# PROGRAMACIÓN EN C++

Transparencias del libro Starting out with C++ : early objects, Tony Gaddis, Judy Walters, Godfrey Muganda

# Chapter 1: Introduction to Computers and Programming

# Topics

1.1 Why Program?

1.2 Computer Systems: Hardware and Software

1.3 Programs and Programming Languages

1.4 What Is a Program Made of?

1.5 Input, Processing, and Output

1.6 The Programming Process

# 1.1 Why Program?

Computer – programmable machine designed to follow instructions

Program/Software – instructions in computer memory to make it do something

Programmer – person who writes instructions (programs) to make computer perform a task

SO, without programmers, no programs; without programs, the computer cannot do anything

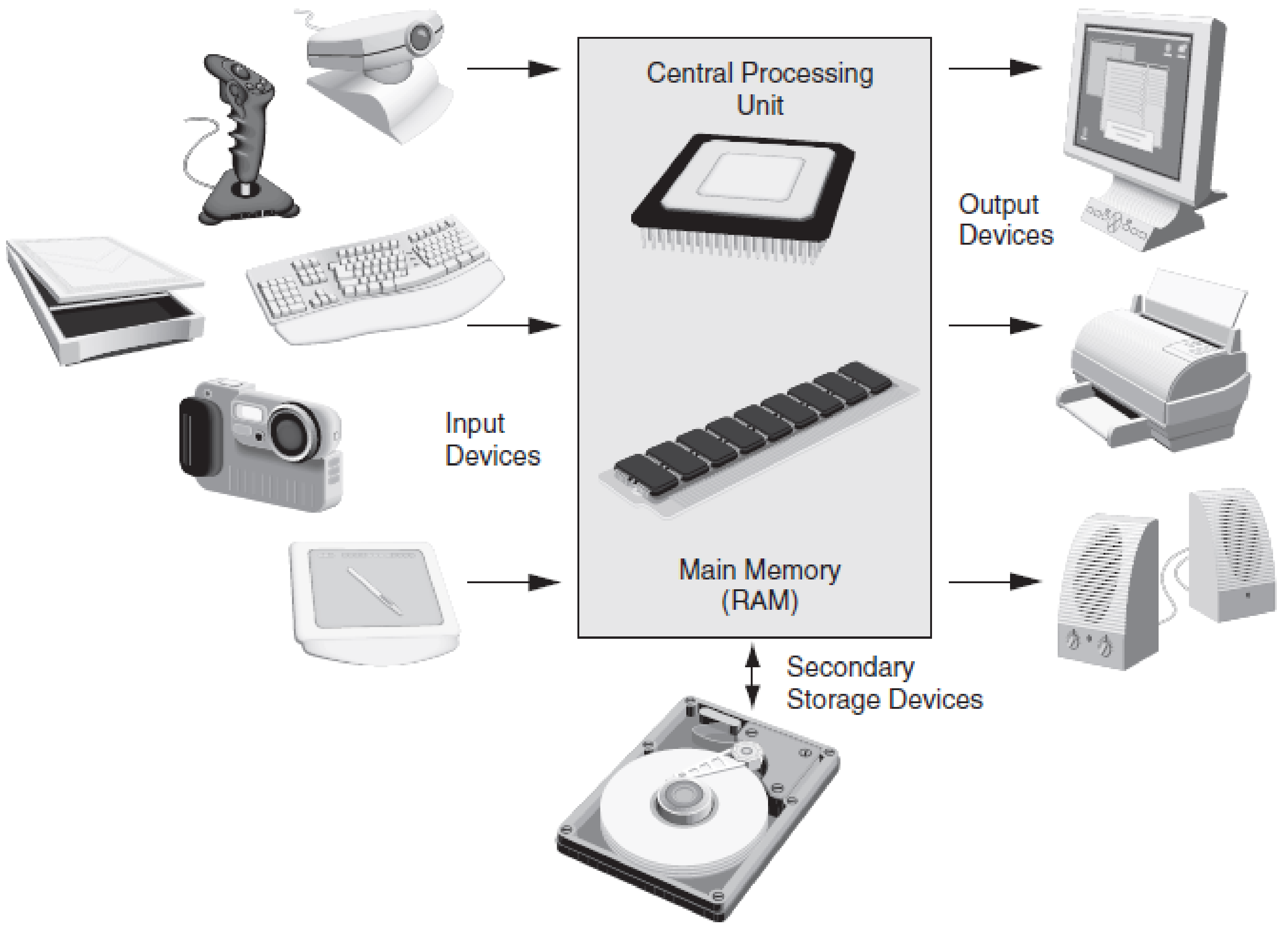
# 1.2 Computer Systems: Hardware and Software

Hardware – Physical components of a computer

Main Hardware Component Categories

1. Central Processing Unit (CPU)
2. Main memory (RAM)
3. Secondary storage devices
4. Input Devices
5. Output Devices

# Main Hardware Component Categories

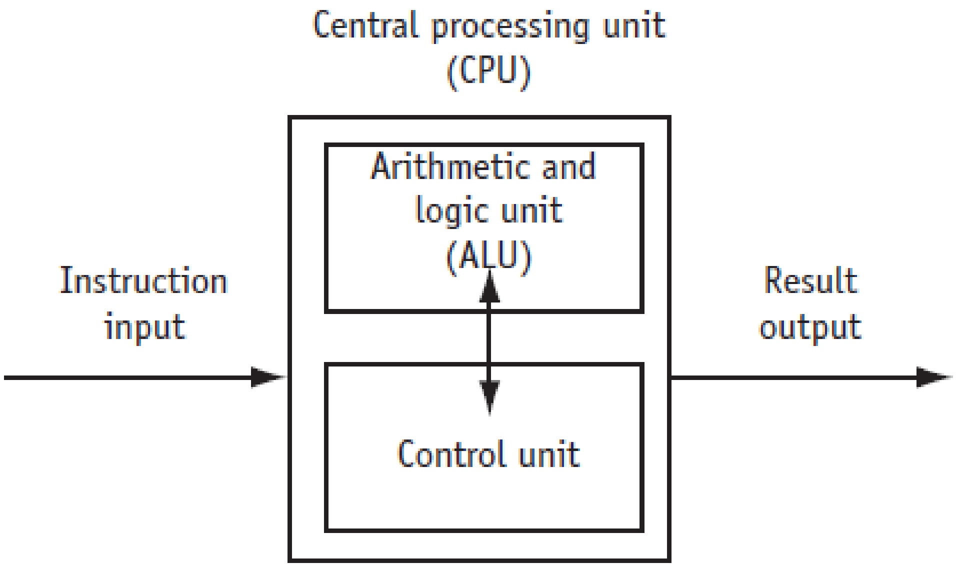


# Central Processing Unit (CPU)

CPU – Hardware component that runs programs

Includes

- Control Unit
  - Retrieves and decode program instructions
  - Coordinates compute operations
- Arithmetic & Logic Ur (ALU)
  - Performs mathematical operations



# The CPU's Role in Running a Program

Cycle through:

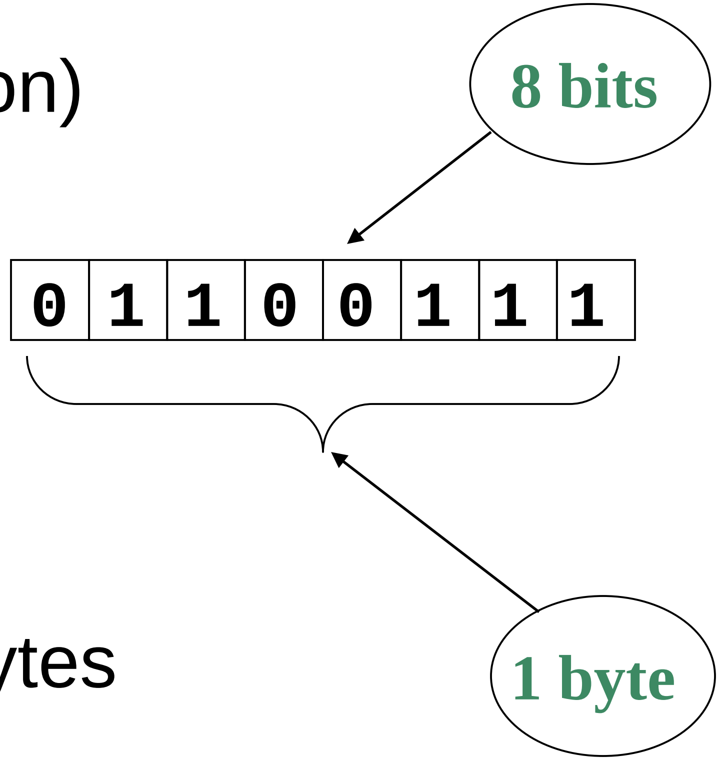
- Fetch: get the next program instruction from main memory
- Decode: interpret the instruction and generate a signal
- Execute: route the signal to the appropriate component to perform an operation

