Discrete Math and C++

R. J. Renka

Department of Computer Science & Engineering
University of North Texas

09/13/2012
1.2.1 Exponents

\[ x^A x^B = x^{A+B} \]

\[ x^A / x^B = x^{A-B} \]

\[ (x^A)^B = x^{AB} \]

\[ x^N + x^N = 2x^N \neq x^{2N} \]

\[ 2^N + 2^N = 2^{N+1} \]
Defn: \( x^A = B \) iff \( \log_x B = A \) for \( x > 0 \).

**Theorem 1.1:** \( \log_A B = \log_C B / \log_C A \) for \( A, B, C > 0, A \neq 1 \).

**proof:** Let \( x = \log_C B, y = \log_C A, z = \log_A B \). Then, by definition, \( C^x = B, C^y = A, \) and \( A^z = B \). Hence
\[ B = C^x = C^{yz} \Rightarrow x = yz \Rightarrow z = x / y. \]

**Example Application:** \( \log_2 x = \ln(x) / \ln(2) \).

**Theorem 1.2:** \( \log AB = \log A + \log B \) for \( A, B > 0 \).

**proof:** Let \( x = \log A, y = \log B, z = \log AB \). Then
\( A = 2^x, B = 2^y, \) and \( AB = 2^z \). Hence
\[ 2^z = 2^x2^y = 2^{x+y} \Rightarrow z = x + y. \]

**Formulas:**
\[
\begin{align*}
\log A/B &= \log A - \log B \\
\log(A^B) &= B \log A \\
\log x &< x \quad \forall x > 0 \\
\log 1 &= 0, \log 2 = 1, \log 1024 = 10, \log 1048576 = 20
\end{align*}
\]
1.2.3 Series

Geometric series:

\[
\sum_{i=0}^{N} 2^i = 2^{N+1} - 1
\]

\[
\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1} \quad \text{for } A \neq 1
\]

\[
= \frac{1 - A^{N+1}}{1 - A} < \frac{1}{1 - A} \quad \text{if } 0 < A < 1
\]

\[
\rightarrow \frac{1}{1 - A} \quad \text{as } N \rightarrow \infty
\]

**proof:** \( S = \sum_{i=0}^{N} A^i = 1 + A + A^2 + \ldots + A^N \Rightarrow \)

\( AS = A + A^2 + \ldots + A^{N+1} \Rightarrow S - AS = 1 - A^{N+1} \Rightarrow \)

\( S = \frac{(1 - A^{N+1})}{(1 - A)} \quad \text{for } A \neq 1. \) \( \square \)
Problem:

\[ S = \sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \cdots \]

Solution:

\[ 2S = 1 + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} + \cdots \Rightarrow \]

\[ S = 2S - S = 1 + \frac{2 - 1}{2^1} + \frac{3 - 2}{2^2} + \frac{4 - 3}{2^3} + \cdots \]

\[ = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots = \sum_{i=0}^{\infty} A^i \text{ for } A = \frac{1}{2} \]

\[ = \frac{1}{1 - 1/2} = 2. \]
Series continued

\[ \sum_{i=1}^{N} i = \frac{N(N+1)}{2} \]

\[ \sum_{k=1}^{N} (3k - 1) = 3N(N + 1)/2 - N \]

\[ \sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3} \]

\[ \sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{k+1} \text{ for } k \geq 0 \]
Harmonic sum:

\[ H_N = \sum_{i=1}^{N} \frac{1}{i} \approx \ln(N) \]

with error \( e \to \gamma \) as \( N \to \infty \), where \( \gamma = .57721566 \) is Euler’s constant.

\[ \sum_{i=1}^{N} f(N) = Nf(N) \]

\[ \sum_{i=k}^{N} f(i) = \sum_{i=1}^{N} f(i) - \sum_{i=1}^{k-1} f(i) \]
1.2.4 Modular (Modulo) Arithmetic (Gauss, 1801)

