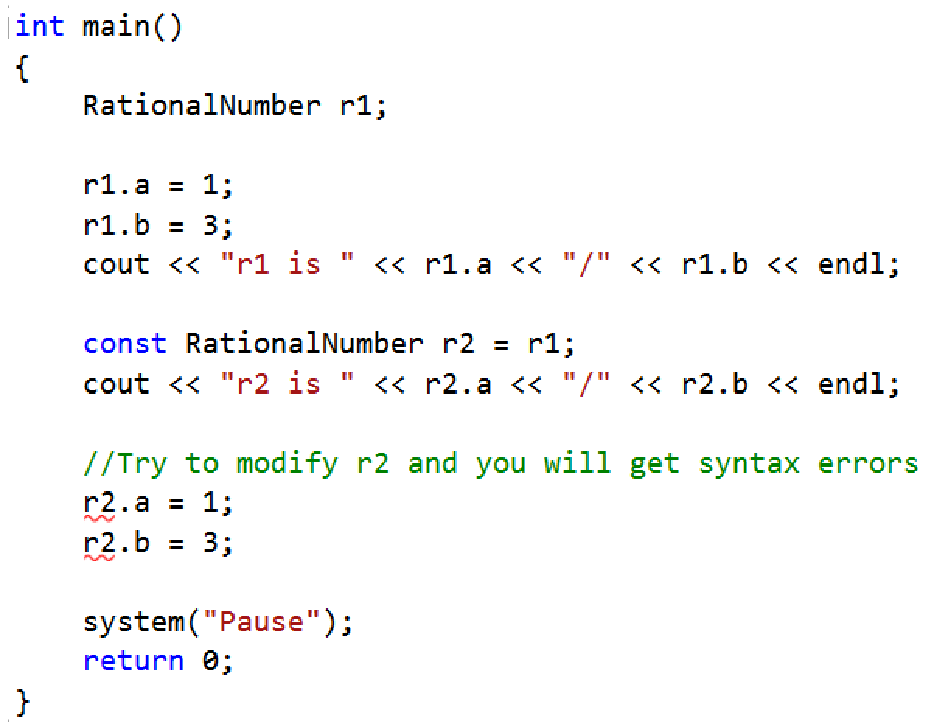
  assigns a copy of the value of the right hand side variable (i.e. r1) to the left hand side variable (i.e. r2)
- That is, copies of the values of the member variables of r1 will be assigned to their corresponding member variables of r2
- We can also define rational number data type variables in the same way we do with C++ primitive data types. See the example below…

Copying and Assigning Objects

```
int main()
{
    RationalNumber r1;
    r1.a = 3;
    r1.b = 4;
    cout << "r1 is " << r1.a << "/" << r1.b << endl;

    RationalNumber r2;
    r2 = r1;    //Assignment operator
    cout << "r2 is " << r2.a << "/" << r2.b << endl;

    RationalNumber r3 = r1; //Variable definition
    cout << "r3 is " << r3.a << "/" << r3.b << endl;

    RationalNumber r4(r2);  //Variable definition
    cout << "r4 is " << r4.a << "/" << r4.b << endl;

    system("Pause");
    return 0;
}
```

Constant Qualifiers

- We may also define constant rational number data type variables
- When a variable is made constant, then its value can NOT be modified
- In the case of rational numbers, this means neither the numerator nor the denominator values can be modified

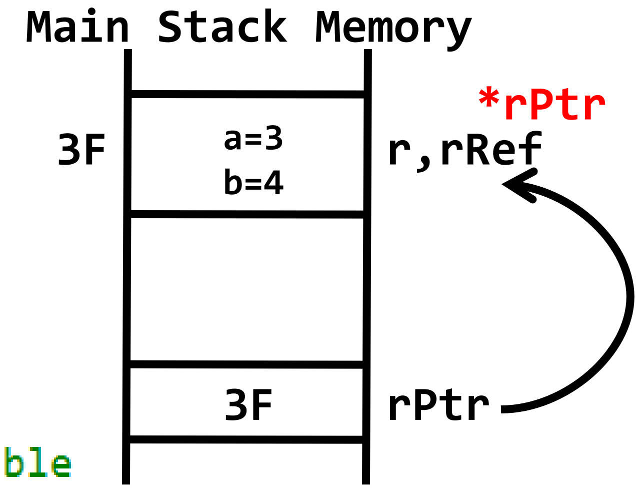


```
int main()
{
    RationalNumber r1;

    r1.a = 1;
    r1.b = 3;
    cout << "r1 is " << r1.a << "/" << r1.b << endl;

    const RationalNumber r2 = r1;
    cout << "r2 is " << r2.a << "/" << r2.b << endl;

    //Try to modify r2 and you will get syntax errors
    r2.a = 1;
    r2.b = 3;

    system("Pause");
    return 0;
}
```

Pointers and References

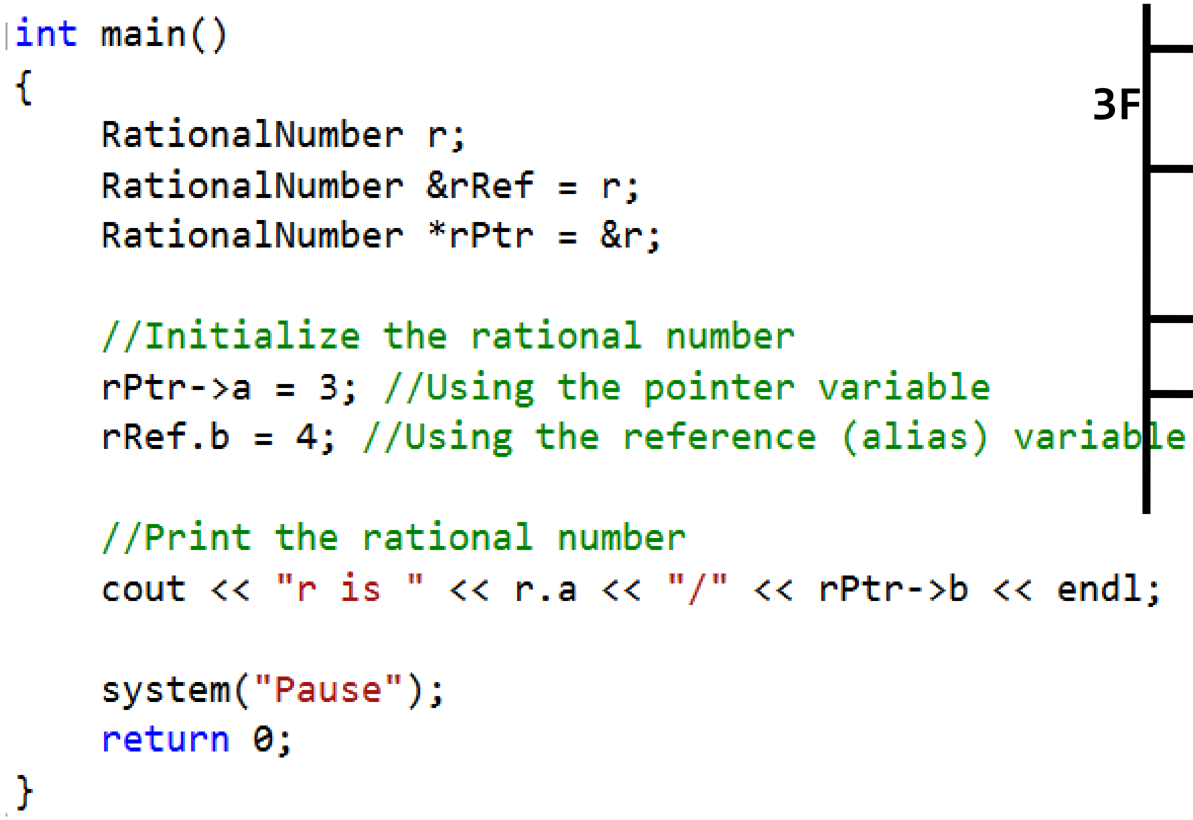
- We may also declare pointers or references to RationalNumber data types as shown below

```
int main()
{
    RationalNumber r;
    RationalNumber &rRef = r;
    RationalNumber *rPtr = &r;

    //Initialize the rational number
    (*rPtr).a = 3; //Using the pointer variable
    rRef.b = 4; //Using the reference (alias) variable

    //Print the rational number
    cout << "r is " << r.a << "/" << (*rPtr).b << endl;

    system("Pause");
    return 0;
}
```

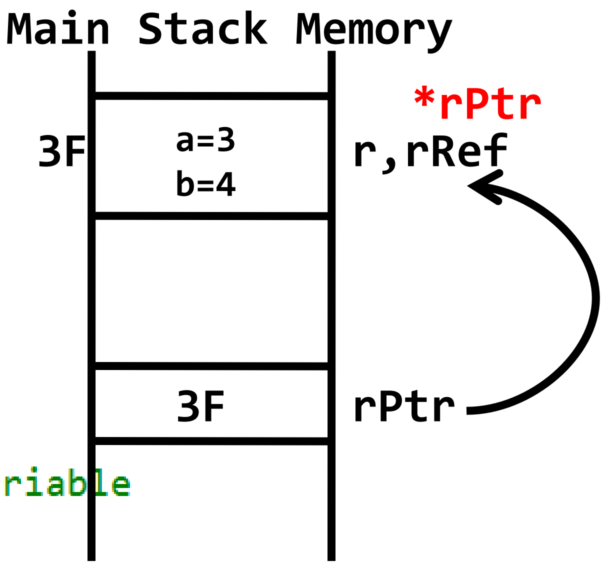


Pointers and References

- An alternative way of accessing member variables with pointers is to use the arrow operator (**->**) instead of the dereference and dot operators. See below

