HIT3303/8303
Data Structures and Patterns
Semester 1, 2011

Convener: Dr Markus Lumpe (EN508c/EN514d)
Lecture: Wednesday 10:30 (EN213)
Labs: Wednesday 08:30 (ATC625), 12:30 (ATC621)
        Thursday 9:30 (EN305)
Grading: Problems sets, mid-term, final exam
Assignments: 6–8
Subject Aims

- How can a given problem be effectively expressed?
- What are suitable data representations for specifying computational processes?
- What is the impact of data and its representation with respect to time and space consumption?
- What are the reoccurring structural artifacts in software and how can we identify them in order to facilitate problem solving?

Selected Objectives

- Solve problems using object-oriented design and implementation techniques.
- Interpret the tradeoffs and issues involved in the design, implementation, and application of various data structures with respect to a given problem.
- Design, implement, and evaluate software solutions using behavioral, creational, and structural software design patterns.
- Explain the purpose and answer questions about data structures and design patterns that illustrate strengths and weaknesses with respect to resource consumption.
Overview

The following gives a tentative list of topics not necessarily in the order in which they will be covered in the subject:

- Introduction
- Sets, Arrays, Indexers, and Iterators
- Basic Data Structures and Patterns
- Abstract Data Types and Data Representation
- One-Dimensional Data Structures
- Hierarchical Data Structures
- Algorithmic Patterns and Problem Solvers

A brief introduction to C++
Why C++

• We need to know more than just Java.
• C++ is highly efficient and provides a better match to implement low-level software layers like device controllers or networking protocols.
• C++ is being widely used to develop commercial applications and is a the center of operating system and modern game development.
• Memory is tangible in C++ and we can, therefore, study the effects of design decisions on memory management more directly.

What is C++

• C++ is a general-purpose, high-level programming language with low-level features.
• Bjarne Stroustrup developed C++ (C with Classes) in 1983 at Bell Labs as an enhancement to the C programming language.
Design Philosophy of C++

- C++ is a hybrid, statically-typed, general-purpose language that is as efficient and portable as C.
- C++ directly supports multiple programming styles like procedural programming, object-oriented programming, or generic programming.
- C++ gives the programmer choice, even if this makes it possible for the programmer to choose incorrectly.
- C++ avoids features that are platform specific or not general purpose, but is itself platform-dependent.
- C++ does not incur overhead for features that are not used.
- C++ functions without an integrated and sophisticated programming environment.

C++ Paradigms

- C++ is a multi-paradigm language.
- C++ provides natural support for
  - the imperative paradigm and
  - the object-oriented paradigm.

- Paradigms must be mixed in any non-trivial project.
Imperative Programming

- This is the oldest style of programming, in which the algorithm for the computation is expressed explicitly in terms of instructions such as assignments, tests, branching and so on.
- Execution of the algorithm requires data values to be held in variables which the program can access and modify.
- Imperative programming corresponds naturally to the earliest, basic and still used model for the architecture of the computer, the von Neumann model.

The von Neumann Architecture

- Memory
- Arithmetic Logic Unit
- Control Unit
  - Accumulator
- Input
- Output
Object-Oriented Programming

- In general, object-oriented languages are based on the concepts of class and inheritance, which may be compared to those of type and variable respectively in a language like Pascal and C.
- A class describes the characteristics common to all its instances, in a form similar to the record of Pascal (structures in C), and thus defines a set of fields.
- In object-oriented programming, instead of applying global procedures or functions to variables, we invoke the methods associated with the instances (i.e., objects), an action called "message passing."
- The basic concept inheritance is used to derive new classes from exiting ones by modifying or extending the inherited class(es).

The Simplest Possible C++ Program

```
.text
.align 1
.globl _main
_main:
    push %ebp
    mov %esp, %ebp
    sub $8, %esp
    mov %0, %eax
    leave
    ret
.globl _maineh
_maineh = 0
.no_dead_strip _maineh
 .construction
    .align 1
    .subsections_via_symbols
```
Let's make the program more responsive!

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

int main() {
    cout << "Enter two numbers: " << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

- C++ does not directly define any I/O primitives.
- I/O operations are provided by standard libraries.

**eclipse for C/C++**
# BookStore

```cpp
#ifndef BOOK_H
#define BOOK_H

#include <iostream>

class Book {
private:
    std::string ISBN;
    unsigned unitsSold;
    double Revenue;

public:
    Book() : unitsSold(0), Revenue(0.0) {}
    Book(const std::string &eBook) : ISBN(eBook), unitsSold(0), Revenue(0.0) {}
    Book(std::istream &isStream) { isStream >> *this; }
    Book &operator<<(const Book &aRHS);
    friend bool operator==(const Book &left, const Book &aRight);
    friend std::ostream &operator<<(std::ostream &osStream, Book &aItem);
    friend std::ostream &operator<<(std::ostream &osStream, const Book &aItem);
    double getAveragePrice() const;
    bool hasNextISBN(const Book &aRHS) const;
};

#endif
```
The Structure of a Class

class X
{
    private:
        // private members
    protected:
        // protected members
    public:
        // public members
};

Never forget the semicolon!

Access Modifiers

- **public:**
  - Public members can be accessed anywhere, including outside of the class itself.

- **protected:**
  - Protected members can be accessed within the class in which they are declared and within derived classes.

- **private:**
  - Private members can be accessed only within the class in which they are declared.
Include File: Book.h

We do not select any namespace yet!

The implementation goes to Book.cpp

Implementation File: Book.cpp

Implementations
C++ Code Organization

• Classes are **defined** in include files (i.e., .h).
• Class members are **implemented** in source files (i.e., .cpp).
• There are exceptions (as usual), when working with templates.

Constructors

• Constructors may be overloaded.
• The concrete constructor arguments determine which constructor to use.
• Constructors are executed automatically whenever a new object is created.
A constructor initializer is a comma-separated list of member initializers, which is declared between the signature of the constructor and its body.
Friends

- Friends are allowed to access private members of classes.
- A class declares its friends explicitly.
- Friends enable uncontrolled access to members.
- The friend mechanism induces a particular programming (C++) style.
- The friend mechanism is not object-oriented!
- I/O depends on the friend mechanism.

The Friend Mechanism

Friends are self-contained procedures (or functions) that do not belong to a specific class, but have access to the members of a class, when this class declares those procedures as friends.
Class Book - The Friends

```cpp
#include "BOOK.h"
#define BOOK_H

class Book {
public:
    static int fSold;
    static int fRevenue;
    int fItemsSold;
    double fRevenue;

    Book() : fItemsSold(0), fRevenue(0.0) {}  // Constructor
    Book(int itemsSold, double revenue) : fItemsSold(itemsSold), fRevenue(revenue) {}  // Parameterized constructor
    Book &operator=(const Book &rhs) {  // Assign operator
        fItemsSold = rhs.fItemsSold;
        fRevenue = rhs.fRevenue;
        return *this;
    }

    friend bool operator==(const Book &left, const Book &right) {
        return fItemsSold == right.fItemsSold &&
               fRevenue == right.fRevenue &&
               hasSameISBN(right);
    }

    // Other member functions...
};

friend FStream &operator<<(FStream &os, const Book &book) {
    return os;
}
```

The Equivalence Operator ==

- The Boolean operator `==` defines a structural equivalence test for Book objects.
- We use `const` references for the `Book` arguments to pass `Book` objects by reference rather than copying their values into the stack frame of the operator `==`. 
The Input Operator `>>`

```cpp
// friend
istream& operator>>( istream& aStream, Book& item )
{
    double lPrice;
    aStream >> item.FISBN >> item.UnitsSold >> lPrice;
    // check that the inputs succeeded
    if ( aStream )
    {
        item.Revenue = item.UnitsSold * lPrice;
    }
    else
    { // reset to default state
        item = Book();
    }
    return aStream;
}
```

Return reference to input stream.

The Output Operator `<<`

```cpp
// friend
ostream& operator<<( ostream& aStream, const Book& item )
{
    aStream << item.FISBN << " \t " << item.UnitsSold << " \t "
    << item.Revenue << " \t " << item.getAveragePrice();
    return aStream;
}
```

Return reference to output stream.
```cpp
#include <iostream>
#include "Book.h"
using namespace std;

int main()
{
    Book bBook;
    cin >> bBook; // read book data
    cout << bBook << endl; // write book data
    return 0;
}
```

**Class Book - Member Operator**

```cpp
#include <iostream>
#include "Book.h"

class Book
{
private:
    static string FISBN;
    unsigned intSales;
    double Revenue;

public:
    Book() : intSales(0), floatSales(0) {} // constructor
    Book(const std::string& book) : FISBN(book), floatSales(0), floatRevenue(0) {} // constructor
    Book(const std::istream& is) { is >> *this; }

    friend bool operator==(const Book& lhs, const Book& rhs);
    friend std::ostream& operator<<(std::ostream& os, const Book& b); // output
    friend std::istream& operator>>(std::istream& is, const Book& b); // input
    double getAveragePrice() const;
    bool hasSameISBN(const Book& rhs) const;
};
```

The Member Operator `+=`

- The operator `+=` is defined as a member of class `Book`!
- Return reference to receiver.

---

Operator Overloading

- C++ supports operator overloading.
- Overloaded operators are like normal functions, but are defined using a pre-defined operator symbol.
- You cannot change the priority and associativity of an operator.
- Operators are selected by the compiler based on the static types of the specified operands.
The Overloaded Operator +

```
// overloaded operator
Book operator+(const Book &left, const Book &right)
{
    Book Result(left);
    Result += right;
    return Result; // return by value
}
```

operator+ : (Book, Book) → Book

---

AddSales

```
#include <iostream>
#include "Book.h"

using namespace std;

// overloaded operator
Book operator+(const Book &left, const Book &right)
{
    Book Result(left);
    Result += right;
    return Result; // return by value
}

int main()
{
    Book lBook1, lBook2;
    cin >> lBook1 >> lBook2;
    // read books
    cout << lBook1 + lBook2 << endl;  // write sales data
    return 0;
}
```

---

Select GT3303 Markus$ ./AddSales
0: 201-78345-X 3 20.00
0: 201-78345-X 2 25.00
0: 201-78345-X 5 110
Class Book – Member Functions

```
#include <iostream>

class Book {
    private:
        string ISBN;
        float Revenue;
        unsigned int unitsSold;
        double AveragePrice;

    public:
        Book() : Revenue(0.0), unitsSold(0) {}
        Book(const string &ISBN, float Revenue) : Revenue(Revenue), unitsSold(0) {}
        Book(const string &ISBN, float Revenue, unsigned int unitsSold) : Revenue(Revenue), unitsSold(unitsSold) {}
        ~Book() {} // Destructor

        friend double operator-(const Book &left, const Book &right); // Overload subtraction
        friend bool operator==(const Book &left, const Book &right); // Overload equality

        double getAveragePrice() const;
        bool hasSameISBN(const Book & rhs) const;

    private:
        friend class Disk;
    }
```

The Member Functions

```
// member function
double Book::getAveragePrice() const
return Revenue / unitsSold;

// member function
bool Book::hasSameISBN(const Book & rhs) const
```

Automatic type conversion

Private member variables are visible within the scope of class Book.
Uniquely C++
Enumerations

- Enumerations provide a mechanism for defining constants and grouping them into sets of integral types.

- An enumeration is defined using the `enum` keyword, followed by an optional enumeration name, and a comma-separated list of enumerators enclosed in braces.

```c
enum CardSuit { Blank, Club, Diamond, Heart, Spade };
```

- Enumerators are `const` values. If not otherwise specified the first enumerator equals 0, whereas the others are implicitly assigned the increment of its predecessor.

Type Definitions

- Type definitions using the `typedef` keyword let us introduce a synonym for a type:

```c
typedef char byte;
```

- Typedefs are commonly used for three purposes:
  - To hide the implementation of a given type.
  - To streamline complex type definitions making them easier to understand, and
  - To allow a single type to be used in different contexts under different names.

- Type definitions establish a nominal equivalence between types.
Const Qualifier

- What are the problems with
  
  ```
  for ( int index = 0; index < 128; index++ ) { ... }
  ```

- We can do better
  
  ```
  for ( int index = 0; index < BufferSize; index++ ) { ... }
  ```

- Defining a const object:
  ```
  const int BufferSize = 128; // initialized at compile time
  ```

- Unlike macro definitions, const objects have an address!

  ```
  #define BUF_SIZE 128 // macro definition
  ```

  ```
  const int BufferSize = BUF_SIZE; // initialized at compile time
  ```

---

C++ Object Models

- C++ supports:
  - A value-based object model
  - A reference-based object model

This can make programming in C++ difficult. Java, for example, has only one model – everything is a reference!
The Value-based Object Model

- Value-based object model:
  - Objects are stored on the stack.
  - Object are accessed through object variables.
  - An object’s memory is implicitly released.