**Congruence Relation**: $A \equiv B \pmod{N}$ iff $A - B$ is an integer multiple of $N$ ($A$ is congruent to $B$ modulo $N$); i.e., $A$ and $B$ have the same remainder when divided by $N$. Since there are $N$ remainders ($0, \ldots, N-1$), there are $N$ residue classes.

\[
A_1 \equiv B_1 \pmod{N} \text{ and } A_2 \equiv B_2 \pmod{N} \implies \\
1. (A_1 + A_2) \equiv (B_1 + B_2) \pmod{N} \\
2. (A_1 - A_2) \equiv (B_1 - B_2) \pmod{N} \\
3. (A_1A_2) \equiv (B_1B_2) \pmod{N}
\]
1.2.5 Proof by Induction, Counterexample, and Contradiction

Fibonacci sequence:

\[ F_0 = 1, \quad F_1 = 1, \quad F_{n+2} = F_n + F_{n+1} \text{ for } n \geq 0 \]

Theorem: \( F_i < (5/3)^i \) for \( i > 0 \).

proof by induction:

base: \( F_1 = 1 < 5/3 \) and \( F_2 = 2 < (5/3)^2 = 25/9 \).

inductive hypothesis: Suppose \( F_i < (5/3)^i \) for \( i = 1, \ldots, k \), \( k \geq 2 \). Then

\[
F_{k+1} = F_{k-1} + F_k < (5/3)^{k-1} + (5/3)^k = (5/3)^{-2}(5/3)^{k+1} + (5/3)^{-1}(5/3)^{k+1} = (9/25 + 3/5)(5/3)^{k+1} = (24/25)(5/3)^{k+1} < (5/3)^{k+1} \]

□
**Theorem:** $F_k > k^2$ for some $k$; i.e., it is not true that $F_k \leq k^2$ for all $k$.

**proof by counterexample:** $F_{11} = 144 > 11^2$  □

**Theorem:** There are infinitely many primes.

**proof by contradiction:** Suppose the primes are ordered $p_1, p_2, \ldots, p_k$, where $p_k$ is the largest. Define $N = p_1 p_2 \ldots p_k + 1$. Then none of $p_1, \ldots, p_k$ divides $N$ because the remainder is 1. Hence, since $N > p_k$, $N$ is not prime and not divisible by a prime. This contradicts the Fundamental Theorem of Arithmetic.  □
Example 1: Implement $f(0) = 0$, $f(x)=2f(x-1)+x^2$ for integer $x > 0$.

```c
int f(int x)
{
    if (x == 0)
        return 0;
    else
        return 2*f(x-1) + x*x;
}
```

What about $f(-1)$?
Example 2: Implement \( f(0) = 0, f(n) = f(n/3+1) + n-1 \) for \( n > 0 \).

```cpp
int bad(int n)
{
    if (n == 0)
        return 0;
    else
        return bad(n/3 + 1) + n-1;
}
```

bad(0) is the only valid call.
Example 3: Print a nonnegative decimal integer using function printDigit.

```c
void printOut(int n) // Print nonnegative integer n
{
    if (n >= 10)
        printOut(n/10);
    printDigit(n % 10);
}
```

proof of correctness: Induction on the number of digits is nearly identical to a description of the algorithm.
Recursion is not efficient but when used properly can simplify algorithm design and lead to clean elegant code. The rules are as follows.

