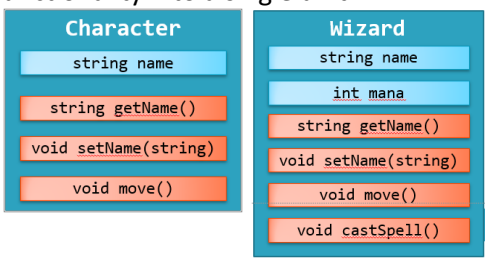
Chapter 15 – Inheritance, Polymorphism, and Virtual Functions (NB)

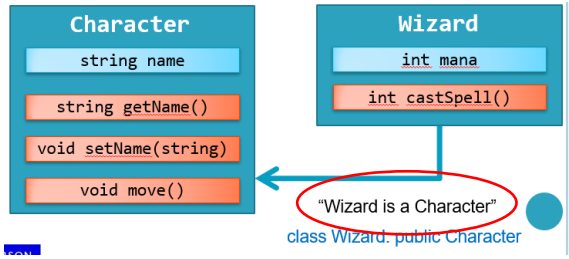
15.1 What is Inheritance?

Class: a user-defined type that binds (encapsulates) attributes and related functionality into a single unit

We may want to write a class that shares some functionalities.



- We can write 2 separate classes, make new one (eg; wizard) by copying and pasting from character class.
  - Potentially need to debug twice now.
- **OR** we can inherit the functionality/ data of the Character class and still add functionality specific to Wizzard.



- Inheritance provides a way to create a new class from an existing class
- **Benefits**:
  - Re-use previously written code!
  - Code re-use means less time spent coding, and less debugging to do
  - Implement elegant design patterns in your software (second year CS)

Aggregation vs Inheritance

- Inheritance establishes an "is a" relationship between classes.
  - A wizard is a character
  - A car is a vehicle
- Aggregation: "has a"
  - Bicycle has a wheel
  - One class may contain objects of another class

Terminology and Syntax

- **Base** class (or parent) – the class you inherit from
- **Derived** class (or child) – inherits from the base class
- **Syntax:**

```
class Character           // base class
{                         // (a.k.a. parent class)
  . . .
};
class Wizard : public Character
{                         // derived class
  . . .                   // (a.k.a. child class)
};
```

What the Child Class inherits

- An object of the **derived** class has:
  - all members declared in parent class
  - all members defined in child class
- An object of the derived class can use:
  - all **public** members defined in parent class
  - all **public** members defined in child class
- What about the **private** members?
  - object of derived class will contain the inherited **private** members, but will only be able to access them through **public** functions defined in the parent class

```
class Character {
 public:
  Character(string, int);
  string getName();
  bool isDead();
  bool isFriendly();
 private:
  string name;
  int health;
};
```

```
class Wizard : public Character {
 public:
  Wizard(string, int);
  void castSpell();
 private:
  int mana;
};
```

```
void Wizard::castSpell()
{
  cout << getName();
  cout << " casts a spell!\n";
}
```

Wizard can only access the name through public interface

15.2 Protected members and Class Access

- **public** members are accessible outside of the class, and **private** members are not accessible outside of the class
- **private** members of the parent class are not directly accessible from the child class

- To keep certain **parent class** members inaccessible from the outside, but allow **derived classes** to access them directly:
  - Use the **protected** access modifier!

```
protected:
        double length;
        double width;
```

```
class Character {
 public:
  Character(string, int);
  string getName();
  bool isDead();
  bool isFriendly();
 protected:
  string name;
  int health;
};
```

```
class Wizard : public Character {
 public:
  Wizard(string, int);
  void castSpell();
 private:
  int mana;
};
```

```
void Wizard::castSpell()
{
  cout << name;
  cout << " casts a spell!\n";
} // ok now: name is protected
```

Parent class access modifier	Access from the outside	Access from a child class
public	Accessible	Accessible
private	Not accessible	Not accessible
protected	Not accessible	Accessible

Protected members are **public** for derived (child) class but **private** for all other classes.
***public** – objects of derived class can be treated as objects of base class (public interface is inherited)
***protected** & **private** – derived classes still inherit the properties, but have to provide their own public interface

Inheritance as public, protected, or private

```
// Inherit from parent publicly
class Child1: public Parent
{
};
// Inherit from parent privately
class Child2: private Parent
{
};
// Inherit from parent protectedly
class Child3: protected Parent
{
};
// Default: private inheritance
class Child4: Parent
{
};
```

Parent Access Modifier	Public Inheritance	Protected Inheritance	Private Inheritance
class Parent	class Child: public Parent	class Child: protected Parent	class Child: private Parent
public: x	public: x	protected: x	private: x
protected: x	protected: x	protected: x	private: x
private: x	Inaccessible	Inaccessible	Inaccessible

Inheritance is private by default!

Public Inheritance

```
class Parent
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
class Child: public Parent    // inherit publicly
{
        Child()
        {
                x = 1; // ALLOWED: x is public
                y = 2; // ALLOWED: y is protected
                z = 3;// COMPILE ERROR: z is private
        }
};
int main()
{
    Child obj;
    obj.x = 1; // ALLOWED: anybody can access public members
    obj.y = 2;// NOT OK: can not access protected members from outside
    obj.z = 3; // NOT OK: can not access private members from outside
}
```

Public inheritance does not **modify** the access rights of the **inherited members**

Protected Inheritance

```
class Parent
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
class Child: protected Parent  // inherit protectedly
{
        Child()
        {
                x = 1; // ALLOWED: x is protected in Child
                y = 2; // ALLOWED: y is protected in Child
                z = 3; // COMPILE ERROR: z is private in Parent
        }
};
int main()
{
    Child obj;
    obj.x = 1; // NOT OK: can not access protected members from outside
    obj.y = 2; // NOT OK: can not access protected members from outside
    obj.z = 3; // NOT OK: can not access private members from outside
}
```

Protected inheritance turns inherited public members into protected members

Private Inheritance

```
class Parent
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
class Child: private Parent  // inherit privately
{
        Child()
        {
                x = 1; // ALLOWED: x is a private member of Child
                y = 2; // ALLOWED: y is a private member of Child
                z = 3; // COMPILE ERROR: z is a private member of Parent
        }
};
int main()
{
    Child obj;
    obj.x = 1; // NOT OK: can not access private members from outside
    obj.y = 2; // NOT OK: can not access private members from outside
    obj.z = 3; // NOT OK: can not access private members from outside
}
```

Private inheritance turns **inherited public** & **protected** members into **private members**

If there was a "**grandchild**" class for example that **inherited** from the **child class**, this grandchild class **would not** be able to **access** any of the **members** that the **child class** has accessed from the parent class.

Example

See protected inheritance example in files

15.3 Constructors and Destructors in base and Derived classes

- Derived classes may have their own member variables – thus, derived classes will need their own **constructors** and **destructors**
- Constructors and destructors are **not inherited**, but **private members** of the parent class are inherited. We need to create constructors and destructors for each child class
- Initialise objects by calling the **constructor** of the **parent class** from the **constructor** of the **derived class**

Constructor initialization lists

- What you're used to (explicit assignment):

```
Person::Person(string n, string s)
{
        name = n;
        surname = s;
}
```

- Another way to do it (implicit assignment):

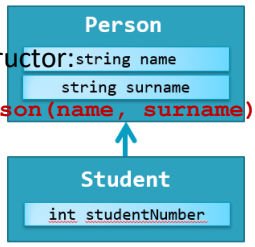
```
Person::Person(string n, string s) : name(n), surname(s)
{
}
// known as member initialisation list
```

- keyword: "another" ;so I can find this later

These two member initialization lists do exactly the same thing

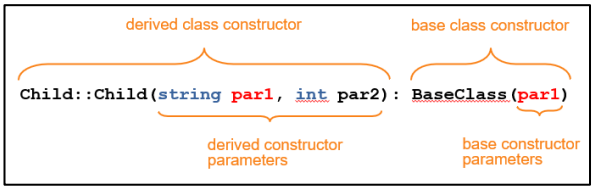
Passing Arguments to Base Class Constructor

- The same syntax can be used to pass arguments to a base class constructor
- Suppose **Student** inherits from **Person**:
  - **class Student : public Person**
- We need to pass arguments to **base class** constructor from the **derived class** constructor:

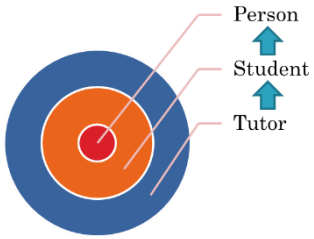



```
Student::Student(string name, string surname,unsigned int number): Person(name, surname)
{
    studentNumber = number;
}
```

- We do not need to invoke the **base class constructor** manually
  - Red line from above, can be omitted and C++ will try to invoke the default base class constructor on your behalf.
  - If the base class does not have a default constructor, this may lead to errors



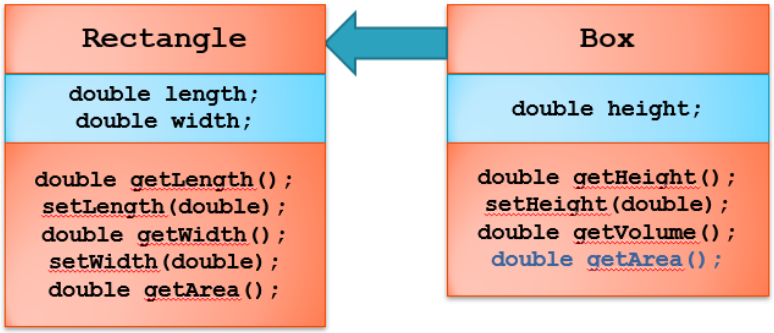
Constructors and Destructors in Base and Derived Classes



- When an object of a **derived class** is **created**:
  - base class's constructor is executed first followed by the derived class's constructor.
- When an object of a **derived class** is **destroyed**:
  - derived class's destructor is called first followed by base class's destructor
  - When the **destructor** of the child class executes, it automatically invokes the destructor of the parent class

15.4 Redefining the base Class

- In C++, we use inheritance to create new classes from existing ones
- Derived classes:
  - inherit data/functionality of the base class and can add their own data/functionality
- what if derived class inherits things from base class that no longer applies to the derived class.



- Box is rectangle with height, but the area of the box is calculated differently from the rectangle
- We will redefine the **getArea()** function…

- **Redefined function**: function in a derived class that has the same name and parameter list as a function in the base class.
  - This will replace a base class function with different behavior in a derived class
  - This **differs from overloading** as:
    - with overloading, parameter lists must be <u>different</u>
    - with redefining, parameter lists must be <u>the same</u>

- Objects of **base class** use **base class** version of function; objects of **derived class** use **derived class** version of function.

```
int main()
{
	Rectangle r(2,3); // create a Rectangle object
	Box b(2,3,4);     // create a Box object

	cout << r.getArea() << endl;// invoke Rectangle's
                                // version of getArea()

	cout << b.getArea() << endl;// invoke Box's
                                // version of getArea()
}
```

- C++ decides what version of **getArea()** to call by looking at the object's class type

```
double Box::getArea() const
{
	double area = 0;
	area += 2 * height * width;
	area += 2 * height * length;
	area += 2 * Rectangle::getArea();
	return area;
}
```

- Derived class can access the base class's version of a redefined function by:
  - Use scope resolution operator ' :: ' to access parent class members that were **redefined**

- Write a function in the base class that uses a redefined function:

```
void Rectangle::printArea() const
{
    cout << "The surface area of this object is ";
    cout << getArea() << endl;
}
```

- If we invoke **printArea()** on an object of type Box, the **getArea()** version of of the base class (Rectangle) will be invoked.
  - This is due to static binding, **next section…**

The problem of static binding

- If we were to invoke the method on a Box object:

```
Box myBox(1,1,1);
myBox.printArea();
```

  - Compiler notices that **printArea()** is defined in the **base class**, and binds the function call to the base class – we call this **static binding**
  - The method will actually print the area for the rectangle class rather than the box class resulting in the wrong area being displayed

- **Static binding (early binding)**: function calls are linked to function bodies at **compile time**, i.e. before the program is run
- **Dynamic binding (late binding)**: function pointers are used by the computer instead of hard-coding the link between function calls/bodies, thus we can substitute the necessary function at **run time**
- To change **static binding** of functions to **dynamic binding:**
  - Make the functions that require dynamic binding **virtual** (**see next…**)

Virtual Functions

```
class Rectangle
{   // put constructor and other functions here
    virtual double getArea() const;
    void printArea() const;
};

void Rectangle::printArea() const
{   // because getArea() is virtual,
    // dynamic binding will take place
    cout << "The surface area of this object is ";
    cout << getArea() << endl;
}
```

- Add "virtual" in the header file
- We tell the compiler that this function will possibly be redefined in child classes
- Now the correct version of **getArea()** will be invoked even on a Box object:

```
Box myBox(1,1,1);
myBox.printArea();
```

- **"getArea()** is a virtual function, a pointer to a function call rather than a direct function call"

- A **virtual function** is dynamically bound to function calls at runtime.
  - At runtime, C++ determines the type of object making the call, and binds the function call to the appropriate version of the function.
  - The compiler will not bind the function to calls. Instead, the program will bind them at runtime.
- To make a function virtual, place the virtual key word before the return type in the base class's declaration:

-
```
virtual double getArea() const;
```

- Redefined function that is declared as **virtual** in the base class is called an **overridden function**

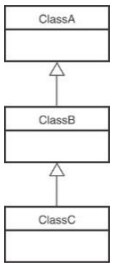
Redefining vs. Overriding

- In C++, **redefined functions** are **statically bound** and **overridden functions** are **dynamically bound**.
  - So, a **virtual function** is **overridden**, and a **non-virtual function** is **redefined**.
- Redefining is a bad idea – you may run into static binding discrepancies any time
  - Use overriding!

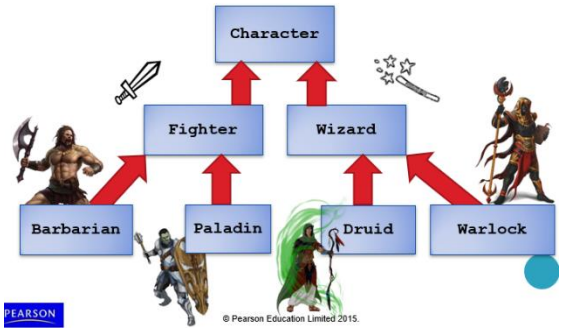
Example

See shapes examples in files

15.5 Class Hierarchies



- In C++, we use inheritance to create new classes from existing ones
- A derived class can become a parent to another derived class.



- Character is the base class:

```
class Character {
 public:
  Character(string, int);
  int getHealth();
  void setHealth(int);
  string getName();
  bool isDead();
  bool isFriendly();
 protected:
  string name;
  int health;
};
```

- If we declare a wizard class all members are **inherited** by Wizard
- If we declare a wizard class all members are accessible by Wizard

- Wizard Class:

```
class Wizard :
public Character {
 public:
  Wizard(string, int, int);
  int getMana();
  void setMana(int);
  void castSpell(Character&);
 protected:
  int mana;
};
```

- If we declare a Druid class, it will inherit all the members from the Wizard as well as the character class
- Druid will have direct access to both the Wizard and the Character Class

15.6 Polymorphism

- Polymorphism: “Changing shapes”
- Inheritance establishes “is a” relationship between classes…
  - This will work because of polymorphism.
- Polymorphism allows **derived** class objects to “hide” behind **base** class interfaces
  - Treat a Box as a Rectangle
  - Treat a Wizard as a Character
- This will work because:
  - Because derived objects **inherit the interface** of the base classes
  - (This only works if public inheritance is taking place)
  - Allows us to create functions at a **higher level** of the hierarchy that would work for **all derived classes**.

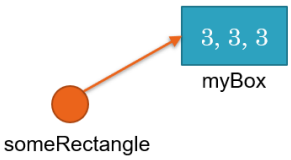
```
Rectangle * someRectangle;
Box myBox(3,3,3);
someRectangle = &myBox; //assign derived to base
cout << someRectangle->getArea() << endl;
```

- If getArea() is overridden (**virtual**), the correct (Box) version will be used!

- When using heirarchies, it is good practice to make all functions that will possibly be redefine, "virtual" functions.

```
Rectangle * someRectangle;
Box myBox(3,3,3);
someRectangle = &myBox; //assign derived to base
cout << someRectangle->getArea() << endl;
```

- Only one object exists in computer memory
- Rectangle pointer simply stores **the address** of an existing object



Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when a **derived** class object is referenced by a **base** class reference variable or a pointer.
- Polymorphism will only work when you use references or pointers
- Polymorphism is "hiding behind the interface of the parent
- This will not work:

```
Rectangle rec;
Box myBox(3,3,3);
rec = myBox; // compiles fine: makes a copy
cout << rec.getArea() << endl;
```

- Rectangle does not hide the box but instead stores one side of the box
- Will execute Rectangle's getArea().

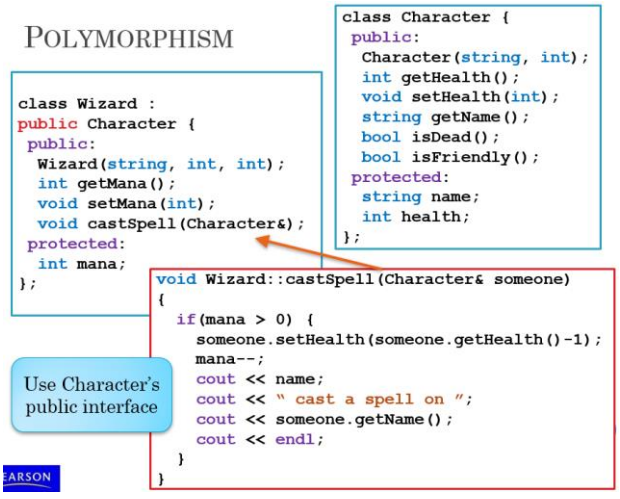
- Known as slicing: we **create** a rectangle from a box, but the unique properties of the box are lost

Base class pointers

- Define a *base* class pointer/reference;
- Assign the address of a *derived* class object to it.
- Example: `Rectangle * rec = new Box(4,2,1);`
    - The correct version of getArea() will be called only if getArea() is virtual.
- Cannot call Box-only functions on rec (eg., getVolume())
    - only Rectangle interface is available to rec
- Polymorphism does not work in reverse!
    - **Box * box = new Rectangle(4,2);**
    - A Box is a Rectangle, but a Rectangle is not a Box
    - Children can pretend to be a Parent but a parent cannot pretend to be a child

True Beauty of Polymorphism

- A wizard can cast a spell on a Character
- Can we make the Wizard cast a spell on a Paladin? A Druid? On another Wizard?
    - Paladin, Druid can pretend to be a character and thus can have spells cast upon them
- We can make a simple function that just needs to damage the character health; thus it does not matter what **type** of character is being attacked, we just need to "look" at the character parameters of the different objects.
    - A Wizard can pretend to be a Character and thus have its health taken away.



```
Wizard rince("Rincewind", 100, 500);
Barbarian ghorak("Ghorak", 300);
Druid miriel("Miriel", 35, 400);

rince.castSpell(ghorak);// castSpell(Character&)
ghorak.fightWith(rince);// fightWith(Character&)

miriel.heal(rince);  // heal(Character&)
miriel.heal(ghorak); // heal(Character&)
```

- This will work
- A Druid can heal **any** Character, a Fighter can fight with **any** Character, a Wizard can cast a spell on **any** Character!
- If we replaced **Character&** with **Character** slicing would occur and instead of applying the function to the object, a copy of the object will be created and the copied object will be altered, leaving the original object unchanged

Destructing Polymorphic Objects

```
Character * merlin = new Wizard("Merlin", 100, 500);
// Go on epic adventures;
// Eventually, delete the object:
delete merlin;
// What destructor is this going to call?
// Character's or Wizard's?
```

- Which destructor is called?
- Compiler will not know which destructor to call
- A destructor is basically a special kind of **function** that is invoked automatically. Thus, destructors are susceptible to **early binding**, too.