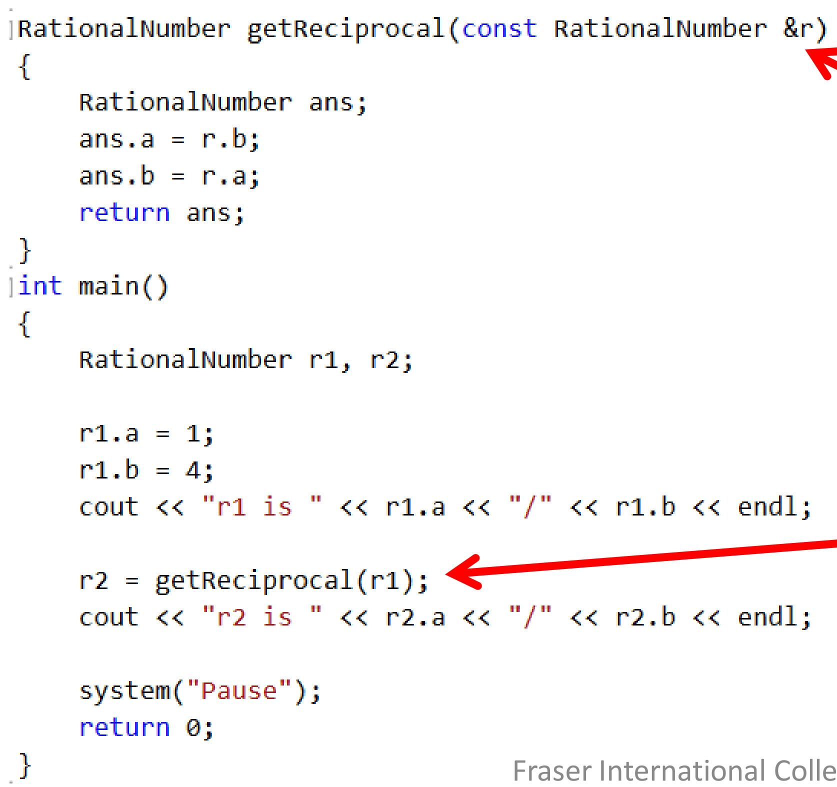


```
int main()
{
    RationalNumber r;
    RationalNumber &rRef = r;
    RationalNumber *rPtr = &r;

    //Initialize the rational number
    rPtr->a = 3; //Using the pointer variable
    rRef.b = 4; //Using the reference (alias) variable

    //Print the rational number
    cout << "r is " << r.a << "/" << rPtr->b << endl;

    system("Pause");
    return 0;
}
```



Functions

- We may also pass rational numbers to functions as arguments and return rational numbers values from functions
- We may use parameter passing by value, reference, or pointer and return by value, reference, or pointer



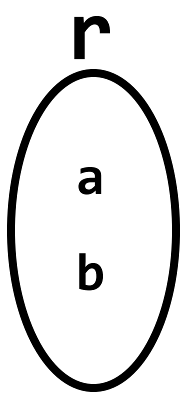
```
RationalNumber getReciprocal(const RationalNumber &r)
{
    RationalNumber ans;
    ans.a = r.b;
    ans.b = r.a;
    return ans;
}
int main()
{
    RationalNumber r1, r2;

    r1.a = 1;
    r1.b = 4;
    cout << "r1 is " << r1.a << "/" << r1.b << endl;

    r2 = getReciprocal(r1);
    cout << "r2 is " << r2.a << "/" << r2.b << endl;

    system("Pause");
    return 0;
}
```

Parameter passing by reference is the most efficient. So use it whenever you can.

For simplicity, assume that the value of the r1 argument is not a zero rational number. Why?

Objects and Data Encapsulation

- C++ structs help us to represent real world or abstract objects in our programs
- For this reason, variables of struct data type are called **objects**
- An object keeps (hides) its member variables inside it
- Thus we work with objects by accessing their member variables with a dot operator
- This process of hiding member variables inside objects is known as **data encapsulation**



Concluding Remarks

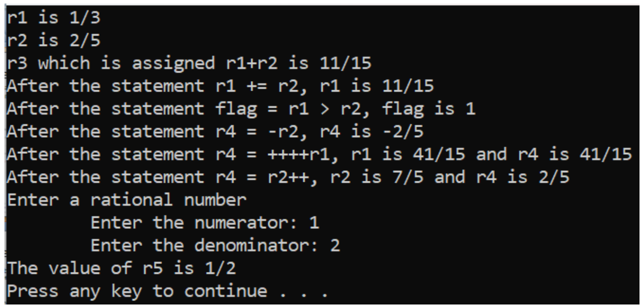
- We work with objects constructed using structs in the same way we work with C++ primitive data types
- Thus declaration or definition of objects, modification of objects, references and pointers to objects, passing objects to functions as arguments, and returning objects from functions all follow the same syntax as any C++ primitive data types
- The only difference is that the actual processing of objects requires the dot operator in order to access the member variables of the objects that are hidden inside the objects

Operators

- Consider the following program and determine the code segments that will cause syntax errors

```
int main()
{
        RationalNumber r1, r2, r3, r4, r5;
        r1.a = 1; r1.b  = 3;
        cout << "r1 is " << r1.a << "/" << r1.b << endl;
        r2.a = 2; r2.b = 5;
        cout << "r2 is " << r2.a << "/" << r2.b << endl;
        r3 = r1 + r2;
        cout << "r3 which is assigned r1+r2 is " << r3.a << "/" << r3.b << endl;
        r1 += r2;
        cout << "After the statement r1 += r2, r1 is " << r1.a << "/" << r1.b << endl;
        bool flag = r1 > r2;
        cout << "After the statement flag = r1 > r2, flag is " << flag << endl;
        r4 = -r2;
        cout << "After the statement r4 = -r2, r4 is " << r4.a << "/" << r4.b << endl;
        r4 = ++++r1;
        cout << "After the statement r4 = ++++r1, r1 is " << r1.a << "/" << r1.b;
        cout << " and r4 is " << r4.a << "/" << r4.b << endl;
        r4 = r2++;
        cout << "After the statement r4 = r2++, r2 is " << r2.a << "/" << r2.b;
        cout << " and r4 is " << r4.a << "/" << r4.b << endl;
        cout << "Enter a rational number";
        cin >> r5;
        cout << "The value of r5 is ";
        cout << r5;
        cout << endl;
        system("Pause");
        return 0;
}
```

Output



```
r1 is 1/3
r2 is 2/5
r3 which is assigned r1+r2 is 11/15
After the statement r1 += r2, r1 is 11/15
After the statement flag = r1 > r2, flag is 1
After the statement r4 = -r2, r4 is -2/5
After the statement r4 = ++++r1, r1 is 41/15 and r4 is 41/15
After the statement r4 = r2++, r2 is 7/5 and r4 is 2/5
Enter a rational number
        Enter the numerator: 1
        Enter the denominator: 2
The value of r5 is 1/2
Press any key to continue . . .
```

Operators

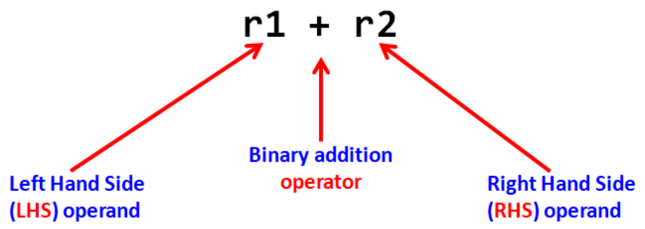
- The program has syntax errors because all the expressions **`r1+r2`**, **`r1+=r2`**, **`r1>r2`**, **`-r2`**, **`++r1`**, **`r4++`**, **`cin>>r5`**, and **`cout<<r5`** are undefined
- Why?
- Because the core of C++ programming language doesn't recognize the RationalNumber data type; after all the RationalNumber data type is a new data type that we declared by ourselves
- So while the core of C++ programming language recognizes the C++ primitive data types and can perform some operations on them, it can't perform any operations on RationalNumber objects
- Thus it will be our responsibility to define these operations for RationalNumber objects