1. Base case must be included
2. Recursive call must make progress toward base case
3. Design rule: assume that all recursive calls work (induction)
4. Compound interest rule: never duplicate work by solving the same instance of a problem in different recursive calls
```cpp
/**
 * A class for simulating an integer memory cell.
 */

class IntCell
{
  public:
    /**
     * Construct the IntCell.
     * Initial value is 0.
     */
    IntCell()
    { storedValue = 0; }

    /**
     * Construct the IntCell.
     * Initial value is initialValue.
     */
    IntCell( int initialValue )
    { storedValue = initialValue; }

    /**
     * Return the stored value.
     */
    int read()
    { return storedValue; }

    /**
     * Change the stored value to x.
     */
    void write( int x )
    { storedValue = x; }

  private:
    int storedValue;
};
```

**Figure 1.5** A complete declaration of an IntCell class
1.4.1 Basic class syntax

- A **private** member may only be accessed by methods in its class.
- A **public** member may be accessed by any method in any class.
- The purpose of private members is *information hiding* — we can change the internal representation of data without affecting parts of the program that use the data.
- The default **visibility** is private.
- The example uses two **constructors**. If no constructor is explicitly defined, one that initializes the data members using language defaults is automatically generated.
Figure 1.6

```cpp
1    /**
2     * A class for simulating an integer memory cell.
3     */
4    class IntCell
5    {
6        public:
7            explicit IntCell( int initialValue = 0 )
8                : storedValue( initialValue ) { }
9            int read( ) const
10                { return storedValue; }
11            void write( int x )
12                { storedValue = x; }
13    
14        private:
15            int storedValue;
16    }
```

Figure 1.6 IntCell class with revisions
1.4.2 Extra Constructor Syntax and Accessors

1. A default parameter (which can be used in any function) implies an optional parameter, and that therefore a zero-parameter constructor is defined.

2. The initializer list (line 8) stores data member values when the object is created. This is required if the data member is const (cannot be changed after the object has been constructed) or is itself a class type that does not have a zero-parameter constructor.

3. An explicit constructor tells the compiler not to allow implicit type conversion (which destroys strong typing) with a one-parameter constructor used to generate an implicit temporary as in the example:

   ```cpp
   IntCell obj; obj = 37;    // IntCell temp=37; obj = temp;
   ```

4. By default all member functions are mutators and can change the state of their object; accessor have 'const' after ')' as part of the signature.
```cpp
#include "IntCell_H"

#define IntCell_H

/**
 * A class for simulating an integer memory cell.
 */

class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 );
    int read() const;
    void write( int x );

  private:
    int storedValue;

};

#endif
```

**Figure 1.7** IntCell class interface in file IntCell.h
```cpp
#include "IntCell.h"

/**
 * Construct the IntCell with initialValue
 */
IntCell::IntCell( int initialValue ) : storedValue( initialValue )
{
}

/**
 * Return the stored value.
 */
int IntCell::read( ) const
{
    return storedValue;
}

/**
 * Store x.
 */
void IntCell::write( int x )
{
    storedValue = x;
}
```

**Figure 1.8** IntCell class implementation in file IntCell.cpp
Figure 1.9

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

int main() {
    IntCell m; // Or, IntCell m(0); but not IntCell m();
    m.write(5);
    cout << "Cell contents: " << m.read() << endl;
    return 0;
}
```

**Figure 1.9** Program that uses IntCell in file TestIntCell.cpp
Interface (.h file): lists class members, both data and functions

Implementation (.cpp file): implements functions

Problem: Complicated projects have files that include other files, and it is illegal to include the same interface twice.

Solution: Each header file uses the preprocessor to define a unique symbol when the class interface is read:

    #ifndef IntCell_H
    #define IntCell_H

    . . .
    #endif

The scoping operator :: is needed in the implementation file to identify the class of each member function. (Otherwise it is in global scope.)
The **signatures** in the class interface and implementation must match. This includes `\texttt{\texttt{const}}` for an accessor, but not default parameters.

**Object Declaration**