# Main Memory

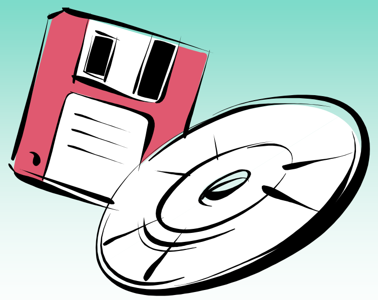
- Holds both program instructions and data
- Volatile – erased when program terminates or computer is turned off
- Also called Random Access Memory (RAM)

# Main Memory Organization

- Bit
  - Smallest piece of memory
  - Stands for **b**inary dig**it**
  - Has values 0 (off) or 1 (on)
- Byte
  - Is 8 consecutive bits
  - Has an address
- Word
  - Usually 4 consecutive bytes



# Secondary Storage

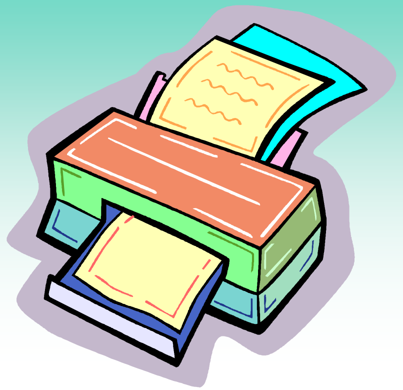


- Non-volatile - data retained when program is not running or computer is turned off
- Comes in a variety of media
  - magnetic: floppy or hard disk drive, internal or external
  - optical: CD or DVD drive
  - flash: USB flash drive

# Input Devices

- Used to send information to the computer from outside
- Many devices can provide input
  - keyboard, mouse, microphone, scanner, digital camera, disk drive, CD/DVD drive, USB flash drive

# Output Devices



- Used to send information from the computer to the outside
- Many devices can be used for output
  – Computer screen, printer, speakers, disk drive, CD/DVD recorder, USB flash drive

# Software Programs That Run on a Computer

- System software
  - programs that manage the computer hardware and the programs that run on the computer
  - Operating Systems
    - Controls operation of computer
    - Manages connected devices
    - Runs programs
  - Utility Programs
    - Support programs that enhance computer operations
    - Examples: anti-virus software, data backup, data compression
  - Software development tools
    - Used by programmers to create software
    - Examples: compilers, integrated development environments (IDEs)

# 1.3 Programs and Programming Languages

- Program

  a set of instructions directing a computer to perform a task

- Programming Language

  a language used to write programs

# Algorithm

Algorithm: a set of steps to perform a task or to solve a problem

Order is important. Steps must be performed sequentially

# Programs and Programming Languages

## Types of languages

- Low-level: used for communication with computer hardware directly.
- High-level: closer to human language

# From a High-level Program to an Executable File

a) Create file containing the program with a text editor.

b) Run preprocessor to convert source file directives to source code program statements.

c) Run compiler to convert source program statements into machine instructions.

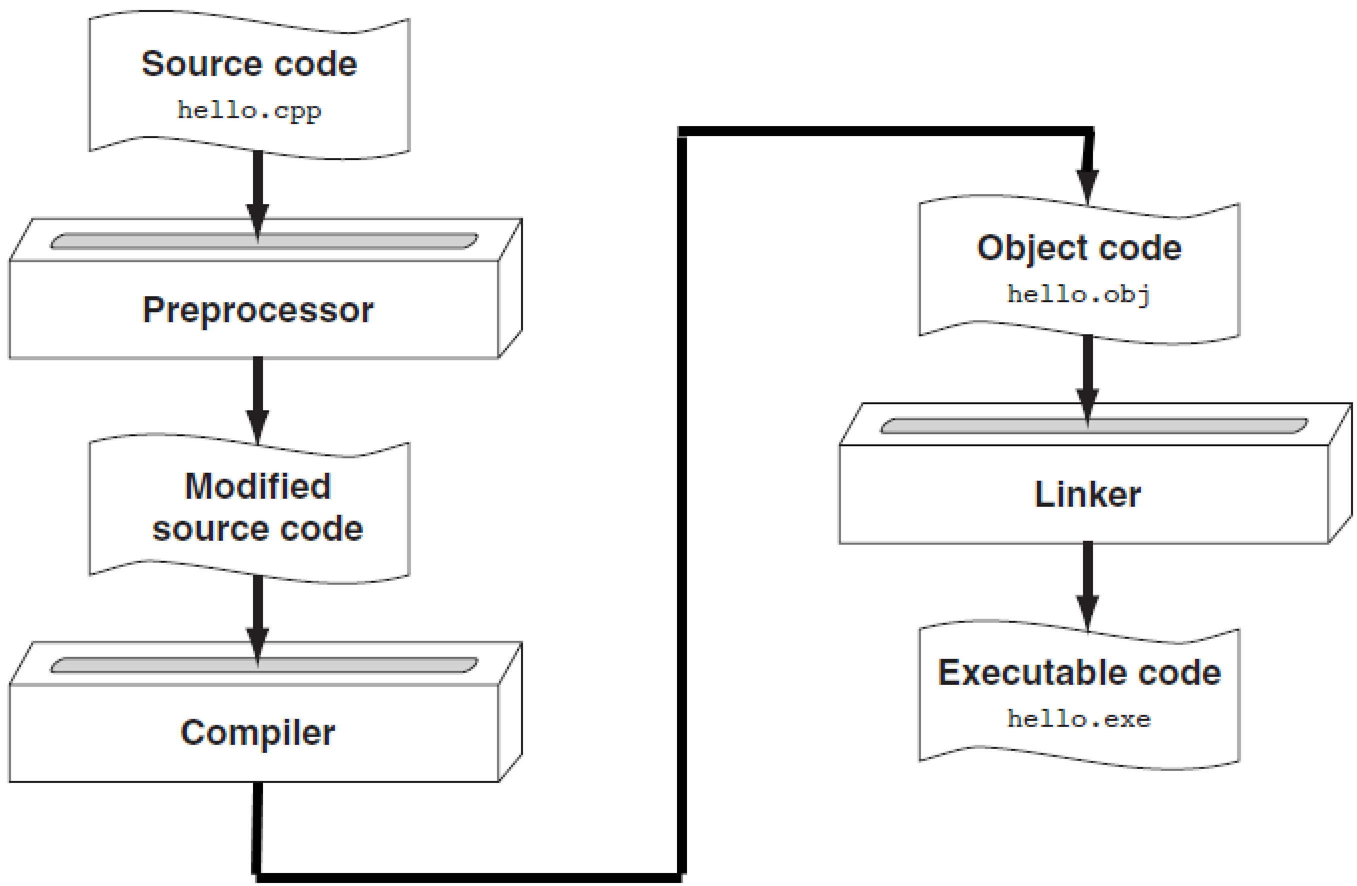
# From a High-level Program to an Executable File

d) Run linker to connect hardware-specific library code to machine instructions, producing an executable file.

Steps b) through d) are often performed by a single command or button click.

Errors occuring at any step will prevent execution of the following steps.