Binary Operators

- Now consider the expression

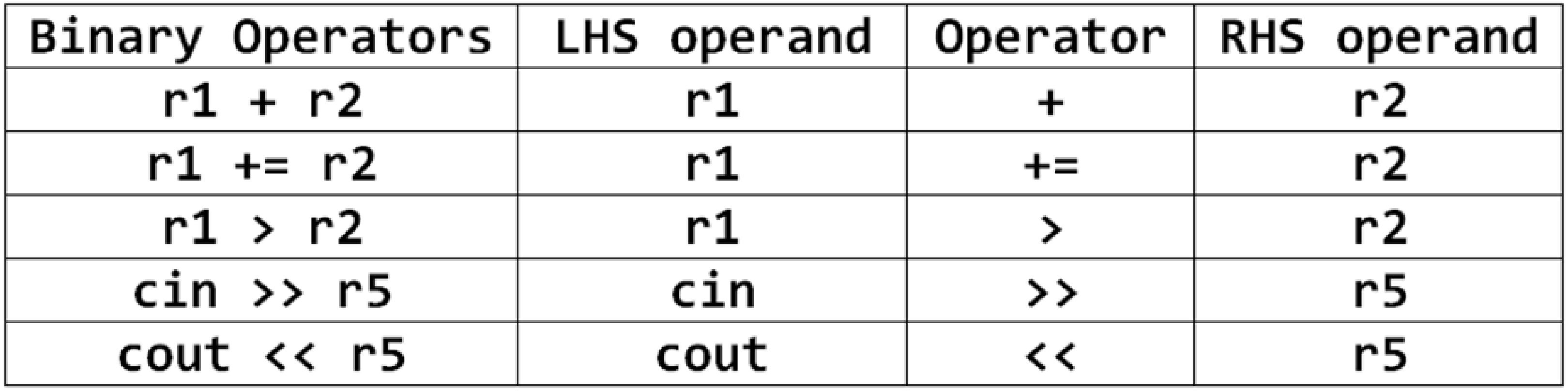
  `r1+r2`

- The expression is made up of three components; namely
  - The left hand side (LHS) operand (**`r1`**)
  - The addition operator (**+**), and
  - The right hand side (RHS) operand (**`r2`**)
- The addition operator has two operands and is known as a **binary operator**



Binary Operators

- The table below summarizes the binary operators being used in the previous program



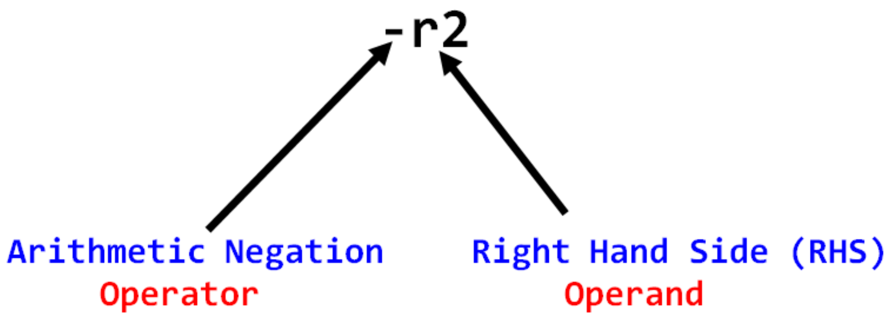
Binary Operators	LHS operand	Operator	RHS operand
r1 + r2	r1	+	r2
r1 += r2	r1	+=	r2
r1 > r2	r1	>	r2
cin >> r5	cin	>>	r5
cout << r5	cout	<<	r5

Unary Operators

- Similarly, consider the expression

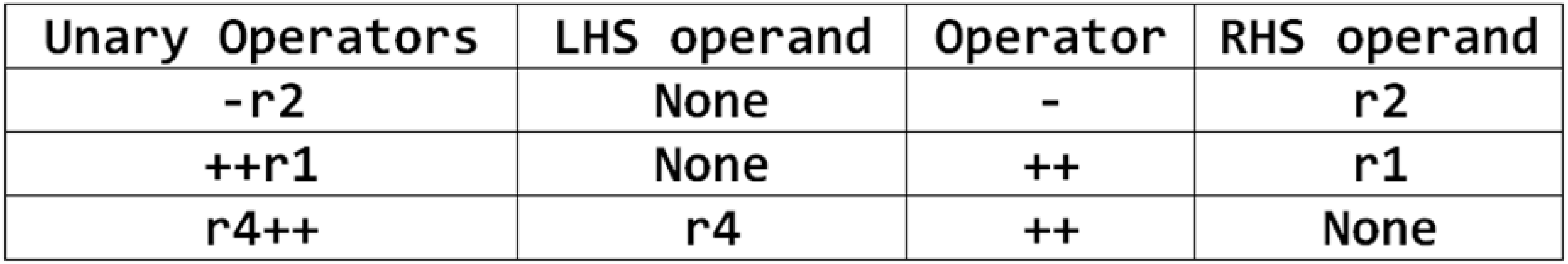
  `-r2`

- The expression is made up of two components; namely
  - ➢ The arithmetic negation operator (**-**), and
  - ➢ The right hand side (RHS) operand (**r2**)
- The arithmetic negation operator has only one operand and is known as a **unary operator**



Unary Operators

- The table below summarizes the unary operators being used in the previous program
- Observe that all unary operators with the exception of the post-increment operator have their operands on the right hand side of the operator
- Thus the post-increment operator is an anomaly in that it doesn't follow the unary operators convention



Unary Operators	LHS operand	Operator	RHS operand
-r2	None	-	r2
++r1	None	++	r1
r4++	r4	++	None

Operator overloading

- C++ defines operators for its primitive data types as functions specified with function name **`operator SYMBOL`**
- For example, the addition operator is defined by a function named **`operator +`**
- Similarly, the arithmetic negation operator is defined by a function named **`operator -`**
- The operands (or operand) of the operators are sent to the functions as arguments
- The functions return the result of the operations
- Thus in order to define the operators for our own data types such as the RationalNumber data type, all we need is to re-define the same functions for our own data types
- This is why defining operators for our own data types is known as **operator overloading**

Operator overloading

- The function declarations to overload the binary addition operator (**+**), binary compound addition operator (**+=**), binary greater than relational operator (**>**), unary arithmetic negation operator (**-**), and unary pre-increment operator (**++**) are therefore given as follows

```
//Binary operator overloading
RationalNumber operator + (const RationalNumber &LHS_op, const RationalNumber &RHS_op);
RationalNumber operator += (RationalNumber &LHS_op, const RationalNumber &RHS_op);
bool operator > (const RationalNumber &LHS_op, const RationalNumber &RHS_op);

//Unary operator overloading
RationalNumber operator - (const RationalNumber &operand); //arithmetic negation
RationalNumber& operator ++ (RationalNumber &operand); //pre-increment
```

- Observe that the unary pre-increment operator overloading should return by reference in order to allow operations such as ++++++r1

Operator overloading

- The operator **`>>`** is known as the input streaming operator
- The LHS operand of the input streaming operator is the input streaming object **`cin`**
- The data type of the cin object is **`istream`**
- Moreover the function that defines the input streaming operator returns the input streaming object cin in order to use it in the subsequent input streamings as in

```
cin >> r1 >> r2;
```

- Also we should note that a C++ program is restricted to have only one input streaming object during its execution and therefore parameter passing by reference and returning by reference must be used for the input streaming object cin
- Therefore the function declaration for the input streaming operator will be

```
istream& operator >> (istream &cin, RationalNumber &r);
```

Operator overloading

- The operator **`<<`** is known as the output streaming operator
- The LHS operand of the output streaming operator is the output streaming object **`cout`**
- The data type of the cin object is **`ostream`**
- Moreover the function that defines the output streaming operator returns the output streaming object cout in order to use it in the subsequent output streamings as in

```
cout << r1 << r2;
```

- Also we should note that a C++ program is restricted to have only one output streaming object during its execution and therefore parameter passing by reference and returning by reference must be used for the output streaming object cout
- Therefore the function declaration for the output streaming operator will be

```
ostream& operator << (ostream &cout, const RationalNumber &r);
```

Operator overloading

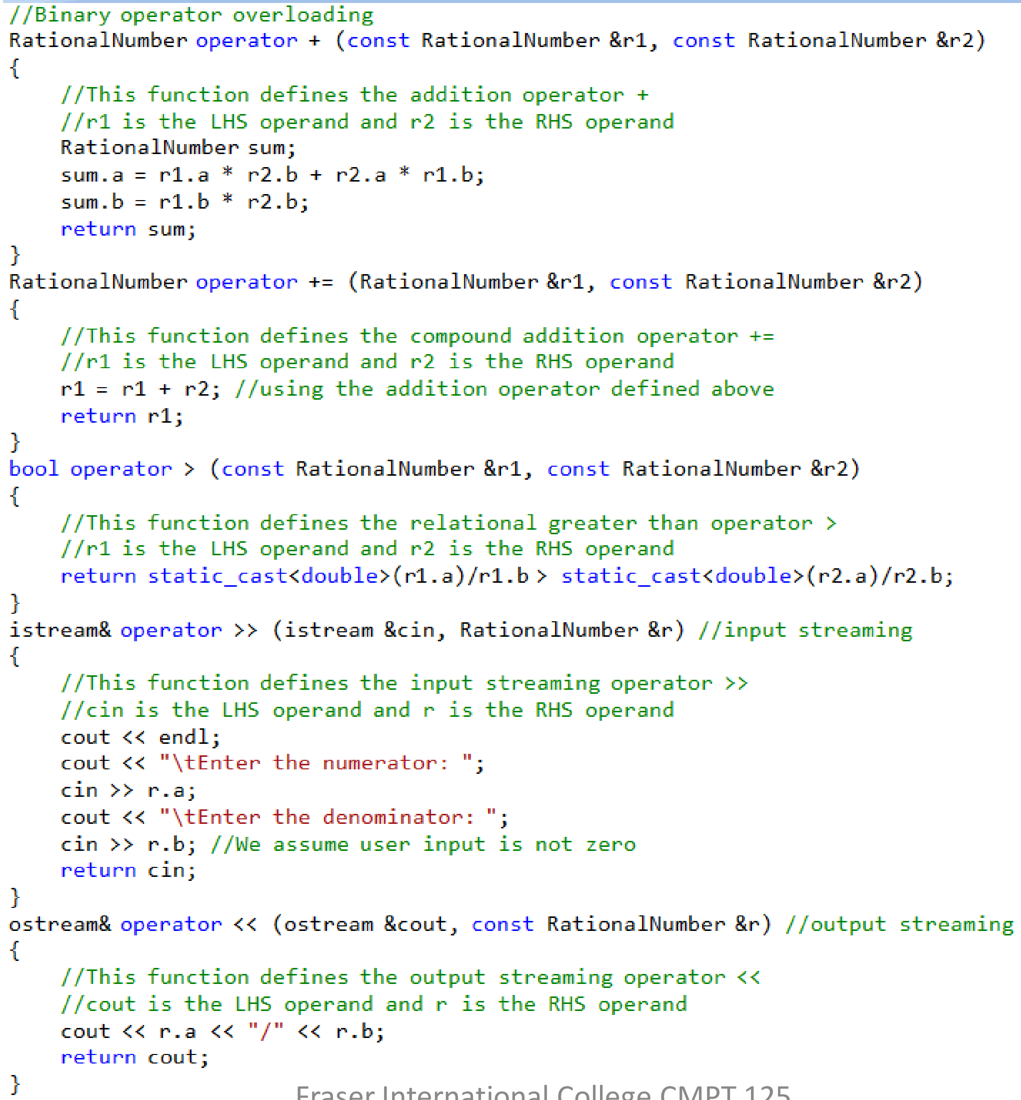
- The unary post-increment operator as in **r4++** is an exception for otherwise its operator overloading function will have identical function declaration as the pre-increment operator
- That would cause a syntax error because it will be a function re-declaration
- For that reason, C++ treats the post increment operator as if it is a binary operator with an integer data type RHS operand usually referred to as a DUMMY operand
- Therefore the function declaration for the post increment operator will be

```
RationalNumber operator ++ (RationalNumber &r, int DUMMY);
```

