

# Recursion in C++

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

```
#include <iomanip.h>
```

```
int factorial (int n) {
```

```
    if (n <= 1) return 1;           // base case
```

```
    else return n * factorial(n-1); // recursive case  
}
```

```
main() {
```

```
    for (i=0; i<=10; i++)
```

```
        cout << setw(2) << i << "! = " << factorial(i) << endl;  
}
```

# Function Overloading

- Different versions must have different argument types.
- The **return** value is not considered when determining which function to call, only the **arguments** are taken into account.

# Function Overloading (2)

*double sqrt(double i);*

*// called with sqrt(25.0);*

*int sqrt(int i);*                      *// called with sqrt(16);*

*double sqrt(int i);*              *// illegal*

*int f(int x = 0);*

*int f(void);*

*// what if the call f()?*

# Function Overloading (3)

***void g(int x);*** // case a

***void g(double x, int y = 0);*** // case b

***g(2.0);*** // case b

***g(0);*** // case a

***g(0, 0);*** // case b

***g('c');*** // ambiguous

# Function Overloading (4)

***`h(double x, int y);`***

***`h(int x, double y);`***

// what if the call ***`h(10, 10);`***

# Function Template

```
int mul(int a, int b) { return a*b; }
```

```
double mul(double a, double b) { return a*b; }
```

```
template <class Type>
```

```
Type mul(Type a, Type b) { return a*b; }
```

```
int a = mul(2, 3);
```

```
double b = mul(5.0, 10.0);
```

# Array Size

- The size must be a constant.
  - a. must be known at **compile** time.
  - b. certainly not changed later.
  - c. **cannot** free up once not needed.
- Does not know its own size, needs to carry around ***additional variable*** with its size.

# Array Initialization

- Zero out (or initialized to default value).
- Prototype to initialize with "normal" array:

***int n[3] = {1, 2, 3};***

***int n[3] = {1, 2};***

***int grades [] = {15, 23, 44, 78, 92};***

***int num\_students = sizeof(grades) /  
sizeof(int);***

- Cannot be assigned as a **whole**, for example, *array1 = array2;*



# Character Array

- Define string literal.
- *char s[] = "test";*

# Array Access

- Index (or subscript) starts with 0, ends with (size - 1).
- No range checking, will be a **run-time** error.

# Array Parameter

- *int arr[20];*  
*void f(int a[]);*  
    *// the call f(arr) is implemented as*  
    *// call-by-reference.*
- *f (int val, int &ref) { val++; ref++; }*  
*g() {*  
    *int i = 1,*  
    *j = 1;*  
    *f(i, j);*  
    *cout << i << j << endl;     // 1 2*  
*}*

# Multidimensional Array

```
int ia[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

```
int a[5][10];      // &a[1] - &a[0] = 40 (10 integers)
```

```
int a1[][10], **a2;
```

```
typedef int A10[10];
```

```
A10 *b = a; b++;      // b = &a[1]
```

# Pointer Type

***int \*p;***

***p = &x;***    // the address of object x

***p = 0;***

      // assignment of the special value 0

***p = (int \*) 2203***

      // an absolute address in memory

# Addressing and Dereferencing

```
int a = 2, b;
```

```
int *p = &a;
```

```
// pointer initialization to the address of a
```

```
b = p;
```

```
// illegal, pointer is not convertible to integer
```

```
b = *p + 1;      // okay
```

```
p = &b;          // p points to b
```

# Simulating Call-by-Reference

- Declare a function parameter to be a pointer.
- Use the dereferenced pointer in the function body.
- Pass an address as an argument when the function is called.

# Simulating Call-by-Reference (cont)

```
swap (int *p, int *q)  
{  
    int temp;  
  
    if (*p > *q) {  
        temp = *p;  
        *p = *q;  
        *q = temp;  
    }  
}
```

```
int i, j;  
swap(&i, &j);
```



# Reference Declarations

***int i = 5;***

*// i is located in memory with value 5*

***int \*p = &i;***

*// p is located in memory with value &i*

***int &r = i;***      *// r and i are the same object*

***int \*&s = p;***

*// s and p are the same object*

# Passing a Reference

```
swap (int &p, int &q)  
{  
    int temp;  
  
    if (p > q) {  
        temp = p;  
        p = q;  
        q = temp;  
    }  
}
```

```
int i, j;  
swap(i, j);
```

# Arrays vs Pointers

```
int a[100], *p;
```

*// p = a; and p = &a[0]; are equivalent*

```
a = p;    // illegal
```

```
++a;      // illegal
```

```
a += 2;   // illegal
```

# Passing Arrays to Functions

```
int sum(int a[], int size) {  
    int i, s = 0;  
    for (i = 0; i < size; i++) s += a[i];  
    return s;  
}
```