# From a High-level Program to an Executable File



# 1.4 What Is a Program Made Of?

Common elements in programming languages:

- Key Words
- Programmer-Defined Identifiers
- Operators
- Punctuation
- Syntax

# Example Program

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

int main()
{
    double num1 = 5,
           num2, sum;
    num2 = 12;

    sum = num1 + num2;
    cout << "The sum is " << sum;
    return 0;
}
```

# Key Words

- Also known as reserved words
- Have a special meaning in C++
- Can not be used for another purpose
- Written using lowercase letters
- Examples in program (shown in green):

```
using namespace std;
int main()
```

# Programmer-Defined Identifiers

- Names made up by the programmer
- Not part of the C++ language
- Used to represent various things, such as variables (memory locations)
- Example in program (shown in green):

```
double num1
```

# Operators

- Used to perform operations on data
- Many types of operators
  - Arithmetic: **`+, -, *, /`**
  - Assignment: **`=`**
- Examples in program (shown in green):

```
num2 = 12;
sum = num1 + num2;
```

# Punctuation

- Characters that mark the end of a statement, or that separate items in a list
- Example in program (shown in green):

```
double num1 = 5,
       num2, sum;
num2 = 12;
```

# Lines vs. Statements

In a source file,

- A line is all of the characters entered before a carriage return.
- Blank lines improve the readability of a program.
- Here are four sample lines. Line 3 is blank:

```
1. double num1 = 5, num2, sum;
2. num2 = 12;
3.
4. sum = num1 + num2;
```

# Lines vs. Statements

In a source file,

- A statement is an instruction to the computer to perform an action.
- A statement may contain keywords, operators, programmer-defined identifiers, and punctuation.
- A statement may fit on one line, or it may occupy multiple lines.
- Here is a single statement that uses two lines:

```
double num1 = 5,
       num2, sum;
```

# Variables

- A variable is a named location in computer memory (in RAM)
- It holds a piece of data. The data that it holds may change while the program is running.
- The name of the variable should reflect its purpose
- It must be *defined* before it can be used. Variable definitions indicate the variable name and the type of data that it can hold.
- Example variable definition:

```
double num1;
```

# 1.5 Input, Processing, and Output

Three steps that many programs perform

1) Gather input data
   - from keyboard
   - from files on disk drives
2) Process the input data
3) Display the results as output
   - send it to the screen or a printer
   - write it to a file

# 1.6 The Programming Process

1. Define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools to create a model of the program.

   Hierarchy charts, flowcharts, pseudocode, etc.
4. Check the model for logical errors.
5. Write the program source code.
6. Compile the source code.

# The Programming Process (cont.)

7. Correct any errors found during compilation.
8. Link the program to create an executable file.
9. Run the program using test data for input.
10. Correct any errors found while running the program.

**Repeat steps 4 - 10 as many times as necessary.**

11. Validate the results of the program.

    Does the program do what was defined in step 1?

# Chapter 2: Introduction to C++

# Topics

2.1 The Parts of a C++ Program

2.2 The **`cout`** Object

2.3 The **`#include`** Directive

2.4 Standard and Prestandard C++

2.5 Variables, Literals, and the Assignment Statement

2.6 Identifiers

2.7 Integer Data Types

2.8 Floating-Point Data Types

# Topics (continued)

2.9 The **`char`** Data Type

2.10 The C++ **`string`** Class

2.11 The **`bool`** Data Type

2.12 Determining the Size of a Data Type

2.13 More on Variable Assignments and Initialization

2.14 Scope

2.15 Arithmetic Operators

2.16 Comments

# 2.1 The Parts of a C++ Program

```
// sample C++ program
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, there!";
    return 0;
}
```

| | |
|---|---|
| `// sample C++ program` | comment |
| `#include <iostream>` | preprocessor directive |
| `using namespace std;` | which namespace to use |
| `int main()` | beginning of function named `main` |
| `{` | beginning of block for `main` |
| `cout << "Hello, there!";` | output statement |
| `return 0;` | send `0` back to operating system |
| `}` | end of block for `main` |

# 2.1 The Parts of a C++ Program

| Statement | Purpose |
|---|---|
| `// sample C++ program` | comment |
| `#include <iostream>` | preprocessor directive |
| `using namespace std;` | which namespace to use |
| `int main()` | beginning of function named `main` |
| `{` | beginning of block for `main` |
| `   cout << "Hello, there!";` | output statement |
| `   return 0;` | send 0 back to the operating system |
| `}` | end of block for `main` |

# Special Characters

| Character | Name | Description |
|---|---|---|
| // | Double Slash | Begins a comment |
| # | Pound Sign | Begins preprocessor directive |
| < > | Open, Close Brackets | Encloses filename used in #include directive |
| ( ) | Open, Close Parentheses | Used when naming function |
| { } | Open, Close Braces | Encloses a group of statements |
| " " | Open, Close Quote Marks | Encloses string of characters |
| ; | Semicolon | Ends a programming statement |

# Important Details

- C++ is <u>case-sensitive</u>. Uppercase and lowercase characters are different characters. '`Main`' is not the same as '`main`'.
- Every `{` must have a corresponding `}`, and vice-versa.

# 2.2 The **`cout`** Object

- Displays information on computer screen
- Use **<<** to send information to cout

```
cout << "Hello, there!";
```

- Can use **<<** to send multiple items to cout

```
cout << "Hello, " << "there!";
```

Or

```
cout << "Hello, ";
cout << "there!";
```

# Starting a New Line

- To get multiple lines of output on screen
  - Use **endl**

```
cout << "Hello, there!" << endl;
```

  - Use **\n** in an output string

```
cout << "Hello, there!\n";
```

# Escape Sequences – More Control Over Output

| Escape Sequence | Name | Description |
|---|---|---|
| \n | Newline | Causes the cursor to go to the next line for subsequent printing. |
| \t | Horizontal tab | Causes the cursor to skip over to the next tab stop. |
| \a | Alarm | Causes the computer to beep. |
| \b | Backspace | Causes the cursor to back up, or move left one position. |
| \r | Return | Causes the cursor to go to the beginning of the current line, not the next line. |
| \\ | Backslash | Causes a backslash to be printed. |
| \' | Single quote | Causes a single quotation mark to be printed. |
| \" | Double quote | Causes a double quotation mark to be printed. |

# 2.3 The **`#include`** Directive

- Inserts the contents of another file into the program
- Is a preprocessor directive
  - Not part of the C++ language
  - Not seen by compiler
- Example:

```
#include <iostream>
```

No ; goes here

# 2.4 Standard and Prestandard C++

Prestandard (Older-style) C++ programs

- Use **`.h`** at end of header files

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

- Do not use **`using namespace`** convention
- May not use **`return 0;`** at the end of function **`main`**
- May not compile with a standard C++ compiler

# 2.5 Variables, Literals, and the Assignment Statement

- Variable
  - Has a name and a type of data it can hold


  - Is used to reference a location in memory where a value can be stored
  - Must be defined before it can be used
  - The value that is stored can be changed, *i.e.*, it can “vary”

# Variables

– If a new value is stored in the variable, it replaces the previous value

– The previous value is overwritten and can no longer be retrieved

```
int age;
age = 17;       // age is 17
cout << age;   // Displays 17
age = 18;       // Now age is 18
cout << age;   // Displays 18
```

# Assignment Statement

- Uses the **=** operator
- Has a single variable on the left side and a value on the right side
- Copies the value on the right into the variable on the left

```
item = 12;
```

# Constants

## Literal

- Data item whose value does not change during program execution
- Is also called a constant