Operator overloading

- We can now implement these operator overloading functions in order to get the program given earlier to compile, link, and run without any errors and give semantically correct results as follows
- **Remark**
  - Considering there are many operators and possibly many data types for their operands, we will be required to define quite a lot of operator overloading functions
  - For this reason, we should make use of the operators that are already defined whenever we define more operators
  - For instance, once we define the binary addition operator +, then the compound addition operator += should make use of it
  - Similarly, once we define the == and > operators, then all the remaining relational operators such as !=, >=, <, and <= should make use of them in order to avoid code duplications

Operator overloading



```
//Binary operator overloading
RationalNumber operator + (const RationalNumber &r1, const RationalNumber &r2)
{
    //This function defines the addition operator +
    //r1 is the LHS operand and r2 is the RHS operand
    RationalNumber sum;
    sum.a = r1.a * r2.b + r2.a * r1.b;
    sum.b = r1.b * r2.b;
    return sum;
}
RationalNumber operator += (RationalNumber &r1, const RationalNumber &r2)
{
    //This function defines the compound addition operator +=
    //r1 is the LHS operand and r2 is the RHS operand
    r1 = r1 + r2; //using the addition operator defined above
    return r1;
}
bool operator > (const RationalNumber &r1, const RationalNumber &r2)
{
    //This function defines the relational greater than operator >
    //r1 is the LHS operand and r2 is the RHS operand
    return static_cast<double>(r1.a)/r1.b > static_cast<double>(r2.a)/r2.b;
}
istream& operator >> (istream &cin, RationalNumber &r) //input streaming
{
    //This function defines the input streaming operator >>
    //cin is the LHS operand and r is the RHS operand
    cout << endl;
    cout << "\tEnter the numerator: ";
    cin >> r.a;
    cout << "\tEnter the denominator: ";
    cin >> r.b; //We assume user input is not zero
    return cin;
}
ostream& operator << (ostream &cout, const RationalNumber &r) //output streaming
{
    //This function defines the output streaming operator <<
    //cout is the LHS operand and r is the RHS operand
    cout << r.a << "/" << r.b;
    return cout;
}
```

Operator overloading

```
//Unary operator overloading
RationalNumber operator - (const RationalNumber &r)
{
	//This function defines the unary arithmetic negation operator -
	//r is the operand
	RationalNumber ans;
	ans.a = -r.a;
	ans.b = r.b;
	return ans;
}
RationalNumber& operator ++ (RationalNumber &r)
{
	//This function defines the unary pre-increment operator ++
	//r is the operand

	//Create a rational number that is equal to 1
	RationalNumber temp;
	temp.a = 1;
	temp.b = 1;
	//Now increment the parameter r
	r = r + temp;
	//return the incremented value of r
	return r;
}
RationalNumber operator ++ (RationalNumber &r, int DUMMY) //post increment
{
	//This function defines the unary post-increment operator ++
	//r is the operand. DUMMY is used for signature purposes only

	//save the value of the parameter r
	RationalNumber temp = r;
	//increment the value of r
	++r;
	//return the original value of r that is stored in temp
	return temp;
}
```

Concluding Remarks

- C++ allows us to overload operators
- The most common operators that we can overload for RationalNumber objects are
  - **Arithmetic operators**

    `r1+r2, r+5, -5+r, r1-r2, r-5, -5-r, r1*r2, r*5, -5*r, r1/r2, r/5, -5/r, r1+=r2, r+=5, r1-=r2, r-=5, r1*=r2, r*=5, r1/=r2, r/=5, -r, ++r, --r, r++, r--`
  - **Comparison operators**

    `r1==r2, r==5, -5==r, r1!=r2, r!=5, -5!=r, r1>r2, r>5, -5>r, r1<r2, r<5, -5<r, r1>=r2, r>=5, -5>=r, r1<=r2, r<=5, -5<=r,`
  - **Input and output streaming operators**

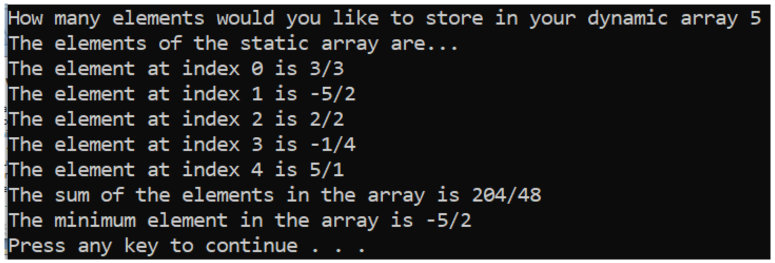
    `cout << r` and `cin >> r`

Arrays and vectors

- We may also store several objects in static arrays, dynamic arrays, or vectors in the same way we do with C++ primitive data types
- We can therefore store a fixed number of objects in a static array, user desired number of objects in a dynamic array, and unknown number of objects in a vector
- See the programs below…

Arrays and vectors

```cpp
#include <iostream>
#include <ctime>
using namespace std;

RationalNumber array_sum(const RationalNumber *A, const int size)
{
    //Declare a sum object
    RationalNumber s;
    //Initialize the sum object to zero rational number
    s.a = 0;
    s.b = 1;
    //Compute the sum of all the elements of the vector in the sum object
    for (int i = 0; i < size; i++)
        s = s + A[i];
    return s;
}
RationalNumber minimum_element(const RationalNumber *A, const int size)
{
    //Define the minimum object to the first element
    RationalNumber m = A[0];
    //Compare the elements to find the minimum
    for (int i = 1; i < size; i++)
    {
        if (m > A[i])
            m = A[i];
    }
    return m;
}
```

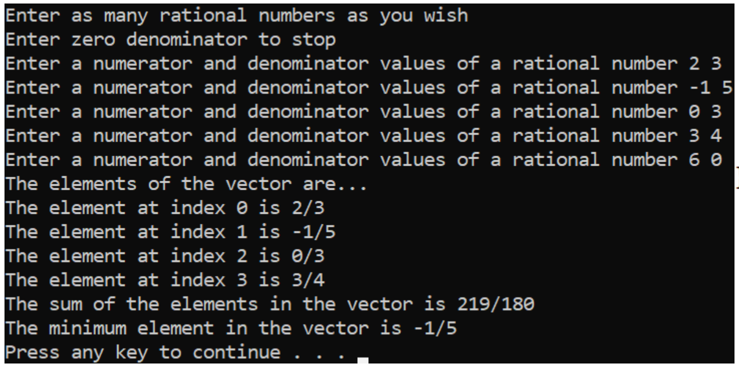


```
How many elements would you like to store in your dynamic array 5
The elements of the static array are...
The element at index 0 is 3/3
The element at index 1 is -5/2
The element at index 2 is 2/2
The element at index 3 is -1/4
The element at index 4 is 5/1
The sum of the elements in the array is 204/48
The minimum element in the array is -5/2
Press any key to continue . . .
```

```cpp
int main()
{
    int size;
    do
    {
        cout << "Enter array size: ";
        cin >> size;
    }while (size <= 0);

    RationalNumber *A = new RationalNumber[size];

    srand(time(0));
    for (int i = 0; i < size; i++)
    {
        int x = rand() % 11 - 5;
        A[i].a = x;
        x = rand() % 5 + 1;
        A[i].b = x == 0 ? 1 : x; //avoid zero denominator
    }

    cout << "The elements of the static array are..." << endl;
    for (int i = 0; i < size; i++)
        cout << "The element at index " << i << " is " << A[i] << endl;

    RationalNumber s = array_sum(A, 5);
    cout << "The sum of the elements in the array is " << s << endl;

    RationalNumber m = minimum_element(A, 5);
    cout << "The minimum element in the array is " << m << endl;

    delete[] A;

    system("Pause");
    return 0;
}
```