```
int v[100];  
sum(v, 100);
```

# Arrays in Java

- ***int [] data;*** // array declaration
- The variable declaration does **not** create the array, this must be done using the keyword ***new***, and at this point the size of the array is set.

***data = new int[100];***  
// a 100 element array

# Arrays in Java (cont)

- Both steps can be done at the same time if the array size is known at that point in the program.

***int [] data = new int[100];***

# Java Collections Framework

- Collection:
  - Object that groups multiple **elements** into one unit.
  - Also called **container**.
- Collection **framework** consists of:
  - Interfaces
    - Abstract data type
  - Implementations
    - Reusable data structures
  - Algorithms
    - Reusable functionality

# Collections Framework (cont)

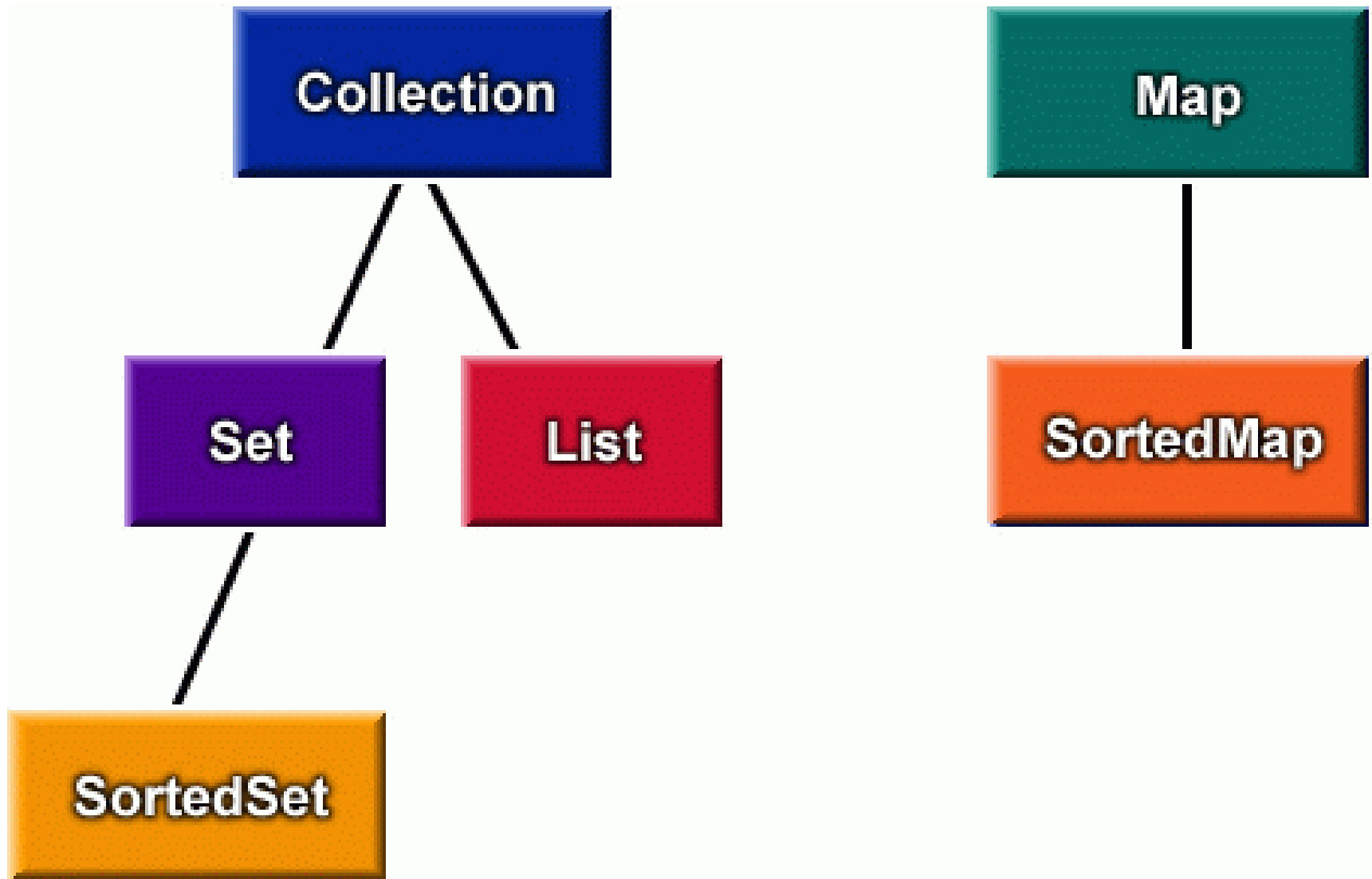
- Goals:
  - Reduce programming effort
  - Make APIs easier to learn
  - Make APIs easier to design and implement
  - Reuse software
  - Increase performance



# Core Collection Interfaces

- **Collection**
  - Group of elements
- **Set**
  - **No** duplicate elements
- **List**
  - **Ordered** collection
- **Map**
  - Maps keys to elements
- **SortedSet, SortedMap**
  - Sorted ordering of elements

# Core Collection Hierarchy



# Collections Interface Implementations

- General implementations:
  - Primary **public** implementation
  - Example
    - List – ArrayList, LinkedList
    - Set – TreeSet, HashSet
    - Map – TreeMap, HashMap
- Wrapper implementations:
  - Combined with other interfaces
  - Example
    - **synchronized**ArrayList, **unmodifiable**HashMap

# Collections Interface Methods

- boolean **add**(Object o)
  - Add specified element
- boolean **contains**(Object o)
  - True if collection contains specified element
- boolean **remove**(Object o)
  - Removes specified element from collection
- Boolean **equals**(Object o)
  - Compares object with collection for equality
- Iterator **iterator**()
  - Returns an iterator over the elements in collection

# Interface Methods (2)

- boolean **addAll**(Collection c)
  - Adds all elements in specified collection
- boolean **containsAll**(Collection c)
  - True if collection contains all elements in collection
- boolean **removeAll**(Collection c)
  - Removes all elements in specified collection
- boolean **retainAll**(Collection c)
  - Retains only elements contained in specified collection

# Interface Methods (3)

- void **clear()**
  - Removes all elements from collection
- boolean **isEmpty()**
  - True if collection contains no elements
- int **size()**
  - Returns number of elements in collection
- Object[] **toArray()**
  - Returns array containing all elements in collection

# Interface Methods (4)

- void **shuffle**(List list, Random rnd)
  - Randomly permute list using rnd
- void **sort**(List list, Comparator c)
  - Sorts list into ascending order
  - According Comparator ordering of elements

# Iterator Interface

- Iterator
  - Common interface for all Collection classes
  - Used to examine all elements in collection
- Properties
  - Order of elements is **unspecified** (may change)
  - Can remove current element during iteration
  - Works for any collection



# Iterator Interface (cont)

- **Interface**

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();           // optional, called once per next()  
}
```