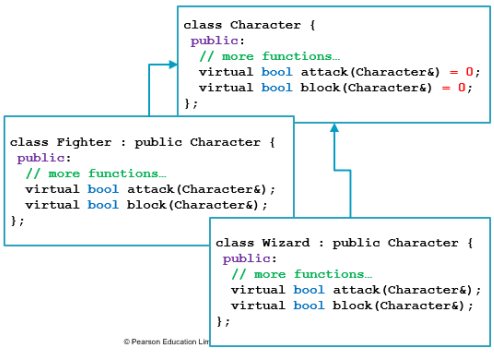
- To fix this: **use virtual destructors**
- Make the destructor "virtual" if the class is to be extended via inheritance
- Otherwise, the compiler will perform static binding on the destructor
  - i.e., if a polymorphic object is destroyed, only invoke the base class constructor, ignoring derived classes
- Why it is important to make destructors virtual: derived classes may use dynamic memory!
- If you fail to invoke their destructors, you will run into a memory leak.
- As a habit, if you are dealing with inheritance make destructors virtual!
- When dealing with class hierarchies / polymorphism, it is important to enforce dynamic (late) binding by making redefined functions virtual.

Pure Virtual Functions

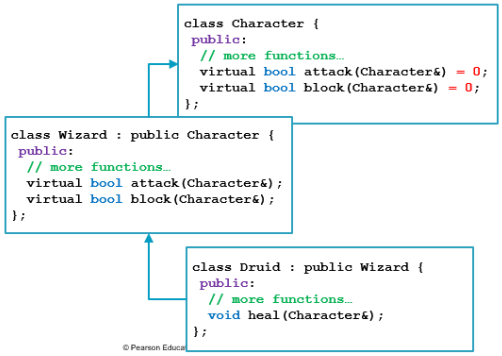
- You can write a **base class** that defines the **public interface** for a certain hierarchy but does not implement the virtual functions.
  - This is done by making your virtual functions pure virtual, or abstract:

```
virtual bool attack(Character&) = 0;
```

    - This function has no **body**, and **must** be implemented by the derived classes
    - All derived classes are forced to override this function. Character class thus acts as an interface for the functions.



- Every character must implement **attack**() and **block**()
  - This means that all derived classes need to implement these functions as well



- For the derived classes of Wizzard:
- The class Wizard needs to override these functions.
- For the Druid class, it inherits the function from the wizard class. because it fights like a wizard, you would not need to overwrite these functions.
- After overriding the functions in the class wizrd. These functions are pure virtual in the character class but in the inherited class, these function will **….** remain pure virtual?? *Answer required*

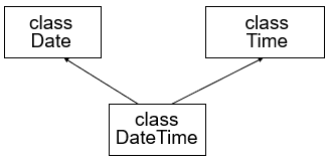
15.7 Abstract Base Classes and Pure Virtual Functions

Abstract classes

- **Character** provides no implementation of the **attack**() or **block**() function
- We cannot create a non-polymorphic character as a class containing at least one pure virtual function becomes an **abstract class.**
    - Abstract classes cannot be instantiated, they can only be used for polymorphism
    - Abstract classes are used to define a public interface to be shared by the derived classes
- Class that contains only pure virtual functions is often called an interface class.
    - It provides no implementation, but declares the functions that every derived class must implement:

```
class Function { // interface class
  public:
    virtual double computeFunction(double) = 0;
    virtual double computeDerivative(double) = 0;
    virtual double* getRange() = 0;
    virtual double* getDomain() = 0;
};
```

15.8 Multiple Inheritance



- A derived class can inherit from more than one base class
- Used when you need a class that combines data functionality of two unrelated parents

```
class Date {
  public:
        Date(int, int, int);
        int getYear();
        int getMonth();
        int getDay();
  protected:
        int day;
        int month;
        int year;
};
```

```
class Time {
  public:
      Time(int, int, int);
      int getHours();
      int getMinutes();
      int getSeconds();
  protected:
      int hour;
      int min;
      int secs;
};
```

```
class DateTime : public Date, public Time {
  public:
      DateTime(int, int, int, int, int, int);
      string getDate();
      string getTime();
      void printDateTime();
};
```

- Now every DateTime object will have all data & functions of Date, as well as all data & functions of Time

```
DateTime::DateTime(int d, int mon, int y, int h, int m, int s): Date(d, mon, y), Time(h, m, s)
{}
```

- Order of execution: same as order of inheritance:

```
class DateTime : public Date, public Time
```

    - In this case, first Date and then Class

```
class Date {
      public:
            Date(int, int, int);
            int getYear();
            int getMonth();
            int getDay();
            void print();
      protected:
            int day;
            int month;
            int year;
};
```

```
class Time {
  public:
      Time(int, int, int);
      int getHours();
      int getMinutes();
      int getSeconds();
      void print();
  protected:
      int hour;
      int min;
      int secs;
};
```

- if both parent classes define a function with the same name, when trying to invoke **print()** in main.cpp the function will not compile as the call is ambiguous.
- Resolve the ambiguity by using scope resolution operator '::'

```
int main() {
    DateTime dt(31, 8, 2015, 15, 20, 0);
    // 31 Aug 2015, 15:20:00
    dt.print(); // Which print??
    return 0;
}
```

```
int main() {
    DateTime dt(31, 8, 2016, 15, 20, 0);
    // 31 Aug 2016, 15:20:00
    dt.Date::print(); // print the date
    dt.Time::print(); // print the time
    return 0;
}
```

- Now the compiler knows exactly which function to call in each case

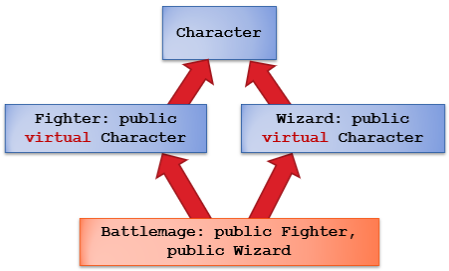
```
class DateTime {
  public:
     DateTime(int, int, int, int, int, int);
     string getDate();
     string getTime();
     void printDateTime();
  private:
     Date date;
     Time time;
};
```

- Most of the time, multiple inheritance can be replaced with single inheritance and aggregation. Use multiple inheritance with caution, and only when it truly makes sense.

Diamond Problem

- RPG hierarchy again:
- Who is responsible for invoking the Character's constructor?
  - C++ will invoke both Fighter and Wizard constructors when Battlemage is created
  - Each of these is going to invoke the Character constructor
  - Two Character objects will be created for each Battlemage
  - This is wrong and will cause ambiguity

- Solution:



- declare inheritance as virtual
- Now the **Battlemage** class must explicitly invoke Character's constructor

Chapter 16 – Exceptions and Templates

16.1 Exceptions

```
int getValue(int index)
{
    if(index < 0 || index >= size)
    {
          cout << "Invalid index" << endl;
          return -1; // return a dummy value
    }
    else return array[index];
}
```

- If invalid input is received:
  - The **getValue()** method will return -1
    - If this method was used to get the value for another array:

```
int result = otherArray[ getValue(12) ];
```

    - This would cause a crash as **-1** is not a valid index for an array

Elegant Error Handling

- **Exceptions** can be used to handle complex error functions
  - **Exception:** a value or an object that signals an error.
  - In C++, a **throw statement** is used to signal that an exception or error has occurred

```
double divide(double num, double den)
{
        if(den == 0) {
              throw "Error: Division by zero\n";    }
        else return num / den;
}
```

- "**Throw**" an exception(send a signal that an error has occurred):
- Instead of returning a meaningless dummy value, we will terminate the function immediately and indicate error.
- To use a throw statement, simply use the throw keyword, followed by a value of any data type you wish to use to signal that an error has occurred.

- Throw point—line containing throw statement
- Types of exceptions that can be thrown
  -

```
throw -1; // throw a value

throw ENUM_INVALID_INDEX; // throw an enum value

throw "Division by zero"; // throw a char const *

throw dX; // throw a defined variable

throw MyException("Fatal Error 404");
// Throw an object of class MyException
```

- A thrown exception can be caught
- To handle an exception the program must have try catch/construct.
- Catch/Handle an exception: process the exception; interpret the signal

```
double x = 25, y = 0, result;
try
{
    result = divide(x, y); // throws an exception
}
catch(char const * errMessage)
{
    cout << errMessage;
}
```

- Catch Block: handles exceptions for a single data type.

```
try {
  doSomething();       // throws an exception
}
catch(char const * errMessage) {
  cout << errMessage; // catch text exception
}
catch(int num) {
  cout << num;         // catch int exception
}
catch(...) {           // catch anything!
}
```

- Can throw more than one exception from the try block and pride a many catch block as necessary
- A try block must have at least one catch block immediately following it but may have multiple catch blocks listed in sequence.
- Just like with functions, if the parameter is not going to be used in the catch block, the variable name can be omitted
- Exceptions of fundamental types can be caught by value, but exceptions of non-fundamental types should be caught by const reference to avoid making an unnecessary copy.

Exceptions

- Indicate that something unexpected has occurred or been detected
- Allow program to deal with the problem in a controlled manner
- Can be as simple or complex as program design requires

Terminology

- Exception: object or value that signals an error
- Throw an exception: send a signal that an error has occurred
- Catch/Handle an exception: process the exception; interpret the signal
- If the exception was caught, the program carries on executing
- If you do not catch a thrown exception the program will terminate.

Flow of Control

- A function that throws an exception is called from within a try block
- If the function throws an exception, the function terminates and the try block is immediately exited. A catch block to process the exception is searched for in the source code immediately following the try block.
- If a catch block is found that matches the exception thrown, it is executed. If no catch block that matches the exception is found, the program terminates.

```
int main()
{
   int num1, num2;   // To hold two numbers
   double quotient; // To hold the quotient of the numbers

   // Get two numbers.
   cout << "Enter two numbers: ";
   cin >> num1 >> num2;

   // Divide num1 by num2 and catch
   // potential exceptions.
   try
   {
      quotient = divide(num1, num2);
      cout << "The quotient is " << quotient << endl;
   }
   catch (string exceptionString)
   {
      cout << exceptionString;
   }
```

```
double divide(int numerator, int denominator)
{
   if (denominator == 0)
   {
      string exceptionString = "ERROR: Cannot divide by zero.\n";
      throw exceptionString;
      //throw "Error constatat
   }

   return static_cast<double>(numerator) / denominator;
}
```

- Note how **devide()** throws the exception but the exception is only caught after the function has been called even though the function terminates(if invalid input is entered)

- What will happens in the try/catch construct:

```
try
{
   quotient = divide(num1, num2);
   cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}
cout << "End of the program.\n";
return 0;
```

If this statement throws an exception...

... then this statement is skipped.

If the exception is a string, the program jumps to this catch clause.

After the catch block is finished, the program resumes here.

- If no exception is thrown:

```
try
{
     quotient = divide(num1, num2);
     cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

- Predefined functions such as new may throw exceptions
- The value that is thrown does not need to be used in **catch** block.
    - in this case, no name is needed in catch parameter definition
    - catch block parameter definition *does* need the **type of exception being caught**

- An exception will not be caught if:
    - it is thrown from outside of a **try{}** block
    - there is no **catch{}** block that matches the data type of the thrown exception
- If an exception is not caught, the program will **terminate**:

```
double x = 25, y = 0, result;
try
{
    result = divide(x, y); // throws an exception
}
// terminate the program - no "catch"!
```

Exceptions and Objects

- Classes may require more detailed exceptions than just a single value or an error message
- An exception class can be defined within a class and thrown as an exception by a member function
    - **exception class**: normal class that is designed specifically to be thrown as an exception.
- An exception class may have:
    - no members: used only to signal an error
    - members: pass error data to the **catch** block
- A class can have more than one exception class

Rectangle.h:

```
class Rectangle
{
   private:
      double width;      // The rectangle's width
      double length;     // The rectangle's length
   public:
      // Exception class
      class NegativeSize
         { };            // Empty class declaration

      // Default constructor
      Rectangle()
```

Rectangle.cpp:

```
void Rectangle::setLength(double len)
{
   if (len >= 0)
      length = len;
   else
      throw NegativeSize();
}
```

Main.cpp:

```
try
{
   myRectangle.setWidth(width);
   myRectangle.setLength(length);
   cout << "The area of the rectangle is "
        << myRectangle.getArea() << endl;
}
catch (Rectangle::NegativeSize)
{
   cout << "Error: A negative value was entered.\n";
}
cout << "End of the program.\n";

return 0;
```

- Once an exception is thrown, the program cannot return to throw point.
- The function executing the **throw** terminates (does not return), after declaring the **throw** statement control is returned to the point where the function containing the throw statement was called (assuming that there was no valid catch statement in the function)
    - If there are multiple nested function calls, functions will be terminated and return to the point of the function call until a valid catch statement is found.
- Other calling functions in **try** block terminate
- This is known as unwinding the stack: continues for the entire chain of nested function calls.

- If objects were **created** in the **try** block and an exception is thrown, **they are destroyed**. (Destructor is automatically executed)
- If exception is thrown by the member function of a class object, then the class destructor is called.
- If exception thrown and not caught other functions calling function will terminate as well.

Nested Try/Catch Blocks

- **try/catch** blocks can occur within an enclosing **try/catch** block
- Exceptions caught at an inner level can be passed up to a **catch** block at an outer level:
- Known as "re-throwing exceptions"

```
catch ( int )
  {
    ...
    throw;  // pass exception up
  }         // to next level
```

Rethrowing Exceptions

```
double function1(double x) {
    try {
      if(x < 0) throw -1;
      return sqrt(x);
    }
    catch ( int ) {
      cout << "negative value";
      throw; // re-throw: let the function who
    }        // called you handle it
}
double function2(double y) {
    try { double x = function1(y); return x; }
    catch(int) { /* handle it!… */ }
}
```

16.2 Function Templates

- To write a function to determine the larger value of two values entered
  - This function will need to be repeated for each data type entered, e.g., int, double, char)
- C++ allows us to write functions that work on any data type
- Such functions are known as generic, or template functions
- Function template: a pattern for a function that can work with many data types
  - When written, parameters are left for the data types
  - When called, compiler generates code for specific data types in function call

```
template <class T>

T max(T x, T y)
{
    if(x >= y) {
          return x;
    }
    else return y;
}
```

- Instead of using a concrete type such as **int**, we use **T**: a placeholder that can be replaced with **any** type
- **template <class T> :** template prefix //see COS110 Prac 6
  - **template <typename T> :** alternative template prefix
- **T** max(**T** x, **T** y) : Type Parameter
- The keyword *template* tells the compiler that what follows is going to be a list of template parameters.
  - To create a template type parameter, use either the keyword *typename* or *class*
- Function templates are compiled on demand; they are compiled based on function calls
  - If **max(T,T)** is never called, it will not be compiled at all
  - If **max(T,T)** is called on two doubles, **T** will be substituted with **double**, and **max(double, double)** will be compiled

```
template<class T>
T max(T x, T y)
{
      if(x >= y) {
            return x;
      }
      else return y;
}
```

What gets generated when `max(T,T)` is called with `ints` :`max(int int)`	What gets generated when `max(T,T)` is called with `doubles` :`max(double,double)`

`int max(int x, int y)`
`{`
`  if(x >= y) {`
`  return x;`
`  }`
`  else return y;`
`}`
	`double max(double x,`
	`double y)`
	`{`
	`  if(x >= y) {`
	`  return x;`
	`  }`
	`  else return y;`
	`}`

```
double a = 37.2;
double b = 20.25;
cout << max(a, b) << endl;

int c = 5;
int d = 10;
cout << max(c, d) << endl;

char one = 'A';
char two = 'Z';
cout << max(one, two) << endl;
char three = 'B';
cout << max(two, three) << endl;
```

- **max()** will be compiled 3 times – once for each data type.

- Compiler knows to create one template instance per set of unique type parameters (per file)
- if you create a template function but do not call it, no template instances will be created

```
Cube a(2); // cube with side = 2
Cube b(3); // cube with side = 3
cout << max(a, b) << endl;
```

- If user defined class objects are passed to max the class must contain code for an overloaded >= operator otherwise it will not work.

- You can substitute any type – primitive type or class type, if all operators used in the template function are implemented for the type.
- Any other letter/word can be used in place of **T**

- Can also define a template to use **multiple data types**: `template<class T1, class T2>`

```
template<class T1, class T2>
double mpg(T1 miles, T2 gallons)
{
    return (miles/gallons);
}
```

- Function templates can be overloaded if each template has a unique parameter list

```
template <class T>
  T someFunction(T num) ...

  template <class T1, class T2>
  T1 someFunction(T1 num1, T2 num2) ...
```

- All data types specified in template prefix must be used in template definition

- Like regular functions, function templates **must be defined before being called**
- A function template is a pattern
- No actual code is generated until the function name in the template is called
- A function template uses no memory
- When passing a class object to a function template, ensure that all operators in the template are defined or overloaded in the class definition

```
// Template definition for square function.
template <class T>
T square(T number)
{
   return number * number;
}

int main()
{
   int userInt;        // To hold integer input
   double userDouble;  // To hold double input

   cout << setprecision(5);
   cout << "Enter an integer and a floating-point value: ";
   cin >> userInt >> userDouble;
   cout << "Here are their squares: ";
   cout << square(userInt) << " and "
        << square(userDouble) << endl;
   return 0;
```

16.4 Class Templates

- Template classes may have data members of an arbitrary type **T**
- **Syntax(Vector.h):**

```
template <class T>
class Vector {
  public:
    Vector(int);
    T & operator[](int i); // return i'th element of type T
    void print();
  private:
    T * coords; // array of vector coordinates of type T
    int dimension;
};
```

- Class name is preceded by a template preamble, **type T** is used throughout the class
- Now we can create a vector of doubles, or ints, or Booleans, or any other type

- Compiling template classes is different to compiling normal classes
- **Syntax for Vector.cpp:**

```
template <class T>
Vector<T> :: Vector(int i) : dimension(i) {
    coords = new T [dimension];
}

template <class T>
T & Vector<T>::operator[] (int i) {
    if(i < dimension && i >= 0)
        return coords[i]; // return i'th element of type T
    else
        throw "Invalid index provided";
}
```

- Every function is preceded by a template preamble, class name is followed by <T>
- Every function of a template class is a <u>template function</u>

Class Templates: Compilation

- Cannot directly compile a template class: `g++ -c Vector.cpp` *– will not work!*
- Cannot compile class templates (generic code)
  - Because the compiler can only check if your code is valid once a concrete type is substituted
  - Can only check if it is valid for every type entered in the generic type **T**
- Your template code is likely to use <u>operators</u>, which may or may not be overloaded for any given type.
- The template class will be re-compiled for every new substituted type
- Compile Class templates by:
  - #include both Vector.h and Vector.cpp in main, compile only main. //method often disliked
  - #include Vector.cpp in Vector.h, compile only main. //more elegant
- It can be good practice to use compilation guards in both the .cpp and .h now

Compiling Option: 1

```
// Vector.h
template <class T>
class Vector {
  public:
    Vector(int);
    T & operator[](int i);
    void print();
  private:
    T * coords;
    int dimension;
};
```

```
// Vector.cpp
template <class T>
Vector<T> :: Vector(int i) : dimension(i) {
    coords = new T [dimension];
}

template <class T>
T & Vector<T>::operator[] (int i) {
    if(i < dimension && i >= 0)
        return coords[i];
    else
        throw "Invalid index provided";
}
```

```
// main.cpp
#include "Vector.h"
#include "Vector.cpp"
int main() {

  Vector<double> doubleVec(3);
  Vector<int> intVec(3);
  Vector<bool> binaryVec(3);
  doubleVec[0] = 3.25;
  intVec[0] = 7;
  binaryVec[0] = true;
}
```

- See <double> in main.cpp, making a template class of type double, compiler will replace all ***T's*** with type double

```
run: main
        ./main

main: main.o
        g++ main.o -o main

main.o:main.cpp Vector.h Vector.cpp
        g++ -c main.cpp
```

- The template class is not compiled explicitly but because it is in main, when main is compiled, the template class is compiled indirectly.

Compiling Option: 2

```
// Vector.h
template <class T>
class Vector {
  public:
    Vector(int);
    T & operator[](int i);
    void print();
  private:
    T * coords;
    int dimension;
};
#include "Vector.cpp"
```

```
// Vector.cpp
template <class T>
Vector<T> :: Vector(int i) : dimension(i) {
    coords = new T [dimension];
}

template <class T>
T & Vector<T>::operator[] (int i) {
    if(i < dimension && i >= 0)
        return coords[i];
    else
        throw "Invalid index provided";
}
```

```
// main.cpp
#include "Vector.h"
int main() {

  Vector<double> doubleVec(3);
  Vector<int> intVec(3);
  Vector<bool> binaryVec(3);
  doubleVec[0] = 3.25;
  intVec[0] = 7;
  binaryVec[0] = 0;
}
```

- In main.cpp, when we #include "Vector.h", because "Vector.cpp" is included in in Vector.h, we do not need to include Vector.cpp in main as well

```
run: main
	./main

main: main.o
	g++ main.o -o main

main.o: main.cpp Vector.h Vector.cpp
	g++ -c main.cpp
```

Default Types

- In your template class definition, you can choose a default value for the type.