Valued-based Objects

- Value-based objects look and feel like records (or structs):
  ```
  Card TenOfSpade( Spade, 10 );
  Card AceOfDiamond( Diamond, 14 );
  Card TestCard = AceOfDiamond;
  if ( TestCard == AceOfDiamond )
      cout << "The test card is " << TestCard.GetName() << endl;
  ```
The Reference-based Object Model

- Reference-based object model:
  - Objects are stored on the heap.
  - Objects are accessed through pointer variables.
  - An object's memory must be explicitly released.

Reference-based Objects

- Reference-based objects require pointer variables and an explicit new and delete:

```cpp
Card* AceOfDiamond = new Card( Diamond, 14 );
Card* TestCard = new Card( Diamond, 14 );

if ( TestCard == AceOfDiamond )
    cout << "The test card is " << TestCard->GetName() << endl;

delete AceOfDiamond;
delete TestCard;
```
Cardinal Rule

- We find both models in C++ code.
- Value semantics provides us with objects that behave like arithmetic types:

  Each object is unique.

Constant References

- A reference introduces a new name for an object:
  
  ```
  int BlockSize = 512;
  int& BufferSize = BlockSize;  // BufferSize is an alias
  ```

- A constant reference yields a new name for a constant object:
  
  ```
  const int FixedBlockSize = 512;
  const int& FixedBufferSize = FixedBlockSize;
  ```

- A constant reference defines an alias to an object.
Constant Reference Parameters

- C++ uses call-by-value as default parameter passing mechanism.
  ```cpp
global Assign( int aPar, int aVal ) { aPar = aVal; }
Assign( val, 3 );    // val unchanged
```
- A reference parameter yields call-by-reference:
  ```cpp
global AssignR( int& aPar, int aVal ) { aPar = aVal; }
AssignR( val, 3 );   // val is set to 3
```
- A const reference parameter yields call-by-reference, but the value of the parameter is read-only:
  ```cpp
global AssignCR( const int& aPar, int aVal ) { aPar = aVal; } // error
```

Object Initialization

- A class defines a new data type. Instances of this data type are objects that need initialization.
- Each class defines explicitly (or implicitly) some special member functions, called constructors, that are executed whenever we create new objects of a class.
- The job of a constructor is to ensure that the data members of each objects are set to some sensible initial values.
Reference Data Members

- Constructor initializers are optional, but there are cases in which they are required.
- Reference data members require a constructor initializer:

```cpp
class RefMember
{
private:
    OtherClass& fRef;
public:
    RefMember(OtherClass& aRef) : fRef(aRef) { ... }
};
```

---

Default Constructor

- A default constructor is one that does not take any arguments.
- The compiler will synthesize a default constructor, when no other constructors have been specified.
- If some data members have built-in or compound types, then the class should not rely on the synthesized default constructor!
Inheritance

- A mechanism for specialization
- A mechanism for reuse
- Fundamental to supporting polymorphism

Account & BankAccount

class Account
{
  private:
  double balance;
  public:
  Account(double balance);
  virtual ~Account();
  void deposit(double amount);
  virtual void withdraw(double amount);
  double getBalance();
};

class BankAccount: public Account
{
  private:
  double interestRate;
  public:
  BankAccount(double rate);
  ~BankAccount()
  virtual void withdraw(double amount);
  void addInterest();
  virtual void chargeFee(double amount);
Access Levels for Inheritance

- public:
  - Public members in the base class remain public.
  - Protected members in the base class remain protected.
  - Yields a "is a" relationship.
- protected:
  - Public and protected members in the base class are protected in the derived class.
  - Yields a "implemented in terms of" relationship.
- private:
  - Public and protected members in the base class become private in the derived class.
  - Yields a stricter "implemented in terms of" relationship.

Constructors and Inheritance

- Whenever an object of a derived class is instantiated, multiple constructors are called so that each class in the inheritance chain can initialize itself.
- The constructor for each class in the inheritance chain is called beginning with the base class at the top of the inheritance chain and ending with the most recent derived class.
**Facts About Base Class Initializers**

- If a base class does not have a default constructor, the derived class must provide a base class initializer for it.

- Base class initializers frequently appear alongside member initializers, which use similar syntax.

- If more than one argument is required by a base class constructor, the arguments are separated by comma.

- Reference members need to be initialized using a member initializer.
Destructors and Inheritance

- Whenever an object of a derived class is destroyed, the destructor for each class in the inheritance chain, if defined, is called.

- The destructor for each class in the inheritance chain is called beginning with the most recent derived class and ending with the base class at the top of the inheritance chain.

Account & BankAccount Destructors

```cpp
class Account
{
private:
    double fBalance;

public:
    Account(double dBalance);
    virtual ~Account() {};
    void deposit(double dAmount);
    virtual void withdraw(double dAmount);
    double getBalance();
};

class BankAccount: public Account
{
private:
    double fInterestRate;

public:
    BankAccount(double dRate);
    ~BankAccount() {};
    void addInterest();
    void chargeFee(double dAmount);
};
```
Virtual Destructors

- When deleting an object using a base class pointer of reference, it is essential that the destructors for each of the classes in the inheritance chain get a chance to run:

  ```cpp
  BankAccount *BAptr;
  Account *Aptr;
  BAptr = new BankAccount(2.25);
  Aptr = BAptr;
  ...
  delete Aptr;
  ```

Virtual Account Destructor

```cpp
class Account
{
public:
    virtual ~Account() { ... }
};
```

Not declaring a destructor virtual is a common source of memory leaks in C++ code.
Virtual Member Functions

- To give a member function from a base class new behavior in a derived class, one overrides it.
- To allow a member function in a base class to be overridden, one must declare the member function virtual.
- Note, in Java all member functions are virtual.

Virtual withdraw Method

```java
class Account
{
    private:
        double balance;
    public:
        Account(double balance);
        virtual Account() {};
        void deposit(double amount);
        double getBalance();
        virtual void withdraw(double amount);
}
```

```java
class BankAccount: public Account
{
    private:
        double interestRate;
    public:
        BankAccount(double rate);
        virtual void withdraw(double amount);
        double getInterest();
        void chargeFee(double amount);
}
```
Overriding the withdraw Method

Call inherited method (this-> optional)

Call overridden method

#include "Accounts.h"

int main()
{
    BankAccount lBankAccount( 2.25 );
    Account* Aptr = &lBankAccount;
    Aptr->withdraw( 50.0 );
    return 0;
}

Calling a Virtual Method

Calls BankAccount::withdraw
Facts About Virtual Members

- Constructors cannot be virtual.
- Declaring a member function virtual does not require that this function must be overridden in derived classes, except the member function is declared pure virtual.
- Once a member function has been declared virtual, it remains virtual.
- Parameter and result types must match to properly override a virtual member function.
- If one declares a non-virtual member function virtual in a derived class, the new member function hides the inherited member function.
- You can declare a private member function virtual to enable polymorphism within the scope of the declaring class.

Data Structures – Basic Concepts

Overview
- Programming Paradigms
- Values, Sets, and Arrays
- Indexer, Iterators, and Pattern Structures

References
Programming Paradigms

- Imperative style:
  \[ \text{program} = \text{algorithms} + \text{data} \]
- Functional style:
  \[ \text{program} = \text{function} \times \text{function} \]
- Logic programming style:
  \[ \text{program} = \text{facts} + \text{rules} \]
- Object-oriented style:
  \[ \text{program} = \text{objects} + \text{messages} \]
- Other styles and paradigms:
  - blackboard, events, pipes and filters, constraints, lists, ...

Object-Oriented Software Development

- Object-oriented programming is about
  - Object-oriented software development
  - Using an object-oriented programming language
- Object-oriented software development is
  - An evolutionary step refining earlier techniques
  - A revolutionary idea perfecting earlier methods
**Object-Oriented Design**

Concrete → Abstract

- Entities
- Types

Domain Analysis

- Objects
- Classes

Prototypes

**APPLICATION DOMAIN**

**SOLUTION DOMAIN**

---

**Concrete vs. Abstract**

Universe of Shapes

- Rectangular
  - Draw()
- Triangle
  - Draw()
- Circle
  - Draw()
- Polygon
  - Draw()
Why is object-oriented software development popular?

- The object-oriented development approach
  - Naturally captures real life
  - Scales well from trivial to complex tasks
  - Focuses on responsibilities, reuse, and composition

Values

- In computer science we classify as a value everything that may be evaluated, stored, incorporated in a data structure, passed as an argument to a procedure or function, returned as a function result, and so on.
- In computer science, as in mathematics, an “expression” is used (solely) to denote a value.
- Which kinds of values are supported by a specific programming environment depends heavily on the underlying paradigm and its application domain.
- Most programming environments provide support for some basic sets of values like truth values, integers, real number, records, lists, etc.
Constants

- Constants are named abstractions of values.
- Constants are used to assign an user-defined meaning to a value.
- Examples:
  - EOF = -1
  - TRUE = 1
  - FALSE = 0
  - PI = 3.1415927
  - MESSAGE = "Welcome to HIT3303/8303"
- Constants do not have an address, i.e., they do not have a location.
- At compile time, applications of constants are substituted by their corresponding definition.

Primitive Values

- Primitive values are values whose representation cannot be further decomposed. We find that some of these values are implementation and platform dependent.
- Examples:
  - Truth values,
  - Integers,
  - Characters,
  - Strings,
  - Enumerands,
  - Real numbers.
  - -1
  - "Hello World!"
  - 3.14159
  - false
  - Red
Composite Values

- Composite values are built up using primitive values and composite values. The layout of composite values is in general implementation dependent.

- Examples:
  - Records
  - Arrays
  - Enumerations
  - Sets
  - Lists
  - Tuples
  - Files

Pointers

- Pointers are references to values, i.e., they denote locations of a values.

- Pointers are used to store the address of a value (variable or function) – pointer to a value, and pointers are also used to store the address of another pointer – pointer to pointer.

- In general, it not necessary to define pointers with a greater reference level than pointer to pointer.

- In modern programming environments, we find pointers to variables, pointers to pointer, function pointers, and object pointers, but not all programming languages provide means to use pointers directly (e.g. Java).
Sets

• A set is a collection of elements (or values), possibly empty.
• All elements satisfy a possibly complex characterizing property. Formally, we write:

\[ \{ x \mid P(x) = \text{True} \} \]

to define a set, where all elements satisfy the property \( P \).
• The basic axiom of set theory is that there exists an empty set, \( \emptyset \), with no elements. Formally,

\[ \forall x, \ x \notin \emptyset \]

In words, "for every \( x \), \( x \) is not an element of \( \emptyset \)."

Inductive Specification

• Sometimes it is difficult to define a set explicitly, in particular if the elements of the set have a complex structure.
• However, it may be easy to define the set in terms of itself. This process is called inductive specification or recursion.
• Example:

Let the set \( S \) be the smallest set of natural numbers satisfying the following two properties:
- \( 0 \in S \), and
- Whenever \( x \in S \), then \( x + 3 \in S \).

The first property is called base clause and the second property is called inductive/recursive clause. An inductive specification may have multiple base and inductive clauses.
The "Smallest Set"

- If we use inductive specification, we always define the smallest set that satisfies all given properties. That is, inductive specification is free of redundancy.
- It is easy to see that there can be only one such set:

  If $S_1$ and $S_2$ both satisfy all given properties, and both are the smallest, then we have $S_1 \subseteq S_2$ (since $S_1$ is the smallest), and $S_2 \subseteq S_1$ (since $S_2$ is the smallest), hence $S_1 = S_2$.

The Set of Strings

$$S = \epsilon | aS,$$

where

- $\epsilon$ is the empty string and
- $a \in \Sigma$, with $\Sigma$ being the alphabet over $S$. 
Regular Sets of Strings

- Operations for building sets of strings:
  - Alternation
    \[ S_1 \mid S_2 = \{ s \mid s \in S_1 \lor s \in S_2 \} \]
  - Concatenation
    \[ S_1 \cdot S_2 = \{ s_1 s_2 \mid s_1 \in S_1, s_2 \in S_2 \} \]
  - Iteration
    \[ S^* = \{ \varepsilon \} \cup S \cdot S \cdot S \cdot S \cdot \ldots \]
    \[ = S^0 \mid S^1 \mid S^2 \mid S^3 \mid \ldots \]

- A set of strings over \( \Sigma \) is said to be regular if it can be built from the empty set \( \varepsilon \) and the singleton set \( \{ a \} \) (for each \( a \in \Sigma \)), using just the operations of alternation, concatenation, and iteration.