```
'A'       // character constant
"Hello"   // string literal
12        // integer constant
3.14      // floating-point constant
```

# 2.6 Identifiers

- Programmer-chosen names to represent parts of the program, such as variables
- Name should indicate the use of the identifier
- Cannot use C++ key words as identifiers
- Must begin with alphabetic character or _, followed by alphabetic, numeric, or _ .  Alphabetic characters may be upper- or lowercase

# Multi-word Variable Names

- Descriptive variable names may include multiple words
- Two conventions to use in naming variables:
  - Capitalize all but first letter of first word. Run words together:

    **quantityOnOrder**

    **totalSales**
  - Use the underscore _ character as a space:

    **quantity_on_order**

    **total_sales**
- Use one convention consistently throughout program

# Valid and Invalid Identifiers

| IDENTIFIER | VALID? | REASON IF INVALID |
| --- | --- | --- |
| `totalSales` | Yes | |
| `total_Sales` | Yes | |
| `total.Sales` | No | Cannot contain period |
| `4thQtrSales` | No | Cannot begin with digit |
| `totalSale$` | No | Cannot contain $ |

# 2.7 Integer Data Types

- Designed to hold whole (non-decimal) numbers
- Can be **`signed`** or **`unsigned`**

  **`12        -6        +3`**

- Available in different sizes (*i.e.*, number of bytes): **`short`**, **`int`**, and **`long`**
- Size of **`short`** ≤ size of **`int`** ≤ size of **`long`**

# Signed vs. Unsigned Integers

- C++ allocates one bit for the sign of the number.  The rest of the bits are for data.
- If your program will never need negative numbers, you can declare variables to be **`unsigned`**.  All bits in unsigned numbers are used for data.
- A variable is signed unless the **`unsigned`** keyword is used.

# Defining Variables

- Variables of the same type can be defined
  - In separate statements

```
int length;
int width;
```

  - In the same statement

```
int length,
    width;
```

- Variables of different types must be defined in separate statements

# Integral Constants

- To store an integer constant in a long memory location, put '**`L`**' at the end of the number: **`1234L`**
- Constants that begin with '**`0`**' (zero) are octal, or base 8: **`075`**
- Constants that begin with '**`0x`**' are hexadecimal, or base 16: **`0x75A`**

# 2.8 Floating-Point Data Types

- Designed to hold real numbers

  **`12.45`**          **`-3.8`**

- Stored in a form similar to scientific notation
- Numbers are all signed
- Available in different sizes (number of bytes): **`float`**, **`double`**, and **`long double`**
- Size of **`float`** ≤ size of **`double`**

  ≤ size of **`long double`**

# Floating-point Constants

- Can be represented in
  - Fixed point (decimal) notation:

    **`31.4159`**  **`0.0000625`**
  - E notation:

    **`3.14159E1`**  **`6.25e-5`**
- Are **`double`** by default
- Can be forced to be float **`3.14159F`** or long double **`0.0000625L`**

# Assigning Floating-point Values to Integer Variables

If a floating-point value is assigned to an integer variable

- The fractional part will be truncated (*i.e.*, “chopped off” and discarded)
- The value is not rounded

```
int rainfall = 3.88;
cout << rainfall;  // Displays 3
```

# 2.9 The **`char`** Data Type

- Used to hold single characters or very small integer values
- Usually occupies 1 byte of memory
- A numeric code representing the character is stored in memory

| SOURCE CODE | MEMORY |
| --- | --- |
| `char letter = 'C';` | `letter` |

**67**

# Character Literal

- A character literal is a single character
- When referenced in a program, it is enclosed in single quotation marks:

```
cout << 'Y' << endl;
```

- The quotation marks are not part of the literal, and are not displayed

# String Literals

- Can be stored as a series of characters in consecutive memory locations

  **"Hello"**

- Stored with the null terminator, **\0**, automatically placed at the end

| | | | | | |
|---|---|---|---|---|---|
| **H** | **e** | **l** | **l** | **o** | **\0** |

- Is comprised of characters between the **" "**

# A character or a string literal?

- A character literal is a single character, enclosed in single quotes:

```
'C'
```

- A string literal is a sequence of characters enclosed in double quotes:

```
"Hello, there!"
```

- A single character in double quotes is a string literal, not a character literal:

```
"C"
```

# 2.10 The C++ **`string`** Class

- Must **`#include <string>`** to create and use string objects
- Can define **`string`** variables in programs

```
string name;
```

- Can assign values to string variables with the assignment operator

```
name = "George";
```

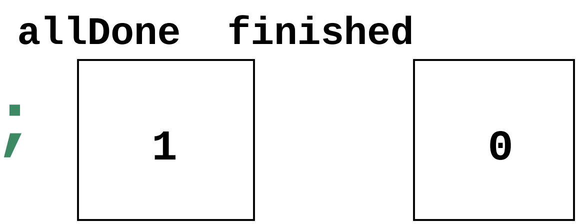
- Can display them with **`cout`**

```
cout << "My name is " << name;
```

# 2.11 The **`bool`** Data Type

- Represents values that are **`true`** or **`false`**
- **`bool`** values are stored as integers
- **`false`** is represented by 0, **`true`** by 1

```
bool allDone = true;
bool finished = false;
```



# 2.12 Determining the Size of a Data Type

The **sizeof** operator gives the size in number of bytes of any data type or variable

```
double amount;
cout << "A float is stored in "
     << sizeof(float) << " bytes\n";
cout << "Variable amount is stored in "
     << sizeof(amount) << " bytes\n";
```

# 2.13 More on Variable Assignments and Initialization

Assigning a value to a variable

- Assigns a value to a previously created variable
- A single variable name must appear on left side of the **=** symbol

```
int size;
size = 5;     // legal
5 = size;     // not legal
```

# Variable Assignment vs. Initialization

Initializing a variable

- Gives an initial value to a variable at the time it is created
- Can initialize some or all of the variables being defined

```
int length = 12;
int width = 7, height = 5, area;
```

# 2.14 Scope

- The scope of a variable is that part of the program where the variable may be used
- A variable cannot be used before it is defined

```
int num1 = 5;
cout >> num1;   // legal
cout >> num2;   // illegal
int num2 = 12;
```

# 2.15 Arithmetic Operators

- Used for performing numeric calculations
- C++ has unary, binary, and ternary operators
  - unary (1 operand) **`-5`**
  - binary (2 operands) **`13 - 7`**
  - ternary (3 operands) **`exp1 ? exp2 : exp3`**

# Binary Arithmetic Operators

| SYMBOL | OPERATION | EXAMPLE | ans |
|---|---|---|---|
| + | addition | `ans = 7 + 3;` | 10 |
| - | subtraction | `ans = 7 - 3;` | 4 |
| * | multiplication | `ans = 7 * 3;` | 21 |
| / | division | `ans = 7 / 3;` | 2 |
| % | modulus | `ans = 7 % 3;` | 1 |

# `/` Operator

- C++ division operator (`/`) performs integer division if both operands are integers

```
cout << 13 / 5;     // displays 2
cout <<  2 / 4;     // displays 0
```

- If either operand is floating-point, the result is floating-point

```
cout << 13 / 5.0;  // displays 2.6
cout << 2.0 / 4;   // displays 0.5
```

# % Operator

- C++ modulus operator (%) computes the remainder resulting from integer division

```
cout << 9 % 2;   // displays 1
```

- % requires integers for both operands

```
cout << 9 % 2.0; // error
```

# 2.16 Comments

- Are used to document parts of a program
- Are written for persons reading the source code of the program
  - Indicate the purpose of the program
  - Describe the use of variables
  - Explain complex sections of code
- Are ignored by the compiler

# Single-Line Comments

- Begin with `//` and continue to the end of line

```
int length = 12; // length in inches
int width = 15;  // width in inches
int area;        // calculated area

// Calculate rectangle area
area = length * width;
```

# Multi-Line Comments

- Begin with **/*** and end with ***/**
- Can span multiple lines

```
/*----------------------------
   Here's a multi-line comment
  ----------------------------*/
```

- Can also be used as single-line comments

```
int area;   /* Calculated area */
```

# Chapter 3: Expressions and Interactivity

# Topics

3.1 The **`cin`** Object

3.2 Mathematical Expressions

3.3 Data Type Conversion and Type Casting

3.4 Overflow and Underflow

3.5 Named Constants

# Topics (continued)

3.6 Multiple and Combined Assignment

3.7 Formatting Output

3.8 Working with Characters and Strings

3.9 Using C-Strings

3.10 More Mathematical Library Functions

# 3.1 The **`cin`** Object

- Standard input object
- Like **`cout`**, requires **`iostream`** file
- Used to read input from keyboard
- Often used with **`cout`** to display a user prompt first
- Data is retrieved from **`cin`** with **`>>`**
- Input data is stored in one or more variables

# The **`cin`** Object

- User input goes from keyboard to the input buffer, where it is stored as characters
- **`cin`** converts the data to the type that matches the variable