```
// Vector.h
template <class T = double>
class Vector {
  public:
  … … … …
```

```
int main() {

  Vector<int> intVec(3);        // vector of ints
  Vector<bool> binaryVec(3);    // vector of booleans
  Vector<> doubleVec(3);        // vector of doubles
```

- If you do not specify a type, the **default** type will be used… As seen in main.cpp

Class Templates

- Class templates allow us to create re-usable **container classes**
- **Example**: create a template class Inventory<T>
  - **Inventory<T>** is a generic container
  - A **student** can store textbooks in it: **Inventory<Book>**
  - A **fighter** can store weapons in it: **Inventory<Weapon>**
  - A **doctor** can store meds in it: **Inventory<Medication>**
  - A **plumber** can store tools in it: **Inventory<Tool>**
- You only must write and debug your re-usable class once, and then use it everywhere!

Class Templates and Inheritance

- Class templates can inherit from other class templates:

```
template <class T>
class Inventory
  { ... };
template <class T>
class SearchableInventory : public Inventory<T>
  { ... };
```

- Derived classes must be template classes, too
- Parent class should be referred to as Parent<T>
- Inherited members should be accessed through **this** pointer

See *TemplateClasses* Example

Template Specialization

- Sometimes, a special kind of behaviour is required from specific types
  - Suppose you write a container class;
  - If **char**acters are stored in it, you want to be able to convert them to uppercase and lowercase at will
  - The rest of the functionality is shared, but only characters can be uppercase or lowercase
  - Specialize the template class for **char** type!
- We can deal with this by Re-writing the class with the specific type already substituted (see next slide for syntax)
- Specialized template classes may change the template functionality for the given type:

```
// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase() { return ++element; }
};
```

```
// class template specialization:
template <>
class mycontainer <char> {
    char element; // use char instead of T
  public:
    mycontainer (char arg) {element=arg;}
    char increase () {return ++element;}
    char uppercase () {
       // convert to uppercase
    }
```

Summary

- C++ allows you to implement **template classes**
  - Template classes can make use of member variables of any type T
  - Every function of a template class is a template function
  - We never compile template classes explicitly / directly!
  - You can create new template classes by inheriting from existing template classes
  - You can also create specialized templates, i.e. variants of the existing template classes that work differently for a specific type.

Chapter 17 – Introduction to the Standard Template Library (STL)

- Template functions and template classes allow us to create highly re-usable code that is applicable to any data type
- Some very useful generic containers and algorithms are a built-in part of standard C++
- Standard Template Library (STL): a library containing templates for frequently used data structures and algorithms

STL components:

- **Containers** (like an array): classes that store data and impose some organization on it
- **Iterators** (universal interface that all containers share): pointer-like objects; generic mechanisms for accessing elements in a container
- **algorithms:** generic operations applied to containers, such as sorting and searching.
    - STL algorithms are not only independent of the data type, they are also independent of the container type!

17.2 STL Container and Iterator Fundamentals

Two types of container classes in STL:

- **sequence containers:** organize and access data sequentially, as in an array. These include **vector**, **deque**, and **list. C++11: array.**
    - *Note that **deque** and **list** will be discussed later in the course, when we talk about the linked list and queue data structures.*
- **associative containers:** store element in order, use keys to allow data elements to be quickly accessed. These include **set**, **multiset**, **map**, and **multimap**

Sequence Containers (special kinds of arrays)

- **`array`:** A fixed-size array. Works exactly like a C++ array, but knows its own size (can be obtained using size() function). Only available since C++11.
- **`vector`:** An expandable array. Values are added at the end, adding values in the middle is less efficient
- **`deque`:** Like vector, but also allows to efficiently add values at the front of the expandable array
    - double ended queue
- **`list`:** A linked list (next chapter). An expandable linear structure that allows efficient insertion of new elements

Array Class (using array<T>)

```
#include <string>
#include <array>

int main() {

  array<double,5> doubleArr; // vector of doubles, size 5
  array<string,3> names;     // vector of strings, size 3

  doubleArr[0] = 2.5; // overloads the [] operator
  cout << doubleArr.size() << endl; // knows its own size
  cout << doubleArr[0] << endl; // use array notation

}
```

Vector Class (using vector<T>)

```
#include <string>
#include <vector>

int main() {

  vector<double> doubleVec; // vector of doubles
  vector<string> names;     // vector of strings

  doubleVec.push_back(2.5); // infinitely expandable
  cout << doubleVec.size() << endl; // knows its own size
  cout << doubleVec[0] << endl; // use array notation
  doubleVec.pop_back(); // remove last element
}
```

- push_back places item at last index of the 'array'

Example *stl1_Examples>>stlExample*

Iterators

- Given a data structure such as **vector** or **deque**, you would want to iterate through its elements, i.e. **use the data** stored there
- Data structures are generic, but each is implemented differently
- When you use it, you don't want to worry about the way it is implemented – you just want to access the data
- Solution: each STL data structure comes with a set of iterators
- Iterators hide implementation details from you and provide a common public interface
- Bridge between the containers and the algorithms

- Think of an iterator as a special kind of pointer:

```
vector<int> vect;
for (int i = 0; i < 6; i++) vect.push_back(i);

vector<int>::iterator it; // declare an iterator object
it = vect.begin(); // assign it to the start of the vector
```

- **`*it`** – returns the value that the iterator is currently pointing to
- **`it++, it--`** – moves iterator one element forward or backward
- **`it1 == it2, it1 != it2`** – checks if two iterators point to the same element
- **`it = vect.end();`** – assign an element position to the iterator

- Every container (vector, deque, map, etc.) provides the following functions to be used with iterators:
  - **`begin()`** – returns an iterator representing the beginning of the elements in the container
  - **`end()`** – returns an iterator representing the past-the-end element (position where you would insert the next element) (element after the last element in the 'array')
  - **`cbegin()`** – returns a const (read-only) iterator representing the beginning of the elements in the container
  - **`cend()`** – returns a const (read-only) iterator representing the element just past the end of the elements
  - **`rbegin()/rend()`** – allow for structure to be traversed in reverse, thus **`rbegin()`** will return location of last element and **`rend()`** will return the location just before the first element
- All containers provide (at least) two types of iterators:
  - **container::iterator** provides a read/write iterator
  - **container::const_iterator** provides a read-only iterator
- Different types of iterators:
  - **forward** (use ++, move in one direction only)
  - **bidirectional** (use ++ and --, move back and forth)
  - **random-access** (move back, forth, or jump to the middle)
  - **input** (can be used with cin and istream objects)
  - **output** (can be used with cout and ostream objects)

```
#include <string>
#include <vector>
int main() {
    vector<int> vect;
    for(int i=0; i < 6; i++) vect.push_back(i);

    vector<int>::const_iterator it; // read-only iterator
    it = vect.begin(); // assign it to the start of the vector
    while (it != vect.end()) // while not at the end
    {
        cout << *it << " "; // print the current value
        it++; // and iterate to the next element
    }
    cout << endl; // Output: 1 2 3 4 5
}
```

- Iterating Example

```cpp
#include <string>
#include <list>
using namespace std;
int main() {
    list<int> ll;
    for(int i = 1; i < 6; i++) ll.push_back(i);

    list<int>::reverse_iterator it; // read-only iter
    it = ll.rbegin(); // assign to the last element
    while (it != ll.rend())// while hasn't reach the end
    {
        cout << *it << " "; // print the current value
        it++; // and iterate to the previous element
    }
    cout << endl; // Output: 5 4 3 2 1
}
```

- Iterating in reverse

Example stl1_Examples>>iterStlExample

Important as it shows how to work with vectors

Using STL containers

- See table 17-8 on page 1073 (physical textbook)
- http://www.cplusplus.com/reference/vector/vector/
- Can also make a 2D vector

Example stl1_Examples>>vectorMulitExample

Associative containers (lecture 3, check the actual name of this chapter in textbook

- Two types of container classes in STL:
    - **sequence containers:** organize and access data sequentially, as in an array. These include **vector**, **deque**, and **list. C++11: array.**
    - **associative containers:** store elements in order, use keys to allow data elements to be quickly accessed. These include **set**, **multiset**, **map**, and **multimap**

STL Map Containers

- Idea: Store key-value pairs
    - Keys are unique (example: student numbers)
    - Values associated with keys may be duplicated (example: student name, final mark, courses registered for, etc.)
    - This structure is common for databases
- Container is sorted by keys

19002002	Jane Doe
19882999	Bob Marley
20121202	Mike Manaka
29009210	John Smith

Map Example

```cpp
#include <string>
#include <map>
int main() {
    map<int, string> myMap = { {1, "banana"}, {2, "milk"} };
    myMap[2] = "muesli"; // overwrites milk (unique keys!)
    myMap[10] = "bread"; // keys can have gaps between them
    myMap[5] = "banana"; // values do not have to be unique
    map<int, string>::const_iterator it; // read-only iterator
    it = myMap.begin(); // assign it to the start of the map
    while (it != myMap.end()) // while not at the end
    {
        cout << it->first << "=" << it->second << " ";
        it++; // iterate to the next element
    }
    cout << endl; // Output will be sorted by key
}
```

- map<int, string> shows the key type and the value type used in the map
- say it aloud: Muesli stored at key 2
- **First**: gives access to key
- **Second**: gives access to the value
- You can use objects of your own class as keys if **operator <** has been overloaded! **NB!!!!**
- See textbook examples on different ways to work with a map: *Insert page number here*

Example: stl2(Map)

- If you insert into “pairs” into the map that share the same key, compiler will only look at the first one ignoring the other attempts to insert at the repeated key.
    - `mymap.insert(make_pair(5, "avocado")); // 5 will not be avacodo`
    - `mymap[5] = "strawberry"; //5 will be changed to strawberry.`
- `Iportant contents covered`

STL Map Variation.

- **unordered_map**
    - More efficient than **map**, since the keys are not sorted
- **multimap**
    - Allows key duplication
    - Store multiple values associated with a key
- **unordered_multimap**
    - Allows key duplication
    - More efficient than **multimap**, since the keys are not sorted

STL Set Container

- Idea: Store unique keys
    - Similar to a mathematical set (of numbers)
    - Keys are unique
    - Keys are sorted

```
#include <string>
#include <set>
int main() {
    set<int> intSet = { 1, 1, 2, -10, 5, 6, 5 }; // -10,1,2,5,6
    set<string> stringSet = {“banana”, “apple”, “peach”};
    set<string>::const_iterator it; // read-only iterator
    it = stringSet.begin(); // assign it to the start
    while (it != stringSet.end()) // while not at the end
    {
        cout << *it << " "; // print the current value
        it++; // and iterate to the next element
    } // Out: apple banana peach (alphabetically sorted)
}
```

- Variations of the set containter:
    - Multiset
    - unordered_set
    - unordered_multiset

Example: stl2(Set)

- useful when printing the set as it does it in ascending/alphabetical order
- `Iportant contents covered`

17.6 Algorithms

- STL contains algorithms implemented as template functions to perform operations on containers.
- Requires **<algorithm>** header file
- **algorithm** includes:

binary_search	count
for_each	find
find_if	max_element
min_element	random_shuffle
sort	and others

- See more examples
    - Textbook describes the above in some detail *(insert page number)*
    - http://www.cplusplus.com/reference/algorithm/

Algorithm Example

```
#include <string>
#include <vector>
#include <algorithm>
int main() {
    vector<int> vect;
    for(int i=0; i < 6; i++) vect.push_back(i);

    vector<int>::iterator it; // iterator

    it = min_element(vect.begin(), vect.end());
    cout << *it << " "; // print the smallest element
    it = max_element(vect.begin(), vect.end());
    cout << *it << " "; // print the largest element
    cout << endl;
}
```

- STL algorithms apply to all STL containers, and are executed using STL iterators

Example: stl2(Algorithm)

- (near end) The template function does not have to be used, can be normal function.
    - Must ensure that it takes only a single input parameter.
    - Input type also needs to match data type of the

Functor

- A **function object** (*functor*) is an object that acts like a function
- Can be used with **STL algorithms**!

```
#include <algorithm>

struct IsEven { // Overload the () operator
  bool operator() (int i) { return i % 2 == 0;}
};

int main() {
    IsEven isEven; // function object
    vector<int> vect;
    for(int i=0; i < 6; i++) vect.push_back(i);

    int countEven = count_if(vect.begin(), vect.end(), isEven);
    int countEv = count_if(vect.begin(), vect.end(), IsEven());
}
```

Summary

- Associative STL containers
    - **maps**
        - Key-value pairs
        - Ordered by unique key
    - **sets**
        - Ordered sets of unique keys
- STL algorithms
    - Use iterators to apply generic algorithms to STL containers
    - Can apply your own functions / function objects to an STL container using an STL algorithm

Chapter 18 – Linked Lists

18.1 Introduction to the Linked List

Abstract Data Types (ADT's)

- A data structure is a way of organizing data in a computer so that it can be used efficiently
  - A university needs to manage a lot of individual students
  - A computer game needs to manage many different characters
  - Working with individual objects is inconvenient
  - Group objects together in a **data structure**!
- Examples of data structures: **Arrays** and **STL vectors**

Arrays

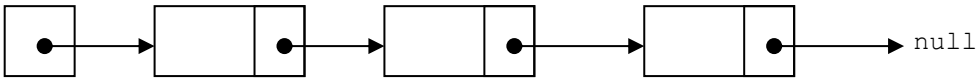
- What's wrong with arrays?
  - Static: need to know the size before you run the code
  - Dynamic: can choose the size at runtime
- Cannot resize an array, a new array of the desired size will need to be created and the desired values copied across to the new array.

STL Vectors

- What's wrong with vectors?
  - Vectors are resized automatically
  - STL vectors are implemented using dynamic arrays making them just as inefficient as arrays
- Shifting a vector elements by one to include an element is not effective

Problems with Arrays and STL Vectors

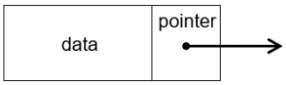
- Increasing the size of an array/vector is not very efficient
- Adding/removing elements from the middle of an array/vector is very inefficient
- **A data structure that does not have these deficiencies is known as a linked list**



List head

Linked Lists

- **Linked list**: set of data units (nodes) that contain references to other data units
- Every **node** has two components:
  - Data variable
  - Pointer to the next node
  - A node is like an advanced array element that knows the address of the next node



- Declaring a node:
  - Need to create a node struct: (this data structure will be used for the node)
  - Each Node needs to be declared as a class or struct

```
struct ListNode
  {
        double data; // data item
        ListNode * next; // address of
                         // the next node
  };
```

- Difference between a class and struct is that members are public by default in structs

18.2 Linked List Operations

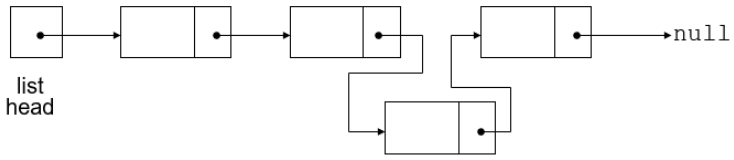
- A list of linked nodes
  - Nodes can be added to or removed from the linked list during execution
  - Linked list needs to store the address of the first node only (list **head**)

- Every node knows how to get to the next node
- Last node points to NULL

```
class LinkedList // linked list class
{
   private:
        struct ListNode // node structure
        {
            double value;     // The value in this node
            ListNode* next;  // Address of the next node
        };
        ListNode* head;     // List head pointer
};
```

Linked Lists vs Arrays and Vectors

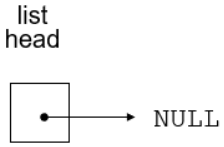
- **Linked lists can grow and shrink** as needed, unlike arrays, which have a fixed size
- Linked lists can **insert a node** between other nodes easily
- Just update two pointers, and you're done!



- see how link is updated and a new node has been added.

Organization of a Linked List

- Linked list contains 0 or more nodes:
- Has a head pointer to point to first node: `ListNode * head`
- Last node must point to null(address 0)
- If a list currently contains 0 nodes, it is an empty list
  - In this case the list head points to **null**



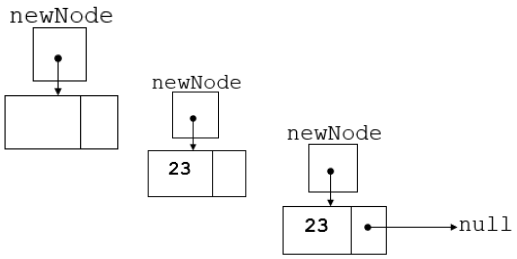
- **Constructing** a linked list:
  - Define a pointer for the head of the list: `ListNode * head = NULL;`
  - Head pointer initialized to NULL indicates an empty list
- **Using** a linked list:
  - Add data
  - Remove data
  - Iterate through the data in the list
- Basic linked list operations (**terminology**):
  - append a node to the end of the list
  - insert a node within the list
  - traverse the linked list
  - delete a node
  - delete/destroy the list

Example of linked list of doubles

```
class NumberList
{
   private:
     struct ListNode // node structure
     {
        double value;           // The value in this node
        ListNode* next;  // Address of the next node
     };
     ListNode* head;            // List head pointer

    public:
     NumberList() { head = NULL; } // constructor: initialize head to 0
     ~NumberList(); // destructor