Indexed Sets

- Sets are unordered collections of data elements.

- In order to obtain an ordering relation over the elements of a given set, we can assign each element in that set a unique element of another ordered set \( I \):
  \[ S_I = \{ a_i \mid a \in S, i \in I \} \]

\( S_I \) is called the “indexed set” of \( S \).
Some Indexed Sets

- Let $A = \{ a, b, c, d \}$ and $I = \mathcal{N}$, then
  \[ A_I = \{ a_1, b_2, c_3, d_4 \} \]

- Let $A = \{ a, b, c, d \}$ and $I = (S \times S, <)$, then
  \[ A_I = \{ a^{a_1}, b^{a_2}, c^{a_3}, d^{a_4} \} \]

Arrays

- An array is a compound data type that consists of
  - a type specifier,
  - an identifier, and
  - a dimension.
- Arrays define an ordered, homogeneous sequence of data of a specific length.
- Arrays define a number-based position association among the elements.
- The compiler can reserve the required amount of space when the program is compiled.
C++ Array Examples

const unsigned int buf_size = 512, max_files = 20;
int staff_size = 27;
const unsigned int sz = get_size();          // const value not known until runtime

char input_buffer[buf_size];
string fileTable[max_files + 1];
int chess_board[8][8];                      // two-dimensional array
double salaries[staff_size];                // error: non const variable
int test_scores[get_size()];                // error: non const expression
int vals[sz];                               // error: sz not known until runtime

 Explicitly Initialized Arrays

- int arr1[20] = { 1 };  
  Array of 20 integers with arr1[0] == 1 and arr1[i] == 0 for 1 ≤ i < 20.

- int arr2[] = { 1, 2, 3, 4, 5 };  
  Array of 5 integers initialized 1, 2, 3, 4, and 5.

- int arr3[10];  
  Array of 10 integers, none initialized.

- Data members of array type are
  - Not initialized if the base type is a built-in data type
  - Initialized (using the default constructor) if the base type is a class type.
Multi-Dimensional Array Initialization

- `int board[3][3] = { {1, 2, 3} };`
  - only the first row is initialized, the remaining elements are set to integer 0
- `char tic-tac-toe[][3] = { {‘_’, ‘_’, ‘_’},
                         {‘_’, ‘_’, ‘_’},
                         {‘_’, ‘_’, ‘_’} };`
  - fully-initialized array of 3x3 characters (‘_’)

C-Strings

- A C-String is an array of characters:
  ```c
  char a_string[] = “Hello World!”;
  ```
- The length of a C-String is the number of characters of the C-String plus one:
  ```c
                      ‘\0’ };
  ```
A C-String Array

#define MAX_ID_LENGTH 5

char keywords[][MAX_ID_LENGTH] =
{
    "if",
    "then",
    "else"
};

The first dimension of an array can be unspecified!

Sum

- The sum of all elements of an array: $\sum_{i=0}^{n} a_i$

```c
int Sum( int a[], unsigned int n )
{
    int Result = 0;
    for ( unsigned int i = 0; i < n; i++ )
    {
        Result += a[i];
    }
    return Result;
}
```

$O(Sum) = n$
Testing Sum

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

int Sum(int a[], unsigned int n)
{
    int Result = 0;
    for (unsigned int i = 0; i < n; i++)
    {
        Result += a[i];
    }
    return Result;
}

int main()
{
    int a[] = {1, 2, 3, 4, 5};
    int val = Sum(a, 5);
    cout << "Sum of [1,2,3,4,5] is " << val << endl;
    return 0;
}
```

Pairs and Maps

- Let A and B be sets. The Cartesian product of A and B, denoted by $A \times B$, is the set of all ordered pairs $(a, b)$ where $a \in A$ and $b \in B$:
  \[ A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \} \]
- A map is an associative container, whose elements are key-value pairs. The key serves as an index into the map, and the value represents the data being stored and retrieved.
**Associative Array**

- An associate array is a map in which elements are indexed by a key rather than by their position.

  \[ a[i] = \begin{cases} 
  v, & \text{if } i \mapsto v \text{ in } a \\
  \bot, & \text{otherwise}
  \end{cases} \]

- Example:

  \[ a = \{ ("u" \mapsto 345), ("v" \mapsto 2), ("w" \mapsto 39), ("x" \mapsto 5) \} \]
  
  \[ a["w"] = 39 \]
  
  \[ a["z"] = \bot \]

---

**From Indices to Keys**

- We can define an adapter class that defines an indexer:

```cpp
#include <string>

class IntArrayIndexer
{
private:
    const int* fArrayElements;
    const int length;

public:
    IntArrayIndexer( const int aArray[], const int aLength );
    
    const int& operator[]( const std::string& aKey ) const;
};
```
Indexer Constructor

Arrays are passed as pointers to the first element to functions in C++.

The Indexer

- We use the const specifier to indicate that the operator[]:
  - is a read-only getter
  - does not alter the elements of the underlying collection
- We use a const reference to avoid copying the original value stored in the underlying collection.
Testing the Indexer

```
int main()
{
    int a[] = { 1, 2, 3, 4, 5 };
    IntArrayIndexer indexer(a, 5);
    string keys[] = { "0", "1", "2", "3", "4" };
    int Sum = 0;
    for ( int i = 0; i < 5; i++ )
    {
        Sum += indexer[keys[i]];
    }
    cout << "Indexed sum of [1,2,3,4,5] is " << Sum << endl;
    return 0;
}
```

How can we define an indexer in Java?
The Transition to Java

- We need to define an Indexer class.
- Java does not support operator overloading. So, we need to map [] to a member function.
- The built-in type Integer provides the required conversion operations.
- We use IndexOutOfBoundsException to signal an index error.

Indexer's at(String aKey) Method

```java
public class Indexer {
    private int[] fArrayElements;
    public Indexer(int[] aArray) {
        fArrayElements = aArray;
    }

    // Indexer behavior
    public int at(String aKey) {
        int index = (new Integer(aKey)).intValue();
        if (index < fArrayElements.length)
            return fArrayElements[index];
        else
            throw new IndexOutOfBoundsException("Index out of bounds!");
    }

    public static void main(String[] args) {
    }
}
```
The Indexer’s main Method

```java
public class Indexer {

    public static void main(String[] args) {
        int[] a = {1, 2, 3, 4, 5};
        Indexer indexer = new Indexer(a);
        String[] keys = {"0", "1", "2", "3", "4"};
        int Sum = 0;
        for (int i = 0; i < keys.length; ++i) {
            Sum += indexer.get(keys[i]);
        }
        System.out.println("Indexed sum of [1,2,3,4,5] is "+ Sum);
    }
}
```

Iterators

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator
Abilities of Iterators

<table>
<thead>
<tr>
<th>Iterator Category</th>
<th>Ability</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Iterator</td>
<td>Read forward</td>
<td>istream</td>
</tr>
<tr>
<td>Output Iterator</td>
<td>Write forward</td>
<td>ostream, inserter</td>
</tr>
<tr>
<td>Forward Iterator</td>
<td>Read and write forward</td>
<td></td>
</tr>
<tr>
<td>Bidirectional Iterator</td>
<td>Read and write forward and backward</td>
<td>list, set, multiset, map, multimap, vector, deque, string, array</td>
</tr>
<tr>
<td>Random Access Iterator</td>
<td>Read and write with random access</td>
<td></td>
</tr>
</tbody>
</table>

Input Iterator

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>*iter</td>
<td>Provides read access to the actual element</td>
</tr>
<tr>
<td>iter-&gt;member</td>
<td>Provides read access to a member of the actual element</td>
</tr>
<tr>
<td>++iter</td>
<td>Steps forward (returns new position)</td>
</tr>
<tr>
<td>iter++</td>
<td>Steps forward (returns old position)</td>
</tr>
<tr>
<td>iter1 == iter2</td>
<td>Returns whether iter1 and iter2 are equal</td>
</tr>
<tr>
<td>iter1 != iter2</td>
<td>Returns whether iter1 and iter2 are not equal</td>
</tr>
</tbody>
</table>
## Output Iterator

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>*iter = value</td>
<td>Provides write access to the actual element</td>
</tr>
<tr>
<td>++iter</td>
<td>Steps forward (returns new position)</td>
</tr>
<tr>
<td>iter++</td>
<td>Steps forward (returns old position)</td>
</tr>
</tbody>
</table>

An output iterator is like a “black hole.”

## Forward Iterator

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>*iter</td>
<td>Provides read access to the actual element</td>
</tr>
<tr>
<td>iter-&gt;member</td>
<td>Provides read access to a member of the actual element</td>
</tr>
<tr>
<td>++iter</td>
<td>Steps forward (returns new position)</td>
</tr>
<tr>
<td>iter++</td>
<td>Steps forward (returns old position)</td>
</tr>
<tr>
<td>iter1 == iter2</td>
<td>Returns whether iter1 and iter2 are equal</td>
</tr>
<tr>
<td>iter1 != iter2</td>
<td>Returns whether iter1 and iter2 are not equal</td>
</tr>
<tr>
<td>iter1 = iter2</td>
<td>Assigns an iterator</td>
</tr>
</tbody>
</table>
Bidirectional Iterator

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>*iter</td>
<td>Provides read access to the actual element</td>
</tr>
<tr>
<td>iter-&gt;member</td>
<td>Provides read access to a member of the actual element</td>
</tr>
<tr>
<td>++iter</td>
<td>Steps forward (returns new position)</td>
</tr>
<tr>
<td>iter++</td>
<td>Steps forward (returns old position)</td>
</tr>
<tr>
<td>--iter</td>
<td>Steps backward (returns new position)</td>
</tr>
<tr>
<td>iter--</td>
<td>Steps backward (returns old position)</td>
</tr>
<tr>
<td>iter1 == iter2</td>
<td>Returns whether iter1 and iter2 are equal</td>
</tr>
<tr>
<td>iter1 != iter2</td>
<td>Returns whether iter1 and iter2 are not equal</td>
</tr>
<tr>
<td>iter1 = iter2</td>
<td>Assigns an iterator</td>
</tr>
</tbody>
</table>

Random Access Iterator

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>iter[n]</td>
<td>Provides read access to the element at index n</td>
</tr>
<tr>
<td>iter += n</td>
<td>Steps n elements forward or backward</td>
</tr>
<tr>
<td>iter -= n</td>
<td>Steps n elements forward or backward</td>
</tr>
<tr>
<td>n+iter</td>
<td>Returns the iterator of the nth next element</td>
</tr>
<tr>
<td>n-iter</td>
<td>Returns the iterator of the nth previous element</td>
</tr>
<tr>
<td>iter - iter2</td>
<td>Returns disjoint distance between iter1 and iter2</td>
</tr>
<tr>
<td>iter1 &lt; iter2</td>
<td>Returns whether iter1 is before iter2</td>
</tr>
<tr>
<td>iter1 &gt; iter2</td>
<td>Returns whether iter1 is after iter2</td>
</tr>
<tr>
<td>iter1 &lt;= iter2</td>
<td>Returns whether iter1 is not after iter2</td>
</tr>
<tr>
<td>iter1 &gt; iter2</td>
<td>Returns whether iter1 is not before iter2</td>
</tr>
</tbody>
</table>
A Read-Only Forward Iterator

```cpp
class IntArrayIterator {
    private:
        const int* fArrayElements;
        const int fLength;
        int fIndex;
    public:
        IntArrayIterator( const int aArray[], const int aLength, int aStart = 0 );
        int& operator*() const;
        int& operator++();  // prefix
        IntArrayIterator& operator++( int );  // postfix (extra unused argument)
        bool operator=( const IntArrayIterator& aOther ) const;
        bool operator=( const IntArrayIterator& aOther );
        IntArrayIterator begin() const;
        IntArrayIterator end() const;
};
```

Forward Iterator Constructor

Arrays are passed as pointers to the first element to functions in C++.

We must use member initializer to initialize const instance variables!

We must not repeat the default value.
The Dereference Operator

- The dereference operator returns the element the iterator is currently positioned on.
- The dereference operator is a const operation, that is, it does not change any instance variables of the iterator.
- We use a const reference to avoid copying the original value stored in the underlying collection.

Prefix Increment

- The prefix increment operator advances the iterator and returns a reference of this iterator.
Postfix Increment

- The postfix increment operator advances the iterator and returns a copy of the old iterator.