Arrays and vectors

```cpp
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;

RationalNumber vector_sum(const vector<RationalNumber> &v)
{
    //Declare a sum object
    RationalNumber s;
    //Initialize the sum object to zero rational number
    s.a = 0;
    s.b = 1;
    //Compute the sum of all the elements of the vector in the sum object
    for (int i = 0; i < v.size(); i++)
        s = s + v[i];
    return s;
}
RationalNumber minimum_element(const vector<RationalNumber> &v)
{
    //Define the minimum object to the first element
    RationalNumber m = v[0];
    //Compare the elements to find the minimum
    for (int i = 1; i < v.size(); i++)
    {
        if (m > v[i])
            m = v[i];
    }
    return m;
}
```

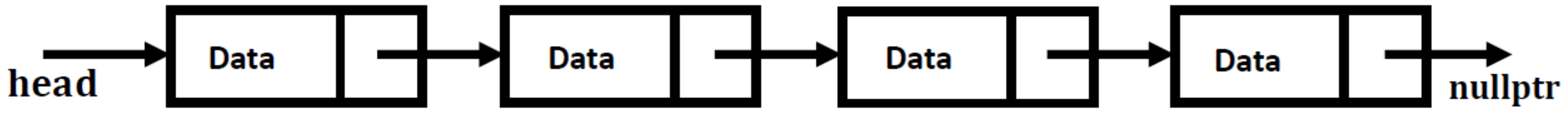


```
Enter as many rational numbers as you wish
Enter zero denominator to stop
Enter a numerator and denominator values of a rational number 2 3
Enter a numerator and denominator values of a rational number -1 5
Enter a numerator and denominator values of a rational number 0 3
Enter a numerator and denominator values of a rational number 3 4
Enter a numerator and denominator values of a rational number 6 0
The elements of the vector are...
The element at index 0 is 2/3
The element at index 1 is -1/5
The element at index 2 is 0/3
The element at index 3 is 3/4
The sum of the elements in the vector is 219/180
The minimum element in the vector is -1/5
Press any key to continue . . .
```

```cpp
int main()
{
    int size;
    vector<RationalNumber> V;

    cout << "Enter as many rational numbers as you wish" << endl;
    cout << "Enter zero denominator to stop" << endl;
    while (true)
    {
        RationalNumber r;
        cout << "Enter a numerator and denominator values of a rational number ";
        cin >> r.a >> r.b;
        if (r.b == 0)
            break;
        else
            V.push_back(r);
    }

    cout << "The elements of the vector are..." << endl;
    for (int i = 0; i < V.size(); i++)
        cout << "The element at index " << i << " is " << V[i] << endl;

    RationalNumber s = vector_sum(V);
    cout << "The sum of the elements in the vector is " << s << endl;

    RationalNumber m = minimum_element(V);
    cout << "The minimum element in the vector is " << m << endl;

    system("Pause");
    return 0;
}
```

Linked Lists

- **Definition:** A linked list is a container designed to store several objects lined up one after the other in such a way that each object is connected to the object that comes after it in the lineup
- Linked list allow inserting new objects anywhere in the linked list (thus expanding the size of the linked list) or removing existing objects from anywhere in the linked list (thus shrinking the size of the linked list)
- The diagram below shows a linked list with four objects

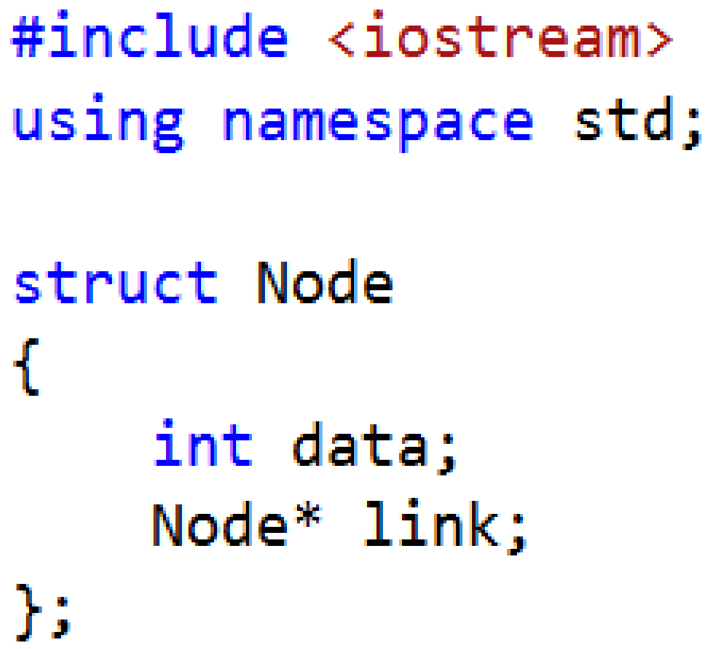


Linked Lists

- Some terminologies commonly used with linked list data structure are listed below
  - Each object in a linked list is called a **node**
  - A node has two parts: a **data part** where we store data and a **pointer part** which points to the next node in the linked list
  - The pointer part of the last node in the linked list is assigned a special pointer value that indicates the end of the linked list. A convenient value that we can easily use will be the **nullptr** value
  - A linked list also requires a pointer that points to the first node in the linked list. This pointer is usually referred to as the **head pointer** of the linked list
  - A linked list with no nodes is known as an **empty linked list**. In this case, the head pointer is assigned a **nullptr** value
  - The first node in a linked list is known as the **head node**
  - The last node in a linked list is known as the **tail node**

Linked Lists

- A Node object in a linked list can conveniently be represented with a struct
- For example, a Node object whose data part is an integer value and whose pointer part is a pointer to a Node can be represented as follows:



```
#include <iostream>
using namespace std;

struct Node
{
    int data;
    Node* link;
};
```

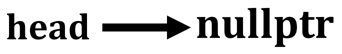
Linked Lists

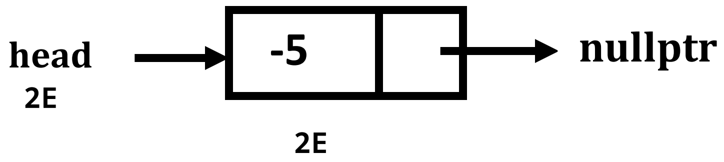
- In order to create a linked list, all we need is therefore create several nodes and connect them
- Connecting the nodes of a linked list is usually known as linking the nodes
- In order to link nodes, all we need is to assign the link member variable of each node the memory address of the node that comes after it in the linked list
- Last but not least, the nodes of a linked list are created on the heap memory so that the linked list can be expanded (by inserting more nodes into it) or shrunk (by removing some nodes from it) during program execution time
- See below…

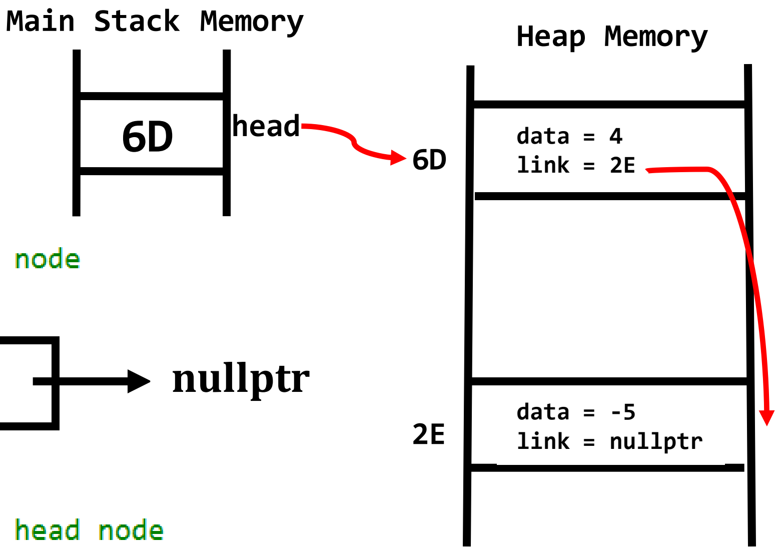
Creating Linked Lists

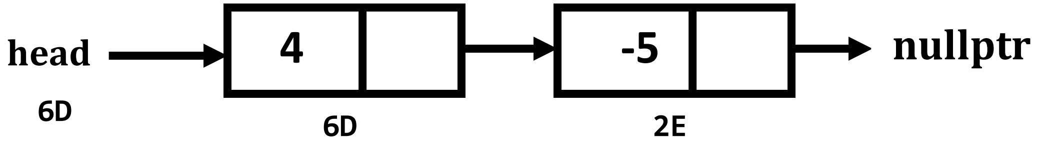
- We can now create a linked list starting from an empty linked list and then inserting Nodes into the linked list as follows

```
int main()
{
    //Create an empty linked list
    Node* head = nullptr;

    //Insert a node whose data is -5 as a head node
    Node* p;
    p = new Node();
    p->data = -5;
    p->link = head;
    head = p;

    //Insert another node whose data is 4 as a head node
    p = new Node();
    p->data = 4;
    p->link = head;
    head = p;

    system("Pause");
    return 0;
}
```