     // Linked list operations:
     void append(double); // add node at the end of the list
     void insert(double); // insert a node (maintain ascending order)
     void delete(double); // remove a node
     void displayList() const;// print list on the screen
};
```

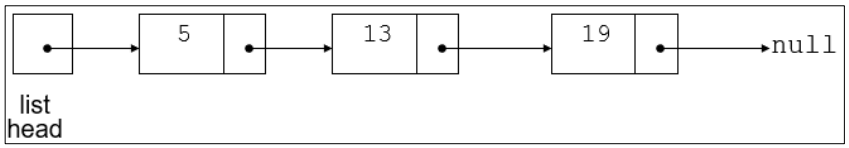
Append: Creating a new Node

- When data is added to the list, a new node is created to contain the data
  1) Allocate memory for the new node:
     ListNode* newNode = new ListNode;
  2) Initialize the contents of the node:
     newNode->value = num;
  3) Set the pointer field to NULL:
     newNode->next = NULL;

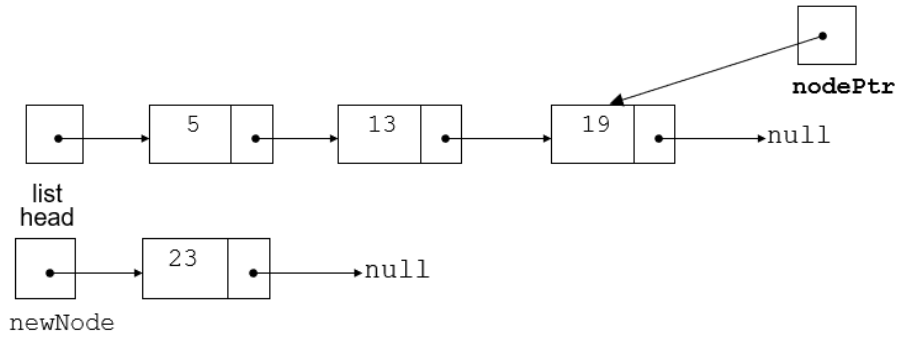


- Append: Add a node to the end of the list
  - Basic process:
    - Create the new node (as discussed)
    - Add the new node to the end of the list:
      - If list is empty,
        - set head pointer to the new node
      - Else,
        - Iterate to the end of the list
        - Set pointer of last node to point to new node

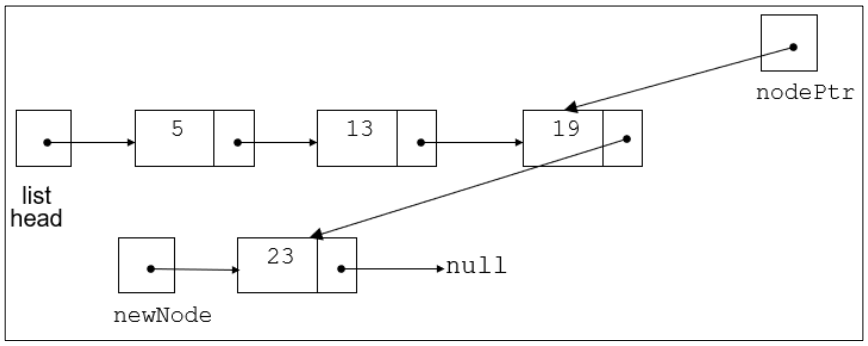
- When we need to create a new node, find the end of the list, and attach the new node to the end of the list



ListNode* nodePtr = head;




```
ListNode* nodePtr = head;
while(nodePtr->next)
{
  nodePtr = nodePtr->next;
}
```



- New node has been added to the end of the list

- Code to add a append a node:

```
11  void NumberList::appendNode(double num)
12  {
13     ListNode *newNode;  // To point to a new node
14     ListNode *nodePtr;  // To move through the list
15
16     // Allocate a new node and store num there.
17     newNode = new ListNode; // must we use "new"?
18     newNode->value = num;   // set value to num
19     newNode->next = NULL;   // set next to NULL
20
21     // If there are no nodes in the list
22     // make newNode the first node:
23     if (!head) // "If head is null"
24        head = newNode; // both are pointers
25     else  // Otherwise, insert newNode at end.
26     {
27        // Initialize nodePtr to head of list.
28        nodePtr = head;
29
30        // Find the last node in the list.
31        while (nodePtr->next) { // "while next != 0"
32           nodePtr = nodePtr->next;
33        }
34        // Insert newNode as the last node
35        nodePtr->next = newNode;
36     }
37  }
```

- Implementation:

Program 17-1

```
1  // This program demonstrates a simple append
2  // operation on a linked list.
3  #include <iostream>
4  #include "NumberList.h"
5  using namespace std;
6
7  int main()
8  {
9     // Define a NumberList object.
10    NumberList list;
11
12    // Append some values to the list.
13    list.appendNode(2.5);
14    list.appendNode(7.9);
15    list.appendNode(12.6);
16    return 0;
17 }
```

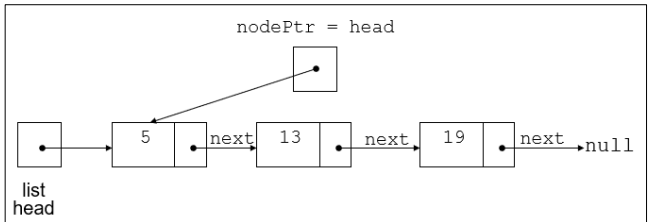
(This program displays no output.)

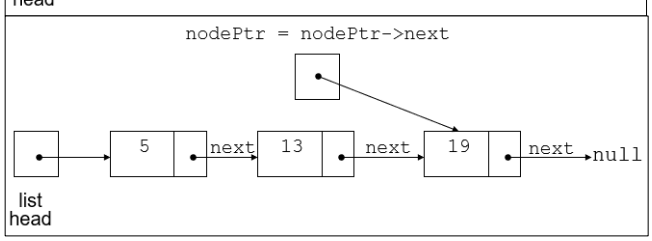
Traversing a Linked List

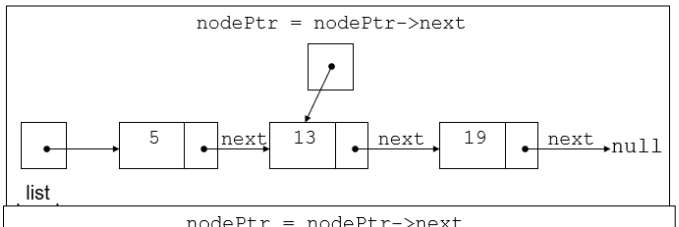
```
void NumberList::printList() const
{
    ListNode * nodePtr = head;
    while (nodePtr) { // while nodePtr != 0
        cout << nodePtr->value << "\t";
        nodePtr = nodePtr->next;
    }
    cout << endl;
}
```

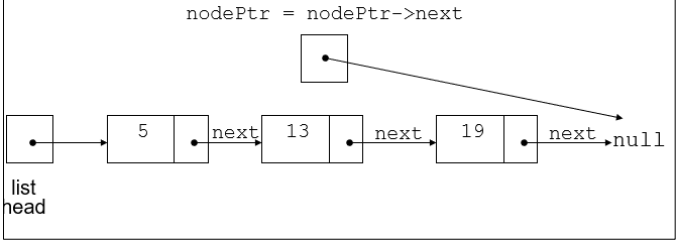
- nodePtr points to the node containing 5, then the node containing 13, then the node containing 19, then points to the null pointer, and the list traversal stops

- see the way the while loop works:

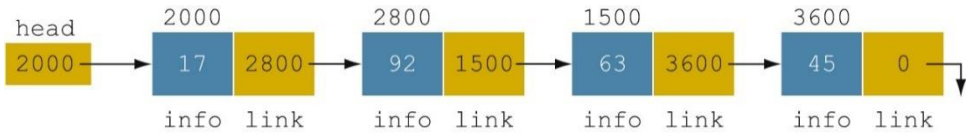








Linked List Properties



	Value	Explanation
`head`	2000	
`head->info`	17	Because `head` is 2000 and the `info` of the node at location 2000 is 17
`head->link`	2800	
`head->link->info`	92	Because `head->link` is 2800 and the `info` of the node at location 2800 is 92

- **`current = head;`**

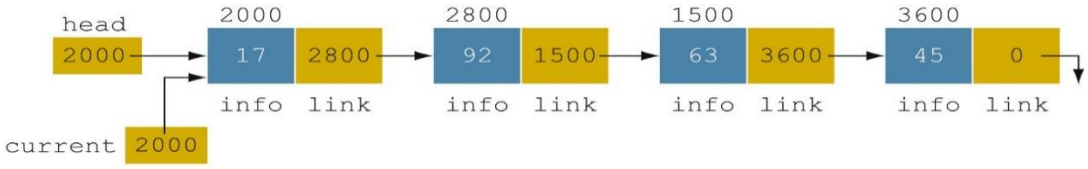
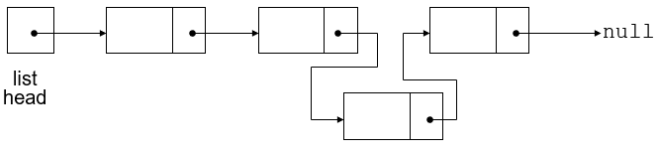


FIGURE 17-5 Linked list after the statement `current = head;` executes

	Value
`current`	2000
`current->info`	17
`current->link`	2800
`current->link->info`	92



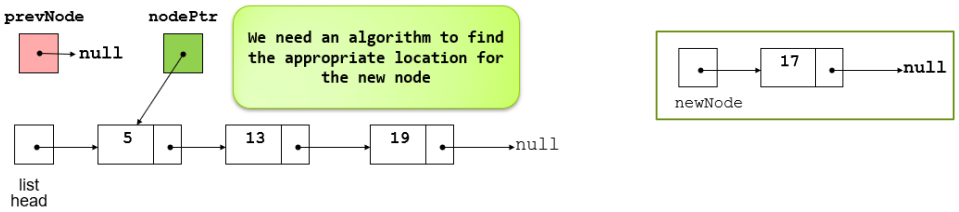
Inserting a node into a linked list

- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node before the point of insertion
  - pointer to locate the node after the point of insertion
- New node is inserted between these two nodes

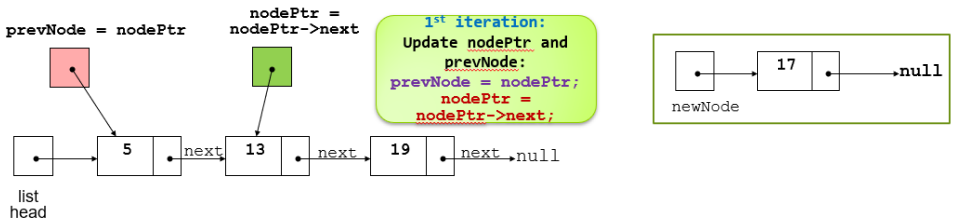
- Algorithm: Numbers need to be in ascending order

```
ListNode* nodePtr = head;
ListNode* prevNode = NULL;
while(nodePtr != NULL && nodePtr->value < newNode->value)
{
  prevNode = nodePtr;
  nodePtr = nodePtr->next;
}
```

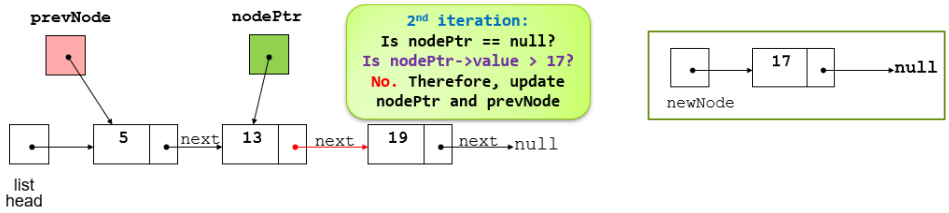
- Process:
- 1.



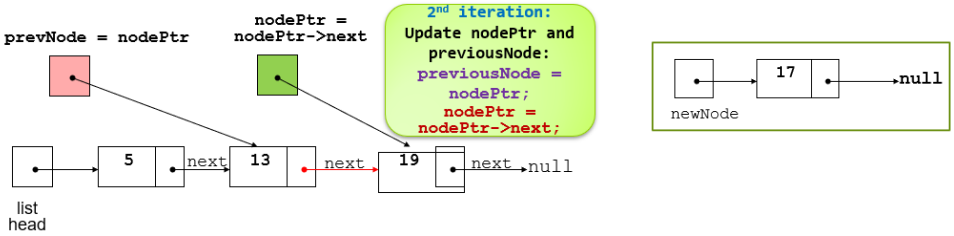
- 2.



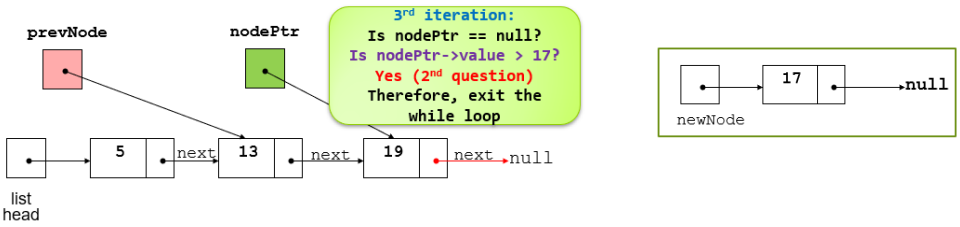
- 3.



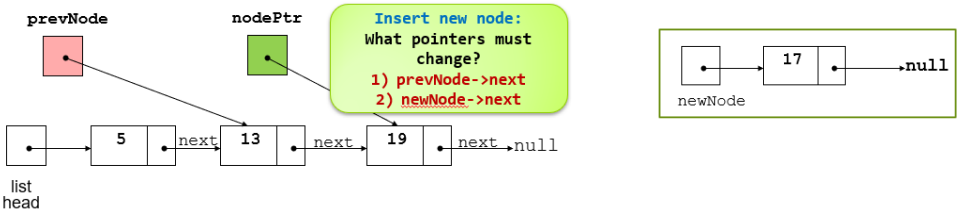
- 4.



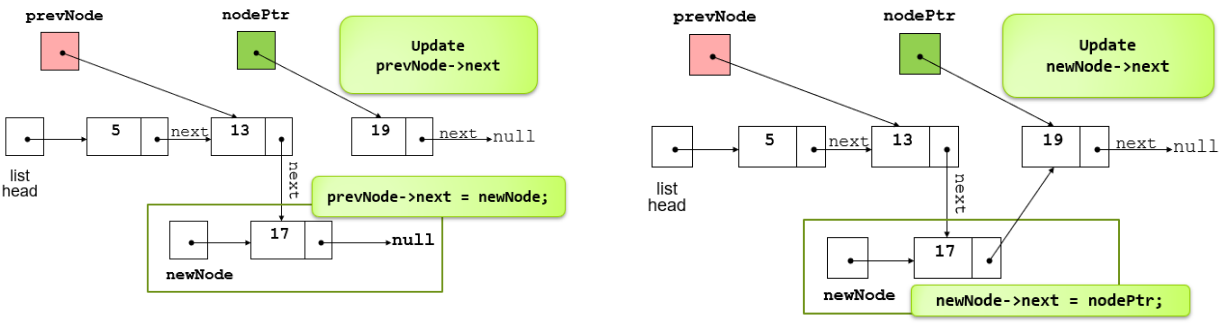
- 5.



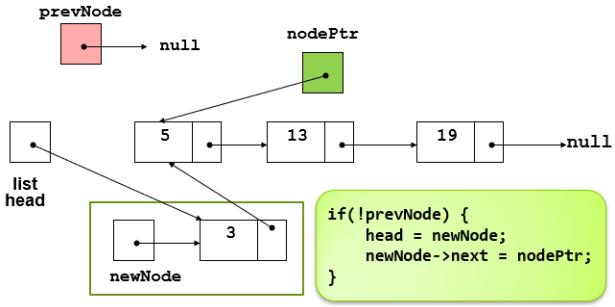
- 6.



- 7.



- What if the new node contains a value smaller than all other values?
- In this case, new node should become the head
    - See 1. On previous page
        - Because 5 > 3, the while loop will never execute:



- We do not need to deallocate (delete) **prevNode**, **nodePtr**, **newNode** when we're done inserting a node:
    - these variables were used to temporarily store addresses
    - now the addresses are permanently stored in the linked list structure
    - the temp pointer variables can just go out of scope when we exit the function
- C++ code used to insert a node

```
void NumberList::insertNode(double num)
{
   ListNode *newNode;  // To point to the new node
   ListNode *nodePtr;  // To move through the list
   ListNode *prevNode; // To store previous ptr

   // Allocate a new node and store num there.
   newNode = new ListNode; // must use "new"
   newNode->value = num;   // set value to num
   newNode->next = NULL;   // set next to NULL

   // If there are no nodes in the list
   // make newNode the first node:
   if (!head) // "If head is null"
      head = newNode; // both are pointers
   else  // Otherwise, find location
   {
      nodePtr = head; // Initialize to head
      prevPtr = NULL;
      // Find the correct location:
      while(nodePtr && nodePtr->value <
                       newNode->value) {
         prevNode = nodePtr;
         nodePtr = nodePtr->next;
      }
      if(!prevNode) { // insert at the head
         head = newNode;
         newNode->next = nodePtr;
      } else { // insert in the middle
         prevNode->next = newNode;
         newNode->next = nodePtr;
      }
   }
}
```

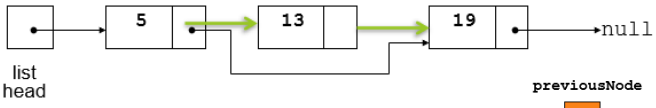
- You can insert a node using only one iterator pointer: **nodePtr**
    - **nodePtr** can be treated as "previous node"
    - **nodePtr->next** can be treated as "next node"

Example: LinkedList

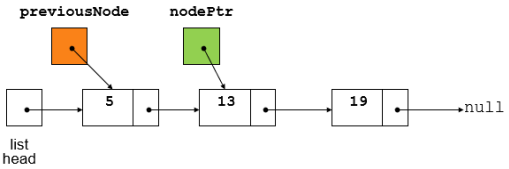
Deleting a node

```
ListNode * newNode = new ListNode;
newNode->value = num;    // set value to num
newNode->next = NULL;    // set next to NULL
```

- Every node is dynamically allocated:
- When a node is deleted from the list, it should be deallocated from memory
- "next" pointer that used to point to the deleted node must be updated to point to the deleted node's "next"



- In order to delete a node, we use 2 pointers:
  - One to locate the node to be deleted
  - One to point to the node before the node to be deleted



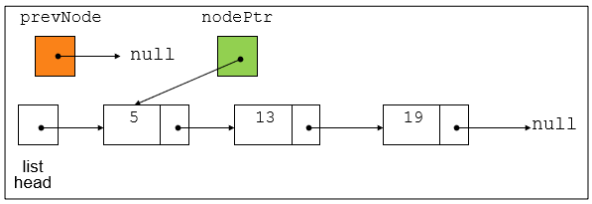
Example of deleting a node containing: num = 13

Initialization:
nodePtr points to 1st element;
prevNode points to null.

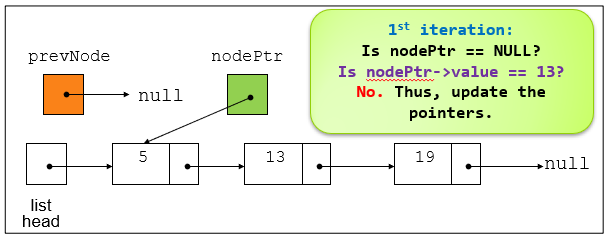
```
ListNode * nodePtr = head;
ListNode * prevNode = 0;
```