```cpp
IntCell obj1;       // Zero-parameter constructor  
IntCell obj2(12);  // One-parameter constructor  
IntCell obj3=37;    // Illegal because constructor is explicit  
IntCell obj4();    // Declares a function with no parameters (defined elsewhere)
```
```cpp
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> squares( 100 );

    for( int i = 0; i < squares.size( ); i++ )
        squares[ i ] = i * i;

    for( int i = 0; i < squares.size( ); i++ )
        cout << i << " " << squares[ i ] << endl;

    return 0;
}
```

Figure 1.10 Using the vector class; stores 100 squares and outputs them
Built-in arrays cannot be copied with $=$, cannot be queried for capacity, and do not test for valid indices. They are not *first-class objects*. The built-in string is an array of characters, and has the liabilities of arrays; e.g., $==$ does not correctly compare strings.

The **vector** is a C++ class intended to replace the array.

The STL (Standard Template Library) vector and string classes create first-class objects. A vector has a size; string objects can be compared with $==$, $<$, etc. Both can be copied with $=$.
```cpp
int main( )
{
    IntCell *m;

    m = new IntCell( 0 );
    m->write( 5 );
    cout << "Cell contents: " << m->read( ) << endl;

    delete m;

    return 0;
}
```

**Figure 1.11** Program that uses pointers to IntCell
Pointers are variables containing addresses.

Dynamic memory allocation (new, delete) uses pointers.

The compiler may not check that a pointer has been assigned a value before being used.

Note that both m = new IntCell(); and m = new IntCell; are legal and use the zero-parameter constructor.

C++ has no garbage collection; failure to delete a dynamically allocated object before the pointer goes out of scope (on return from the function that did the allocation, for example) results in a memory leak.
The following function returns the average of the first $n$ integers in $\text{arr}$, and sets $\text{errorflag}$ to true if $n$ is larger than $\text{arr.size()}$ or smaller than 1.

\[
\text{double avg(const vector<int> & arr, int n, bool & errorflag);}
\]

- $\text{arr}$ uses call by constant reference
- $n$ uses call by value
- $\text{errorflag}$ uses call by reference
Two versions to find the maximum string; only the first is correct.

```c++
const string & findMax( const vector<string> & arr )
{
    int maxIndex = 0;

    for( int i = 1; i < arr.size( ); i++ )
        if( arr[ maxIndex ] < arr[ i ] )
            maxIndex = i;

    return arr[ maxIndex ];
}

const string & findMaxWrong( const vector<string> & arr )
{
    string maxValue = arr[ 0 ];

    for( int i = 1; i < arr.size( ); i++ )
        if( maxValue < arr[ i ] )
            maxValue = arr[ i ];

    return maxValue;
}
```
1.5.3 Return Passing
Return by *reference* can avoid a copy, but it is necessary that the returned object, unlike local variables, exist after the return. Use of constant reference prevents the returned object from being altered after the call. It cannot therefore appear on the left side of an assignment statement.

1.5.4 Reference Variables
Reference variables can be used as local variables and class data members as well as for parameter passing.

1. Avoid a copy by using a constant reference as a return value:
   ```cpp
   const string & x = findMax(arr);
   ```

2. Rename an object that would otherwise require a complicated expression:
   ```cpp
   list<T> & whichlist = theLists[hash(x, theLists.size())];
   ```
These three functions are automatically created for every class.

- **Destructor** is called when an object goes out of scope or is deleted. It should free up resources by calling delete for each new, close files, etc.

- **Copy constructor** is called by a
  1. declaration with initialization: IntCell B=C; or IntCell B(C);, but not B=C;
  2. an object passed using call by value
  3. an object returned by value

- **operator=** (copy assignment operator): lhs = rhs, where both objects already exist

In all three cases, the implementation applies the operator to each data member in turn.

**Problems with Defaults:** If a data member is a pointer, the defaults don’t work: it is not deleted by the destructor, and only shallow copies are made by the other operators.
Figure 1.13

1    IntCell::~IntCell() 
2    { 
3        // Does nothing, since IntCell contains only an int data 
4        // member. If IntCell contained any class objects, their 
5        // destructors would be called. 
6    } 
7
8    IntCell::IntCell( const IntCell & rhs ) : storedValue( rhs.storedValue ) 
9    { 
10   }
11
12   const IntCell & IntCell::operator=( const IntCell & rhs ) 
13   { 
14       if( this != &rhs )  // Standard alias test 
15           storedValue = rhs.storedValue; 
16       return *this; 
17   }

**Figure 1.13** The defaults for the big three
class IntCell {
    public:
        explicit IntCell( int initialValue = 0 )
        { storedValue = new int( initialValue ); }

        int read() const
        { return *storedValue; }

        void write( int x )
        { *storedValue = x; }

    private:
        int *storedValue;
    };
```cpp
1   int f( )
2   {
3       IntCell a( 2 );
4       IntCell b = a;
5       IntCell c;
6
7       c = b;
8       a.write( 4 );
9       cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;
10      return 0;
11   }
```

**Figure 1.15** Simple function that exposes problems in Figure 1.14