- **Example usage**

```
Iterator i = myCollection.iterator();  
while (i.hasNext()) {  
    myCollectionElem x = (myCollectionElem) i.next();  
}
```

# Collection Classes in Java

- In addition to ***Arrays***, Java has **Vectors** and **Hashtables**.
- A ***Vector*** is a dynamically expandable/contractable collection of objects – these are referencable in a similar way to the elements of an array.

# Collection Classes (2)

- A ***Hashtable*** is a dynamically expandable collection of **key/value** pairs. Both key and value are objects.
- This allows **random access** to values by supplying the key (implemented by a hashing algorithm defined in the class object, but overridable).

# Collection Classes (3)

- As both collection classes accommodate only **objects**, int's, short's, double's etc must first be wrapped using a **wrapper** class.
- Vectors can easily be processed sequentially, but to do this with Hashtables it is necessary to employ an **Enumeration** object.

# Vectors

```
public class Garage {  
  
    private Vector fleet = new Vector();  
  
    public int addVehicle(Vehicle v) {  
        fleet.addElement(v);  
        return fleet.size() - 1;  
    } // end addVehicle
```

# Vectors (2)

```
public Vehicle getVehicle(int num) {  
    Vehicle v = (Vehicle)fleet.elementAt(num);  
    return v;  
} // end getVehicle
```

```
public void serviceFleet() {  
    for(int i = 0; i < fleet.size(); i++)  
        fleet.elementAt(i).service();  
} // end serviceFleet
```

```
} // end class Garage
```

# Vectors (3)

- To add items to vector use the method `addElement()`.
- To access items, use the method `elementAt()` – Vectors are indexed like arrays.
- The item returned from a Vector is assumed to be of class object and has to be cast to the correct type – called `DownCasting`.

# Vectors (4)

- The number of elements in a Vector is given by the method `size()`.
- Processing is similar to arrays although **Enumerations** can be used (by calling `elements()` method).



# Wrapper Classes

```
public class Grades {  
  
    private Vector marks = new Vector();  
  
    public void addScore(int grade) {  
        marks.addElement(new Integer(grade));  
    } // end addScore
```

# Wrapper Classes (2)

```
public int getScoreAt(int pos) {  
    Integer val;  
    val = (Integer)marks.elementAt(pos);  
    return val.intValue();  
} // end getScoreAt  
  
} // end class Grades
```

# Wrapper Classes (3)

- As Vectors and Hashtables can only contain **objects**, to store simple data types, such as int, long, double etc wrapper classes are needed.
- There are separate wrapper classes for all basic data types e.g. Byte, Short, Long, Float, Character, Boolean.

# Wrapper Classes (4)

- Each instance of a wrapper class contains a single attribute of the corresponding simple type.