```
ListNode* nodePtr = head;
ListNode* prevNode = NULL;
while(nodePtr != NULL && nodePtr->value != num)
{
  prevNode = nodePtr;
  nodePtr = nodePtr->next;
}
```

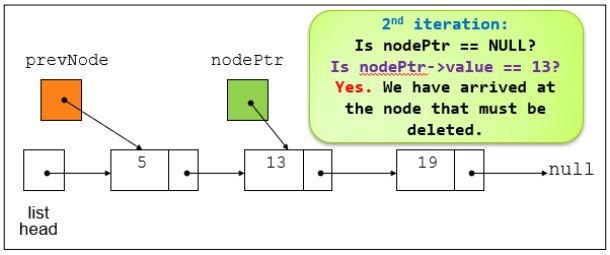
- 1.



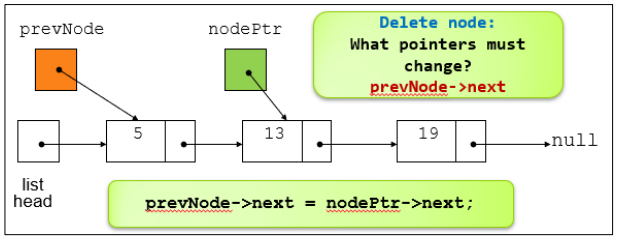
- 2.



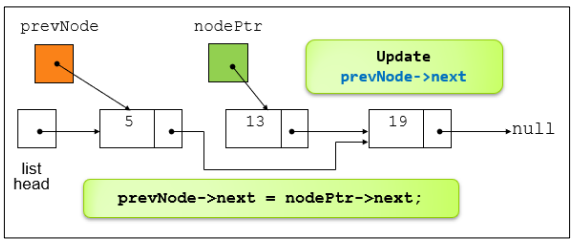
- 3.

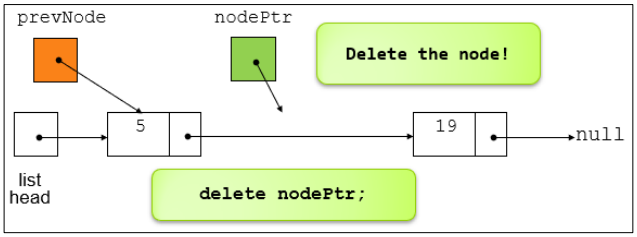


- 4.

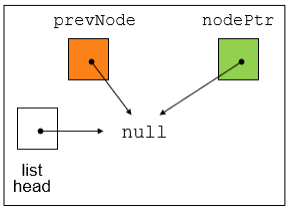


- 5/6



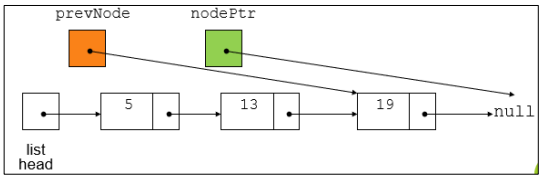


Special Cases When Deleting a node

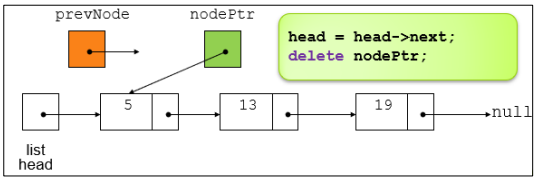


- If the list is empty:
  - Just exit the function: nothing to delete!

- Desired node is not found:
  - If nodePtr ==NULL, just exit the function: nothing to delete!



- First node needs to be deleted:
  - Must update the head pointer



C++ code for deleting a node

```
void NumberList::deleteNode(double num)
   {
       ListNode *nodePtr = head;  // current
       ListNode *prevNode = NULL; // previous
      // Attempt deletion only on non-empty list:
      if(head) { // "If head is NOT null"
          while(nodePtr && nodePtr->value != num) {
              prevNode = nodePtr;
              nodePtr = nodePtr->next;
          }
          if(nodePtr) { // if the value has been found
                if(nodePtr == head) { // delete head
                head = head->next;
                delete nodePtr;
              }
              else { // Otherwise, delete from the middle
                prevNode->next = nodePtr->next;
                delete nodePtr;
              }
          }
       }
   }
```

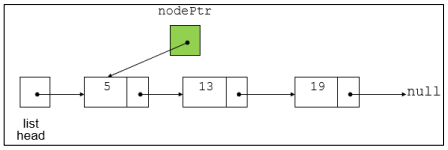
- When deleting a node, you cannot delete using nodePtr only as the address will be lost, you need to use a temporary pointer

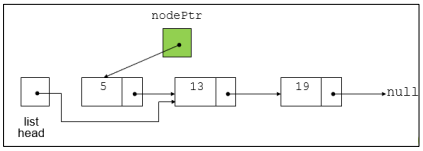
Destroying a linked list

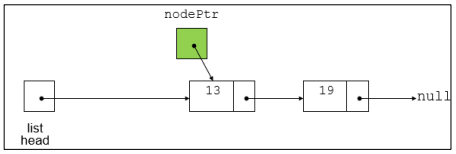
- Linked List destructor must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - Free the node's memory
- Set the list head to NULL

- **The deletion process:**
- At every iteration:
  - Move head forward;
  - Delete nodePtr;
  - Set nodePtr to head

```
ListNode* nodePtr = head;
while(nodePtr)
{
  head = head->next;
  delete nodePtr;
  nodePtr = head;
}
```







loops until nodePtr == NULL

Example: LinkedList

18.3 Linked List Template

- When declaring a linked list, must specify the type of data to be held in each node
- **Using templates, can declare a linked list that can hold data type determined at list definition time**
- See NumberList above
- Everything wrong with having a linked list for just doubles:
  - It can only store numbers!
  - What if you want a linked list of chars, or names, or wizards, or students, or unicorns?
  - Solution: implement a generic Linked list
    - Use a template class
    - Choose type when creating a list object:

```
LinkedList<double> doubleList;
LinkedList<Unicorn> unicornList;
```

- To change the linked list class to be template class:
  - All we need to do is change double to <T>

```
// LinkedList.h
template <class T>
class LinkedList
{
   private:
     struct ListNode // node structure
     {
        T value;               // The value in this node
        struct ListNode *next; // Address of next node
     };
     ListNode *head;           // List head pointer
     // The rest of the class functions go here
};
```

- All linked list algorithms still apply
- Operators applied to node values must be available for value's type
- What operator must be implemented in class T for this code to work?

```
template <class T>
void LinkedList<T>::deleteNode(T searchValue)
{
   ListNode *nodePtr, *previousNode = NULL;
   if (!head) return;

   if (head->value == searchValue)
   {
      // rest of the algorithm goes here
```

```
template <class T>
   void LinkedList<T>::appendNode(T num)
   {
      ListNode *newNode;  // To point to a new node
      ListNode *nodePtr;  // To move through the list

      // Allocate a new node and store num there.
      newNode = new ListNode; // dynamically allocate
      newNode->value = num;   // what assumption is made?
      newNode->next = NULL;   // set next to NULL

      // If there are no nodes in the list
      // make newNode the first node:
      if (!head) // "If head is null"
         head = newNode; // both are pointers

      else  // Otherwise, insert newNode at end.
      {
         // Initialize nodePtr to head of list.
         nodePtr = head;

         // Find the last node in the list.
         while (nodePtr->next) { // "while next != 0"
            nodePtr = nodePtr->next;
         }
         // Insert newNode as the last node
         nodePtr->next = newNode;
      }
   }
```

Allocating new Nodes

```
// Allocate a new node and store num there.
newNode = new ListNode; // allocate on the heap
newNode->value = num;   // set value to num
newNode->next = NULL;   // set next to NULL
```

- Adding a constructor to the ListNode can simplify the code:

```
// LinkedList.h
template <class T>
class LinkedList
{
   private:
     struct ListNode // node structure
     {
        T value;                // The value in this node
        struct ListNode *next;  // Address of next node
        ListNode(T nodeValue) : value(nodeValue) {
              next = NULL;
        }
     };
     ListNode *head;            // List head pointer
     // The rest of the class functions go here
};
```

- Method to append a node:

```
template<class T>
   void LinkedList<T>::appendNode(T num)
   {
      ListNode *newNode;
      ListNode *nodePtr;  // Iterator to move through list

      // Use a single line of code to create a new node
      newNode = new ListNode(num);


      // If there are no nodes in the list
      // make newNode the first node:
      if (!head) // "If head is null"
         head = newNode; // both are pointers


      // rest of the algorithm goes here
```

Linked List Template

- Can also make **ListNode** a template class:

```
template <class T>
struct ListNode // node structure
{
    T value;                    // The value in this node
    struct ListNode<T> *next;  // Address of next node
    ListNode(T nodeValue) : value(nodeValue) {
       next = NULL;
    }
};
template <class T>
class LinkedList
{
   private:
       ListNode<T> *head;             // List head pointer
   // The rest of the class functions go here
};
```

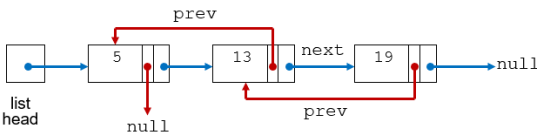
```
template<class T>
void LinkedList<T>::appendNode(T num)
{
   ListNode<T> *newNode;  // To point to a new node
   ListNode<T> *nodePtr;  // To move through the list

   // Use a single line of code to create a new node
   newNode = new ListNode<T>(num);


   // If there are no nodes in the list
   // make newNode the first node:
   if (!head) // "If head is null"
      head = newNode; // both are pointers


   // rest of the algorithm goes here
```

Overloading << for the template LinkedList

```
template <class T>
class LinkedList; // forward declaration
template <class T>
ostream& operator<<(ostream& o, LinkedList<T>& ll); //prototype
template <class T
class LinkedList
{ // The rest of the class functions go here
   friend ostream& operator<< <>(ostream& o,
                                 LinkedList<T>& ll);
};
template <class T>
ostream& operator<<(ostream& o, LinkedList<T>& ll)
{ // operator implementation goes here
}
```

18.4 Variations of the Linked List: Doubly-Linked List

- doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list



- blue shows normal linked list
- Red shows doubly-linked list

- Traversal in either direction is possible
  - In addition to a head, store a tail – pointer to the last node
- For complex operations such as insertion and deletion, no extra previous node pointer is required.
  - This means that less temp variables may need to be used.

- Modified Node Struct for this linked list:

```
struct ListNode    // node structure
     {
       double value;    // The value in this node
       ListNode *next;  // Address of next node
       ListNode *prev;  // Address of previous node
       ListNode(double nodeValue) : value(nodeValue) {
              next = NULL;
              prev = NULL;
       }
     };
```

- Template:

```
template <class T>
class LinkedList
{
   private:
     struct ListNode // node structure
     {
        T value;         // The value in this node
        ListNode *next;  // Address of next node
        ListNode *prev;  // Address of prev. node
        ListNode(T nodeValue) : value(nodeValue) {
              next = NULL;
              prev = NULL;
        }
     };
     ListNode * head;            // List head pointer
     ListNode * tail;            // List tail pointer
     // The rest of the class functions go here
};
```

Append a node of type T

- (normal linked list on left, doubly linked on right)

```
template <class T>
void LinkedList<T>::appendNode(T num)
{
   ListNode *newNode;  // To point to a new node
   ListNode *nodePtr;  // To move through the list

   newNode = new ListNode(num);

   if (!head) // "If head is null"
      head = newNode; // both are pointers
   else  // Otherwise, insert newNode at end.
   {
      nodePtr = head;
      // find the end of the list
      while (nodePtr->next) { // "while next != 0"
         nodePtr = nodePtr->next;
      }
      nodePtr->next = newNode;
   }
}
```

```
template <class T>
void LinkedList<T>::appendNode(T num)
{
   ListNode *newNode;  // To point to a new node
   ListNode *nodePtr;  // To move through the list

   newNode = new ListNode(num);

   if (!head)
   {
      head = newNode;
      tail = newNode;
   }
   else // Otherwise, insert newNode at the end
   {
      tail->next = newNode;
      newNode->prev = tail;
      tail = newNode;
   }
}
```

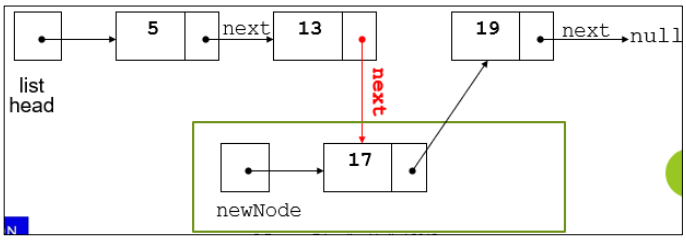
1) Update tail as well as the head

2) Instead of iterating, go directly to the last element

3) Update the previous pointer and the tail

Insert a Node of Type T (Singularly linked list):

- Singularly linked list:

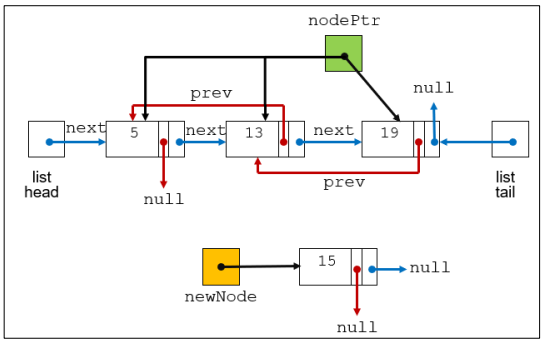



```
template <class T>
void LinkedList<T>::insertNode(T num)
{
   ListNode *newNode;  // To point to a new node
   ListNode *nodePtr;  // To move through the list
   ListNode *prevNode; // To store prev. node position
   newNode = new ListNode(num); // dynamically allocate

   if (!head){ // "If head is null"
      head = newNode; // both are pointers
      newNode->next = NULL;
   }
    else  // Otherwise, insert newNode at correct pos
   {
      nodePtr = head;
      while (nodePtr!=NULL && nodePtr->value < num)
      {
           prevNode = nodePtr;
           nodePtr = nodePtr->next;
      }
    if (prevNode == NULL) // Will be 1st node in list
       {
           head = newNode;
           newNode->next = nodePtr;
       }
       else  // Otherwise, insert newNode at correct pos
       {
           prevNode->next = newNode;
           newNode->next = nodePtr;
       }
    }
 }
```

Insert a Node of Type T (Doubly linked list):

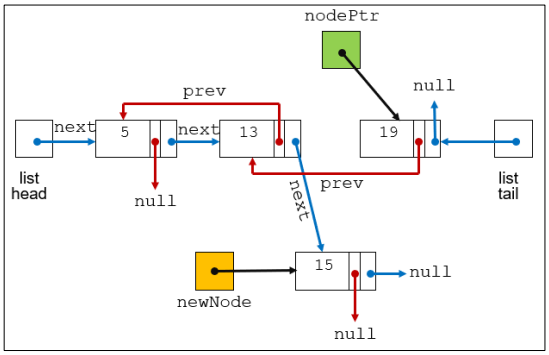
- Doubly linked list

- 1. (searching for correct place)




```
while (nodePtr != NULL && nodePtr->value < newValue)
{
   nodePtr = nodePtr->next;
}
```

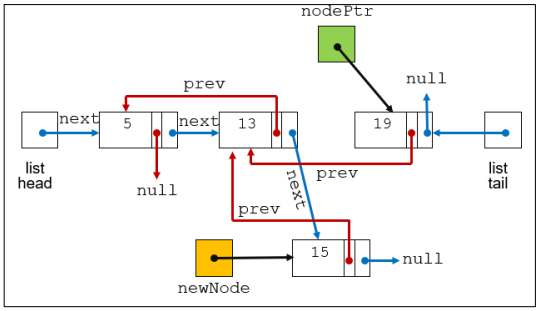
- 2. (Attaching, inserting, the node)




```
nodePtr->prev->next = newNode;
```

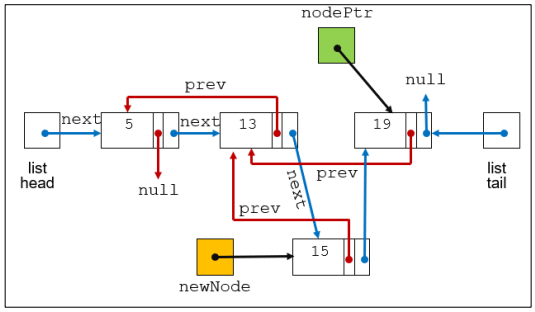
- Make sure that when adding the addresses, you do not loose any important information by overwriting it.

- 3. (Attaching, inserting, the node)



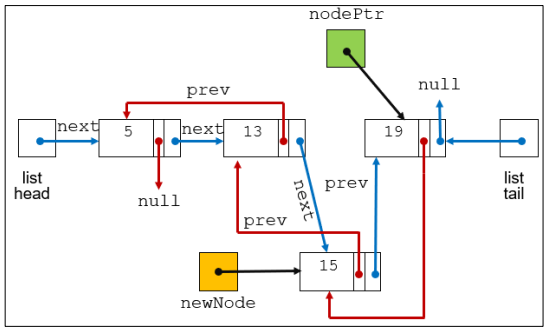

```
newNode->prev = nodePtr->prev;
```

- 4. (Attaching, inserting, the node)




```
newNode->next = nodePtr;
```

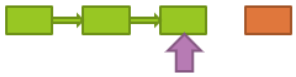
- 5. (Attaching, inserting, the node)

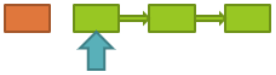



```
nodePtr->prev = newNode;
```

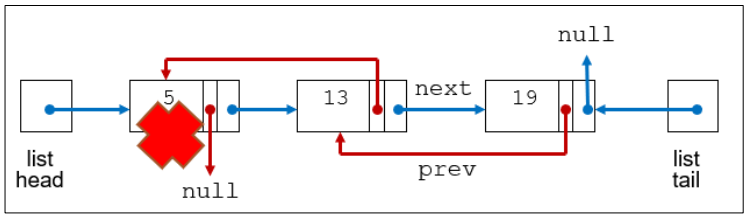
Insertion a Node of Type T:

```
template <class T>
void LinkedList<T>::insertNode(T newValue) {
    ListNode *newNode, *nodePtr = NULL;
    newNode = new ListNode(newValue);

    if (!head) { // if list is empty
        head = newNode;
        tail = newNode;
    }
    else // Otherwise, insert newNode
    {
        nodePtr = head;
        // Skip nodes whose value is less than newValue
        while (nodePtr != NULL && nodePtr->value < newValue)
        {
            nodePtr = nodePtr->next;
        }
        if (!nodePtr) // if new node is to be last node
        {
          tail->next = newNode;
          newNode->prev = tail;
          tail = newNode;
        }
        // If the new node is to be the 1st in the list,
        // insert it before all other nodes.
        else if (nodePtr->prev == NULL) {
           newNode->next = nodePtr;
           nodePtr->prev = newNode;
           head = newNode;
        }
         else // Otherwise, insert it after the prev. node.
        {
           nodePtr->prev->next = newNode;
           newNode->prev = nodePtr->prev;
           newNode->next = nodePtr;
           nodePtr->prev = newNode;
        }
    }
}
```





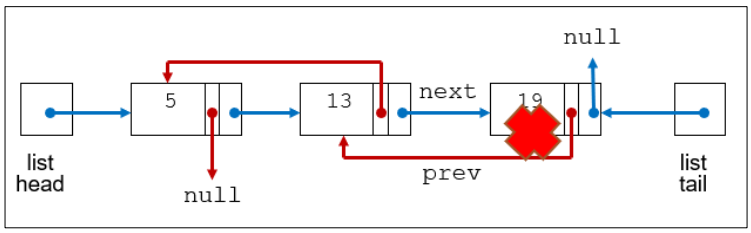
Deletion of Node (head):