```cpp
class IntCell
{
    public:
        explicit IntCell( int initialValue = 0 );

        IntCell( const IntCell & rhs );
        ~IntCell();
        const IntCell & operator=( const IntCell & rhs );

        int read() const;
        void write( int x );

    private:
        int *storedValue;
    }

    IntCell::IntCell( int initialValue )
    {
        storedValue = new int( initialValue );
    }

    IntCell::IntCell( const IntCell & rhs )
    {
        storedValue = new int( *rhs.storedValue );
    }

    IntCell::~IntCell()
    {
        delete storedValue;
    }

    const IntCell & IntCell::operator=( const IntCell & rhs )
    {
        if( this != &rhs )
            *storedValue = *rhs.storedValue;
        return *this;
    }

    int IntCell::read() const
    {
        return *storedValue;
    }

    void IntCell::write( int x )
    {
        *storedValue = x;
    }
```

**Figure 1.16** Data member is a pointer; big three needs to be written
1.6 Templates

Templates provide the following:

- a powerful *software re-use* feature of C++
- *generic programming*
- *type-independent algorithms* such as sorting and searching

1.6.1 Function Templates

- Specify a range of *overloaded functions*
- Are not actually functions but are used to generate function-template specializations (expansions; instantiations) as needed, each adding to the memory requirements
- Are like macros but enable full type checking
```cpp
/**
 * Return the maximum item in array a.
 * Assumes a.size() > 0.
 * Comparable objects must provide operator< and operator=
 */

template <typename Comparable>
const Comparable & findMax( const vector<Comparable> & a )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size(); i++ )
        if( a[ maxIndex ] < a[ i ] )
            maxIndex = i;
    return a[ maxIndex ];
}
```

**Figure 1.17** findMax function template
```cpp
int main() {
    vector<int> v1(37);
    vector<double> v2(40);
    vector<string> v3(80);
    vector<IntCell> v4(75);

    // Additional code to fill in the vectors not shown

    cout << findMax(v1) << endl; // OK: Comparable = int
    cout << findMax(v2) << endl; // OK: Comparable = double
    cout << findMax(v3) << endl; // OK: Comparable = string
    cout << findMax(v4) << endl; // Illegal; operator< undefined

    return 0;
}
```

**Figure 1.18** Using `findMax` function template
Class templates are parameterized types.

Most C++ compilers require the template to be completely defined in the source code file that uses it. The header file that declares the template must therefore include the implementation of member functions and must be included in the source file. Popular implementations of the STL follow this strategy; the interface and implementation are not separated.
```cpp
/*
 * A class for simulating a memory cell.
 */

template <typename Object>
class MemoryCell
{
    public:
        explicit MemoryCell(const Object & initial\Value = Object( ) )
            : storedValue( initial\Value ) { }
        const Object & read( ) const
            { return storedValue; }
        void write( const Object & x )
            { storedValue = x; }
    private:
        Object storedValue;
};

Figure 1.19 MemoryCell class template without separation
```
Figure 1.20 Program that uses MemoryCell class template (Note the errors)