# Hashtables

```
public class Garage {  
  
    private Hashtable fleet =  
        new Hashtable();  
  
    public void addVehicle(Vehicle v) {  
        fleet.put(v.getReg(), v);  
    } // end addVehicle
```

# Hashtables (2)

```
public Vehicle getVehicle(String reg) {  
    return (Vehicle)fleet.get(reg);  
} // end getVehicle
```

# Hashtables (3)

```
public void serviceFleet() {  
    Vehicle v;  
    Enumeration e = fleet.elements();  
    while (e.hasMoreElements()) {  
        v = (Vehicle)e.nextElement();  
        v.service();  
    } // end while  
} // end serviceFleet  
} // end class Garage
```

# Hashtables (4)

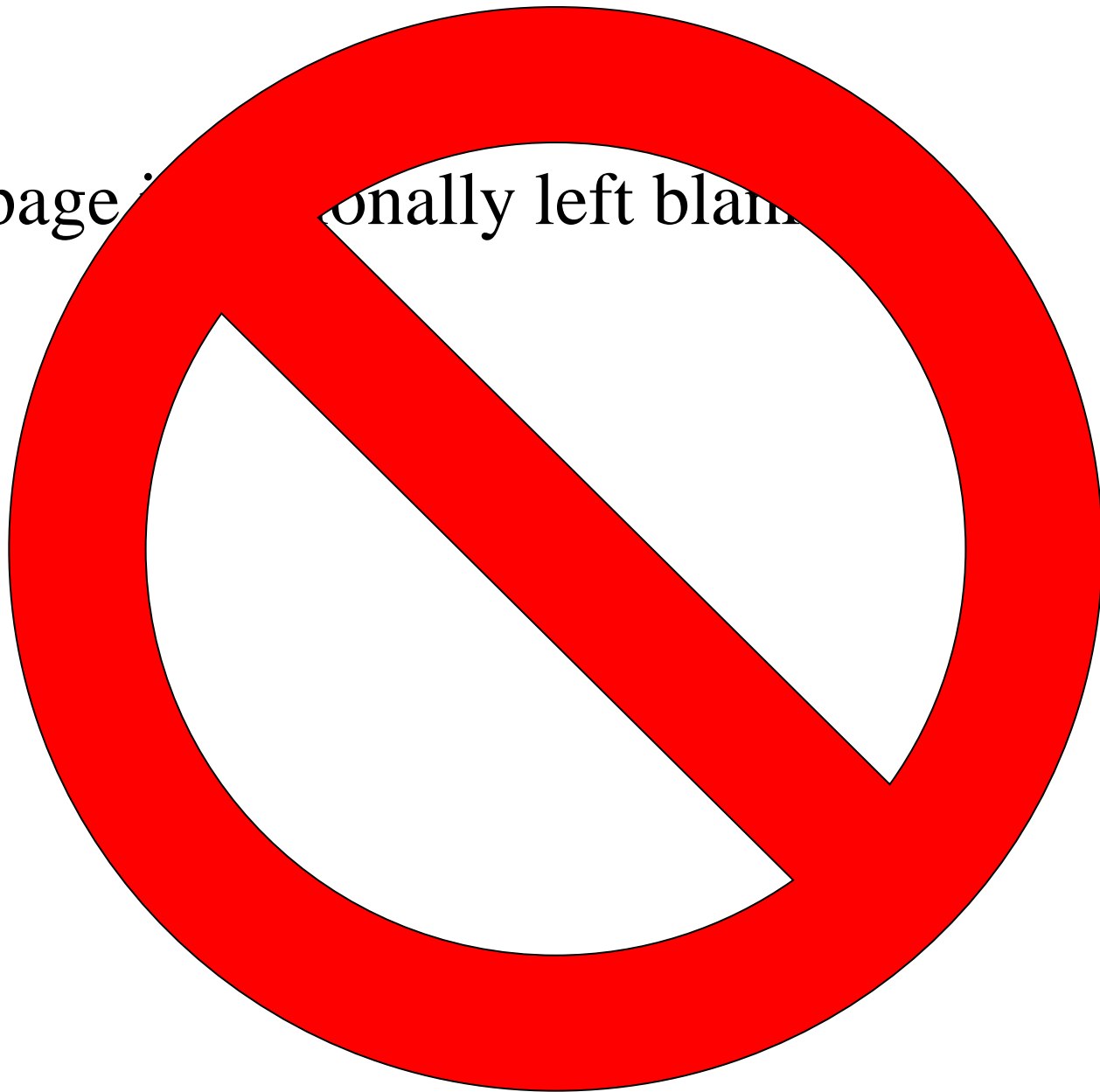
- The collection is keyed on registration number.
- Traversing all items in a **Hashtable** requires the use of an **Enumeration** instance.



# Hashtables (5)

- Enumerations operate like a file, starting at the 'first' item and working sequentially through until ***end-of-file***.
- Must obtain a new enumeration to repeat the process.
- An ***Enumeration*** makes **no** guarantees about the order in which the values are returned.