```
head = head->next;
delete head->prev;
head->prev = NULL;
```

- This method will fail if:
  - Head = NULL
  - There is only one element in the list: (head->next == NULL)
- We can fix this:
  - If head->next == NULL, then the head should be deleted straight away, then set Head to NULL,
  - Make sure that head is not Null and that head->next is not NULL either

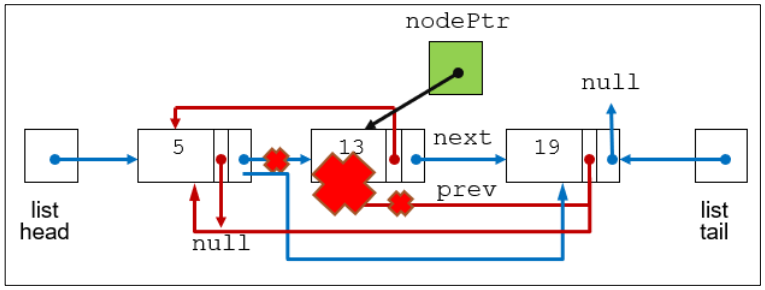
Deletion of a Node (tail):




```
tail = tail->prev;
delete tail->next;
tail->next = NULL;
```

- This method will fail if:
  - Tail = NULL
  - There is only one element in the list: (head->prev == NULL)
- We can fix this:
  - If tail->prev == NULL, then the tail should be deleted straight away, then set tail to NULL, don't try to interact with its predecessors.
  - Make sure that head is not Null and that head->next is not NULL either

Deletion of a Node (middle):



- assumption that code is in while loop and the correct node has been found

```
nodePtr->prev->next = nodePtr->next;
nodePtr->next->prev = nodePtr->prev;
delete nodePtr;
```

- Deletes blue arrow, then red arrow and then the value 13.

Reasons for doubly linked lists

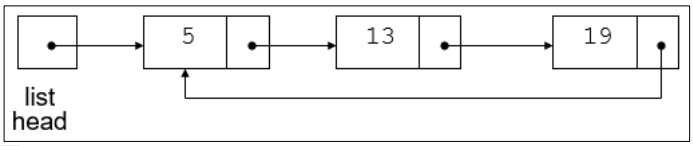
- More efficient if
  - You insert at the front and at the end
  - You traverse front to back and back to front
- Memory wise, doubly linked lists are bulkier but save more time and thus are better.

Summary for 18.4

- Doubly Linked lists are bi-directional linear structures
- Two new variables are introduced:
- "previous" pointer added to each node
- "tail" pointer added to the linked list class
- Singly Linked list algorithms are adjusted to take care of "previous" and "tail" pointers

18.4 Variations of the Linked List: Circular Linked List

- circular linked list: the last node in the list points to the first node in the list, **not to** the **null** pointer
- Useful for applications that are cyclic in nature, e.g. multiplayer game where each player has a turn, or timesharing between users

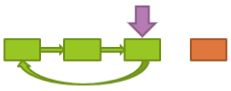


Appending a node of Type T

```
template <class T>
void LinkedList<T>::appendNode(T num)
{
   ListNode *newNode;  // To point to a new node
   ListNode *nodePtr;  // To move through the list
   newNode = new ListNode(num);
   if (!head) {
      head = newNode;
      newNode->next = head;
    }
    else {
      nodePtr = head;
      // find the end of the list
      while( nodePtr->next != head ) { // "while next != head"
         nodePtr = nodePtr->next;
      }
      nodePtr->next = newNode;
   }
   newNode->next = head;
}
```

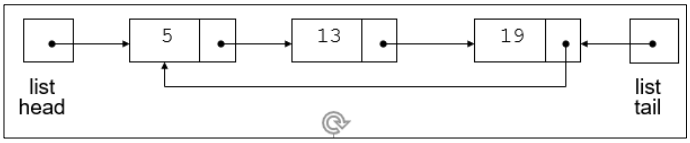
1) Compare to the head instead of NULL

2) Re-link the chain by setting `newNode->next` to head

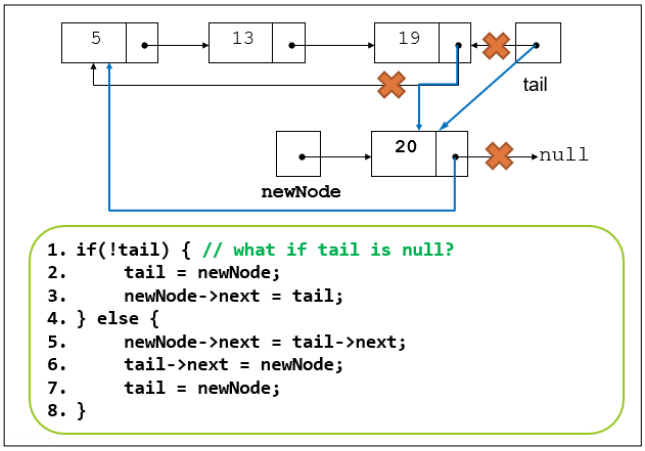


- Same methods for singular linked list will not work, while loop condition needs to be updated in order to loop through nodes correctly
- This is working with the assumption that the head is the insertion point into the linked list

- Tail is better pointer to keep because tail->next will point you to the head
  - tail pointer has more useful information



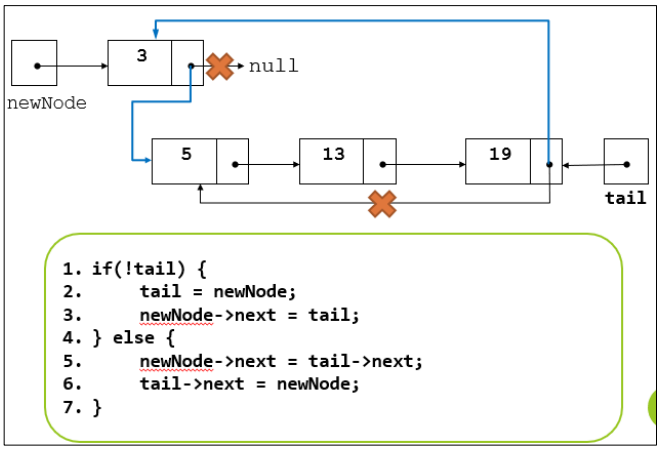
- Appending to linked List(adding at the end)




```
1. if(!tail) { // what if tail is null?
2.      tail = newNode;
3.      newNode->next = tail;
4. } else {
5.      newNode->next = tail->next;
6.      tail->next = newNode;
7.      tail = newNode;
8. }
```

- If we change the order of operations that the different pointers point to, we can potentially lose information.
  - Order is important

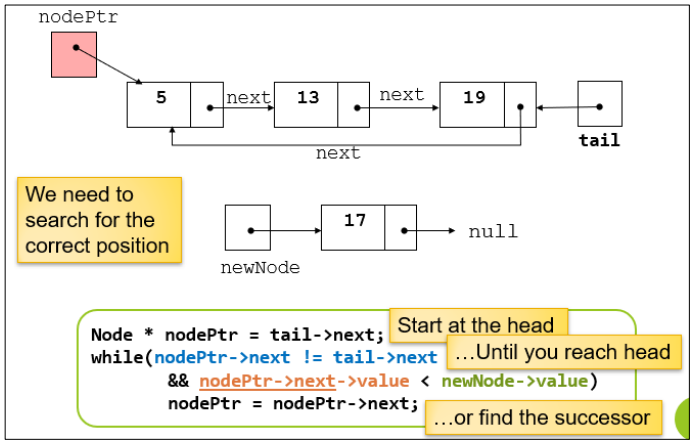
- Prepending to a Linked List(Adding at the beginning)

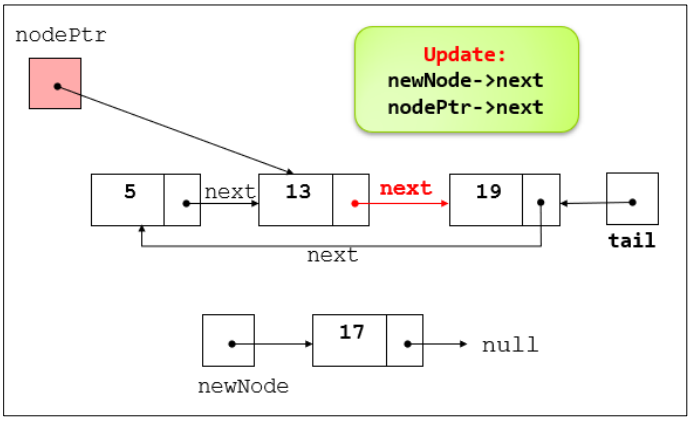


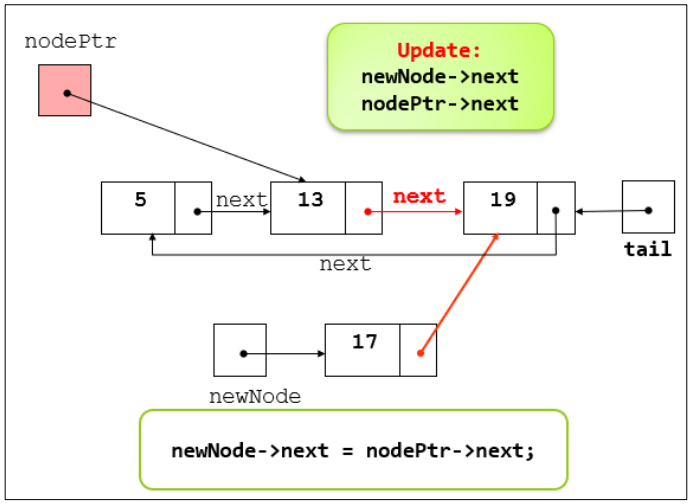

```
1. if(!tail) {
2.      tail = newNode;
3.      newNode->next = tail;
4. } else {
5.      newNode->next = tail->next;
6.      tail->next = newNode;
7. }
```

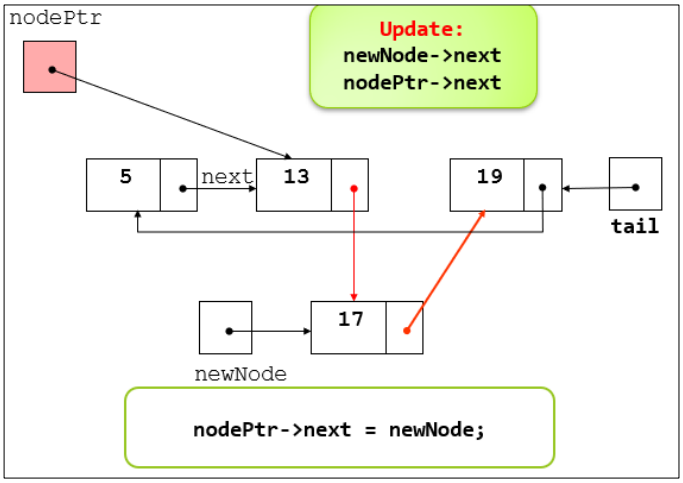
- Code is very similar to add item at the end of the list except you just don't have to update the tail pointer

- Inserting a Node into the middle of a Linked List



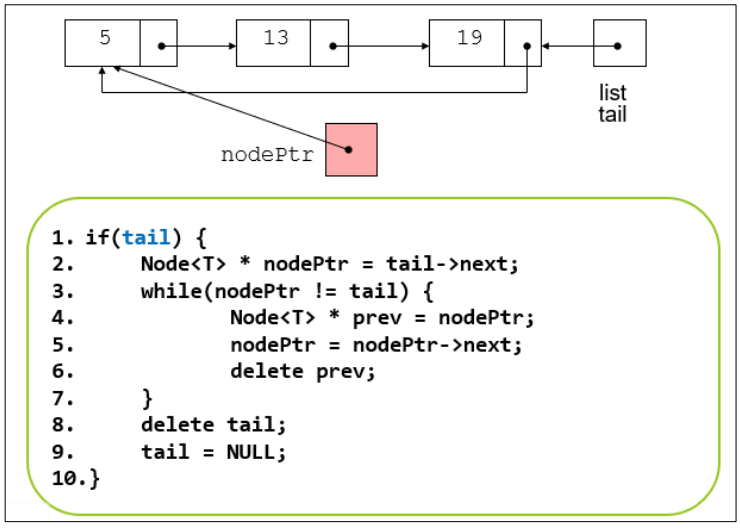




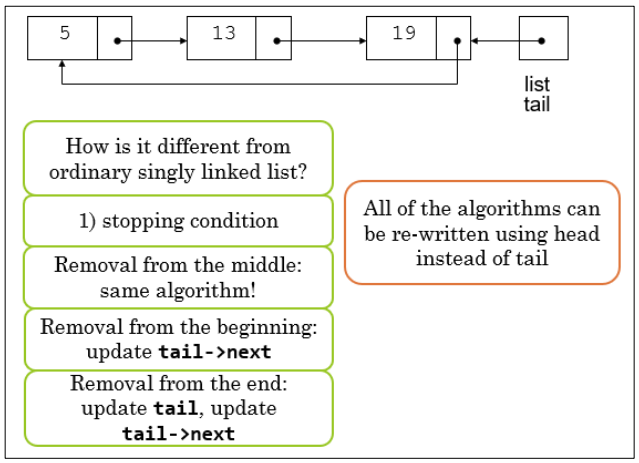


- When inserting into a circular linked list, you should cater for the following cases:
- The above-mentioned cases can also work if written using the head rather than the tail. Maybe some updates will be needed;
    - Inserting into empty list
    - Prepending
    - Appending
    - Inserting into the middle of the list

Destructing a Circular Linked List

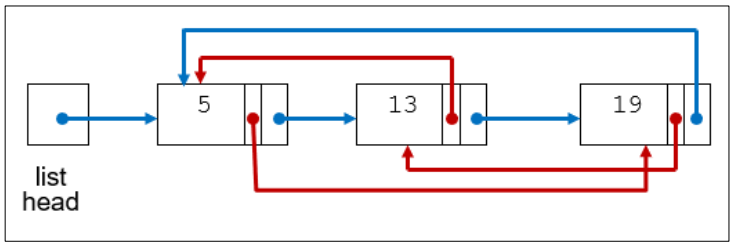


Deleting a node from a circular Linked List



- Many of the methods will be the exact same as the deletion of nodes from a singly linked list

Circular Doubly Linked List



- **LOTS OF HINTS AT THIS BEING IN AN EXAM!!**

Summary of Circular Linked Lists

- Circular linked lists are cyclic linear structures
- Last node's next points to the first element
- **Store the tail instead of the head**
- Singly-linked list algorithms are adjusted to work with tail and tail->next

18.5 The STL List and Forward_List Containers

- **Note to self: 18.5 was only presented halfway through 19.5…**

Recap:

- Standard Template Library(STL): A library containing templates for frequently used data structures and algorithms
- Two types of container classes in STL:
  - Sequence containers: organize and access data sequentially, as in an array. These include **vector, deque, and list**
  - **Associative containers:** store element in order, use keys to allow data elements to be quickly accessed.

STL List

- Lists are sequence containers found in the Standard Template Library(STL)
- STL List Implemented as doubly-Linked lists.
- Similar to **forward_list** (discussed later- are implemented as singly-linked list)

- Lists generally perform better than other standard sequence containers (**array, vector and deque**)
  - **Insertion**-Perform insertion faster than vectors because they don't have to shift elements when a new element is added
  - **Efficient at adding elements at their back** because they have built in pointer to last element
- Lists lack direct access to elements by their position.
  - **e.g.to access a fifth element in list, iterate from a known position(like beginning or end) to that position (linear time)**
- Each node keeps two pointers-one to next element and one to preceding element.

- **Allows iteration in both directions.**
- Consume Extra memory to keep the linking information associated to each element.

To use a list include header file #include<list>

```
#include<iostream>
#include<string>
#include<list>
int main{
list<int> name; //empty list of ints
list<string> first;  // empty list of string
list<int> second (4,100);//four ints with value 100
}
```

- Defining a list:

When defining list container objects- contents are initialized depending on the constructor version used.

The general format for list definitions

```
1.list<dataType> name; //creates empty list
  e.g. list<int> name; //empty list of ints

2.list<dataType> name(size,value)//fill constructor
    e.g list<int> second (4,100);//100 100 100 100

3.list<dataType> name(list2)//-Copy constructor

4.list<dataType> name(iterator1, iterator2) //Range Constructor
```

- *See Table 18-1 (pg 1186) for a list of constructors*

- ***See constructingList.cpp (shows different ways of initializing and defining a list)***

STL member functions

- STL list implements the following member functions:
  - **locating beginning, end of list:** `front, back, end`
  - **adding elements to the list**: `insert, merge, push_back, push_front`
  - **removing elements from the list**: `erase, pop_back, pop_front, unique`
- *See table 18-2(pg 1188) for a list of member functions*

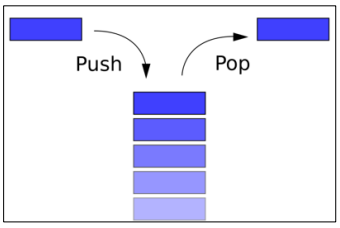
- ***See ListMemberFunctions.cpp (use of different member functions)***

STL Forward_List Container

- **forward lists** are sequence containers implemented as singly-linked lists.
- You can only step forward in a forward_list.
- Each node keeps only one pointer to the next node
- List container keeps two links in a node
- A **forward_list** uses slightly less memory than a **list** and takes slightly less time for inserting and removing nodes.
- Provides most, but not all, of the same member functions as the list container
- See www.cppreference.com or www.cplusplus.com

Chapter 19 – Stacks and Queues

19.1 Introduction to the Stack ADT

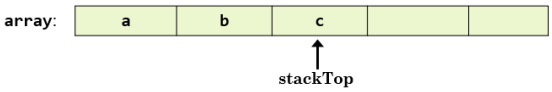


- **Stack**: a last in, first out (LIFO) data structure
- Examples:
  - Plates in a cafeteria, books on a coffee table…
  - Return addresses for function calls
  - Stacks are used extensively in computer architecture
  - Local variables are stored on a system stack
- **Implementation**:
  - **static**: fixed size, implemented using an array
  - **dynamic**: variable size, implemented using a linked list

Stack Data Structure

- Main operations on a stack:
  - **push**
    - Add element to the top
  - **top**
    - Access top element
  - **pop**
    - Remove top element
  - **isEmpty**
    - Check if the stack is empty
  - **isFull**
    - Check if the stack if full

Static Stack ADT: Array



- Stacks can be implemented using an array or a linked list
- Array: a random-access structure
- Stack interface has to be written such that access is restricted to the top element
- Variables:
  - **T * array;**
  - **int maxStackSize;**
  - **int stackTop;**

- The size of **array** is stored in the **maxStackSize** variable.
- **stackTop < 0** means that the stack is empty
- To access the top element of the stack, we'll use:
  - **array[stackTop]**

```
template <class Type>
class Stack
{
public:
   Stack(int stackSize = 100); // Constructor
   Stack(const Stack<Type>& otherStack); // Copy constructor
   const Stack<Type>& operator=(const Stack<Type>&); // Do we need this?
   ~Stack();             // Destructor
   bool isEmpty() const;// determine whether the stack is empty
   bool isFull() const; // determine whether the stack is full
   void push(Type newItem);// add newItem to the stack
   Type top() const;    // return the top element of the stack
   void pop(Type&);    // remove the top element of the stack
private:
   int maxStackSize; // maximum stack size
   int stackTop;     // point to the top of the stack
   Type *array; // array that holds the stack elements
   void copyStack(const Stack<Type>& otherStack); // helper
};
```

- Constructor(No error checking (left) and error checking, (right)):

```
template <class Type>
Stack<Type>::Stack(int stackSize)
{ // What is missing in this constructor?
       maxStackSize = stackSize;
       stackTop = -1;
       array = new Type[maxStackSize];
}
```

```
template <class Type>
Stack<Type>::Stack(int stackSize)
{
       if(stackSize <= 0)
       {
         cout << "Size must be positive." << endl;
         cout << "Creating an array of size 100." << endl;
         maxStackSize = 100; // default size
       }
       else
         maxStackSize = stackSize;

       stackTop = -1;
       array = new Type[maxStackSize];
}
```

- In main, if we say:
  - **Stack<int> intStack(10);**
    - **maxStackSize == 10**
    - **stackTop == -1**
    - The stack is empty.

- If **stackTop < 0**, then the stack is empty.

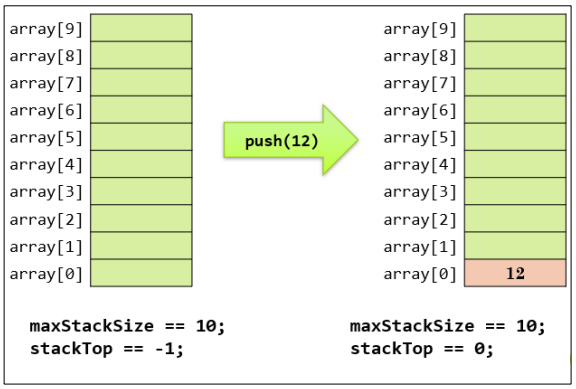
```
template <class Type>
bool Stack<Type>::isEmpty() const
{
       return (stackTop == -1);
}
```

- If **stackTop == maxStackSize - 1,** then the stack is full.