```cpp
IntArrayIterator IntArrayIterator::operator++( int )
{ IntArrayIterator temp = *this;
  fIndex++;
  return temp;
}
```

Return a copy of the old iterator (position unchanged).

---

Iterator Equivalence

Two iterators are equal if and only if they refer to the same element:

- fIndex is the current index into the array
- Arrays are passed as a pointer to the first element that is constant throughout runtime.
Iterator Inequality

We implement `!=` in terms of `==`.

Auxiliary Methods

We use the default value, 0, here.

- The methods `begin()` and `end()` return fresh iterators set to the first element and after the last element, respectively.
Putting Everything Together

```java
int main() {
    int[] a = {1, 2, 3, 4, 5};
    int Sum = 0;
    for (IntArrayIterator iter(a, 5); iter != iter.end(); iter++)
        Sum += *iter;
    cout << "Iterated sum of [1,2,3,4,5] is " << Sum << endl;
    return 0;
}
```

How can we define an iterator in Java?
**Iterator Interface - java.util.Iterator**

- boolean hasNext():
  - Returns true if the iteration has more elements. In other words, returns true if next() would return an element rather than throwing an exception.

- E next():
  - Returns the next element in the iteration. Calling this method repeatedly until the hasNext() method returns false will return each element in the underlying collection exactly once.

- void remove():
  - Removes the last element returned from the underlying collection. This is an optional operation.

---

**IntArrayIterator in Java**

import java.util.*;

public class IntArrayIterator implements Iterator<Integer>
{
    private int[] fArrayElements;
    private int fIndex;

    public IntArrayIterator(int[] array) { fArrayElements = array; }

    public boolean hasNext() { return fIndex < fArrayElements.length; }

    public Integer next() {
        if (hasNext()) {
            return new Integer(fArrayElements[fIndex++]);
        }
    }

    public void remove() { // intentionally empty
    }

    public static void main(String[] args) { ... }
}
The Iterator's main Method

```java
public class IntArrayIterator implements Iterator<Integer>
{
    ...
    public static void main(String[] args)
    {
        int[] a = { 1, 2, 3, 4, 5};
        IntArrayIterator iter = new IntArrayIterator(a);
        int Sum = 0;
        while (iter.hasNext())
        {
            Sum += iter.next().intValue();
        }
        System.out.println("Iterated sum of {1,2,3,4,5} is " + Sum);
    }
}
```

What is a Design Pattern?

Christopher Alexander says:

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same thing twice.”
Essential Design Pattern Elements

A pattern has four essential elements:

- The pattern name that we use to describe a design problem,
- The problem that describes when to apply the pattern,
- The solution that describes the elements that make up the design, and
- The consequences that are the results and trade-offs of applying the pattern.

Design Patterns Are Not About Design

- Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is.
- Design patterns are not complex, domain-specific designs for an entire application or subsystem.
- Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
Creational Patterns

- Creational patterns abstract the instantiation process. They help to make a system independent of how its objects are created, composed, and represented.
- Main forms:
  - Creational patterns for classes use inheritance to vary the class that is instantiated.
  - Creational patterns for objects delegate instantiation to another object.

Example: Factory Method

- Intent:
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Collaborations:
  - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.
Structure of Factory Method

Classical Example

- A classical example of factory method is that of iterators.

- An iterator provides access to elements of a collection. A concrete iterator methods isolate the caller from knowing which class to instantiate.
**Structural Patterns**

- Structural patterns are concerned with how classes and object are composed to form larger structures:
  - Structural class patterns use inheritance to compose interfaces or implementations.
  - Structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition.

**Example: Adapter**

- **Intent:**
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

- **Collaborations:**
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.
Behavioral Patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between classes and objects:
  - Behavioral class patterns use inheritance to distribute behavior between classes.
  - Behavioral object patterns use composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.
- The classic example of a behavioral pattern is Model-View-Controller (MVC), where all views of the model are notified whenever the model’s state changes.

Example: Iterator

- Intent:
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Collaborations:
  - A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.
Structure of Iterator

Linked Lists, Recursion, and ADTs

Overview
- Recursion
- Singly-Linked Lists
- Abstract Data Types

References
Recursion

- If a procedure contains within its body calls to itself, then this procedure is said to be recursively defined.
- This approach of program specification is called recursion and is found not only in programming.
- If we define a procedure recursively, then there must exist at least one sub-problem that can be solved directly, that is without calling the procedure again.
- A recursively defined procedure must always contain a directly solvable sub-problem. Otherwise, this procedure does not terminate.

Impossible Structures: M.C. Escher

http://www.mcescher.com/
Problem-Solving with Recursion

- Recursion is an important problem-solving technique in which a given problem is reduced to smaller instances of the same problem.
- The general structure of a recursive definition is

\[ \text{X} = \ldots \text{X} \ldots \]

Left-hand side
Right-hand side

Fibonacci

- The Fibonacci numbers are the following sequence of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- In mathematical terms, the sequence \( F(n) \) of Fibonacci numbers is defined recursively as follows:

\[
\begin{align*}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2)
\end{align*}
\]
Recursive Problem-Solving: Factorials

• The factorial for positive integers is
  \[ n! = n \times (n-1) \times \ldots \times 1 \]

• The recursive definition:
  \[
  n! = \begin{cases} 
  1 & \text{if } n = 0 \\
  n \times (n-1)! & \text{if } n > 0 
  \end{cases}
  \]

Calculating Factorials

• The recursive definition tells us exactly how to calculate a factorial:
  \[
  4! = 4 \times 3! = 4 \times (3 \times 2) = 4 \times (3 \times (2 \times 1)) = 4 \times (3 \times (2 \times (1 \times 1))) = 24
  \]
  Recursive step: n=4
  Recursive step: n=3
  Recursive step: n=2
  Recursive step: n=1
  Stop condition: n=0
Recursive Factorial

```cpp
#include <iostream>
using namespace std;
long factorial( unsigned long n )
{
    if (n == 0)
        return 1;
    else
        return n * factorial( n-1 );
}
int main()
{
    cout << "6! = " << factorial( 6 ) << endl;
    return 0;
}
```

Types of Recursion

- Direct left-recursive specification: $X = XA$
  
  Pure iteration: do { $X$ } while ( $A$ );

- Direct right-recursive specification: $X = AX$
  
  Pure iteration: while ( $A$ ) { $X$ }
  
  Note, a direct left-recursive function does not terminate!
Tail-Recursion

- A function is called tail-recursive if it ends in a recursive call that does not build-up any deferred operations.

```c
long gcd( long x, long y )
{
    if (y == 0)
        return x;
    else
        return gcd( y, x % y );
}
gcd( 1246, 234 );
gcd( 234, 76 )
gcd( 76, 6 )
gcd( 6, 4 )
gcd( 4, 2 )
gcd( 2, 0 )
gcd( 0, 0 )
gcd( 0, 0 )
gcd( 0, 0 )
gcd( 0, 0 )
```

Towers of Hanoi

- Problem:
  - Move disks from a start peg to a target peg using a middle peg.

- Challenge:
  - All disks have a unique size and at no time must a bigger disk be placed on top of a smaller one.
Towers of Hanoi: Configuration

Start:

Start

Middle

Target

Finish:

Start

Middle

Target

A Recursive Solution

1. Move n-1 disks from Start to Middle:

Start

Middle

Target

2. Move 1 disk from Start to Target:

Start

Middle

Target

3. Move n-1 disks from Middle to Target:

Start

Middle

Target
A Recursive Solution: Intermediate

The Recursive Procedure

```cpp
#include <iostream>

using namespace std;

void move( int n, string start, string target, string middle )
{
    if ( n > 0 )
    {
        move( n-1, start, middle, target );
        cout << "Move disk " << n << " from " << start
             << " to " << target << ".
" << endl;
        move( n-1, middle, target, start );
    }

int main()
{
    move( 3, "Start", "Target", "Middle" );
    return 0;
}
```

Move disk 1 from Start to Target.
Move disk 2 from Start to Middle.
Move disk 1 from Middle to Target.
Move disk 3 from Start to Target.
Move disk 1 from Middle to Start.
Move disk 2 from Middle to Target.
Move disk 3 from Middle to Start.
Move disk 1 from Start to Target.
Recursion is a prerequisite for linked lists!

Problems with Arrays

- An array is a contiguous storage that provides insufficient abstractions for handling addition and deletion of elements.
- Addition and deletion require n/2 shifts on average.
- The computation time is O(n).
- Resizing effects performance.
Deletion Requires Shift

Delete 5

Addition Requires Shift

Insert 50 after 29
Linked Lists

- A linked list is a sequence of data items, each connected to the next by a pointer called `next`.

```
Data   →   Data   →   ... →   Data   →   Nil
```

- A data item may be a primitive value, a composite value, or even another pointer.
- A linked list is a recursive data structure whose nodes refers to nodes of the same type.

Nodes

- The use of a const reference member variable in `Node` prevents any accidental copies. The associated effect is one of the most difficult to understand and can lead extremely hard-to-find memory leaks if it is not properly used.

```
64 class Node
65 {
66    public:
67    const DataType& data;
68    Node* next;
69    
70    Node( const DataType& data, Node* aNext = (Node*)0 ) : data(aData) 
71    { 
72        next = aNext;
73    }
74
75};
```
Values, References, and Pointers

- C++ supports three forms of member variables:
  - values,
  - references, and
  - pointers.
- The use of references and pointers prevents copying the underlying object. But
  - References are aliases to objects that have to reside in a fixed and unique location as long the reference is active (i.e., in use). Creating references to parameters passed as actual values to a member function can violate the uniqueness criterion for locations of references.
  - Objects referenced by pointers can denote both values located on the heap and values located on the stack. Without additional information, we cannot distinguish these cases, which can lead to memory leaks.

A Simple List of Integers

```cpp
#include <iostream>
#include "Node.h"

using namespace std;

int main()
{
    Node One(1);
    Node Two(2, &One);
    Node Three(3, &Two);
    Node* lTop = &Three;
    while ( lTop != (Node*)0 )
    {
        cout << "value: " << lTop->data << endl;
        lTop = lTop->next;
    }
    return 0;
}
```

```plaintext
value: 1
value: 2
value: 3
```
Can we be more specific?

Templates

- Templates are blueprints from which classes and/or functions automatically generated by the compiler based on a set of parameters.
- Each time a template is used with different parameters is used, a new version of the class or function is generated.
- A new version of a class or function is called specialization of the template.
Node Class Template

template<
  class DataType,
  class Node
>
public:
  const DataType& data;
  Node* next;

Node(
  const DataType& aData, Node* aNext = (Node*)0
) : data(aData)
{
  next = aNext;
}

The New Main

We instantiate the template Node to Node<int>.
Class Template Instantiation

typedef Node<int> IntegerNode;
typedef Node<IntegerNode> NodeOfIntegerNode;

- Types used as arguments cannot be classes with local scope.
- Once instantiated, a class template can be used as any other class.

Using IntegerNode

```cpp
#include <iostream>
#include "Node.h"
using namespace std;
typedef Node<int> IntegerNode;

int main()
{  
    IntegerNode One( 1 );
    IntegerNode Two( 2, One );
    IntegerNode Three( 3, &Two );
    const IntegerNode* lTop = &Three;
    while ( lTop != (IntegerNode*)0 )
    {  
        cout << "value: " << lTop->data << endl;
        lTop = lTop->next;
    }
    return 0;
}  
```
NodeIterator

```cpp
#include "Node.h"

template<
class T>

class NodeIterator
{
public:
    Node* pNode;

typedef NodeIterator<T> Iterator; // Iterator type definition

NodeIterator( Node* pNode );

const T& operator*() const;
Iterator& operator++(); // prefix
Iterator operator++( int ); // postfix (extra unused argument)

bool operator==( const Iterator& aOther ) const;

};

};

// end() const;
```

NodeIterator Test

```cpp
#include <iostream>
#include "Node.h"
#include "NodeIterator.h"

using namespace std;

typedef Node<int> IntegerNode;

int main()
{
    IntegerNode One( 1 );
    IntegerNode Two( 2, &One );
    IntegerNode Three( 3, &Two );

    for ( NodeIterator<int> iter( &Three ); iter != iter.end(); ++iter )
        cout << "value: " << *iter << endl;

    return 0;
```
The Need for Pointers

- A linked-list is a dynamic data structure with a varying number of nodes.
- Access to a linked-list is through a pointer variable in which the base type is the same as the node type:

  ```c
  Node<int>* pListOfInteger = (Node<int>*)0;
  ```

  Here `(Node<int>*)0` stands for Nil.

The situation is actually a bit more complicated, but pointers work fine for now.
Node Construction

IntegerNode *p, *q;
p = new IntegerNode(5);
q = new IntegerNode(7, p);

Node Access

int a = p->data;
int b = q->next->data;
Inserting a Node

```c
IntegerNode *r;
r = new IntegerNode(6);
r->next = p;
q->next = r;
```

Deleting a Node

```c
q->next = q->next->next;
```
Insert at the Top

IntegerNode *p = (IntegerNode*)0;
P = new IntegerNode(5, p);

p → 5 → Nil

P = new IntegerNode(7, p);

P → 7 → 5 → Nil

Insert at the End

- To insert a new node at the end of a linked list we need to search for the end:

```c
IntegerNode *p = (IntegerNode*)0;
IntegerNode **pPtr = &p;

while (*pPtr != (IntegerNode*)0)
pPtr = &(*pPtr)->next;  // pointer to next
*pPtr = new IntegerNode(9);
```
Insert at the End: The Pointers

Insert at the end preserves the order of list nodes.
Insert at the End with Aliasing

- Rather than using a Pointer-to-Pointer we can just record the last next pointer.