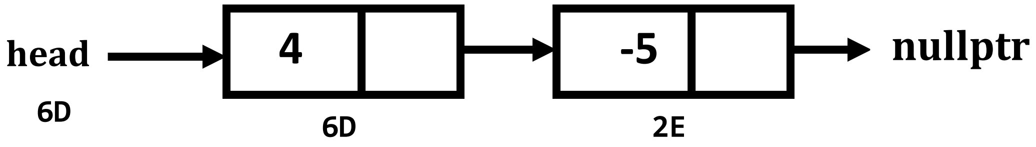


Linked List Traversal

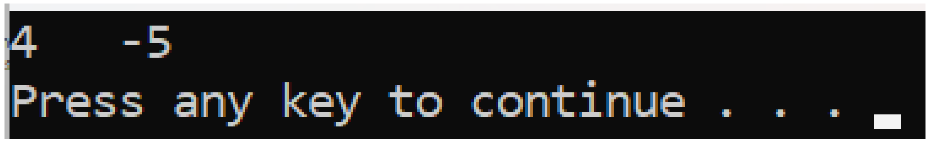
- In order to go through (i.e. traverse) through the nodes of a linked list, we need to start from the head pointer and then go through the nodes in the linked list one after the other
- **Thus a linked list allows ONLY sequential access of elements**
- That is, the only way to access a particular node in a linked list is to start at the head pointer and then follow the links in the linked list in order to reach the node of our interest
- **This means the whole linked list is valid if and only if the head pointer is pointing to the head node of the linked list; otherwise the linked list will be corrupted because there will be no way to access the nodes in the linked list**
- Thus in order to print the data part of the nodes in the linked list, we need to start from the head pointer and then traverse the linked list sequentially as shown below

Linked List Traversal

```cpp
int main()
{
    //Create an empty linked list
    Node* head = nullptr;

    //Insert a node whose data is -5 as a head node
    Node* p;
    p = new Node();
    p->data = -5;
    p->link = head;
    head = p;

    //Insert another node whose data is 4 as a head node
    p = new Node();
    p->data = 4;
    p->link = head;
    head = p;

    //Print the linked list
    while (head != nullptr)
    {
        cout << head->data << "   ";
        head = head->link;
    }
    cout << endl;

    system("Pause");
    return 0;
}
```



Output



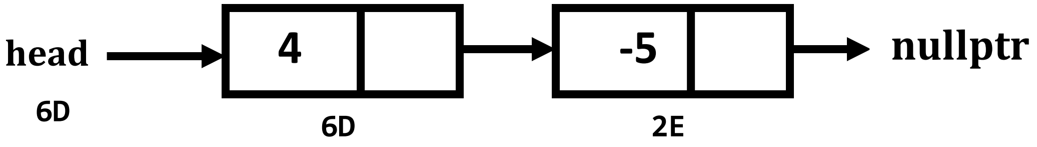
```
4   -5
Press any key to continue . . .
```

Linked List Traversal

- Observe that our linked list will be **corrupted** at the end of the code fragment we have written in order to print the linked list in the previous example
- This is because the head pointer was assigned the value of the link member variable (which is the memory address of the next node in the linked list) at each iteration
- This means the head pointer will be assigned a nullptr value at the end of the loop
- Thus we have now lost the linked list (i.e. all the nodes in the linked list) because the head pointer is no more pointing to the head node of the linked list
- In order to prevent this corruption of our linked list, we need to avoid modifying the head pointer
- Thus we correct the printing loop as follows

Linked List Traversal

```
int main()
{
    //Create an empty linked list
    Node* head = nullptr;


    //Insert a node whose data is -5 as a head node
    Node* p;
    p = new Node();
    p->data = -5;
    p->link = head;
    head = p;

    //Insert another node whose data is 4 as a head node
    p = new Node();
    p->data = 4;
    p->link = head;
    head = p;

    //Print the linked list
    Node* temp = head;
    while (temp != nullptr)
    {
        cout << temp->data << "    ";
        temp = temp->link;
    }
    cout << endl;

    system("Pause");
    return 0;
}
```



By copying the value of the head pointer (which is the memory address of the head node) to a temporary pointer and looping using the temporary pointer, we guarantee that the head pointer will still point to the head node of the linked at the end of the loop.

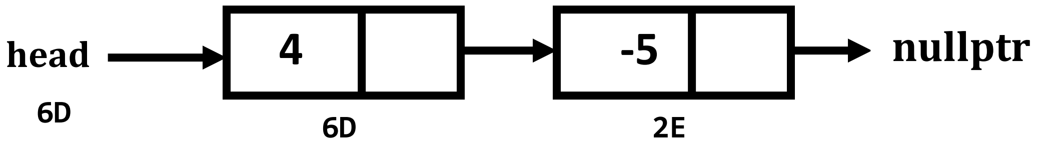
This way we will still have access to the head node of the linked list using the head pointer at the end ofn the loop and can continue to work with the linked list.

Linked List Traversal

- Observe that the nodes in the previous program are stored in the heap memory
- This means they will not be cleared from the heap memory after the program has finished execution
- **This means the program has memory leak**
- Therefore we need to clear all the nodes of the linked list using the delete operator in order to avoid any memory leak in the program
- The same program modified to clear all the nodes at the end of the program is shown below

Linked List Traversal

```cpp
int main()
{
    //Create an empty linked list
    Node* head = nullptr;
    //Insert a node whose data is -5 as a head node
    Node* p;
    p = new Node();
    p->data = -5;
    p->link = head;
    head = p;
    //Insert another node whose data is 4 as a head node
    p = new Node();
    p->data = 4;
    p->link = head;
    head = p;

    //Print the linked list
    Node* temp = head;
    while (temp != nullptr)
    {
        cout << temp->data << "    ";
        temp = temp->link;
    }
    cout << endl;

    //Delete the linked list
    while (head != nullptr)
    {
        Node* temp = head;
        head = head->link;
        delete temp;
    }
    system("Pause");
    return 0;
}
```



Processing Linked Lists

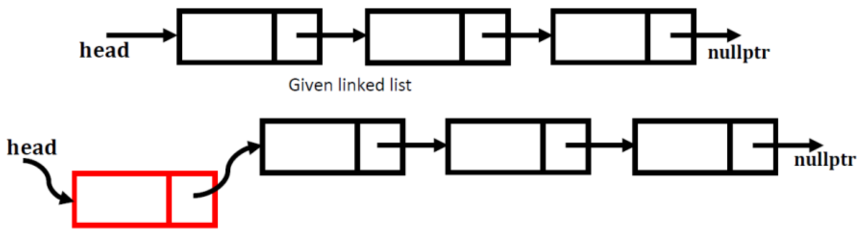
- Often linked lists are processed by defining functions for the common tasks we perform on linked lists such as
  - Inserting nodes into linked lists
  - Printing linked lists
  - Searching for a node in a linked list
  - Removing a node from a linked list
  - etc
- We demonstrate such functions in the next sections

Head Insert Algorithm

- Inserting a new node before the head node of a linked list so that the new node becomes the head node of the linked list is known to as **head insert** algorithm
- We define a function named **`head_insert`** in order to implement the head insert algorithm
- Moreover, we also define functions to print linked lists as well as to delete linked lists from the heap memory
- We also use **`typedef`** and assert statements to make our code readable and easy to debug
- The following program demonstrates performing head insert operations, printing linked lists, and deleting linked lists using functions

Head Insert Algorithm

```
#include <iostream>
using namespace std;
struct Node
{
    int data;
    Node* link;
};
typedef Node* NodePtr;

void head_insert(NodePtr &head, const int &data_value)
{
    //Pre-condition: head is the head of a linked list
    //Post-condition: A new node is inserted into the linked list before the head node
    //Remark: Parameter passing by reference must be used in this function
    NodePtr p = new Node();
    p->data = data_value;
    p->link = head; //Now p is pointing to the head node
    head = p; //Now head is pointing to the node pointed by p
}
```



Head insert algorithm

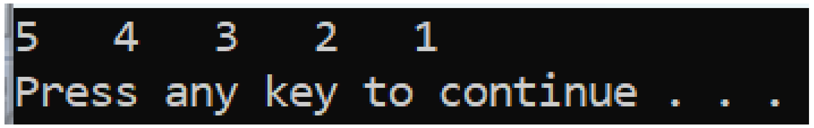
Head Insert Algorithm

```
void print_linked_list(const NodePtr &head)
{
    //Remark: Parameter passing by value can be used in this function
    NodePtr temp = head;
    while (temp != nullptr)
    {
        cout << temp->data << "   ";
        temp = temp->link;
    }
    cout << endl;
}

void delete_linked_list(NodePtr &head)
{
    //Pre-condition: head is the head of a linked list
    //Post-condition: All the nodes in the linked list are deleted from the heap memory
    //               But the head pointer must NOT be modified in order not to corrupt the linked list
    //Remark: Parameter passing by reference must be used in tbis function
    while (head != nullptr)
    {
        NodePtr temp = head; //Now temp is pointing to the head node (first node)
        head = head->link; //Now head is pointing to the second node
        delete temp; //Now the head node is deleted
    }
}
```

Head Insert Algorithm

```
int main()
{
    //Create an empty linked list
    NodePtr head = nullptr;

    //Insert several nodes using head_insert
    for (int i = 0; i < 5; i++)
    {
        int x = i+1;
        head_insert(head, x);
    }

    //Print the linked list
    cout << "The linked list is: ";
    print_linked_list(head);

    //Delete the linked list (i.e. all the nodes in the linked list)
    delete_linked_list(head);

    system("Pause");
    return 0;
}
```