This page intentionally left blank



# Chapter 3 - Functions

## Outline

- 3.1 Introduction
- 3.2 Program Components in C++
- 3.3 Math Library Functions
- 3.4 Functions
- 3.5 Function Definitions
- 3.6 Function Prototypes
- 3.7 Header Files
- 3.8 Random Number Generation
- 3.9 Example: A Game of Chance and Introducing `enum`
- 3.10 Storage Classes
- 3.11 Scope Rules
- 3.12 Recursion
- 3.13 Example Using Recursion: The Fibonacci Series
- 3.14 Recursion vs. Iteration
- 3.15 Functions with Empty Parameter Lists



# Chapter 3 - Functions

## Outline

- 3.16      Inline Functions**
- 3.17      References and Reference Parameters**
- 3.18      Default Arguments**
- 3.19      Unary Scope Resolution Operator**
- 3.20      Function Overloading**
- 3.21      Function Templates**



## 3.3 Math Library Functions

- Perform common mathematical calculations
  - Include the header file `<cmath>`
- Example

```
cout << sqrt( 900.0 );
```

  - All functions in math library return a **double**



| Method              | Description  | Example   |
|---------------------|--|---|
| <b>ceil( x )</b>    | rounds $x$ to the smallest integer not less than $x$   | ceil( 9.2 ) is 10.0<br>ceil( -9.8 ) is -9.0                       |
| <b>cos( x )</b>     | trigonometric cosine of $x$ ( $x$ in radians)          | cos( 0.0 ) is 1.0   |
| <b>exp( x )</b>     | exponential function $ex$                              | exp( 1.0 ) is 2.71828<br>exp( 2.0 ) is 7.38906                    |
| <b>fabs( x )</b>    | absolute value of $x$                                  | fabs( 5.1 ) is 5.1<br>fabs( 0.0 ) is 0.0<br>fabs( -8.76 ) is 8.76 |
| <b>floor( x )</b>   | rounds $x$ to the largest integer not greater than $x$ | floor( 9.2 ) is 9.0<br>floor( -9.8 ) is -10.0                     |
| <b>fmod( x, y )</b> | remainder of $x/y$ as a floating-point number          | fmod( 13.657, 2.333 ) is 1.992                                    |
| <b>log( x )</b>     | natural logarithm of $x$ (base $e$ )                   | log( 2.718282 ) is 1.0<br>log( 7.389056 ) is 2.0                  |
| <b>log10( x )</b>   | logarithm of $x$ (base 10)                             | log10( 10.0 ) is 1.0<br>log10( 100.0 ) is 2.0                     |
| <b>pow( x, y )</b>  | $x$ raised to power $y$ ( $xy$ )                       | pow( 2, 7 ) is 128<br>pow( 9, .5 ) is 3                           |
| <b>sin( x )</b>     | trigonometric sine of $x$ ( $x$ in radians)            | sin( 0.0 ) is 0   |
| <b>sqrt( x )</b>    | square root of $x$                                     | sqrt( 900.0 ) is 30.0<br>sqrt( 9.0 ) is 3.0                       |
| <b>tan( x )</b>     | trigonometric tangent of $x$ ( $x$ in radians)         | tan( 0.0 ) is 0   |

Fig. 3.2 Math library functions.



## 3.4 Functions

- Functions
  - Modularize a program
  - Software **reusability**
    - Call function multiple times
- Local variables
  - Known only in the function in which they are defined
  - All variables declared in function definitions are local variables
- Parameters
  - **Local** variables passed to function when called
  - Provide outside information



## 3.5 Function Definitions

- Format for function definition

```
return-value-type function-name( parameter-list )  
{  
    declarations and statements  
}
```

- Parameter list
  - If no arguments, use **void** or leave blank
- Return-value-type
  - Data type of result returned (use **void** if nothing returned)





## 3.5 Function Definitions

- Example function

```
int square( int y )  
{  
    return y * y;  
}
```

- **return** keyword

- Returns data, and control goes to function's caller
  - If no data to return, use **return;**
- Function ends when reaches right brace
  - Control goes to caller



## 3.6 Function Prototypes

- Function prototype
  - Only needed if function definition **after** function call
- Prototype must match function definition
  - Function prototype

```
double maximum( double, double, double );
```
  - Definition

```
double maximum( double x, double y, double z )  
{  
    ...  
}
```



## 3.8 Random Number Generation

- **rand** function (**<cstdlib>**)
  - **i = rand( );**
  - Generates unsigned integer between 0 and RAND\_MAX (usually 32767)
- Scaling and shifting
  - Modulus (remainder) operator: %
    - **10 % 3** is **1**
    - **x % y** is between **0** and **y - 1**
  - Example
    - i = rand( ) % 6 + 1;**
    - **"Rand( ) % 6"** generates a number between **0** and **5** (scaling)
    - **" + 1"** makes the range **1** to **6** (shift)