```cpp
IntegerNode *plist = ...;
IntegerNode *pLastNode = (IntegerNode*)0;
IntegerNode *pNewNode = new IntegerNode( 9 );
if ( plist == (IntegerNode*)0 )
    { pLastNode = pNewNode; // make new node last }
    plist = pLastNode;  // set first node
else
    { pLastNode->next = pNewNode; // append new node }
    pLastNode = pNewNode; // make new node last
```

Complications with Singly-Linked Lists

- The deletion of a node at the end of a list requires a search from the top to find the new last node.
Abstraction

New Sets of Values

- The definition of a new data type (i.e., a new set of values) consists of two ingredients:
  - Some set, called the interface, that serves as representation of the newly define data type, and
  - Some set of procedures, called the implementation, that provides the operations, which can be used to manipulate entities of the newly defined data type.
Representation Independence

- The representation of new data types can be often very complex.
- When working with new data types, we usually do not want to be concerned with their actual representation. In fact, programs become more reliable and robust, if they do not depend on the actual representation of data type. Data types that do not expose their actual representation are called representation transparent.
- Data types in C/C++ are in general not representation transparent (e.g. the size of integers in C/C++ is platform dependent).
- Data types in Java are basically representation transparent (arrays are an exception, since they are represented by objects).

Opaque Representation

- A data type is opaque if there is no way to find out its representation, even by printing.

Node is not opaque as we have unrestricted access to data and next!
Object-Oriented Encapsulation

- Object-oriented encapsulation is a principle that provides the means to obtain an opaque data type:
  - All instance variables have private visibility.
  - All member functions have public visibility.
  - The extent to which this scheme is used can differ!

Pros & Cons

- Opaque data types enforce the use of defining procedure (i.e., a constructor).
- Opaque data types are more secure. Access to values of opaque data types is only possible by means of access procedures defined in an interface.
- Transparent data types are easier to debug and to extend.
- The fact that transparent data types expose their internal representation is also a disadvantage (limited security).
Abstract Data Type

- The technique used to define new data types independently of their actual representation is called data abstraction.

- A data type, which has been defined in this way is called abstract data type. A client (program) can use values of an abstract data type by means of the interface without knowing their actual representation (which can change over time).

---

Representation Strategies for ADTs

- Given an interface for a data type we can change the underlying representation if needed using different strategies.
Data Abstraction

- Data abstraction enforces representation independence.
- Data abstraction divides the data types in interfaces and implementations:
  - Interfaces are used to specify the set of values the data types represents, the operations, which are available for that data type, and properties these operations may be guaranteed to have.
  - Implementations provide a specific representation of the data and code for the operations.

Examples of Abstract Data Types

- Files
- Lists, hash tables, vectors, bags
- Strings, records, arrays
- Objects with private instance variables and public methods
- Standardized integers (e.g. in Java the type int is represented using 32 bits and big endian format, on every platform)
Constructors and Access Procedures

- In order to create, manipulate, and verify that a given value is of the desired data type, we need the following ingredients:
- Constructors that allow us to build values of a given data type,
- A predicate that tests whether a given value is a representation of a particular data type, and
- Some access procedures that allow us to extract a particular information from a given representation of a data type.

Can we represent lists as abstract data types?
```cpp
// List implementation

template<
    class T>

class List {
    private:
        typedef Node<T>* ListImpl; // private representation
        ListImpl fTop;              
        ListImpl fLast;             
        int fCount;                 

    public:
        typedef NodeIterator<T> ListIterator;  

        List();                      
    ~List();

        bool isEmpty() const;       // empty list predicate
        int size() const;           // get number of elements
        void addC(const T &element); // add element at end
        void addFirst(const T &element); // add element at top

        void dropC(const T &element); // delete first element
        void dropLast();              // delete last element

        const T & operator[](int index) const; // List indexer
        ListIterator begin() const;      // List iterator
        ListIterator end() const;        // List iterator

};
```

---

```cpp
#include <iostream>
#include "List.h"

using namespace std;

int main()
{
    List<int> list;

    list.addFirst(1);
    list.addFirst(2);
    list.addFirst(3);

    for (List<int>::ListIterator iter = list.begin(); iter != list.end(); ++iter)
    {
        cout << "value: " << *iter << endl;
    }

    return 0;
}
```
The type `list<T>` is part of the standard template library.

Container Types, Stacks, and Queues

Overview
- Stacks
- Container types and references

References
Stacks

- A stack is a special version of a linear list where items are added and deleted at only one end called the top.

```
push
  20  top
  29
  5
  3
```

```
pop
  5
  3
```

Stack Behavior

- Stacks manage elements in last-in, first-out (LIFO) manner.
- A stack underflow happens when one tries to pop on an empty stack.
- A stack overflow happens when one tries to push onto a full stack.
Applications of Stacks

- Reversal of input
- Checking for matching parentheses (e.g., stack automata in compiler implementations)
- Backtracking (e.g., Prolog or graph analysis)
- State of program execution (e.g., storage for parameters and local variables of functions)
- Tree traversal

Stack Frames (PASCAL)

incoming arguments

frame pointer

outgoing arguments

stack pointer

higher addresses

previous frame

current frame

next frame
Stack Interface

- When defining a container type we wish to minimize the number of value copies required for the objects stored in the container. In order to achieve this, we use const references (e.g. const T&).

Container Types

- Stacks belong to a special group of data types called container types.
- The de facto standard approach for the definition of container types in C++ is to use value-based semantics.
- Other examples of container types are Lists, Queues, Hash Tables, Maps, Arrays, or Trees.
The Stack’s Private Interface

- Inside Stack we need to be able to store objects of type T. Hence we need to allocate dynamically memory (i.e., an array of type T) and store the address of the first element in a matching pointer variable.

Notes on Templates

- When using templates, the C++ compiler must have access not only the specification of a template class but also to all method implementations in order to properly instantiate the template.
- Templates are not to confused with library classes. Templates are blueprint that have to be instantiated for each separate type application.
- Think of templates as special forms of macros.
- When defining a template class, we need to implement, like in Java, all methods within the class definition, usually in a header file.
Stack Constructor

```cpp
Stack( int aSize )
{
    if ( aSize <= 0 )
        throw std::invalid_argument( "Illegal stack size." );
    else
    {
        mElements = new T[aSize];
        mStackPointer = 0;
        mStackSize = aSize;
    }
}
```

Stack Destructor

- There are two forms of delete:
  - `delete ptr` - release the memory associated with pointer `ptr`.
  - `delete[] arr` - release the memory associated with all elements of array `arr` and the array `arr` itself.
- Whenever one allocates memory for an array, for example `char* arr = new char[10]`, one must use the array form of `delete` to guarantee that all array cells are released.
Stack Auxiliaries

- `isEmpty()`: Boolean predicate to indicate whether there are elements on the stack.
- `size()`: returns the actual stack size.

Push

- The `push` method stores an item at the next free slot in the stack, if there is room.
Pop

- The `pop` method shifts the stack pointer to the previous slot in the stack, if there is such a slot. Note, the element in the current slot itself is not yet destroyed.

Top

- The `top` method returns a const reference to the item in the current slot in the stack, if there is such a slot.
Stack Sample

```cpp
int main()
{
    Stack<int> sStack(10);
    sStack.push(2);
    sStack.push(34);
    sStack.push(68);
    cout << "Number of elements on the stack: " << sStack.size() << endl;
    cout << "Top: " << sStack.top() << endl;
    sStack.pop();
    cout << "Top: " << sStack.top() << endl;
    sStack.pop();
    cout << "Top: " << sStack.top() << endl;
    sStack.pop();
    cout << "Number of elements on the stack: " << sStack.size() << endl;
    return 0;
}
```

Dynamic Stack

- We can define a dynamic stack that uses a list as underlying data type to host an arbitrary number of elements:

```cpp
#include <iostream>
#include <list>

template<class T>
class DynamicStack
{
private:
    std::list<T> mElements;
public:
    bool isEmpty() const;
    int size() const;
    void push(const T &tItem);
    void pop();
    const T &top() const;
};
```
Reverse Polish Notation

- Reverse Polish Notation (RPN) is a prefix notation wherein operands come before operators.

RPN: \[ a \ b \ * \ c \ + \]

Infix: \[ a \ * \ b \ + \ c \]

RPN Calculation

- \( x = 3 \* 5 + 29 \)

Stack:

\[
\begin{array}{ccc}
3 & 5 & 15 \\
15 & 29 & 44 \\
\end{array}
\]

Steps:
1. Load 3
2. Load 5
3. Multiply (3 * 5)
4. Load 29
5. Add (15 + 29)
6. Store \( x \)
**Queues**

- A queue is a special version of a linear list where access to items is only possible at its front and end.

```
enqueue  20 5  dequeue
29        3
```

**Queue Behavior**

- Queues manage elements in first-in, first-out (FIFO) manner.
- A queue underflow happens when one tries to dequeue on an empty queue.
- A queue overflow happens when one tries to enqueue on a full queue.
A Queue Interface

```cpp
#include "list.h"

template<class T>
class Queue
{
private:
List<T> &elements;

public:
Queue();
~Queue();
bool is_empty() const; // empty queue predicate
int size() const; // get number of elements
void enqueue(const T &element); // insert element at end
const T &dequeue(); // remove element from front
};
```

Queue Service Members

```cpp
Queue()
{}
~Queue()
{}
bool is_empty() const // empty queue predicate
{
return elements.is_empty();
}
int size() const // get number of elements
{
return elements.size();
}
```
Queue Semantics

```c
void enqueue( const T &element ) // insert element at end
{
    elements.add( element );
}

const T &dequeue() const // remove element from front
{
    if ( isEmpty() )
    {
        const T &result = elements[0];
        elements.dropFirst();
        return result;
    } else
    {
        throw std::underflow_error( "Queue is empty!" );
    }
}
```

Queue Test

```c
#include <iostream>
#include "Queue.h"

using namespace std;

int main()
{
    Queue<int> lQueue;
    lQueue.enqueue( 20);
    lQueue.enqueue( 3 );
    lQueue.enqueue( 37 );
    cout << "Number of elements in the queue: " << lQueue.size() << endl;
    cout << "value: " << lQueue.dequeue() << endl;
    cout << "value: " << lQueue.dequeue() << endl;
    cout << "value: " << lQueue.dequeue() << endl;
    cout << "Number of elements in the queue: " << lQueue.size() << endl;
    return 0;
}
```
Requirements for a Priority Queue

• The underlying data structure for a priority queue must be sorted (e.g., `SortedList<T>`).

• Elements are queued using an integer to specify priority. We use a `Pair<Key, T>` to store elements with their associated priority.

• We need to provide a matching `operator<` on key values to sort elements in the priority queue.

Priority Queue

```
(5,29)  
\(\downarrow\)  
(1,14) (4,20) (5,30) (10,6) (10,5)  
\(\downarrow\)  
(1,14) (4,20) (5,29) (5,30) (10,6)  
```

```
<table>
<thead>
<tr>
<th>enqueue</th>
<th>dequeue</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,14)</td>
<td>(10,6)</td>
</tr>
<tr>
<td>(4,20)</td>
<td>(10,5)</td>
</tr>
<tr>
<td>(5,30)</td>
<td>(5,29)</td>
</tr>
<tr>
<td>(10,6)</td>
<td></td>
</tr>
</tbody>
</table>
```
The Pair Class