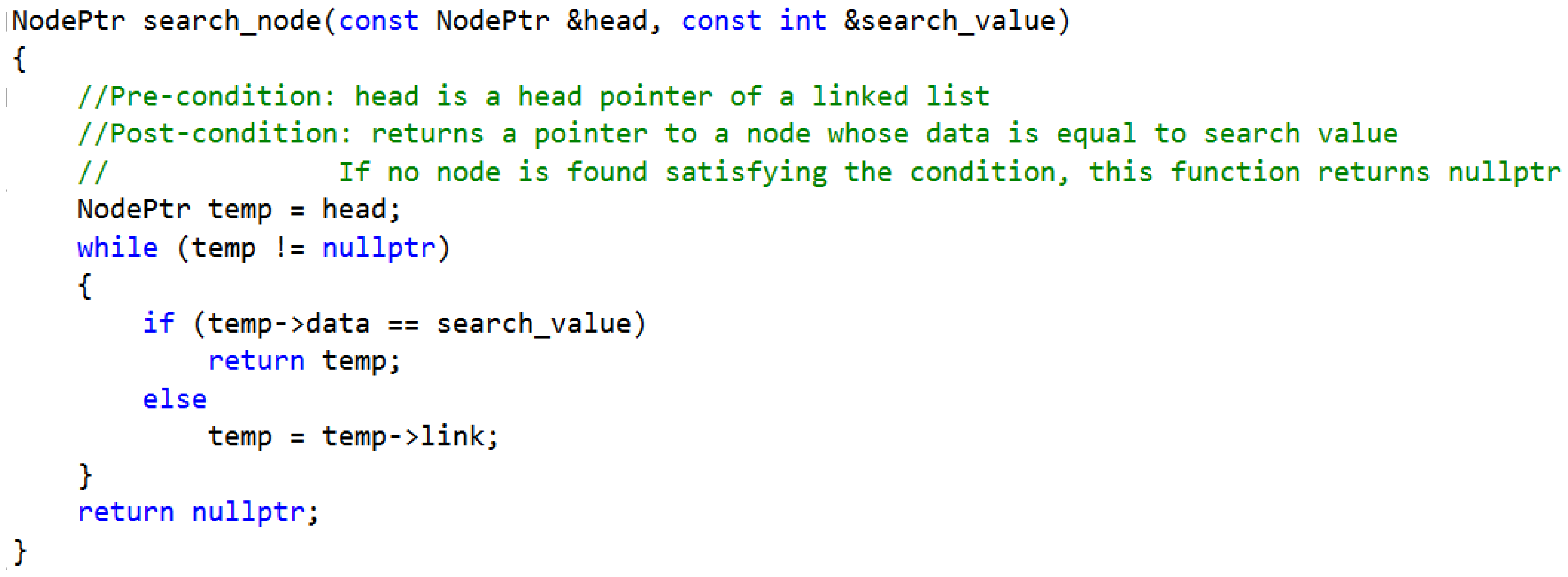
Output



```
5    4    3    2    1
Press any key to continue . . .
```

Searching in Linked Lists

- Searching for a particular node in a linked list requires traversing the nodes in the linked list sequentially starting from the head node until the required node is found or the end of the linked list is reached

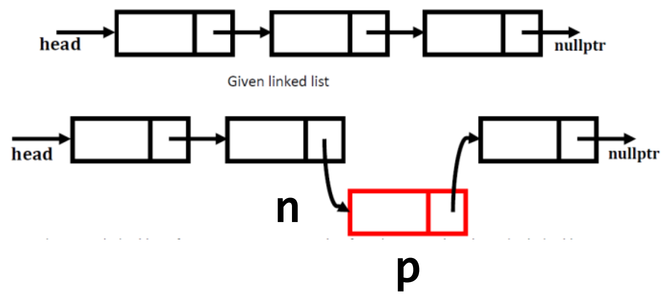


```cpp
NodePtr search_node(const NodePtr &head, const int &search_value)
{
    //Pre-condition: head is a head pointer of a linked list
    //Post-condition: returns a pointer to a node whose data is equal to search value
    //              If no node is found satisfying the condition, this function returns nullptr
    NodePtr temp = head;
    while (temp != nullptr)
    {
        if (temp->data == search_value)
            return temp;
        else
            temp = temp->link;
    }
    return nullptr;
}
```

Insert After Algorithm

- We may also insert a node after a particular node in a linked list
- We call it the **insert after** algorithm

```
void insert_after(const NodePtr &head, const int &search_value, const int &data_value)
{
    //Pre-condition: head is the head pointer of a linked list
    //Post-condition: inserts a new node whose data is the data value argument
    //              after a node whose data is the search value argument
    NodePtr n = search_node(head, search_value);
    if (n == nullptr)
        return; //There is no node matching the search value
    else
    {
        NodePtr p = new Node();
        p->data = data_value;
        p->link = n->link;
        n->link = p;
    }
}
```



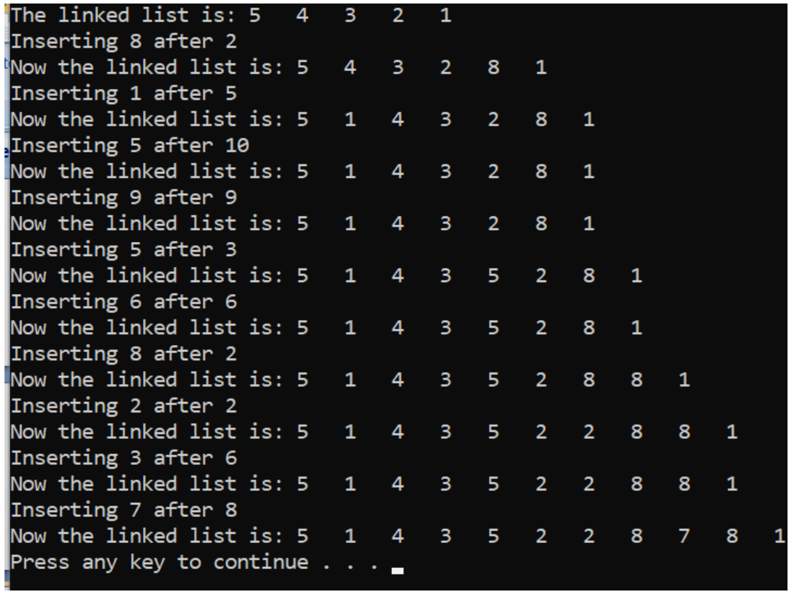
Insert after algorithm

Example

- A test program for the **`search_node`** and **`insert_after`** functions is shown below

```cpp
int main()
{
        //Create an empty linked list
        NodePtr head = nullptr;

        //Insert several nodes using head_insert
        for (int i = 0; i < 5; i++)
        {
                int x = i+1;
                head_insert(head, x);
        }

        //Print the linked list
        cout << "The linked list is: ";
        print_linked_list(head);

        //Now insert several nodes using insert_after function
        for (int i = 0; i < 10; i++)
        {
                int search_value = rand() % 10 + 1;
                int data_value = rand() % 10 + 1;
                cout << "Inserting " << data_value << " after " << search_value << endl;
                insert_after(head, search_value, data_value);
                cout << "Now the linked list is: ";
                print_linked_list(head);
        }

        //Delete the linked list (i.e. all the nodes in the linked list)
        delete_linked_list(head);
        system("Pause");
        return 0;
}
```

Output

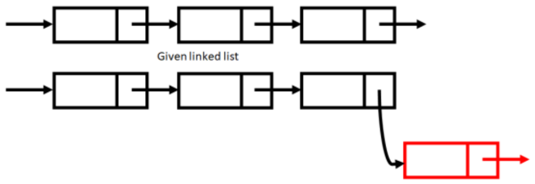


```
The linked list is: 5   4   3   2   1
Inserting 8 after 2
Now the linked list is: 5   4   3   2   8   1
Inserting 1 after 5
Now the linked list is: 5   1   4   3   2   8   1
Inserting 5 after 10
Now the linked list is: 5   1   4   3   2   8   1
Inserting 9 after 9
Now the linked list is: 5   1   4   3   2   8   1
Inserting 5 after 3
Now the linked list is: 5   1   4   3   5   2   8   1
Inserting 6 after 6
Now the linked list is: 5   1   4   3   5   2   8   1
Inserting 8 after 2
Now the linked list is: 5   1   4   3   5   2   8   8   1
Inserting 2 after 2
Now the linked list is: 5   1   4   3   5   2   2   8   8   1
Inserting 3 after 6
Now the linked list is: 5   1   4   3   5   2   2   8   8   1
Inserting 7 after 8
Now the linked list is: 5   1   4   3   5   2   2   8   7   8   1
Press any key to continue . . .
```

Tail Insert Algorithm

- We may also insert a node after a particular node in a linked list which is known as the **tail insert** algorithm