```c++
int main()
{
    MemoryCell<int> m1;
    MemoryCell<string> m2; ("hello");

    m1.write(37);
    m2.write(m2.read() + "world");
    cout << m1.read() << endl << m2.read() << endl;

    return 0;
}
```
Comparable can be a class type, such as `Employee`: 

```cpp
class Employee {
  public:
    void setValue(const string & n, double s) {
      name = n; salary = s;
    }

    const string & getName() const
    return name;

    void print(ostream & out) const
    { out << name << " (* <= salary <= *)"; }

    bool operator< (const Employee & rhs) const
    { return salary < rhs.salary; }

    // Other general accessors and mutators, not shown
    private:
    string name;
    double salary;
};

// Define an output operator for Employee
ostream & operator<<(ostream & out, const Employee & rhs) {
  rhs.print(out);
  return out;
}

int main() {
  vector<Employee> v(3);

  v[0].setValue("George Bush", 4000000.00);
  v[1].setValue("Bill Gates", 2000000000.00);
  v[2].setValue("Dr. Phil", 13000000.00);
  cout << findMax(v) << endl;
  return 0;
}
```

**Figure 1.21**
Figure 1.22

```cpp
1 // Generic findMax, with a function object, Version #1.
2 // Precondition: a.size() > 0.
3 template <typename Object, typename Comparator>
4 const Object & findMax( const vector<Object> & arr, Comparator cmp )
5 {
6    int maxIndex = 0;
7
8    for( int i = 1; i < arr.size(); i++ )
9        if( cmp.isLessThan( arr[ maxIndex ], arr[ i ] ) )
10           maxIndex = i;
11
12    return arr[ maxIndex ];
13 }
14
15 class CaseInsensitiveCompare
16 {
17    public:
18        bool isLessThan( const string & lhs, const string & rhs ) const
19            { return strcmp( lhs.c_str(), rhs.c_str() ) < 0; }
20    }
21
22 int main( )
23 {
24    vector<string> arr( 3 );
26    cout << findMax( arr, CaseInsensitiveCompare( ) ) << endl;
27
28    return 0;
29 }
```

Figure 1.22 Simplest idea of using a function object as a second parameter to findMax; output is ZEBRA
Figure 1.23 Using a function object C++ style, with a second version of findMax; output is ZEBRA then crocodile
1.6.3, 4 Object, Comparable, and Function Objects

Note the idiom for providing an output function in Figure 1.21: member function `print` and non-class (global) function `operator<<`. This is necessary because `Employee` is not a primitive and therefore not recognized by `<<`.

1.6.4 Function Objects

- Instances of classes with no data and one member function
- Used to pass functions as parameters
```cpp
#ifndef MATRIX_H
#define MATRIX_H

#include <vector>
using namespace std;

template<typename Object>
class matrix
{
  public:
    matrix(int rows, int cols) : array(rows)
    {
      for(int i = 0; i < rows; i++)
        array[i].resize(cols);
    }

    const vector<Object> & operator[](int row) const
    { return array[row]; }
    vector<Object> & operator[](int row)
    { return array[row]; }

    int numrows() const
    { return array.size(); }
    int numcols() const
    { return numrows() ? array[0].size() : 0; }

  private:
    vector<vector<Object>> array;
};

#endif

Figure 1.24 A complete matrix class
1.7 Using Matrices

A matrix is a vector of vectors of Objects.

1.7.1 The Data Members, Constructor, and Basic Accessors
The constructor creates array with rows zero-length vectors. Then the body of the constructor resizes the rows to have cols columns.

1.7.2 operator[]
- m[i] is the vector corresponding to row i of matrix m
- m[i][j] is row i, column j
- vector<Object> is returned by reference for copy-to, and by constant reference for copy-from
- The constant version is made an accessor (with a different signature) in order to allow two functions with the same name