## 3.8 Random Number Generation

- Calling `rand()` repeatedly
  - Gives the same sequence of numbers
- **Pseudorandom** numbers
  - Preset sequence of "random" numbers
  - Same sequence generated whenever program run
- To get different random sequences
  - Provide a seed value
    - Like a random starting point in the sequence
    - The same seed will give the same sequence
  - **`srand(seed);`**
    - **`<cstdlib>`**
    - Used **before** **`rand()`** to set the seed



## 3.8 Random Number Generation

- Can use the current time to set the seed
  - No need to explicitly set seed every time
  - `srand( time( 0 ) );`
  - `time( 0 );`
    - `<ctime>`
    - Returns current time in *seconds*
- General shifting and scaling
  - $Number = shiftingValue + rand( ) \% scalingFactor$
  - `shiftingValue` = first number in desired range
  - `scalingFactor` = width of desired range



## 3.9 Example: Game of Chance and Introducing enum

- Enumeration

- Set of integers with identifiers

**enum** *typeName* { *constant1*, *constant2*... } ;

- Constants start at 0 (default), incremented by 1

- Constants need **unique** names

- Cannot assign integer to enumeration variable

- Must use a previously defined enumeration type

- Example

```
enum Status {CONTINUE, WON, LOST};
```

```
Status enumVar;
```

```
enumVar = WON; // cannot do enumVar = 1
```



## 3.9 Example: Game of Chance and Introducing enum

- Enumeration constants can have preset values

```
enum Months { JAN = 1, FEB, MAR, APR, MAY,  
             JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

- Starts at 1, increments by 1



## 3.10 Storage Classes

- Variables have attributes
  - Have seen name, type, size, value
  - Storage class
    - How long variable exists in memory
  - Scope
    - Where variable can be referenced in program
  - Linkage
    - For multiple-file program, which files can use it





## 3.13 Example Using Recursion: Fibonacci Series

- Order of operations
  - `return fibonacci( n - 1 ) + fibonacci( n - 2 );`
- Do **not** know which one executed first
  - C++ does not specify
  - Only **&&**, **||** and **? :** guaranteed **left-to-right** evaluation
- Recursive function calls
  - Each level of recursion doubles the number of function calls
    - 30<sup>th</sup> number =  $2^{30} \sim 4$  billion function calls
  - Exponential complexity



## 3.14 Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between **performance** (iteration) and **good software engineering** (recursion)



## 3.15 Functions with Empty Parameter Lists

- Empty parameter lists
  - **void** or leave parameter list empty
  - Indicates function takes no arguments
  - Function **print** takes no arguments and returns no value
    - `void print();`
    - `void print( void );`



## 3.16 Inline Functions

- Inline functions
  - Keyword **inline** before function
  - Asks the compiler to copy code into program instead of making function call
    - Reduce function-call overhead
    - Compiler can **ignore inline**
  - Good for small, often-used functions

- Example

```
inline double cube( const double s )  
    { return s * s * s; }
```

- **const** tells compiler that function does not modify **s**



## 3.17 References and Reference Parameters

- Call by value
  - Copy of data passed to function
  - Changes to copy do not change original
  - Prevent unwanted side effects
- Call by reference
  - Function can directly access data
  - Changes affect original



## 3.17 References and Reference Parameters

- Pointers
  - Another way to pass-by-reference
- References as aliases to other variables
  - Refer to same variable
  - Can be used within a function

```
int count = 1; // declare integer variable count
Int &cRef = count; // create cRef as an alias for count
++cRef; // increment count (using its alias)
```

- References must be **initialized** when declared
  - Otherwise, compiler error
  - Dangling reference
    - Reference to undefined variable





**fig03\_22.cpp**  
(1 of 1)

**fig03\_22.cpp**  
output (1 of 1)

```

1  // Fig. 3.22: fig03_22.cpp
2  // References must be initialized.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10     int x = 3;
11     int &y;           // Error: y must be initialized
12
13     cout << "x = " << x << endl << "y = " << y << endl;
14     y = 7;
15     cout << "x = " << x << endl << "y = " << y << endl;
16
17     return 0;  // indicates successful termination
18
19 } // end main

```

Uninitialized reference –  
compiler error.

*Borland C++ command-line compiler error message:*

Error E2304 Fig03\_22.cpp 11: Reference variable 'y' must be  
initialized- in function main()

*Microsoft Visual C++ compiler error message:*

D:\cpphttp4\_examples\ch03\Fig03\_22.cpp(11) : error C2530: 'y' :  
references must be initialized

## 3.18 Default Arguments

- Function call with omitted parameters
  - If not enough parameters, rightmost go to their defaults
  - Default values
    - Can be constants, global variables, or [function calls](#)

- Set defaults in function prototype