```
template <class Type>
bool Stack<Type>::isFull() const
{
       return (stackTop == maxStackSize - 1);
}
```

- Push function:

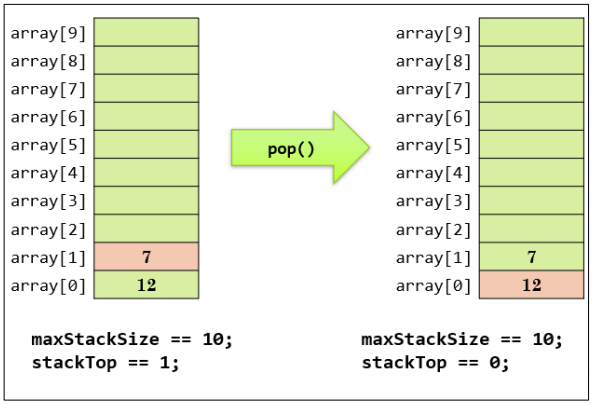
```
template <class Type>
void Stack<Type>::push(Type newItem)
{
      if (isFull())
      {
        cout << "Cannot add to a full stack.";
        cout << endl;
      }
      else
      {
        stackTop++; // increment stackTop
        // add newItem to the top of stack:
        array[stackTop] = newItem;
      }
}
```



- Pop Function (removes whatever is on top of stack):

```
template <class Type>
void Stack<Type>::pop(Type& item)
{
	if (isEmpty()) {
		cout << "Cannot remove from ";
		cout << "an empty stack." << endl;
	}
	else {
		item = array[stackTop--];
		// decrement stackTop
	}
}
```

```
Stack<int> st(10);
st.push(3);
int x;
st.pop(x);
// Now x = 3
```



- Top Function (returns a read only version of value on top of stack):

```
template <class Type>
Type stackType<Type>::top() const
{
	//if stack is empty, throw an exception:
	if (isEmpty()) {
		throw “Stack is empty";
	}
	// otherwise, return array[stackTop]:
	return array[ stackTop ];
}
```

- Applying the abovementioned methods:

```
int main()
{
	Stack<int> stack(10);

	stack.push(12);
	stack.push(7);
	cout << stack.top() << endl; // outputs 7
	int var;
	stack.pop(var); // removes 7, puts 7 into var
	cout << stack.top() << endl; // outputs 12
	stack.pop(var); // removes 12, puts 12 into var

	return 0;
}
```

- Can override the pop() method so that you do not need to parse a value in each time. Instead of making it void, you can make it to return the value that is being popped

Stack Applications

- *Uses can be used a lot in IDE's*
- **Reverse the order of elements**
- **Code parsing**: eg; check code for matching { } brackets
  - When you encounter { , push it on the stack
  - When you encounter }, pop the stack
  - If there's nothing to pop, then opening { is missing
  - If the stack is not empty at the end of the program, then closing } is missing
- **Backtracking**
  - A calls B, B calls C, C calls D
  - When you enter a function, push it on the stack
  - When the function exist, pop it from the stack
  - Top element will hold the function you came from

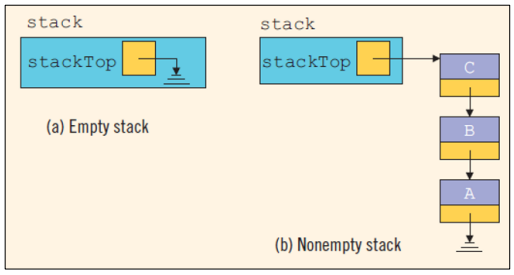
Array or Linked List

- Disadvantages of array implementation:

- Fixed size
- Random access is available, but not used

- What if we use a linked list instead?
  - Dynamic memory allocation of each element: **no maximum size**
  - Linked structure has no random access
- Arrays: **stackTop** is used to store the **index** of the top stack element
- Linked lists: **stackTop** is a **pointer** storing the address of the top stack element

Stack ADT: Linked List




```
template <class Type>
class DynamicStack {
    private:
        struct Node
        {
                Type data;
                Node *next;
        };
        Node *stackTop; // pointer to the top of the stack
    public:
        DynamicStack();  // constructor
        ~DynamicStack(); // destructor
        DynamicStack(const DynamicStack<Type>& otherStack); // copy constr.
        const DynamicStack<Type>& operator=(const DynamicStack<Type>&);
        bool isEmpty() const; // determine whether the stack is empty
        void push(const Type& newItem); // add element to the top
        Type top() const;           // access element at the top
        void pop(Type&);            // remove element from the top
};
```

- Constructor:

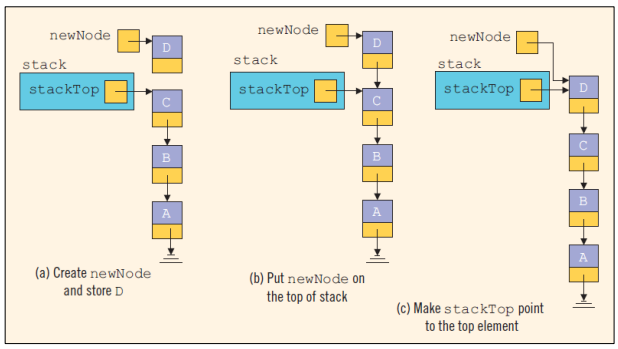
```
template <class Type>
DynamicStack<Type>::DynamicStack()
{
    stackTop = NULL;
}
```

- Methods to check if stack is full or empty

```
template <class Type>
bool DynamicStack<Type>::isEmpty() const
{
    return (stackTop == NULL);
}
```

```
template <class Type>
bool DynamicStack<Type>::isFull() const
{
    return false;  // always return false:
                   // there is no max size
}
```

- Push() method:



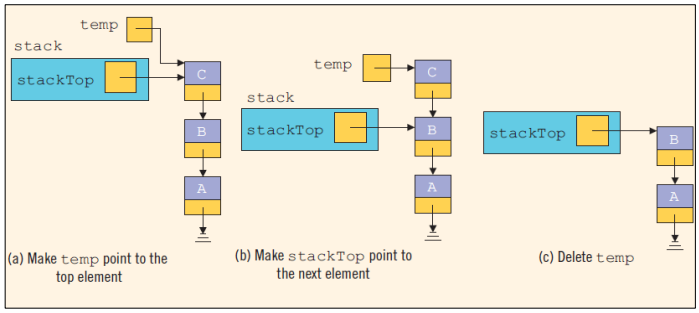
- It is a prepend operation as the node is added to the head of the stack.
- Simplified version of normal linked list adding nodes as nodes are always added to the head.

```
template <class Type>
void DynamicStack<Type>::push(const Type& newElement)
{
    Node * newNode = new Node;  // create a new node
    newNode->data = newElement; // store data in the node
    newNode->next = stackTop;   // put the new node on top
    stackTop = newNode;         // update stackTop
}
```

- Top() method (access top element):

```
template <class Type>
Type DynamicStack<Type>::top() const
{
     if(!isEmpty()) // if not empty
             return stackTop->data; // return the top element
     else throw “Stack is empty”;
}
```

- Pop() method (remove an element from the top of the stack):




```
template <class Type>
void DynamicStack<Type>::pop(Type& var)
{
    if (!isEmpty())
    {
        var = stackTop->data;
        Node * temp = stackTop; // point to the top node
        stackTop = stackTop->next; // update stackTop
        delete temp; // delete the top node
    }
    else throw "Stack is empty";
}
```

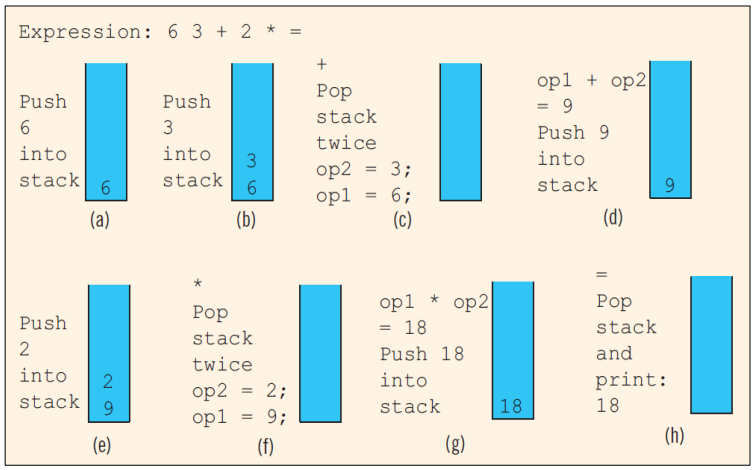
- Deallocation(delete all existing elements and set stackTop to NULL:

```
template <class Type>
DynamicStack<Type>::~DynamicStack()
{
    Node *temp; //pointer to delete the node
    while (stackTop) { // != NULL
        temp = stackTop; // point to the current node
        stackTop = stackTop->next;
        delete temp; //deallocate memory occupied by temp
    }
}
```

- This method is **Very** similar to the pop() method and one can actually change the destructor to say: **while(stack is not empty): pop()**

Reasons for using the stack (postfix calculator):

- **Infix**: (easy for humans to understand)
  - (2 + 3) * 7
- **Prefix**: (easier for computers to implement)
  - * 7 + 2 3
- **Postfix**: (easier for computers to implement)
  - 2 3 + 7 *
- More postfix examples:
  - (a – b) * (c + d) -> a b – c d + *
  - (a + b) / d + c -> a b + d / c +
- Operands (values) precede operators

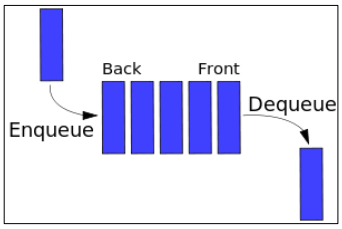


- If you encounter a number:
  - Push it onto the stack
- If you encounter a symbol:
  - If it's an operator (+, –, *, /):
    - Pop two numbers off the stack, evaluate, push the result back onto the stack
  - If it's the equals sign (=):
    - Pop the stack – the result will be stored there, and it will be the only element if the expression was valid
  - Otherwise, report an error: invalid symbol/expression!

Summary:

- **Stack**: Last-In-First-Out (LIFO) data structure
- **Static stack**: implemented using an array
- **Dynamic stack**: implemented using a linked list
- Example applications
  - Used excessively in computer architecture
  - Useful for parsing (code parsing, notation parsing, etc)

19.4 Introduction to the Queue ADT



- **Queue**: a **FIFO** (first in, first out) data structure.
- Like a stack, queue is a restricted linear structure:
  - Add elements to the rear end of the queue
  - Remove elements from the front end of the queue
- Examples:
  - people in line at the cashier
  - print jobs sent to a printer
  - computer instructions sent to a CPU

Locations and Operations

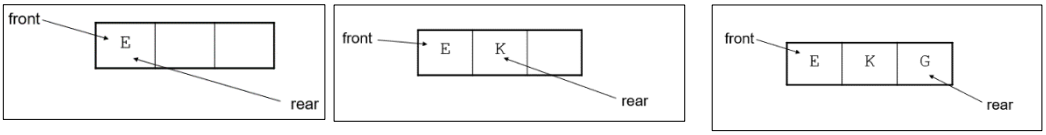
- **rear**: position where elements are added
- **front**: position from which elements are removed

- **enqueue**: add an element to the rear of the queue
- **dequeue**: remove an element from the front of a queue
- No access to the elements in the middle!

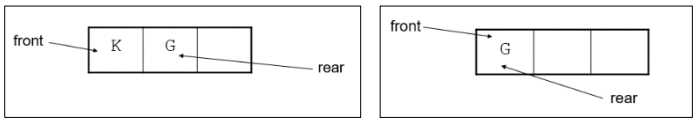
- Implementation:
  - **static**: fixed size, implemented as **array**
  - **dynamic**: variable size, implemented as **linked list**

Queue Operations – Example

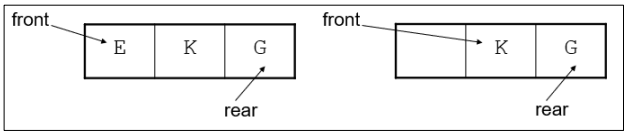
- A currently empty queue that can hold **char** values:
- `enqueue('E');`
- `enqueue('K');`
- `enqueue('G');`



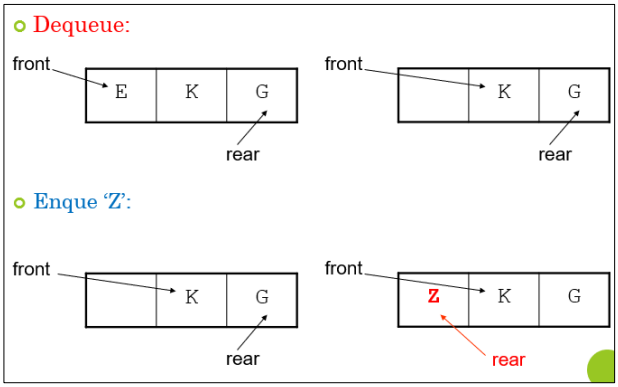
- Not an efficient way: shifting elements to the front
- `dequeue(); // remove E`
- `dequeue(); // remove K`



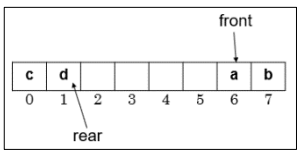
- When removing an element from a queue, remaining elements must shift to front
  - This is not very efficient!
- **Solution**:
  - Let the front index move as elements are removed
  - This solution creates a problem as now the gap is in the front so trying to add an element to the end would cause a problem as the end is technically full.



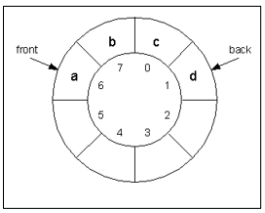
- The use above solution, and let rear index **"wrap around"** to front of array, treating array as circular instead of linear:



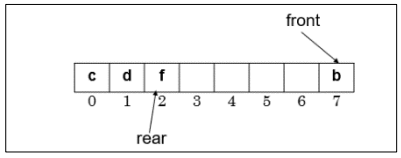
- Essentially:



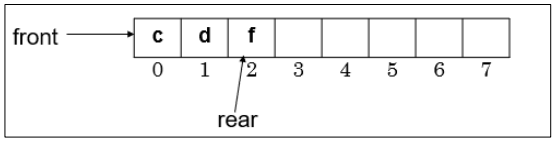
- Is Identical to:



```
enqueue('f'); // where will f go?
// At the rear end: index 2
// rear will be incremented
dequeue(); // what element will be removed?
// Element at the front end: 'a'
// front will be incremented
```



- What will happen to front if we remove 'b'?
  - It will go back to the beginning of the array
  - I.e., it will wrap around the array, imitating a circular structure.
- Same logic applies to rear:
  - when rear is the last element in the array, the rear counter wraps around.
- So: when you reach the end of the array with either front or rear counters, set counter to 0




```
if(counter == size - 1)
    counter = 0;
else
    counter++;
```

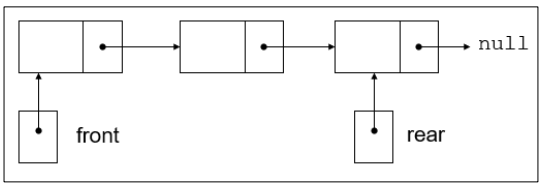
Queue ADT: Arrays

```
template <class Type>
class Queue {
   T * queueArray; // the array to store queue elements
   int queueSize; // max size
   int numItems; // number of items
   int front;
   int rear;
   int increment(int); // handle the wrapping of indices
   public:
   Queue(int);
   ~Queue();
   void enqueue(T);
   void dequeue(T &);
   bool isEmpty();
   bool isFull();
   void clear(); // remove all elements from the queue
   };
```

See *queue.cpp* for implementation

19.5 Dynamic Queues

- Like a **stack**, a queue can be implemented using a **linked list**
- Advantages of a linked list implementation:
  - Dynamic size
  - No need to either shift elements or wrap indices
- Two pointers: **front** and **rear** (think head and tail)




```
template <class T>
class Dynque
{
        private:
                struct QueueNode
                {
                        T value;
                        QueueNode *next;
                };

                QueueNode *front;
                QueueNode *rear;
                int numItems;
        public:
         Dynque();
         ~Dynque();
         void enqueue(T);
         void dequeue(T &);
         bool isEmpty();
         bool isFull();
         void clear();
};
```

Enqueue

```
template <class T>
void Dynque<T>::enqueue(T num)
{
        QueueNode * newNode = 0;

        newNode = new QueueNode; // create new node
        newNode->value = num;
        newNode->next = NULL;

        if (isEmpty())
                front = rear = newNode; // only 1 element in queue

        else {
                rear->next = newNode;
                rear = newNode; // no need to update the front
        }

        numItems++; // keep track on # items
}
```

Dequeue

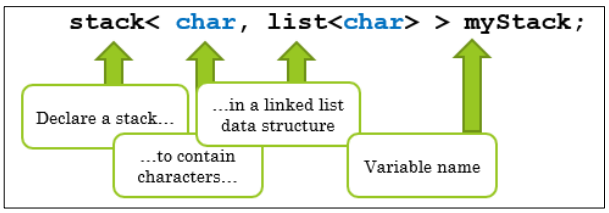
```
template <class T>
void Dynque<T>::dequeue(T &num)
{
        QueueNode *temp = NULL;

        if (isEmpty())
                cout << "The queue is empty.\n";
        else
        {
                num = front->value; // store front value
                temp = front->next; // store next item's address
                delete front;       // delete current front
                front = temp;       // set front to the next el.
                numItems--;         // keep track of # items
        }
}
// no need to update the rear!
```

See D*ynamicQueue.cpp* for implementation

- **Note to self: 18.5 was only presented halfway through 19.5… here**

19.3 The STL stack Container

- C++ Standard Template Library (STL) provides a stack container
- Stack template can be implemented as a **vector**, a linked list, or a **deque** (double-ended queue)
- Container type must be chosen when the variable is declared:



- Defining a stack of chars, named cstack, implemented using a vector:
  - **`stack< char, vector<char> > cstack;`**
- implemented using a list:
  - **`stack< char, list<char> > cstack;`**
- implemented using a deque (default):
  - **`stack< char > cstack;`**
- When using a compiler that is older than C++ 11, be sure to put spaces between the angled brackets that appear next to each other.
  - **`stack< char, vector<char> > cstack;`**

STL stack implements the following functions:

- **push:**
  - **`stack<int> mystack;`**
  - **mystack.push(5); // add 5 to the stack**

- **top**: reference to element on top of the stack
  - `int x = mystack.top(); // retrieve top element`
- **pop:**
  - `mystack.pop(); // remove top element`
- **empty:**
  - `if(!mystack.empty()) // return true if empty`
- **size**: number of elements on the stack
  - `cout << "Number of elements on stack: ";`
  - `cout << mystack.size() << endl;`

19.6 The STL deque and queue Containers

- **deque**: a double-ended queue. Has member functions to:
  - enqueue (**push_back**)
  - dequeue (**pop_front**)
- **queue**: container ADT that can be used to provide queue as a **vector**, **list**, or **deque**. Has member functions to:
  - enque (**push**)
  - dequeue (**pop**)
- **queue** constructors:
  - `queue<int> first;`
  - `queue<int, vector<int> > second; // Can be any STL structure: vector, list, deque`
  - Default container: **deque**

- ***See dequeue.cpp***

Chapter 20 – Recursion

20.1 Introduction to Recursion

- The programs you have written so far were generally structured as functions that call one another
- For some types of problems, it's useful to have functions call themselves
- A recursive function is a function that calls itself either directly or indirectly through another function