```
int height;
cout << "How tall is the room? ";
cin  >> height;
```

# The `cin` Object

- Can be used to input multiple values

```
cin >> height >> width;
```

- Multiple values from keyboard must be separated by spaces or [Enter]
- Must press [Enter] after typing last value
- Multiple values need not all be of the same type
- Order is important; first value entered is stored in first variable, etc.

# 3.2 Mathematical Expressions

- An expression can be a constant, a variable, or a combination of constants and variables combined with operators
- Can create complex expressions using multiple mathematical operators
- Examples of mathematical expressions:

```
  2
height
a + b / c
```

# Using Mathematical Expressions

- Can be used in assignment statements, with **`cout`**, and in other types of statements
- Examples:

```
area = 2 * PI * radius;
cout << "border is: " << (2*(l+w));
```

This is an expression

These are expressions

# Order of Operations

- In an expression with > 1 operator, evaluate in this order

**Do first:** ( ) expressions in parentheses

**Do next:** **-** (unary negation) in order, left to right

**Do next:** *** / %** in order, left to right

**Do last:** **+ -** in order, left to right

- In the expression **2 + 2 * 2 – 2 ,**

**Evaluate 2nd** (+) **Evaluate 1st** (*) **Evaluate 3rd** (–)

# Associativity of Operators

- **-** (unary negation) associates right to left
- ***   /   %   +   -**   all associate left to right
- parentheses **( )** can be used to override the order of operations

```
 2 + 2  *  2 – 2  = 4
(2 + 2) *  2 – 2  = 6
 2 + 2  * (2 – 2) = 2
(2 + 2) * (2 – 2) = 0
```

# Algebraic Expressions

- Multiplication requires an operator

  $Area = lw$ is written as `Area = l * w;`

- There is no exponentiation operator

  $Area = s^2$ is written as `Area = pow(s, 2);`

  (note: **`pow`** requires the **`cmath`** header file)

- Parentheses may be needed to maintain order of operations

  $$m = \frac{y_2 - y_1}{x_2 - x_1}$$ is written as

  `m = (y2-y1)/(x2-x1);`

# 3.3 Data Type Conversion and Type Casting

- Operations are performed between operands of the same type
- If operands do not have the same type, C++ will automatically convert one to be the type of the other
- This can impact the results of calculations

# Hierarchy of Data Types

- Highest
- Lowest

```
long double
double
float
unsigned long
long
unsigned int
int
```

- Ranked by largest number they can hold

# Type Coercion

- Coercion: automatic conversion of an operand to another data type
- Promotion: converts to a higher type
- Demotion: converts to a lower type

# Coercion Rules

1) **`char`**, **`short`**, **`unsigned short`** are automatically promoted to **`int`**
2) When operating on values of different data types, the lower-ranked one is promoted to the type of the higher one.
3) When using the **`=`** operator, the type of expression on right will be converted to the type of variable on left

# Coercion Rules – Important Notes

1) If demotion is required to use the = operator,
   - the stored result may be incorrect if there is not enough space available in the receiving variable
   - floating-point values are truncated when assigned to integer variables

2) Coercion affects the <u>value</u> used in a calculation. It does not change the type associated with a variable.

# Type Casting

- Used for manual data type conversion
- Format

```
static_cast<Data Type>(Value)
```

- Example:

```
cout << static_cast<int>(4.2);
                    // Displays 4
```

# More Type Casting Examples

```
char ch = 'C';
cout << ch << " is stored as "
     << static_cast<int>(ch);

gallons = static_cast<int>(area/500);

avg = static_cast<double>(sum)/count;
```

# Older Type Cast Styles

```
double Volume = 21.58;
int intVol1, intVol2;
intVol1 = (int) Volume; // C-style
                        // cast
intVol2 = int (Volume); //Prestandard
                        // C++ style
                        // cast
```

C-style cast uses prefix notation

Prestandard C++ cast uses functional notation

**`static_cast`** is the current standard

# 3.4 Overflow and Underflow

- Occurs when assigning a value that is too large (overflow) or too small (underflow) to be held in a variable
- The variable contains a value that is ‘wrapped around’ the set of possible values

# Overflow Example

```
// Create a short int initialized to
// the largest value it can hold
short int num = 32767;

cout << num;      // Displays 32767
num = num + 1;
cout << num;      // Displays -32768
```

# Handling Overflow and Underflow

Different systems handle the problem differently. They may

- display a warning / error message, or display a dialog box and ask what to do
- stop the program
- continue execution with the incorrect value

# 3.5 Named Constants

- Also called constant variables
- Variables whose content cannot be changed during program execution
- Used for representing constant values with descriptive names

```
const double TAX_RATE = 0.0675;
const int NUM_STATES = 50;
```

- Often named in uppercase letters

# Benefits of Named Constants

- Makes program code more readable by documenting the purpose of the constant in the name:

```
const double TAX_RATE = 0.0675;
…
salesTax = purchasePrice * TAX_RATE;
```

- Simplifies program maintenance:

```
const double TAX_RATE = 0.0725;
```

# `const` vs. `#define`

## `#define`

- C-style of naming constants

  `#define NUM_STATES 50`

  no ; goes here

- Interpreted by pre-processor rather than compiler
- Does not occupy a memory location like a constant variable defined with **`const`**
- Instead, causes a text substitution to occur. In above example, every occurrence in program of **`NUM_STATES`** will be replaced by **`50`**

# 3.6 Multiple and Combined Assignment

- The assignment operator (**=**) can be used multiple times in an expression

```
x = y = z = 5;
```

- Associates right to left

```
x = (y = (z = 5));
```

Done 3rd (first =), Done 2nd (second =), Done 1st (third =)

# Combined Assignment

- Applies an arithmetic operation to a variable and assigns the result as the new value of that variable
- Operators: **+=  -=  *=  /=  %=**
- Also called compound operators or arithmetic assignment operators
- Example:

`sum += amt;` is short for `sum = sum + amt;`

# More Examples

```
x += 5;   means   x = x + 5;
x -= 5;   means   x = x – 5;
x *= 5;   means   x = x * 5;
x /= 5;   means   x = x / 5;
x %= 5;   means   x = x % 5;
```

The right hand side is evaluated before the combined assignment operation is done.

```
x *= a + b;   means   x = x * (a + b);
```

# 3.7 Formatting Output

- Can control how output displays for numeric and string data
  - size
  - position
  - number of digits
- Requires **`iomanip`** header file

# Stream Manipulators

- Used to control features of an output field
- Some affect just the next value displayed
  - **`setw(x)`**: Print in a field at least **x** spaces wide. It will use more spaces if specified field width is not big enough.

# Stream Manipulators

- Some affect values until changed again
  - **`fixed`**: Use decimal notation (not E-notation) for floating-point values.
  - **`setprecision(x)`**:
    - When used with **`fixed`**, print floating-point value using **`x`** digits after the decimal.
    - Without **`fixed`**, print floating-point value using **`x`** significant digits.
  - **`showpoint`**: Always print decimal for floating-point values.
  - **`left, right`**: left-, right justification of value

# Manipulator Examples

```
const float e = 2.718;
float price = 18.0;
cout << setw(8) << e << endl;
cout << left << setw(8) << e
     << endl;
cout << setprecision(2);
cout << e << endl;
cout << fixed << e << endl;
cout << setw(6) << price;
```

Displays

```
^^^2.718

2.718^^^

2.7
2.72
^18.00
```

# 3.8 Working with Characters and Strings

- **`char`**: holds a single character
- **`string`**: holds a sequence of characters
- Both can be used in assignment statements
- Both can be displayed with **`cout`** and **`<<`**

# String Input

Reading in a string object

```
string str;

cin >> str;              // Reads in a string
                         // with no blanks

getline(cin, str);       // Reads in a string
                         // that may contain
                         // blanks
```

# Character Input

Reading in a character:

```
char ch;
cin >> ch;       // Reads in any non-blank char
cin.get(ch);     // Reads in any char
ch = cin.get;    // Reads in any char
cin.ignore();    // Skips over next char in
                 // the input buffer
```

# String Operators

**=** Assigns a value to a string

```
string words;
words = "Tasty ";
```

**+** Joins two strings together

```
string s1 = "hot", s2 = "dog";
string food = s1 + s2; // food = "hotdog"
```

**+=** Concatenates a string onto the end of another one

```
words += food; // words now = "Tasty hotdog"
```

# string Member Functions

- **`length()`** – the number of characters in a string

```
string firstPrez="George Washington";
int size=firstPrez.length(); // size is 17
```

- assign() – put repeated characters in a string. Can be used for formatting output.