```cpp
template<
class K, class V>

class Pair{

public:
    K key;
    V value;

    Pair(const K &aKey, const V &aValue) : key(aKey), value(aValue) {}

    bool operator<(const Pair<K,V> &aOther) const
    {
        return key < aOther.key;
    }

};
```

SortedList uses an increasing order.
A Priority Queue

```cpp
template<class T>
class PriorityQueue;
private:
  SortedList<T> elements;  // I must define a partial order
public:
  PriorityQueue();
  ~PriorityQueue();
  bool isEmpty() const;  // empty queue predicate
  int size() const;  // get number of elements
  void enqueue(const T& element);  // insert element
  const T& dequeue();  // remove element from front
```

Priority Queue Semantics

```cpp
void enqueue(const T& element)  // insert element
{
  elements.add(element);
}

const T& dequeue()  // remove element from front
{
  if (isEmpty())
  {
    // increasing order of priorities
    const T& result = elements[elements.size()-1];
    elements.removeLast();
    return result;
  }
  else
    throw std::underflow_error("Queue is empty!");
}
A PriorityQueue Test

```cpp
int main()
{
    PriorityQueue<Pair<int, int>> q;
    Pair<int, int> p1(4, 28);
    Pair<int, int> p2(5, 38);
    Pair<int, int> p3(5, 29);
    q.enqueue(p1);
    q.enqueue(p2);
    q.enqueue(p3);
    cout << "Number of elements in the queue: " << q.size() << endl;
    cout << "value: " << q.dequeue().value << endl;
    cout << "value: " << q.dequeue().value << endl;
    cout << "value: " << q.dequeue().value << endl;
    cout << "Number of elements in the queue: " << q.size() << endl;
    return 0;
}
```

Copy Control and Memory Management

Overview

- Types of memory
- Copy constructor, assignment operator, and destructor
- Reference counting with smart pointers

References

Static Read-Write Memory

- C++ allows for two forms of global variables:
  - Static non-class variables,
  - Static class variables.
- Static variables are mapped to the global memory. Access to them depends on the visibility specifies.
- We can find a program's global memory in the so-called read-write .data segment.

The Keyword static

- The keyword *static* can be used to
  - mark the linkage of a variable or function internal,
  - retain the value of a local variable between function calls,
  - declare a class instance variable,
  - define a class method.
Read-Write Static Variables

- Static class variables must be initialized outside the class.

Static Read-Only Memory

- In combination with the const specifier we can also define read-only global variables or class variables:

- Const variables are often stored in the program's read-only text segment.
Program Memory: Stack

- All value-based objects are stored in the program's stack.
- The program stack is automatically allocated and freed.
- References to stack locations are only valid when passed to a callee. References to stack locations cannot be returned from a function.

Stack Frames (C)

- incoming arguments
- stack pointer
- outgoing arguments

High addresses

Previous frame

Current frame

Next frame

temporaries
save registers
arguments
return address
temporaries
save registers
arguments
Program Memory: Heap

- Every program maintains a heap for dynamically allocated objects.
- Each heap object is accessed through a pointer.
- Heap objects are not automatically freed when pointer variables become inaccessible (i.e., go out of scope).
- Memory management becomes essential in C++ to reclaim memory and to prevent the occurrences of so-called memory leaks.

List::dropFirst()

```cpp
void dropFirst()
{
    if (fTop != (ListImpl*)0)
    {
        ListImpl*lNode = fTop;
        fTop = fTop->next;
        if (fTop == (ListImpl*)0)
            fLast = (ListImpl*)0;
        delete lNode;
        fCount--;
    }
}
```

Release memory associated with Node object on the heap.
The Dominion Over Objects

- Alias control is one of the most difficult problems to master in object-oriented programming.
- Aliases are the default in reference-based object models used, for example, in Java and C#.
- To guarantee uniqueness of value-based objects in C++, we are required to define copy constructors.

The Copy Constructor

- Whenever one defines a new type, one needs to specify implicitly or explicitly what has to happen when objects of that type are copied, assigned, and destroyed.
- The copy constructor is a special member, taking just a single parameter that is a const reference to an object of the class itself.
SimpleString

```cpp
class SimpleString
{
private:
    char *fCharacters;
public:
    SimpleString();
    ~SimpleString();
    SimpleString &operator=(const char &Character);
    const char *operator() const;
};
```

SimpleString: Constructor & Destructor

```cpp
#include <iostream>
#include "SimpleString.h"
using namespace std;

SimpleString::SimpleString()
{
    fCharacters = new char[1];
    *fCharacters = 'A';
}

SimpleString::~SimpleString()
{
    delete fCharacters;
}
```
SimpleString: The Operators

```cpp
SimpleString & SimpleString::operator= (const char *Character)
{
    char* Temp = new char[strlen(Character) + 2];
    unsigned int i = 0;
    for (; i < strlen(Character); i++)
        Temp[i] = Character[i];
    Temp[i] = '\0';
    delete fCharacters;
    fCharacters = Temp;
    return *this;
}

const char* SimpleString::operator*() const
{
    return fCharacters;
}
```

Implicit Copy Constructor

```cpp
int main()
{
    SimpleString s1;
    s1 = "A";
    SimpleString s2 = s1;
    s2 = "B";
    cout << "S1: " << *s1 << endl;
    cout << "S2: " << *s2 << endl;
    return 0;
}
```
What Has Happened?

Shallow copy:  
`s2.fCharacters = s1.fCharacters`

Double free:  
delete s2.fCharacters, which was called in s2 + 'B'.

We need an explicit copy constructor!

class SimpleString
{
  private:
    char* fCharacters;
  public:
    SimpleString();
    -SimpleString();
    SimpleString(const SimpleString& otherString);
    SimpleString& operator=(const char* character);
    const char* operator*(const);
};
The Explicit Copy Constructor

- When a copy constructor is called, then all instance variables are uninitialized in the beginning.

Explicit Copy Constructor in Use
What Has Happened?

```cpp
int main()
{
SimpleString s1;
s1 = 'A';
SimpleString s2 = s1;
s2 = 'B';
cout << "S1: " << s1 << endl;
cout << "S2: " << s2 << endl;
<delete s1;>
<delete s2;>
return 0;
}
```

Deep copy: s2.fCharacters != s1.fCharacters

That's it. No more problems, or?
A Simple Assignment

```c
int main()
{
    SimpleString s1;
    s1 = 'A';
    SimpleString s2 = s1;
    s2 += 'B';

    s1 = s2;
    cout << "s1: " << *s1 << endl;
    cout << "s2: " << *s2 << endl;
    return 0;
}
```

What Has Happened?

Shallow copy: `s2.characters == s1.characters`

Double free: `delete s1;` and `delete s2;` is the same as `delete s1.characters;`
Rule Of Thumb

- Copy control in C++ requires three elements:
  - a copy constructor
  - an assignment operator
  - a destructor
- Whenever one defines a copy constructor, one must also define an assignment operator and a destructor.

We need an explicit assignment operator!

```cpp
class SimpleString
{
private:
    char* fCharacters;

public:
    SimpleString();
    SimpleString();
    SimpleString(const SimpleString& aOtherString);
    SimpleString& operator=(const SimpleString& aOtherString);
    SimpleString& operator=(const char* aCharacter);
    const char* operator*() const;
};
```
The Explicit Assignment Operator

When the assignment operator is invoked, then all instance variables are initialized in the beginning. We need to release the memory first!

Explicit Assignment Operator in Use
What Has Happened?

```cpp
int main()
{
    SimpleString s1;
    s1 += 'A';
    SimpleString s2 = s1;
    s2 += 'B';
    s1 = s2;
    cout << "s1: " << *s1 << endl;
    cout << "s2: " << *s2 << endl;
    delete s1;
    delete s2;
    return 0;
}
```

Deep copy: s2.fCharacters != s1.fCharacters

Cloning: Alias Control for References
Copying Pointers

```cpp
int main()
{
    SimpleString* ps1 = new SimpleString();
    (*ps1) = 'A';
    SimpleString* ps2 = ps1;
    (*ps2) = 'B';
    cout << "S1: " << *ps1 << endl;
    cout << "S2: " << *ps2 << endl;
    delete ps1;
    delete ps2;
    return 0;
}
```

What Has Happened?

- **Shallow copy:** `ps2 == ps1`
- **Double free:** `delete ps2`, which is the same as `ps1`. 
Solution: A clone() Method

```cpp
class SimpleString {
private:
    char *Characters;
public:
    SimpleString();
    virtual ~SimpleString();
    SimpleString(const SimpleString& aOtherString);
    SimpleString& operator=(const SimpleString& aOtherString);
    virtual SimpleString* clone() const;
    SimpleString& operator=(const char aCharacter);
    const char* operator*() const;
};
```

- It is best to define the destructor of a class virtual always in order to avoid problems later.

The Use of clone()

```cpp
SimpleString* SimpleString::clone() {
    return new SimpleString(*this);
}
```
Problems With Cloning

- The member function clone() must be defined virtual to allow for proper redefinition in subtypes.
- Whenever a class contains a virtual function, then its destructor is required to be defined virtual as well.
- The member function clone() can only return one type. When a subtype redefines clone(), only the super type can be returned.

Non-virtual Cloning Does Not Work!

- One could define clone() non-virtual and use overloading. But this does not work as method selection starts at the static type of the pointer.

```cpp
SimpleString* pToSimpleString = new SubtypeOfSimpleString();
SimpleString* cl = pToSimpleString->clone(); // SimpleString::clone()
```
Reference-based Semantics: When Do We Destroy Objects?

Reference Counting

- A simple technique to record the number of active uses of an object is reference counting.
- Each time a heap-based object is assigned to a variable the object's reference count is incremented and the reference count of what the variable previously pointed to is decremented.
- Some compilers emit the necessary code, but in case of C++ reference counting must be defined (semi-)manually.
Smart Pointers: Handle

```c
template<class T>
class Handle
{
  private:
  T* mPointer;
  int mCount;
  void addRef();
  void releaseRef();
  
  public:
  Handle(T* aPointer = (T*)0);
  Handle(const Handle<T>& aOtherHandle);
  ~Handle();
  Handle& operator=(Handle<T>& aOtherHandle);
  T& operator*();
  T* operator->();
};
```

The Use of Handle

- The template class Handle provides a pointer-like behavior:
  - Copying a Handle will create a shared alias of the underlying object.
  - To create a Handle, the user will be expected to pass a fresh, dynamically allocated object of the type managed by the Handle.
  - The Handle will own the underlying object. In particular, the Handle assumes responsibility for deleting the owned object once there are no longer any Handles attached to it.
Handle: Constructor & Destructor

Create a shared counter

Decrement reference count

Handle: addRef & releaseRef

Increment reference count

Decrement reference count and delete object if it is no longer referenced anywhere.
Handle: Copy Control

```cpp
Handle( const Handle& aOtherHandle )
{
    fPointer = aOtherHandle.fPointer;
    fCount = aOtherHandle.fCount;  // increment use
    addRef();
}

Handle& operator=(Handle& aOtherHandle)
{
    aOtherHandle.addRef();  // increment use
    releaseRef();  // release old handle
    fPointer = aOtherHandle.fPointer;
    fCount = aOtherHandle.fCount;
    return *this;
}
```

Handle: Pointer Behavior

```cpp
T& operator*()
{
    if (fPointer)
        return *fPointer;
    else
        throw std::runtime_error("Dereference of unbound handle!");
}

T* operator->()
{
    if (fPointer)
        return fPointer;
    else
        throw std::runtime_error("Access through unbound handle!");
}
```
Using Handle

```c++
int main()
{
    Handle<SimpleString> hs1( new SimpleString() );
    "ms1 = 'A';
    Handle<SimpleString> hs2( hs1->clone() );
    "ms2 = 'B';
    Handle<SimpleString> hs3 = hs1;
    cout << "HS1: ' " "ms1 << endl;
    cout << "HS2: ' " "ms2 << endl;
    cout << "HS3: ' " "ms3 << endl;
    return 0;
}
```

Reference Counting Limits

- Reference counting fails on circular data structures like double-linked lists.
- Circular data structures require extra effort to reclaim allocated memory. **Know solution: Mark-and-Sweep**
Trees

Overview
- Trees
- Search Trees

References

Basics

- A tree $T$ is a finite, non-empty set of nodes,

\[ T = \{r\} \cup T_1 \cup T_2 \cup ... \cup T_n, \]

with the following properties:
- A designated node of the set, $r$, is called the root of the tree.
- The remaining nodes are partitioned into $n \geq 0$ subsets $T_1, T_2, ..., T_n$, each of which is a tree.
Tree Examples

$T_A$: A
$T_B$: B
$T_C$: D

$T_A = \{A\}$
$T_B = \{B, \{C\}\}$
$T_C = \{D, \{E, \{H\}\}, \{F, \{G, \{I\}\}, \{J, \{K\}, \{L\}\}\}\}$

Parent and Child

- The root node $r$ of tree $T$ is the parent of all the roots $r_i$ of the subtrees $T_i$, $1 < i \leq n$.
- Each root $r_i$ of subtree $T_i$ of tree $T$ is called a child of $r$. 
**Degree**

- The degree of a node is the number of subtrees associated with that node. For example, the degree of $T_C = \{D, \{E, \{H\}\}, \{F, \{G, \{I\}\}, \{J, \{K\}, \{L\}\}\}\}$ is 2.