```
int myFunction( int x = 1, int y = 2, int z = 3 );
```

- **myFunction(3)**
  - **x = 3**, **y** and **z** get defaults (rightmost)
- **myFunction(3, 5)**
  - **x = 3**, **y = 5** and **z** gets default





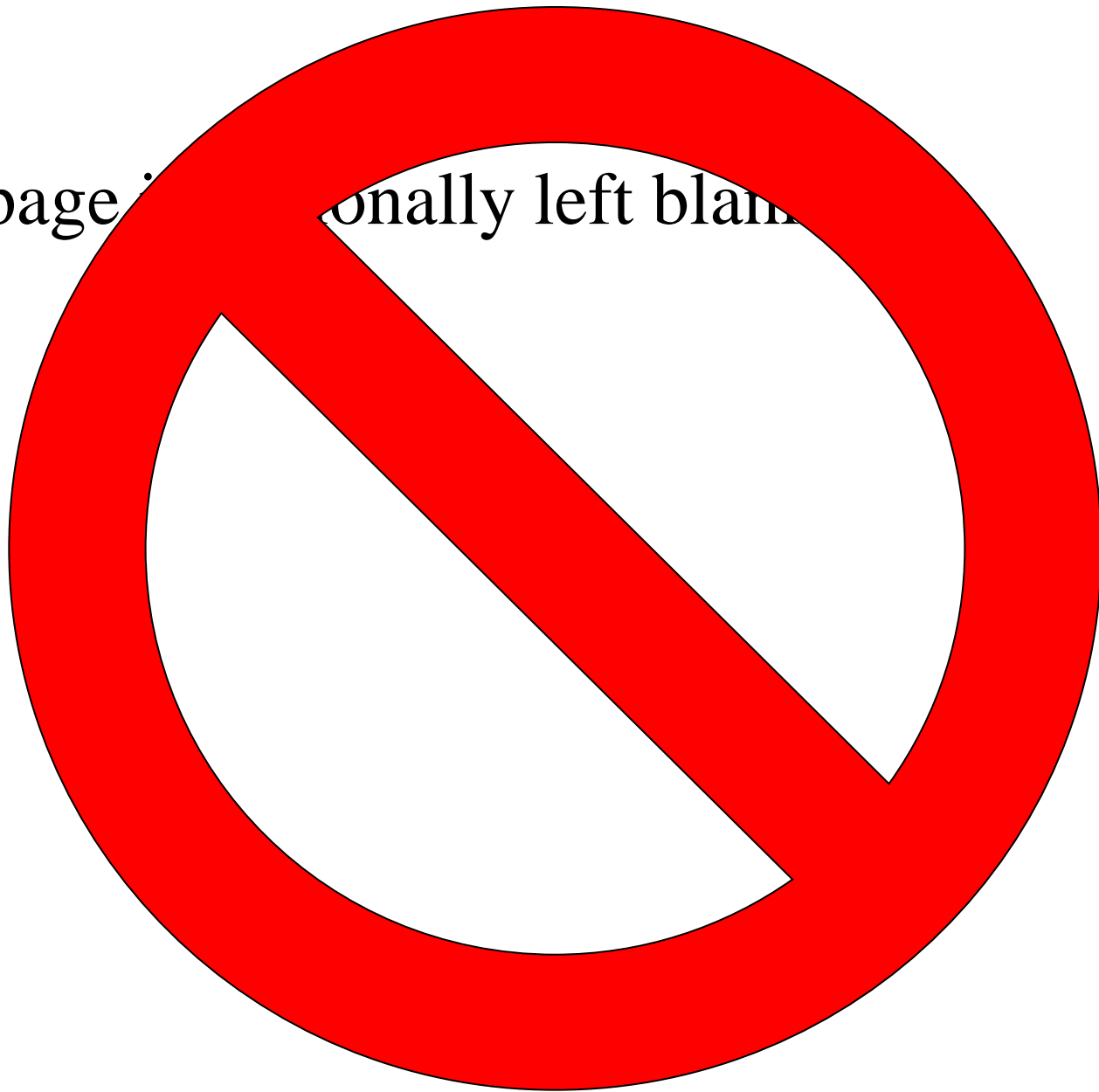
**fig03\_24.cpp**  
(1 of 2)

```
1  // Fig. 3.24: fig03_24.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setprecision;
11
12 // define global constant PI
13 const double PI = 3.14159265358979;
14
15 int main()
16 {
17     // define local constant PI
18     const float PI = static_cast< float >( ::PI );
19
20     // display values of local and global PI constants
21     cout << setprecision( 20 )
22         << "   Local float value of PI = " << PI
23         << "\nGlobal double value of PI = " << ::PI << endl;
24
25     return 0; // indicates successful termination
```

Access the global **PI** with  
**::PI**.

Cast the global **PI** to a  
**float** for the local **PI**. This  
example will show the  
difference between **float**  
and **double**.

This page intentionally left blank



# Chapter 4 - Arrays

## Outline

- 4.1 Introduction**
- 4.2 Arrays**
- 4.3 Declaring Arrays**
- 4.4 Examples Using Arrays**
- 4.5 Passing Arrays to Functions**
- 4.6 Sorting Arrays**
- 4.7 Case Study: Computing Mean, Median and Mode Using Arrays**
- 4.8 Searching Arrays: Linear Search and Binary Search**
- 4.9 Multiple-Subscripted Arrays**



## 4.1 Introduction

- Arrays
  - Structures of related data items
  - Static entity (same size throughout program)
- A few types
  - Pointer-based arrays (C-like)
  - Arrays as objects (C++)



## 4.4 Examples Using Arrays

- Initializing arrays

- For loop

- Set each element

- Initializer list

- Specify each element when array declared