```
string equals;
equals.assign(80,'=');
…
cout << equals << endl;
cout << "Total: " << total << endl;
```

# 3.9 Using C-Strings

- C-string is stored as an array of characters
- Programmer must indicate maximum number of characters at definition

```
const int SIZE = 5;
char temp[SIZE] = "Hot";
```

- NULL character (**`\0`**) is placed after final character to mark the end of the string

| | | | | |
|---|---|---|---|---|
| H | o | t | \0 | |

- Programmer must make sure array is big enough for desired use; `temp` can hold up to 4 characters plus the **`\0`**.

# C-String Input

- Reading in a C-string

```
const int SIZE = 10;
char Cstr[SIZE];
cin >> Cstr;    // Reads in a C-string with no
                // blanks. Will write past the
                // end of the array if input string
                // is too long.
cin.getline(Cstr, 10);
                // Reads in a C-string that may
                // contain blanks. Ensures that <= 9
                // chars are read in.
```

- Can also use **`setw()`** and **`width()`** to control input field widths

# C-String Initialization vs. Assignment

- A C-string can be initialized at the time of its creation, just like a string object

```
const int SIZE = 10;
char month[SIZE] = "April";
```

- However, a C-string cannot later be assigned a value using the = operator; you must use the **`strcpy()`** function

```
char month[SIZE];
month = "August"             // wrong!
strcpy(month, "August"); //correct
```

# C-String and Keyboard Input

- Must use **`cin.getline()`** to put keyboard input into a C-string
- Note that **`cin.getline()`** ≠ **`getline()`**
- Must indicate the target C-string and maximum number of characters to read:

```
const int SIZE = 25;
char name[SIZE];
cout << "What's your name? ";
cin.getline(name, SIZE);
```

# 3.10 More Mathematical Library Functions

- These require **`cmath`** header file
- Take **`double`** arguments and return a **`double`**
- Commonly used functions

| | |
|---|---|
| **`abs`** | Absolute value |
| **`sin`** | Sine |
| **`cos`** | Cosine |
| **`tan`** | Tangent |
| **`sqrt`** | Square root |
| **`log`** | Natural (e) log |
| **`pow`** | Raise to a power |

# More Mathematical Library Functions

These require **`cstdlib`** header file

- **`rand`**
  - Returns a random number between **`0`** and the largest **`int`** the computer holds
  - Will yield the same sequence of numbers each time the program is run
- **`srand(x)`**
  - Initializes random number generator with **`unsigned int x`**. **`x`** is the "seed value".
  - Should be called at most once in a program

# More on Random Numbers

- Use **`time()`** to generate different seed values each time that a program runs:

```
#include <ctime> //needed for time()
…
unsigned seed = time(0);
srand(seed);
```

- Random numbers can be scaled to a range:

```
int max=6;
int num;
num = rand() % max + 1;
```

# Chapter 4:  Making Decisions

# Topics

4.1 Relational Operators

4.2 The **`if`** Statement

4.3 The **`if/else`** Statement

4.4 The **`if/else if`** Statement

4.5 Menu-Driven Programs

4.6 Nested **`if`** Statements

4.7 Logical Operators

# Topics (continued)

4.8 Validating User Input
4.9 More About Block and Scope
4.10 More About Characters and Strings
4.11 The Conditional Operator
4.12 The **`switch`** Statement
4.13 Enumerated Data Types

# 4.1 Relational Operators

- Used to compare numeric values to determine relative order
- Operators:

| | |
|---|---|
| `>` | Greater than |
| `<` | Less than |
| `>=` | Greater than or equal to |
| `<=` | Less than or equal to |
| `==` | Equal to |
| `!=` | Not equal to |

# Relational Expressions

- Relational expressions are Boolean (*i.e.*, evaluate to **`true`** or **`false`**)
- Examples:

  **`12 > 5`** is **`true`**

  **`7 <= 5`** is **`false`**

  if **`x`** is 10, then

  **`x == 10`** is **`true`**,

  **`x <= 8`** is **`false`**,

  **`x != 8`** is **`true`**, and

  **`x == 8`** is **`false`**

# Relational Expressions

- Can be assigned to a variable

```
bool result = (x <= y);
```

- Assigns **`0`** for **`false`**, **`1`** for **`true`**
- Do not confuse **`=`** (assignment) and **`==`** (equal to)

# 4.2 The **`if`** Statement

- Supports the use of a decision structure
- Allows statements to be conditionally executed or skipped over
- Models the way we mentally evaluate situations

  "If it is cold outside,
  wear a coat and wear a hat."

# Format of the **if** Statement

```
if (condition)
{
   statement1;
   statement2;
        …
   statementn;
}
```

No
; goes here

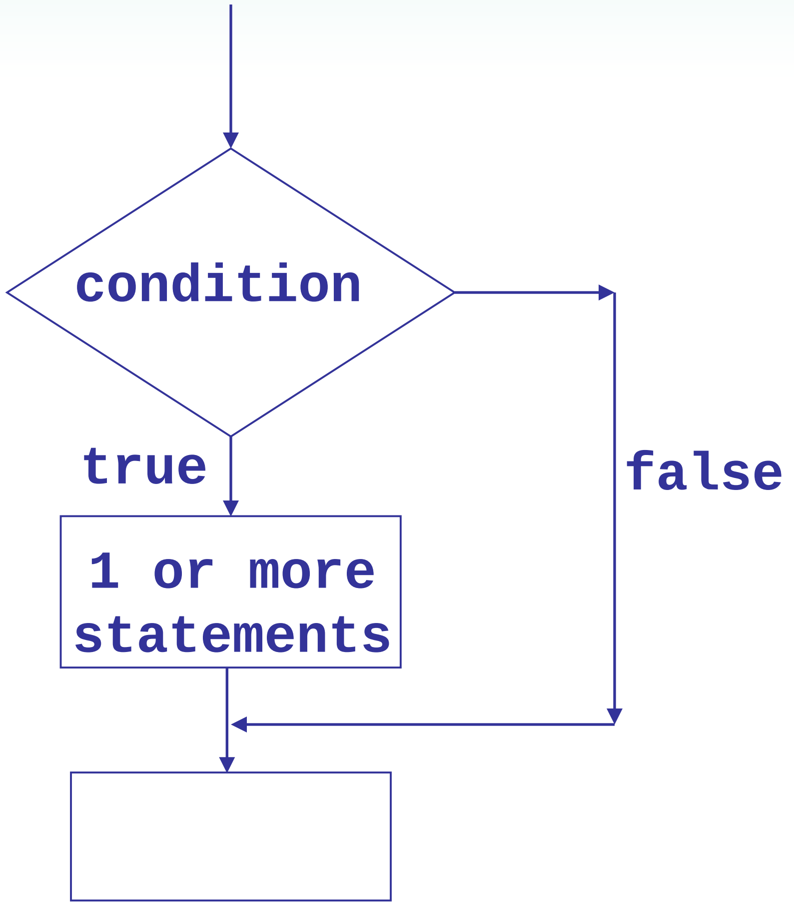
; goes here

The block inside the braces is called the <u>body</u> of the **if** statement. If there is only 1 statement in the body, the **{ }** may be omitted.

# How the **`if`** Statement Works

- If ***`(condition)`*** is **`true`**, then the ***`statement(s)`*** in the body are executed.
- If ***`(condition)`*** is **`false`**, then the ***`statement(s)`*** are skipped.

# `if` Statement Flow of Control



# Example **if** Statements