- A node of degree zero has no subtrees. Such a node is called a leaf. For example, the leaves of $T_C$ are $\{H, I, K, L\}$.

- Two roots $r_i$ and $r_j$ of distinct subtrees $T_i$ and $T_j$ with the same parent in tree $T$ are called siblings. For example, $T_i = \{G, \{I\}\}$ and $T_j = \{J, \{K\}, \{L\}\}$ are siblings in $T_C$.

---

**Path and Path Length**

- Given a tree $T$ containing the set of nodes $R$, a path in $T$ is defined as a non-empty sequence of nodes

$$P = \{r_1, r_2, \ldots, r_k\}$$

where $r_i \in R$, for $1 \leq i \leq k$ such that the $i$th node in the sequence, $r_i$, is the parent of the $(i+1)$th node in the sequence $r_{i+1}$.

- The length of path $P$ is $k-1$. 
Depth and Height

- The depth of a node \( r_i \in R \) in a tree \( T \) is the length of the unique path in \( T \) from its root to the node \( r_i \).
- The height of a node \( r_i \in R \) in a tree \( T \) is the length of the longest path from node \( r_i \) to a leaf. Therefore, the leaves are all at height zero.
- The height of a tree \( T \) is the height of its root node \( r \).

Nodes With the Same Degree

- The general case allows each node in a tree to have a different degree. We now consider a variant of trees in which each node has the same degree.
- Unfortunately, it is not possible to construct a tree that has a finite number of nodes, which all have the same degree \( N \), except the trivial case \( N = 0 \).
- We need a special notion, called empty tree, to realize trees in which all nodes have the same degree.
**N-ary Trees**

- An N-ary tree $T$, $N \geq 1$, is a finite set of nodes with one of the following properties:
  - Either the set is empty, $T = \emptyset$, or
  - The set consists of a root, $R$, and exactly $N$ distinct N-ary trees. That is, the remaining nodes are partitioned into $N \geq 1$ subsets, $T_1, T_2, \ldots, T_N$, each of which is an N-ary tree such that
    \[ T = \{ R, T_1, T_2, \ldots, T_N \}. \]

**N-ary Tree Examples**

- $T_A$: $N = 3$
  - $T_A = \{ A, \emptyset, \emptyset, \emptyset \}$

- $T_B$: $N = 4$
  - $T_B = \{ D, \{ E, \emptyset, \emptyset, \emptyset \}, \emptyset, \{ F, \{ G, \emptyset, \emptyset, \emptyset \}, \emptyset, \emptyset \} \}$
The Empty Tree

- The empty tree, $T = \emptyset$, is a tree.
- From the modeling point of view an empty $N$-ary tree has no key and has to have the same type as a non-empty $N$-ary tree.
- To use null (i.e., $0$) to denote an empty $N$-ary tree is inappropriate, as null refers to nothing at all!

Sentinel Node: NIL

- A sentinel node is a programming idiom used to facilitate tree-based operations.
- A sentinel node in tree structures indicates a node with no children.
- Sentinel nodes behave like null-pointers. However, unlike null-pointers, which refer to nothing, sentinel nodes denote proper, yet empty, subtrees.
We do not wish to allow clients to create empty NTrees.

The Empty NTree Constructor

- We use (T*), the null pointer, to initialize the fKey with a suitable value of type pointer to T.
- Each subtree-node is set to null in the empty NTree.
The Empty NTree Constructor

- We store the address of the reference aKey in fKey.
- Each child node in a non-empty NTree leaf node is set to the location of NIL, the sentinel node for NTree.

The NTree Destructor

- In the destructor of NTree only non-sentinel nodes are destroyed.
The NTree Sentinel

- Static instance variables, like the NTree sentinel NIL, need to be initialized outside the class definition.
- Here, NIL is initialized using the private default constructor.
- The scope of NIL is NTree, which means that all members of NTree are available, including the private constructor to initialize NIL.

The NTree Auxiliaries

```cpp
bool isEmpty() const
{
    return this == NIL;
}

const T& key() const
{
    if (isEmpty())
        throw std::domain_error("Empty NTree!");
    return *Key;
}
```
**Attaching a New Subtree**

```cpp
void attachNTree(int aIndex, NTree<T, N>* aNTree)
{
  if (isEmpty())
    throw std::domain_error("Empty NTree!");
  if ( (aIndex >= 0) && (aIndex < N) )
  {
    if ( Nodes[aIndex] != NIL )
      throw std::domain_error("Non-empty subtree present!");
    Nodes[aIndex] = aNTree;
  }
  else
    throw std::out_of_range("Illegal subtree index!");
}
```

**Accessing a Subtree**

```cpp
NTree& operator[](int aIndex) const
{
  if ( isEmpty() )
    throw std::domain_error("Empty NTree!");
  if ( (aIndex >= 0) && (aIndex < N) )
  {
    return *Nodes[aIndex]; // return reference to subtree
  }
  else
    throw std::out_of_range("Illegal subtree index!");
}
```

* We return a reference to the subtree rather than a pointer. This way, we prevent accidental manipulations outside the tree structure.
Removing a Subtree

```cpp
NTree* detachNTree(int aIndex)
{
    if (isEmpty())
        throw std::domain_error("Empty NTree!");
    if ((aIndex >= 0) && (aIndex < N))
    {
        NTree* Result = fNodes[aIndex];
        fNodes[aIndex] = NULL;
        return Result; // return subtree
    }
    else
        throw std::out_of_range("Illegal subtree index!");
}
```

A NTree Example

```cpp
int main()
{
    string s1( "Hello World!" );
    string s2( "A" );
    string s3( "B" );
    string s4( "C" );
    NTree<string,ASTree> aTree1(s1);
    NTree<string,ASTree> aTree2(s2);
    NTree<string,ASTree> aTree3(s3);
    NTree<string,ASTree> aTree4(s4);
    aTree1.attachNTree( 0, &aTree1 );
    aTree1.attachNTree( 1, &aTree2 );
    aTree1.attachNTree( 2, &aTree3 );
    cout << "Key: " << aTree1.getKey() << endl;
    cout << "Key: " << aTree2.getKey() << endl;
    aTree1.detachNTree( 0 );
    aTree1.detachNTree( 1 );
    aTree1.detachNTree( 2 );
    return 0;
}
```
2-ary Trees: Binary Trees

- A binary tree $T$ is a finite set of nodes with one of the following properties:
  - Either the set is empty, $T = \emptyset$, or
  - The set consists of a root, $R$, and exactly 2 distinct binary trees $T_L$ and $T_R$, $T = \{R, T_L, T_R\}$.
- The tree $T_L$ is called the left subtree of $T$ and the tree $T_R$ is called the right subtree of $T$.

We cannot just create a type alias!

```cpp
template<class T>
  typedef NTrees<2> BTree<T>;
```

Many different algorithms for manipulating trees exist, but these algorithms have in common that they systematically visit all the nodes in the tree.

There are essentially two methods for visiting all nodes in a tree:

- Depth-first traversal,
- Breadth-first traversal.
Depth-first Traversal

- **Pre-order traversal:**
  - Visit the root first; and then
  - Do a preorder traversal of each of the subtrees of the root one-by-one in the order given (from left to right).

- **Post-order traversal:**
  - Do a postorder traversal of each of the subtrees of the root one-by-one in the order given (from left to right); and then
  - Visit the root.

- **In-order traversal:**
  - Traverse the left subtree; and then
  - Visit the root; and then
  - Traverse the right subtree.

Breadth-first Traversal

- Breadth-first traversal visits the nodes of a tree in the order of their depth (from left to right).
**Pre-order Traversal Example**

Depth = 0:

D

Depth = 1:

2  D  E  F  H

Depth = 2:

3  5  G  I

Depth = 3:

6  8  I  K  J  L

D-E-H-F-G-I-J-K-L

**Post-order Traversal Example**

Depth = 0:

D 9

Depth = 1:

2  D  E  F

Depth = 2:

1  F  G  H

Depth = 3:

3  G  I  K  J  L 6

H-E-I-G-K-L-J-F-D
**In-order Traversal Example**

Depth = 0:  
D

Depth = 1:  
3  
H-E-D-I-G-F-K-J-L  
2  
E  
6  
F

Depth = 2:  
1  
H  
5  
G  
8  
J

Depth = 3:  
4  
I  
7  
K  
9  
L

**Breadth-first Traversal Example**

Depth = 0:  
D

Depth = 1:  
1  
D-E-F-H-G-J-I-K-L  
2  
E  
3  
F

Depth = 2:  
4  
H  
5  
G  
6  
J

Depth = 3:  
7  
I  
8  
K  
9  
L
The Visitor Pattern

- **Intent:**
  - Represent an operation to be performed on the elements of an object structure. Visitor lets one define a new operation without changing the classes of the elements on which it operates.

- **Collaborations:**
  - A client that uses the Visitor pattern must create a `ConcreteVisitor` object and then traverse the object structure, visiting each element with the visitor.
  - When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.
A Tree Visitor

```cpp
#include <iostream>

template<class T>
class TreeVisitor
{
public:
  virtual ~TreeVisitor() {} // virtual default destructor

  // default behavior
  virtual void preVisit(const T &aKey) const {}
  virtual void postVisit(const T &aKey) const {}
  virtual void inVisit(const T &aKey) const {}

  virtual void visit(const T &aKey) const
  {
    std::cout << aKey << " ";
  }
};
```

PreOrderVisitor

```cpp
template<class T>
class PreOrderVisitor : public TreeVisitor<T>
{
public:

  // override pre-order behavior
  virtual void preVisit(const T &aKey) const
  {
    visit(aKey); // invoke default behavior
  }
};
```
PostOrderVisitor

```cpp
template<class T>
class PostOrderVisitor : public TreeVisitor<T>
{
public:
    // override post-order behavior
    virtual void postVisit( const T& aKey ) const
    {
        visit( aKey );  // invoke default behavior
    }
};
```

InOrderVisitor

```cpp
template<class T>
class InOrderVisitor : public TreeVisitor<T>
{
public:
    // override in-order behavior
    virtual void inVisit( const T& aKey ) const
    {
        visit( aKey );  // invoke default behavior
    }
};
```
Depth-first Traversal for BTree

```c++
void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const
{
    if (!isEmpty())
    {
        aVisitor.preVisit( key() ); // Show if PreOrder
        left().traverseDepthFirst( aVisitor );
        aVisitor.inVisit( key() ); // Show if InOrder
        right().traverseDepthFirst( aVisitor );
        aVisitor.postVisit( key() ); // Show if PostOrder
    }
}
```

```c++
int main()
{
    string s1( "Hello World!" );
    string s2[ "A" ];
    string s3[ "B" ];
    string s4[ "C" ];
    BTree<int> ATree( s1 );
    BTree<int> BTree1[ s2 ];
    BTree<int> BTree2[ s3 ];
    BTree<int> BTree3[ s4 ];
    ATree.attachLeft( &BTree1 );
    ATree.attachRight( &BTree2 );
    ATree.left().attachLeft( &BTree3 );
    cout << "Key: " << ATree.key() << endl;
    cout << "Key: " << ATree.left().left().key() << endl;
    ATree.traverseDepthFirst( PreOrderVisitor<string>() );
    cout << endl;
    ATree.left().detachLeft();
    ATree.detachLeft();
    ATree.detachRight();
    return 0;
}
```
Breadth-first Traversal Implementation