- ```
int n[ 5 ] = { 1, 2, 3, 4, 5 };
```

- If not enough initializers, rightmost elements 0

- If too many syntax error

- To set every element to same value

- ```
int n[ 5 ] = { 0 };
```

- If array size omitted, initializers determine size

- ```
int n[] = { 1, 2, 3, 4, 5 };
```

- 5 initializers, therefore 5 element array





fig04\_07.cpp  
(1 of 1)

fig04\_07.cpp  
output (1 of 1)

Uninitialized **const** results in a syntax error. Attempting to modify the **const** is another error.

```
1 // Fig. 4.7: fig04_07.cpp
2 // A const object must be initialized
3
4 int main()
5 {
6     const int x; // Error: x must be initialized
7
8     x = 7; // Error: cannot modify a const variable
9
10    return 0; // indicates successful termination
11
12 }
```

```
d:\cpphttp4_examples\ch04\Fig04_07.cpp(6) : error C2734: 'x' :
const object must be initialized if not extern
d:\cpphttp4_examples\ch04\Fig04_07.cpp(8) : error C2166:
l-value specifies const object
```

## 4.4 Examples Using Arrays

- Strings

- Arrays of characters
- All strings end with **null** (`'\0'`)
- Examples

- `char string1[] = "hello";`

- Null character implicitly added

- `string1` has 6 elements

- `char string1[] = { 'h', 'e', 'l', 'l', 'o', '\0' };`



**fig04\_12.cpp**  
(1 of 2)

```
1  // Fig. 4_12: fig04_12.cpp
2  // Treating character arrays as strings.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  int main()
10 {
11     char string1[ 20 ], //
12     char string2[] = "string literal";
13
14     // read string from user into a
15     cout << "Enter the string \"hello\" ";
16     cin >> string1; // reads "hello" [space terminates input]
17
18     // output strings
19     cout << "string1 is: " << string1
20         << "\nstring2 is: " << string2;
21
22     cout << "\nstring1 with spaces between characters is:\n";
23 }
```

Two different ways to declare strings. **string2** is initialized, and its size determined automatically.

Examples of reading strings from the keyboard and printing them out.



## 4.9 Multiple-Subscripted Arrays

- To initialize
  - Default of 0
  - Initializers grouped by row in braces

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Row 0                      Row 1

|   |   |
|---|---|
| 1 | 2 |
| 3 | 4 |

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

|   |   |
|---|---|
| 1 | 0 |
| 3 | 4 |



## 4.4 Examples Using Arrays

- Recall static storage
    - If **static**, local variables save values between function calls
    - Visible only in function body
    - Can declare local arrays to be static
      - Initialized to zero on first function call
- ```
static int array[3];
```



## 4.5 Passing Arrays to Functions

- Arrays passed-by-reference
  - Functions can modify original array data
  - Value of name of array is address of first element
    - Function knows where the array is stored
    - Can change original memory locations
- Individual array elements passed-by-value
  - Like regular variables
  - **square( myArray[3] );**



## 4.5 Passing Arrays to Functions

- Functions taking arrays
  - Function prototype
    - `void modifyArray( int b[], int arraySize );`
    - `void modifyArray( int [], int );`
      - Names optional in prototype
    - Both take an integer array and a single integer
  - No need for array size between brackets
    - Ignored by compiler
  - If declare array parameter as **const**
    - Cannot be modified (compiler error)
    - `void doNotModify( const int [] );`



## 4.9 Multiple-Subscripted Arrays

- Function prototypes
  - Must specify sizes of subscripts
    - First subscript not necessary, as with single-scripted arrays
  - **`void printArray( int [ ][ 3 ] );`**



**fig04\_22.cpp**  
(1 of 2)

```
1  // Fig. 4.22: fig04_22.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  void printArray( int [][ 3 ] );
9
10 int main()
11 {
12     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
13     int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
14     int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
15
16     cout << "Values in array1 by row are:" << endl;
17     printArray( array1 );
18
19     cout << "Values in array2 by row are:" << endl;
20     printArray( array2 );
21
22     cout << "Values in array3 by row are:" << endl;
23     printArray( array3 );
24
25     return 0; // indicates successful termination
26
27 } // end main
```

Note the format of the prototype.

Note the various initialization styles. The elements in **array2** are assigned to the first row and then the second.