```
if (score >= 60)
    cout << "You passed." << endl;

if (score >= 90)
{
    grade = 'A';
    cout << "Wonderful job!" << endl;
}
```

# **`if`** Statement Notes

- **`if`** is a keyword. It must be lowercase
- ***`(condition)`*** must be in ( )
- Do not place `;` after ***`(condition)`***
- Don't forget the { } around a multi-statement body

# `if` Statement Style Recommendations

- Place each ***`statement;`*** on a separate line after **(*`condition`*)**
- Indent each statement in the body
- When using { and } around the body, put { and } on lines by themselves

# What is `true` and `false`?

- An expression whose value is 0 is considered **`false`**.
- An expression whose value is non-zero is considered **`true`**.
- An expression need not be a comparison – it can be a single variable or a mathematical expression.

# Flag

- A variable that signals a condition
- Usually implemented as a **`bool`**
- Meaning:
  - **`true`**: the condition exists
  - **`false`**: the condition does not exist
- The flag value can be both set and tested with **`if`** statements

# Flag Example

Example:

```
bool validMonths = true;
        …
if (months < 0)
    validMonths = false;
        …
if (validMonths)
    moPayment = total / months;
```

# Integer Flags

- Integer variables can be used as flags
- Remember that 0 means false, any other value means true

```
int allDone = 0;  // set to false
          …
  if (count > MAX_STUDENTS)
     allDone = 1;  // set to true
          …
  if (allDone)
     cout << "Task finished";
```

# 4.3 The **`if/else`** Statement

- Allows a choice between statements depending on whether **`(`*`condition`*`)`** is **`true`** or **`false`**
- Format:

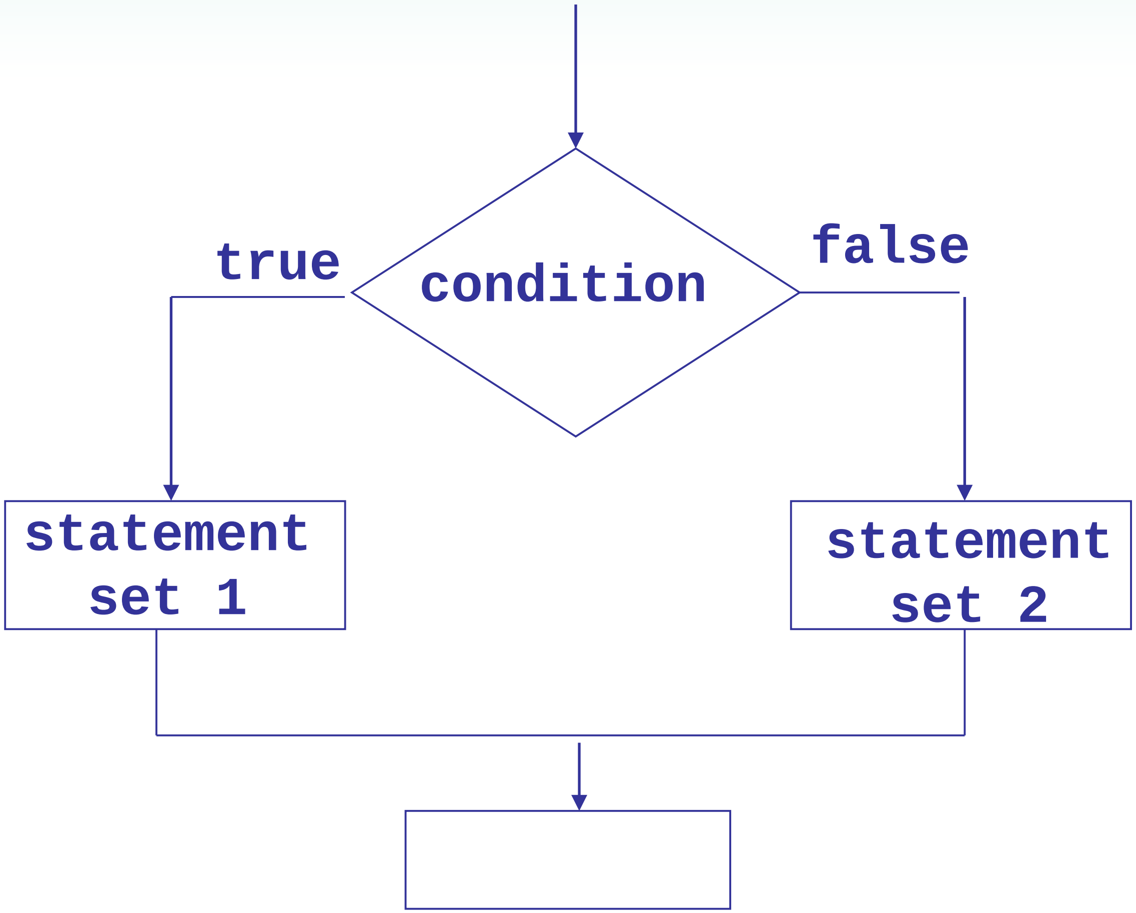
```
if (condition)
{
    statement set 1;
}
else
{
    statement set 2;
}
```

# How the **`if/else`** Works

- If **`(`*`condition`*`)`** is **`true`**, ***`statement set 1`*** is executed and ***`statement set 2`*** is skipped.

- If **`(`*`condition`*`)`** is **`false`**, ***`statement set 1`*** is skipped and ***`statement set 2`*** is executed.

# `if/else` Flow of Control



# Example `if/else` Statements

```
if (score >= 60)
   cout << "You passed.\n";
else
   cout << "You did not pass.\n";

if (intRate > 0)
{  interest = loanAmt * intRate;
   cout << interest;
}
else
   cout << "You owe no interest.\n";
```

# Comparisons with floating-point numbers

- It is difficult to test for equality when working with floating point numbers.
- It is better to use
  - greater than, less than tests, or
  - test to see if value is very close to a given value

# 4.4 The `if/else if` Statement

- Chain of **`if`** statements that test in order until one is found to be true
- Also models thought processes

“If it is raining, take an umbrella,
else, if it is windy, take a hat,
else, if it is sunny, take sunglasses.”

# `if/else if` Format

```
if (condition 1)
{   statement set 1;
}
else if (condition 2)
{   statement set 2;
}
        …
else if (condition n)
{   statement set n;
}
```

# Using a Trailing **`else`**

- Used with **`if/else if`** statement when all of the conditions are false
- Provides a default statement or action that is performed when none of the conditions is true
- Can be used to catch invalid values or handle other exceptional situations

# Example **if/else if** with Trailing **else**

```
if (age >= 21)
   cout << "Adult";
else if (age >= 13)
   cout << "Teen";
else if (age >= 2)
   cout << "Child";
else
   cout << "Baby";
```

# 4.5 Menu-Driven Program

- Menu: list of choices presented to the user on the computer screen
- Menu-driven program: program execution controlled by user selecting from a list of actions
- Menu can be implemented using **`if/else if`** statements

# Menu-driven Program Organization

- Display list of numbered or lettered choices for actions.
- Input user's selection of number or letter
- Test user selection in **(*condition*)**
  - if a match, then execute code to carry out desired action
  - if not, then test with next **(*condition*)**

# 4.6 Nested `if` Statements

- An **`if`** statement that is part of the **`if`** or **`else`** part of another **`if`** statement
- Can be used to evaluate > 1 data item or condition

```
if (score < 100)
{
    if (score > 90)
        grade = 'A';
}
```

# Notes on Coding Nested **`if`**s

- An **`else`** matches the nearest previous **`if`** that does not have an **`else`**

```
if (score < 100)
   if (score > 90)
      grade = 'A';
   else ...  // goes with second if,
             // not first one
```

- Proper indentation aids comprehension

# 4.7 Logical Operators

Used to create relational expressions from other relational expressions

| Operator | Meaning | Explanation |
|---|---|---|
| && | AND | New relational expression is true if both expressions are true |
| \|\| | OR | New relational expression is true if either expression is true |
| ! | NOT | Reverses the value of an expression; true expression becomes false, false expression becomes true |

# Logical Operator Examples