D
E
F
G
H
I
J
K
L

Traversal Queue

D
E
F
H
G
J
I
K
L

D-E-F-H-G-J-I-K-L

Breadth-first Traversal for BTee

```cpp
void traverseBreadthFirst( const TreeVisitor& v ) const
{
    std::queue<T*> q;
    if ( !q.empty() ) q.enqueue( *this );
    // start with root node
    while ( !q.empty() )
    {
        const T* head = q.dequeue();
        if ( !head ) v.visit( head->key() );
        // output
        if ( head->left() ) q.enqueue( head->left() );
        if ( head->right() ) q.enqueue( head->right() );
        // enqueue left
        if ( head->left() ) // enqueue right
            v.visit( head->left()->key() );
            // enqueue right
    }
}```
M-way Search Tree

An M-ary search tree $T$ is a finite set of nodes with one of the following properties:

- either the set is empty, $T = \emptyset$, or

- for $2 \leq n \leq M$, the set consists of $n$ M-ary subtrees $T_1, T_2, \ldots, T_{n-1}$ and

and the keys and nodes satisfy the data ordering properties:

- The keys in each node are distinct and ordered, i.e., $k_i < k_j$, for $1 \leq i \leq n-1$.

- All the keys contained in subtree $T_{i-1}$ are less than $k_i$, i.e., $\forall k \in T_{i-1}: k < k_i$, for $1 \leq i \leq n-1$. The tree $T_{i-1}$ is called left subtree with respect the key $k_i$.

- All the keys contained in subtree $T_i$ are greater than $k_i$, i.e., $\forall k \in T_i: k > k_i$, for $1 \leq i \leq n-1$. The tree $T_i$ is called right subtree with respect the key $k_i$. 
2-way Search Tree

- A 2-ary (binary) search tree $T$ is a finite set of nodes with one of the following properties:
  - either the set is empty, $T = \emptyset$, or
  - the set consists of one key, $r$, and exactly 2 binary subtrees $T_L$ and $T_R$ such that following properties are satisfied:
    - All the keys in the left subtree, $T_L$, are less than $r$, i.e., $\forall k \in T_L: k < r$.
    - All the keys contained in the right subtree, $T_R$, are greater than $r$, i.e., $\forall k \in T_R: k > r$. 
A Binary Search Tree Example

Traversing a Binary Search Tree

- Binary Tree Search:
  - Traverse the left subtree, and then
  - Visit the root, and then
  - Traverse the right subtree.

- We use in-order traversal to search for a given key in an M-ary search tree.
Binary Search Tree Operations

Insert 8:

1. Insert 8:

2. The predecessor of 25: 

Delete 25:
We define the representation of a binary search tree in class BSTNode in the private section of BSTree.
bool insert( const T &key )
{
    BSTNode<T>* x = fRoot;
    BSTNode<T>* y = &BSTNode<T>::NIL;
    BSTNode<T>* z = new BSTNode<T>( key );

    while( x != &BSTNode<T>::NIL )
    {
        y = x;
        if ( aKey == *(x->key) )
            return false; // duplicate key - error
        if ( aKey < *(x->key) )
            x = x->left;
        else
            x = x->right;
    }

    if ( y == &BSTNode<T>::NIL )
        fRoot = z; // z is new root
    else
        if ( aKey < *(y->key) )
            y->left = z;
        else
            y->right = z;

    return true; // insert succeeded
}

Remove and Depth-first Traversal

bool remove( const T &key )
{
    return fRoot->remove( aKey, &BSTNode<T>::NIL );
}

void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const
{
    fRoot->traverseDepthFirst( aVisitor );
}

- The class BSTree defines an adapter for BSTNode.
- The operation insert can be defined as a simple while-loop over BSTNodes from the root node.
- The operations remove and traverseDepthFirst use recursion to explore BSTNodes. In the beginning, both start with the root node.
Other Tree Variants

- Rose Trees (directories)
- Expression Trees (internal program representation)
- Multi-rooted trees (C++: multiple inheritance)
- Red-Black Trees (directories in compound documents, java.util.TreeMap)
- AVL Trees (Adelson-Velskii & Landis balanced BTrees)

AVL vs. Red-Black Trees

- Both AVL trees and Red-Black trees are self-balancing binary search trees. However, the operations to balance the trees are different.
- AVL and Red-Black trees have different height limits. For a tree of size $n$:
  - An AVL tree's height is limited to $1.44 \log_2(n)$.
  - A Red-Black tree's height is limited to $2 \log_2(n)$.
- The AVL tree is more rigidly balanced than Red-Black trees, leading to slower insertion and removal but faster retrieval.
Algorithmic Patterns

Overview
- Algorithm Efficiency
- Solution Strategies

References

The “Best” Algorithm

- There are usually multiple algorithms to solve any particular problem.
- The notion of the “best” algorithm may depend on many different criteria:
  - Structure, composition, and readability
  - Time required to implement
  - Extensibility
  - Space requirements
  - Time requirements
**Time Analysis**

- **Example:**
  
  Algorithm A runs 2 minutes and algorithm B takes 1 minute and 45 seconds to complete for the same input.

- **Is B “better” than A? ➞ Not necessarily!**:
  
  - We have tested A and B only on one (fixed) input set. Another input set might result in a different runtime behavior.
  
  - Algorithm A might have been interrupted by another process.
  
  - Algorithm B might have been run on a different computer.

- A reasonable measurement should be machine-independent.

---

**Running Time in Terms of Input Size**

- What is the one of most interesting aspect about algorithms?
  
  - How many seconds does it take to complete for a particular input size n?
  
  - How does an increase of size n effect the running time?

- An algorithm requires
  
  - Constant time if the running time remains the same as the size n changes,
  
  - Linear time if the running time increases proportionally to size n.
  
  - Exponential time if the running time increases exponentially with respect to size n.
What is computable?

- Computation is usually modeled as a mapping from inputs to outputs, carried out by a "formal machine", or program, which processes its input in a sequence of steps.
- An "effectively computable" function is one that can be computed in a finite amount of time using finite resources.
- Church's Thesis: It is not possible to build a machine that is more powerful than a Turing machine.

The Ackermann Function

- The Ackermann function is a simple example of a computable function that grows much faster than polynomials or exponentials even for small inputs.
- The Ackermann function is defined recursively for non-negative integers \( m \) and \( n \) as follows:
  \[
  \begin{align*}
  A(0, n) &= n + 1 \\
  A(m+1, 0) &= A(m, 1) \\
  A(m+1, n+1) &= A(m, A(m+1, n))
  \end{align*}
  \]
The Big-O

- An algorithm is O(f(n)), read “has order f(n)”, if there exists constants C > 0 and integer k such that the algorithm requires at most C * f(n) steps for all input sizes n ≥ k.

- Example:
  - Algorithm A takes 4n + 3 steps, that is, it is O(n).
  - Choose C = 5 and k = 4, then 4n + 3 < 5n for all n ≥ 4.
Facts About Big-O

- Big-O focuses on growth rate as running time of input size approaches infinity \( n \to \infty \).
- Big-O does not say anything about the running time on small input.
- The function \( g(n) \) in \( O(g(n)) \) is a simple function for comparison of different algorithms:
  - \( 1, n, \log n, n \log n, n^2, \ldots \)

On Running Time Estimation

- Big-O ignore lower-order terms:
  - Lower order terms in the computation steps count functions that are concerned with initialization, secondary arguments, etc.
- Big-O does not consider the multiplier in higher order terms:
  - These terms are machine-dependent.
Performance Analysis

- **Best-Case (Lower Bound):**
  - The search for a given element A in an array of size \( n \) can be \( O(1) \), if the element is the first. (Also applies to binary search trees.)

- **Worst-Case (Upper Bound):**
  - The search for a given element A in an array of size \( n \) is \( O(n) \), if the element is the last in the array. (For binary search trees, it is also \( O(n) \), if the element is the last in a totally unbalanced binary search tree, but \( O(\log n) \) for a balanced search tree.)

- **Average-Case:**
  - The search for a given element A in an array of size \( n \) takes on average \( n/2 \), whereas the lookup in a binary search tree is \( O(\log n) \).

Constant Time

- Algorithm A requires 2,000,000 steps: \( O(1) \)

- As a young boy, the later mathematician Carl Friedrich Gauss was asked by his teacher to add up the first hundred numbers, in order to keep him quiet for a while. As we know today, this did not work out, since:

  \[
  \text{sum}(n) = \frac{n(n+1)}{2}
  \]

  which is \( O(1) \).
Polynomial Time

- $4n^2 + 3n + 1$:
  - Ignore lower-order terms: $4n^2$
  - Ignore constant coefficients: $O(n^2)$
- $a_kn^k + a_{k-1}n^{k-1} + \ldots + a_1n + a_0 = O(n^k)$

Logarithmic & Exponential Functions

- $\log_{10} n = \log_2 n / \log_2 10 = O(\log n)$:
  - Ignore base: $\log_{10} n$
- $345n^{4536} + n(\log n) + 2^n = O(2^n)$:
  - Ignore lower-order terms $345n^{4536}$ and $n(\log n)$
Ranking of Big-O

- Fastest:  
  - O(1)
  - O(log n)
  - O(n)
  - O(n log n)
  - O(n^2)
  - O(n^2 log n)
  - O(n^3)
  - O(2^n)

- Slowest:  
  - O(n!)

Algorithmic Patterns

- Direct solution strategies:
  - Brute force and greedy algorithms

- Backtracking strategies:
  - Simple backtracking and branch-and-bound algorithms

- Top-down solution strategies:
  - Divide-and-conquer algorithms

- Bottom-up solution strategies:
  - Dynamic programming

- Randomized strategies:
  - Monte Carlo algorithms
Brute-force Algorithms

- Brute-force algorithms are not distinguished by their structure.
- Brute-force algorithms are separated by their way of solving problems.
- A problem is viewed as a sequence of decisions to be made. Typically, brute-force algorithms solve problems by exhaustively enumerating all the possibilities.

Greedy Algorithms

- Greedy algorithms do not really explore all possibilities. They are optimized for a specific attribute.
- Example: Knapsack Problem
  - Profit - Maximal value of items
  - Weight - Maximal weight stored first
  - Density - Maximal profit per weight
- Greedy algorithms produce a feasible solution, but do not guarantee an optimal solution.
Sudoku

A hard puzzle:

```
 5 8 7 9
 1 3
 6 9 8
 4 6 3
 1 2 3 6
 5 1 7
 1 9 2
 2 4
 9 6 3 8
```

Sudoku: Solution Last Step

```
 5 1 3 8 4 6 7 9 2
 8 7 4 1 2 9 6 3 5
 2 6 9 3 7 5 8 1 4
 9 4 7 6 3 2 5 8 1
 1 5 2 7 9 8 3 4 6
 6 3 8 4 5 1 2 7 9
 4 8 1 5 6 7 9 2 3
 3 2 5 9 8 4 1 6 7
 7 9 6 2 1 3 4 5 8
```
Sudoku Solver: Greedy

1. Find $M[i,j]$ with the minimal number of choices.
2. Let $C$ be the possibilities for $M[i,j]$.
4. Solve puzzle recursively with new configuration.
5. If 4. succeeds, then report result and exit.
6. If 4. fails, then set $M[i,j]$ to next element if $C$. If there is no more element, then report error.
7. Continue with 4.
Backtracking Algorithms

- A backtracking algorithm systematically considers all possible outcomes for each decision.
- Backtracking algorithms are distinguished by the way in which the space of possible solutions is explored. Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary.
- Example: Knapsack Problem

Sudoku Solver: Backtracking

1. Find $M[i,j]$ with the minimal number of choices.
2. Let $C$ be the possibilities for $M[i,j]$.
4. Solve puzzle recursively with new configuration.
5. If 4. succeeds, then report result and exit.
6. If 4. fails, then set $M[i,j]$ to next element if $C$. If there is no more element, then report error.
7. Continue with 4.
Divide-and-Conquer

- Top-down algorithms use recursion to divide-and-conquer the problem space.
- This class of algorithms has the advantage that not all possibilities have to be explored.
- Example: Binary Search, Merge Sort, Quick Sort

Insert 8:

1. Insert 8:
   - 8 < 10
   - 8 < 25
   - 8 < 37

2. 2 steps required!
Bottom-up

• Bottom-up algorithms employ dynamic programming.
• Bottom-up algorithms solve a problem by solving a series of subproblems.
• These subproblems are carefully devised in such a way that each subsequent solution is obtained by combining the solutions to one or more of the subproblems that have already been solved.
• Example: Parsing, Pretty-Printing

Randomized Algorithms

• Randomized algorithms behave randomly.
• Randomized algorithms select elements in a random order to solve a given problem.
• Eventually, all possibilities are explored, but different runs can produce results faster or slower, if a solution exists.
• Example: Monte Carlo Methods, Simulation
This is only the beginning