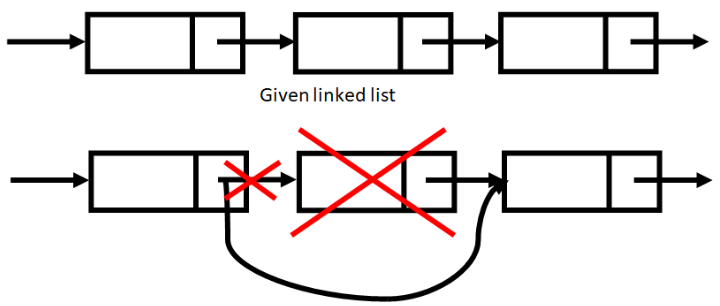
```
void tail_insert(NodePtr &head, const int &data_value)
{
    //Pre-condition: head is the head of a linked list
    //Post-condition: A new node is inserted into the linked list as a tail node
    //Remark: Parameter passing by reference must be used in this function
    if (head == nullptr) //There is no head node
        head_insert(head, data_value);
    else
    {
        NodePtr temp1 = head; //temp1 is the head node
        NodePtr temp2 = temp1->link; //temp2 is the node after temp1
        while (temp2 != nullptr)
        {
            temp1 = temp2;
            temp2 = temp2->link;
        }
        //Now temp1 is the last node. So we insert after temp1
        NodePtr p = new Node();
        p->data = data_value;
        p->link = nullptr;
        temp1->link = p;
    }
}
```



Remove Node Algorithm

- We may also remove a particular node from a linked list which is known as the **remove node** algorithm

```
bool remove_node(NodePtr &head, const int &data_value)
{
    //Pre-condition: head is the head of a linked list
    //Post-condition: A node whose data is equal to the data_value argument is removed
    //Remark: Parameter passing by reference must be used in this function
    if (head == nullptr) //The linked list is empty. So do nothing and return false
        return false;
    else if (head->data == data_value) // remove the head node
    {
        NodePtr temp = head;
        head = head->link;
        delete temp;
        return true;
    }
    else //remove a node that is not a head node
    {
        NodePtr temp1 = head; //temp1 is the head node
        NodePtr temp2 = temp1->link; //temp2 is the node after temp1
        while (temp2 != nullptr)
        {
            if (temp2->data == data_value) //remove temp2 node and return true
            {
                temp1->link = temp2->link;
                delete temp2;
                return true;
            }
            else  //Advance both temp1 and temp2
            {
                temp1 = temp2;
                temp2 = temp2->link;
            }
        }
        //temp2 has reached nullptr. There is no node to be removed. So return false
        return false;
    }
}
```



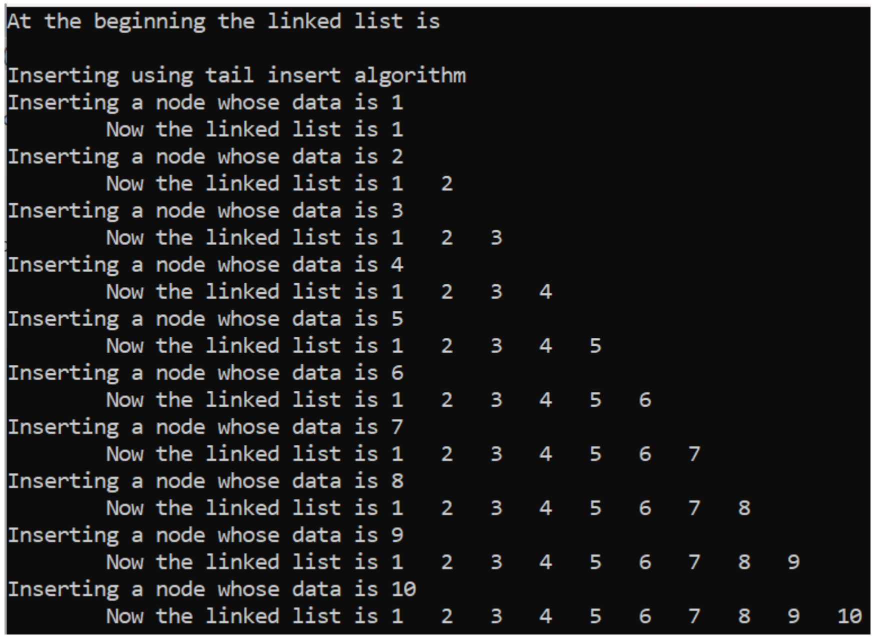
Example

```
int main()
{
    //Create an empty linked list
    NodePtr head = nullptr;
    cout << "At the beginning the linked list is ";
    print_linked_list(head);

    //Insert nodes using tail insert algorithm
    cout << endl << "Inserting using tail insert algorithm" << endl;
    for (int i = 0; i < 10; i++)
    {
        cout << "Inserting a node whose data is " << i+1 << endl;
        tail_insert(head, i+1);
        cout << "\tNow the linked list is " ;
        print_linked_list(head);
    }

    //Remove some nodes
    cout << endl << "Removing some nodes..." << endl;
    for (int i = 0; i < 10; i++)
    {
        int data_value = rand() % 10 + 1;
        cout << "Removing a node whose data is " << data_value;
        bool flag = remove_node(head, data_value);
        flag ? cout << ": Node successfully removed" << endl : cout << ": Node not found" << endl;
        cout << "\tNow the linked list is " ;
        print_linked_list(head);
    }

    //Now we don't need the linked list, so delete it
    delete_linked_list(head);
    system("Pause");
    return 0;
}
```

Example

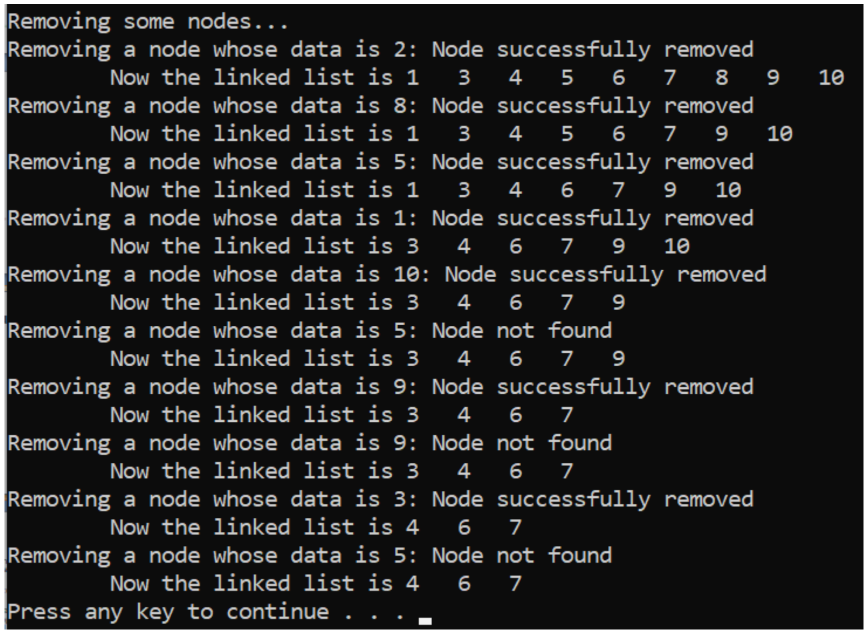
- A sample run output of the program is shown below

Tail Insert Output



```
At the beginning the linked list is

Inserting using tail insert algorithm
Inserting a node whose data is 1
        Now the linked list is 1
Inserting a node whose data is 2
        Now the linked list is 1   2
Inserting a node whose data is 3
        Now the linked list is 1   2   3
Inserting a node whose data is 4
        Now the linked list is 1   2   3   4
Inserting a node whose data is 5
        Now the linked list is 1   2   3   4   5
Inserting a node whose data is 6
        Now the linked list is 1   2   3   4   5   6
Inserting a node whose data is 7
        Now the linked list is 1   2   3   4   5   6   7
Inserting a node whose data is 8
        Now the linked list is 1   2   3   4   5   6   7   8
Inserting a node whose data is 9
        Now the linked list is 1   2   3   4   5   6   7   8   9
Inserting a node whose data is 10
        Now the linked list is 1   2   3   4   5   6   7   8   9   10
```

Remove Node Output



```
Removing some nodes...
Removing a node whose data is 2: Node successfully removed
        Now the linked list is 1   3   4   5   6   7   8   9   10
Removing a node whose data is 8: Node successfully removed
        Now the linked list is 1   3   4   5   6   7   9   10
Removing a node whose data is 5: Node successfully removed
        Now the linked list is 1   3   4   6   7   9   10
Removing a node whose data is 1: Node successfully removed
        Now the linked list is 3   4   6   7   9   10
Removing a node whose data is 10: Node successfully removed
        Now the linked list is 3   4   6   7   9
Removing a node whose data is 5: Node not found
        Now the linked list is 3   4   6   7   9
Removing a node whose data is 9: Node successfully removed
        Now the linked list is 3   4   6   7
Removing a node whose data is 9: Node not found
        Now the linked list is 3   4   6   7
Removing a node whose data is 3: Node successfully removed
        Now the linked list is 4   6   7
Removing a node whose data is 5: Node not found
        Now the linked list is 4   6   7
Press any key to continue . . .
```

Data Structures

- A data structure is a mechanism in programming **that is used to store, organize, process, and retrieve data in a computer**
- C/C++ **struct**s allow us to create data structures in order to work with data more efficiently
- For example, by organizing the numerator and denominator of a rational number object together, we are able to store, retrieve, and process rational number objects more efficiently and conveniently
- Thus we may say the RationalNumber struct discussed earlier is a data structure
- Similarly, a linked list is a data structure that allows us to work with several objects in an organized, convenient, and efficient way