```
void recursiveFunction()
{
  recursiveFunction(); // recursive call
}
```

```
void countDown(int num)
{
   if (num == 0)
       cout << "Blastoff!";
   else
    {
       cout << num << "...\n";
       countDown(num - 1); // recursive call
   }
}
```

- A stopping condition is necessary to avoid infinite recursion
- **Base case**: `num == 0 // Blastoff!`
- Breaking the problem into **sub-problems**:

```
countDown(num - 1);   // decrement num by 1:
                      // problem becomes smaller
```

If a program contains a line like `countDown(2);`

- `countDown(2)` generates the output 2..., then it calls `countDown(1)`
- `countDown(1)` generates the output 1..., then it calls `countDown(0)`
- `countDown(0)` generates the output Blastoff! then returns to `countDown(1)`
- `countDown(1)` returns to `countDown(2)`
- `countDown(2)` returns to the calling function

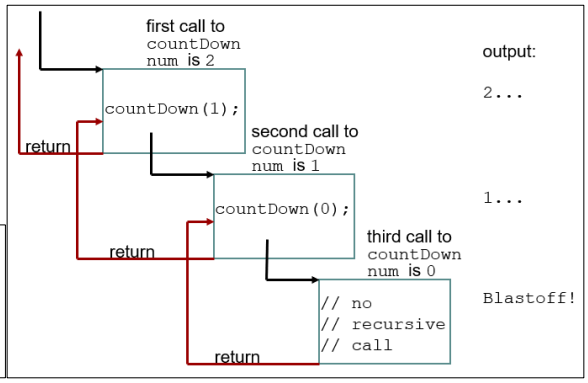
- Recursive functions are used to reduce a complex problem into a simpler-to-solve problem.
- The simpler-to-solve problem is known as the base case
- Recursive calls stop when the base case is reached
- **Divide-and-conquer:**
  - Define a problem as a combination of sub-problems of the same type
  - Solve each sub-problem by dividing it into sub-sub-problems
  - Solve each sub-sub-problem by dividing it into sub-sub-sub-problems
  - …
  - Until you reach the simplest sub-n-problem (base case) that is easily solvable

Stopping the Recursion

- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call
- In the sample program, the test is: `if (num == 0)`
- Recursion uses a process of breaking a problem down into smaller problems until the problem can be solved
- In the countDown function, a different value is passed to the function each time it is called
- Eventually, the parameter reaches the value in the test, and the recursion stops

What Happens during the Recursion Countdown?

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function




```
int main()
{
  countdown(2); // how many instances of countdown
                // will be created?
  cout << "Rocket successfully launched!" << endl;
}
```

Types of Recursion:

- **Direct:**
  - a function calls itself
- **Indirect:**
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls …, which calls function A

The Recursion Factorial Function:

- The factorial of a non-negative integer *n*, written *n*! (pronounced "n factorial"), is the product of all integers between *n* and 1:
  - n * (n – 1) * (n – 2) * … * 1
  - with 1! equal to 1, and 0! defined to be 1
- 5! is the product 5 * 4 * 3 * 2 * 1 = 120
- The factorial of an integer number ≥ 0 can be calculated iteratively (non-recursively) using a for loop.

```
int factorial = 1;
for(int counter = number; counter >= 1; counter--)
  {
    factorial *= counter;
  }
```

- **We need to Identify**:
  - **Base case** (stopping condition)
  - **Recursive case** (how do we simplify this problem into a sub-problem)?
- We know that for 0 <= n <= 1, n! = 1
- Therefore, if n == 1, we have arrived at the base case, no further simplification is possible
- Decrement n at every recursive step to get from given **n** to 1
- How do we combine the sub-problem solutions?

- A recursive definition of the factorial function is arrived at by observing the following relationship:

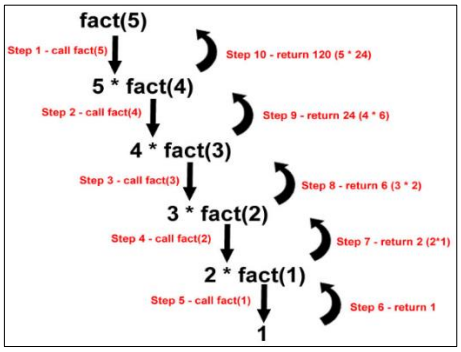
$n! = n * (n - 1)!$

E.g. $5! = 5 * 4!$

- $5! = 5 * 4 * 3 * 2 * 1$
- $5! = 5 * (4 * 3 * 2 * 1)$
- $5! = 5 * (4!)$

- To combine partial solution with sub problem solution:
  - Multiply n by factorial of n
- Now we have all the components to put a recursive function together!

```
int fact(int num) // factorial
{
  if (num > 1) // recursive case
       return num * factorial(num - 1); // sub-problem
  else // base case
       return 1;
}
```



- **Alternative to iterating:**
  - Functions invoking themselves until a stopping condition is met
- **At every recursive step:**
  - Compare the current problem state to the base case
  - If base case not reached, break the problem down into subproblems, and re-invoke the recursive function
  - Every step of recursion should take you one step closer to the base case

Calculating the Sum of first N Natural Numbers recursively

- Given: number n
- Task: Find the Sum of the first N natural numbers
    - `E.g.1+2+3+4+5+6 =21//sum of first six natural numbers`
    - `sum(n) =1+2+3+4+5+6+…+n`
    - `sum(n)=1+2+3+4+5+6+…+(n-1)+n`
    - `sum(n)=sum(n-1)+n //recursive definition of sum(n)`
- Base case:
    - `n==0`
- What is the recursive case?
    - `sum(n)=sum(n-1)+ n`
- Using recursion(left) and using a for loop (right):

```
int sum(int n)
{
    if(n==0)    //Base Case
        return 0;
  else
   return  sum(n-1)+n; //Recursive case
}
```

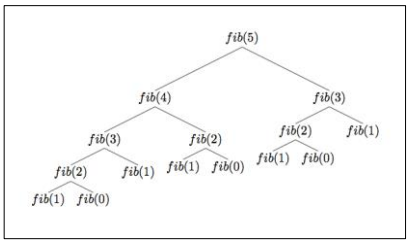
```
int sum(int n)
{
    int total=0;
    for(int i=1; i<=n; i++)
        total=total + i;
    return total;
}
```

20.4 Solving Recursively Defined Problems

Recursions in Nature

- **The Fibonacci series:**
- 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- begins with 0 and 1
- Each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers
- Occurs in nature and, in particular, describes a form of spiral
- The ratio of successive Fibonacci numbers converges to a constant value of 1.618 – the golden ratio
- **Calculating the Fibonacci numbers recursibly**
- Given: number n
- Task: output n^{th} Fibonacci number
    - Every number except 0 and 1 is the sum of two previous numbers
- **What are the base cases?**
    - `n == 0, n == 1`
- **What is the recursive case?**
    - `fib(n) = fib(n – 1) + fib(n – 2);`

```
int fib(int n)
{
   if (n == 0)          // base case 1: n == 0
          return 0;
   else if (n == 1)     // base case 2: n == 1
          return 1;
   else                 // recursive case
          return fib(n – 1) + fib(n – 2);
}
```



- Idk why but fib(0) refers to the first element so fib(1) = 1 and not 0

20.5 Recursive Linked List Operation

- It is often convenient to write recursive functions for data structures
- **Some applications:**
    - Compute the size of (number of nodes in) a linked list
    - Traverse the list in reverse order
    - Clear the list (delete all nodes)
- **Recursive code** is much more **compact** and **elegant** than iterative code
- In general, *we do recursion because it is beautiful*
- Some problems are easily solved with recursion, while an iterative solution is hard to derive

Counting the nodes in a linked list

- Uses a pointer to visit each node
- **Algorithm**:
  - pointer starts at head of list
  - If pointer is a null pointer, return 0 (base case)
  - else, return 1 + number of nodes in the list pointed to by current node (recursive case)

```
int NumberList::countNodes(ListNode *nodePtr) const
{
    if (nodePtr != nullptr)
        return 1 + countNodes(nodePtr->next);
    else
        return 0;
}
```

- eg: 1+1+1+1+1+0 = 5 nodes

- Invoke private countNodes(head) from public numNodes()

Contents of a List in reverse order!

- **Algorithm**:
  - pointer starts at head of list
  - If the pointer is null pointer, return (base case)
  - If the pointer is not null pointer, advance to next node (recursive case)

```
void NumberList::showReverse(ListNode *nodePtr) const
{
    if (nodePtr != nullptr)
    {
        showReverse(nodePtr->next);
        cout << nodePtr->value << " ";
    }
}
```

- Upon returning from recursive call, display contents of current node
- What will happen if you change the order?
  - Won't work

20.6 Binary Search

- Requires array elements to be in order

  1. Divides the array into three sections:
     - middle element
     - elements on one side of the middle element
     - elements on the other side of the middle element
  2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
  3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine

- **Example:**
  - Array numlist2 contains:

2	3	5	11	17	23	29

  - Searching for the value 11, binary search examines 11 and stops
  - Searching for the value 7, binary search examines 11, 3, 5, and stops

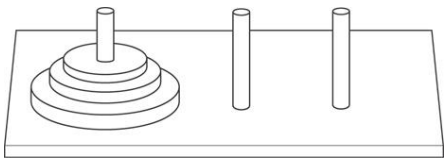
```
int binarySearch(int array[], int size, int value)
{
   int first = 0,              // First array element
       last = size - 1,        // Last array element
       middle,                 // Mid point of search
       position = -1;          // Position of search value
   bool found = false;         // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;     // Calculate mid point
      if (array[middle] == value)      // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;           // If value is in upper half
   }
   return position;
}
```

A Recursive Binary Search Function

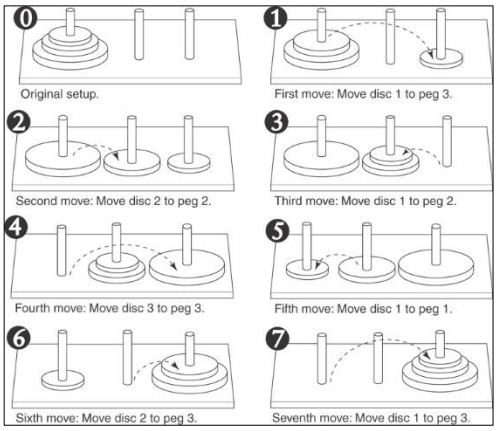
- Binary search algorithm can easily be implemented recursively
- **Base cases:** desired value is found, or no more array elements to search
- **Algorithm** (assuming array in ascending order):
  - If middle element of array segment is desired value, then done
  - Else, if the middle element is too large, repeat binary search on left half of array segment
  - Else, if the middle element is too small, repeat binary search on the right half of array segment

```
 int binarySearch(int array[], int first, int last, int value)
{
   int middle;   // Mid point of search

   if (first > last)
      return -1;
   middle = (first + last) / 2;
   if (array[middle] == value)
      return middle;
   if (array[middle] < value)
      return binarySearch(array, middle+1,last,value);
   else
      return binarySearch(array, first,middle-1,value);
}
```

20.7 The Towers of Hanoi



- The object of the game is to move the discs from the first peg to the third peg. Here are the rules:
  - Only one disc may be moved at a time.
  - A disc larger disk cannot be placed on top of a smaller disc.
  - All discs must be stored on a peg except while being moved.

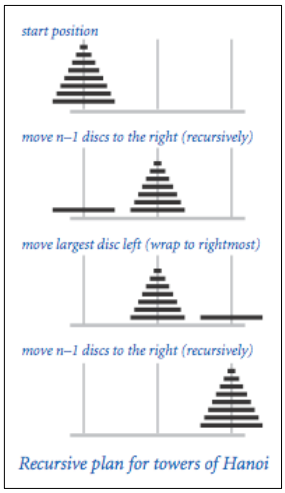


- The following statement describes the overall solution to the problem:
- Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.
- **Algorithm:**

```
To move n discs from peg A to peg C, using peg B as
a temporary peg:
If n > 0 Then
   Move (n – 1) discs from peg A to peg B, using
   peg C as a temporary peg.

   Move the remaining disc from the peg A to peg C.

   Move n – 1 discs from peg B to peg C, using
   peg A as a temporary peg.
End If
```

Program 19-10

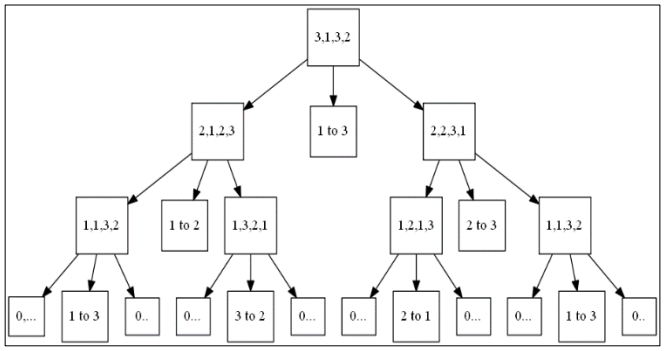
```
1  // This program displays a solution to the Towers of
2  // Hanoi game.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  void moveDiscs(int, int, int, int);
8
9  int main()
10 {
11    const int NUM_DISCS = 3;    // Number of discs to move
12    const int FROM_PEG = 1;     // Initial "from" peg
13    const int TO_PEG = 3;       // Initial "to" peg
14    const int TEMP_PEG = 2;     // Initial "temp" peg
15
16    // Play the game.
17    moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
18    cout << "All the pegs are moved!\n";
19    return 0;
20 }
21
22 //*************************************************
23 // The moveDiscs function displays a disc move in   *
24 // the Towers of Hanoi game.                        *
25 // The parameters are:                              *
26 //    num:     The number of discs to move.         *
27 //    fromPeg: The peg to move from.                *
28 //    toPeg:   The peg to move to.                  *
29 //    tempPeg: The temporary peg.                   *
30 //*************************************************
31
32 void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
33 {
34    if (num > 0)
35    {
36       moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
37       cout << "Move a disc from peg " << fromPeg
38            << " to peg " << toPeg << endl;
39       moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
40    }
41 }
```



Recursive plan for towers of Hanoi

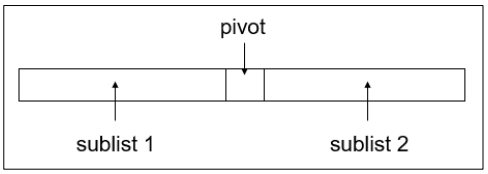
Program Output

```
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3
All the pegs are moved!
```



20.8 The QuickSort Algorithm

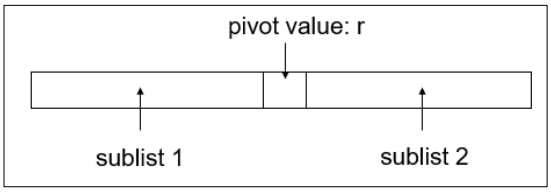
- Recursive algorithm that can efficiently sort an array or a linear linked list
- Determines an element/node to use as pivot value:



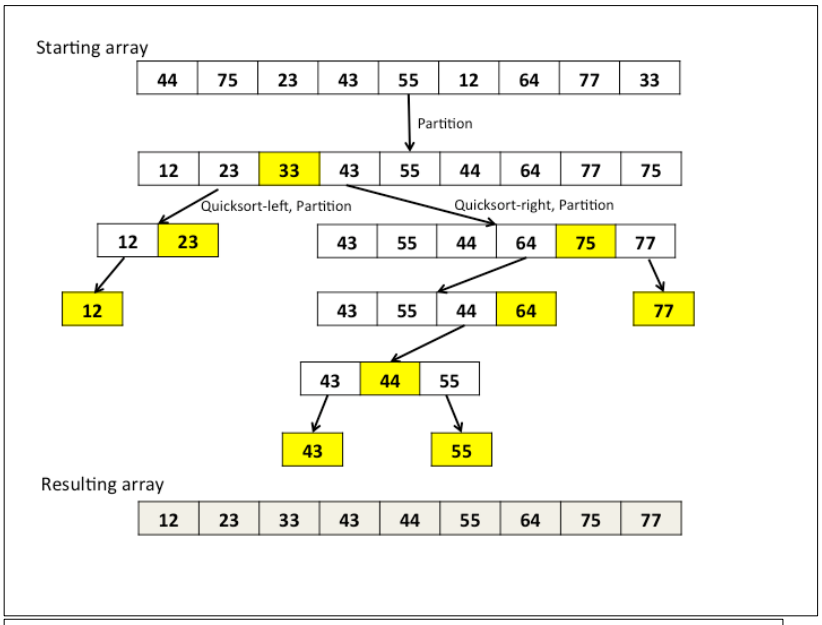
- Once pivot value is determined, values are shifted so that:
  - elements in sublist1 are < pivot
  - elements in sublist2 are > pivot
- Algorithm then sorts sublist1 and sublist2
- Base case: sublist has size 1

Example:

- QuickSort(A,p,q)
  - If p < q
    - Then r <- Partition(A,p,q)
      QuickSort(A,p,r-1)
      QuickSort(A,r+1,q)
- Initial Call: QuickSort(A,0,n)



Example:



- in this case, the last value has been chosen as the keyword value, but it doesn't really matter
- Textbook does this example differently *pg1278*

```
int partition (int arr[], int low, int high)
{
        int pivot = arr[high]; // pivot
        int j=high;
        int i = (low - 1); // Index of smaller element
        while (i< j)
        {
                do{
                        i++
                }while(arr[i]<=pivot)// If current element is smaller than or
                        // equal to pivot

                do{
                        j--
                }while(arr[j]>pivot)// If current element is bigger than pivot

                if(i<j)
                        swap(arr[i],arr[j])
        }
                swap(arr[high], arr[j])
                return j;
}
```

20.9 Exhaustive Algorithms

- Exhaustive algorithm: search a set of combinations to find an optimal one

Coin Example

- Example: change for a certain amount of money that uses the fewest coins
- Uses the generation of all possible combinations when determining the optimal one.
- E.g.Coin change Problem
  - consider all the different ways you can make change for M5.00 using
  - Coins={1,2,3}
- Maximum number of coins to give as change=7
- we are given coins of denomination 1, 2, and 3. The asked sum is 5, then there are five possible solutions to this problem, viz. {1, 1, 1, 1, 1}, {1, 1, 1, 2}, {1, 2, 2}, {1, 1, 3}, {2, 3}
- Then choose best combination
- The key idea to solve this problem is to consider whether **a coin will be present in the solution or not.**
- *See the example on pg1284*

20.10 Recursion vs Iteration

- **Benefits (+), disadvantages(-) for recursion**:
  - Models certain algorithms most accurately (+)
  - Results in shorter, simpler functions (+)
  - May not execute very efficiently (-)
- **Benefits (+), disadvantages(-) for iteration:**
  - Executes more efficiently than recursion (+)
  - Often is harder to code or understand (-)