```
int x = 12, y = 5, z = -4;
```

| | |
|---|---|
| `(x > y) && (y > z)` | `true` |
| `(x > y) && (z > y)` | `false` |
| `(x <= z) \|\| (y == z)` | `false` |
| `(x <= z) \|\| (y != z)` | `true` |
| `!(x >= z)` | `false` |

# Logical Precedence

| | |
|---|---|
| Highest | **!** |
| | **&&** |
| Lowest | **\|\|** |

Example:

```
(2 < 3) || (5 > 6) && (7 > 8)
```

is true because AND is evaluated before OR

# More on Precedence

| | |
|---|---|
| Highest | arithmetic operators |
| ↓ | relational operators |
| Lowest | logical operators |

Example:

8 < 2 + 7 || 5 == 6 is true

# Checking Numeric Ranges with Logical Operators

- Used to test if a value is within a range

```
if (grade >= 0 && grade <= 100)
    cout << "Valid grade";
```

- Can also test if a value lies outside a range

```
if (grade <= 0 || grade >= 100)
    cout << "Invalid grade";
```

- Cannot use mathematical notation

```
if (0 <= grade <= 100) //Doesn’t
                       //work!
```

# 4.8 Validating User Input

- Input validation: inspecting input data to determine if it is acceptable
- Want to avoid accepting bad input
- Can perform various tests
  - Range
  - Reasonableness
  - Valid menu choice
  - Zero as a divisor

# 4.9 More About Blocks and Scope

- Scope of a variable is the block in which it is defined, from the point of definition to the end of the block
- Variables are usually defined at the beginning of a function
- They may instead be defined close to the place where they are first used

# More About Blocks and Scope

- Variables defined inside **{ }** have local or block scope
- When in a block that is nested inside another block, you can define variables with the same name as in the outer block.
  - When the program is executing in the inner block, the outer definition is not available
  - This is generally not a good idea

# 4.10 More About Characters and Strings

- Can use relational operators with characters and string objects

```
if (menuChoice == 'A')
if (firstName == "Beth")
```

- Comparing characters is really comparing ASCII values of characters
- Comparing string objects is comparing the ASCII values of the characters in the strings. Comparison is character-by-character
- Cannot compare C-style strings with relational operators

# Testing Characters

require **`cctype`** header file

| FUNCTION | MEANING |
|---|---|
| **`isalpha`** | **`true`** if arg. is a letter, **`false`** otherwise |
| **`isalnum`** | **`true`** if arg. is a letter or digit, **`false`** otherwise |
| **`isdigit`** | **`true`** if arg. is a digit 0-9, **`false`** otherwise |
| **`islower`** | **`true`** if arg. is lowercase letter, **`false`** otherwise |

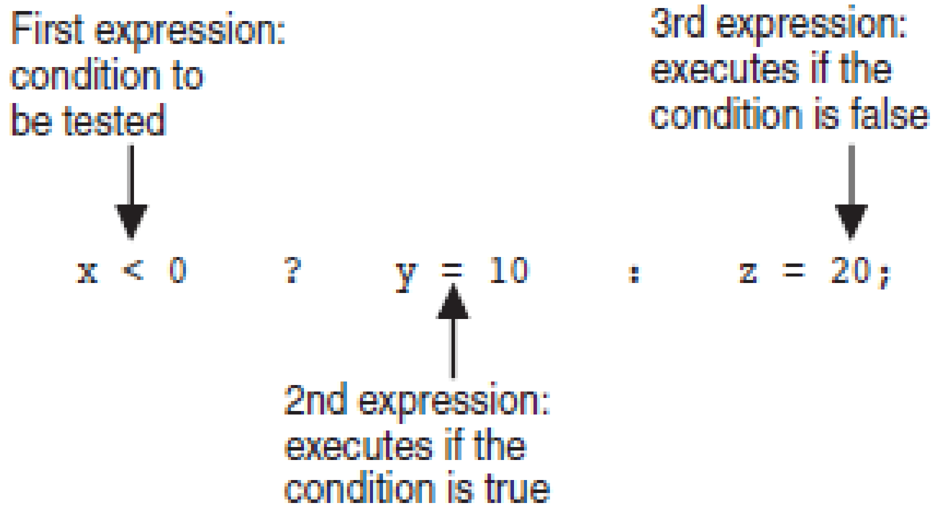
# Character Testing

require **cctype** header file

| FUNCTION | MEANING |
|---|---|
| **isprint** | **true** if arg. is a printable character, **false** otherwise |
| **ispunct** | **true** if arg. is a punctuation character, **false** otherwise |
| **isupper** | **true** if arg. is an uppercase letter, **false** otherwise |
| **isspace** | **true** if arg. is a whitespace character, **false** otherwise |

# 4.11 The Conditional Operator

- Can use to create short **`if/else`** statements
- Format: **`expr ? expr : expr;`**



# 4.12 The **`switch`** Statement

- Used to select among statements from several alternatives
- May sometimes be used instead of **`if/else if`** statements

# **switch** Statement Format

```
switch (IntExpression)
{
  case exp1: statement set 1;
  case exp2: statement set 2;
  ...
  case expn: statement set n;
  default:   statement set n+1;
}
```

# `switch` Statement Requirements

1) ***`IntExpression`*** must be a **`char`** or an integer variable or an expression that evaluates to an integer value
2) ***`exp1`*** through ***`expn`*** must be constant integer type expressions and must be unique in the **`switch`** statement
3) **`default`** is optional but recommended

# How the **`switch`** Statement Works

1) ***`IntExpression`*** is evaluated

2) The value of ***`intExpression`*** is compared against ***`exp1`*** through ***`expn`***.

3) If ***`IntExpression`*** matches value ***`expi`***, the program branches to the statement(s) following ***`expi`*** and continues to the end of the **`switch`**

4) If no matching value is found, the program branches to the statement after **`default:`**

# The **`break`** Statement

- Used to stop execution in the current block
- Also used to exit a **`switch`** statement
- Useful to execute a single **`case`** statement without executing statements following it

# Example **switch** Statement

```
switch (gender)
{
  case 'f': cout << "female";
            break;
  case 'm': cout << "male";
            break;
  default : cout << "invalid gender";
}
```

# Using **`switch`** with a Menu

**`switch`** statement is a natural choice for menu-driven program

- display menu
- get user input
- use user input as **`IntExpression`** in **`switch`** statement
- use menu choices as **`exp`** to test against in the **`case`** statements

# 4.13 Enumerated Data Types

- Data type created by programmer
- Contains a set of named constant integers
- Format:

  **enum *name* {*val1*, *val2*, … *valn*};**
- Examples:

```
enum Fruit {apple, grape, orange};
enum Days {Mon, Tue, Wed, Thur, Fri};
```

# Enumerated Data Type Variables

- To define variables, use the enumerated data type name

```
Fruit snack;
Days workDay, vacationDay;
```

- Variable may contain any valid value for the data type

```
snack = orange;      // no quotes
if (workDay == Wed) // none here
```

# Enumerated Data Type Values

- Enumerated data type values are associated with integers, starting at 0

```
enum Fruit {apple, grape, orange};
                ↑       ↑       ↑
                0       1       2
```

- Can override default association

```
enum Fruit {apple = 2, grape = 4,
            orange = 5}
```

# Enumerated Data Type Notes

- Enumerated data types improve the readability of a program
- Enumerated variables can not be used with input statements, such as **`cin`**
- Will not display the name associated with the value of an enumerated data type if used with **`cout`**

# Chapter 5:  Looping

# Topics

5.1 Introduction to Loops: The **`while`** Loop

5.2 Using the **`while`** loop for Input Validation

5.3 The Increment and Decrement Operators

5.4 Counters

5.5 The **`do-while`** loop

5.6 The **`for`** loop

5.7 Keeping a Running Total

# Topics (continued)

5.8 Sentinels

5.9 Deciding Which Loop to Use

5.10 Nested Loops

5.11 Breaking Out of a Loop

5.12 Using Files for Data Storage

5.13 Creating Good Test Data

# 5.1 Introduction to Loops: The `while` Loop

- Loop: part of program that may execute > 1 time (*i.e.*, it repeats)
- **`while`** loop format:

```
while (condition)
{  statement(s);
}
```

No ; here

- The **`{}`** can be omitted if there is only one statement in the body of the loop

# How the **`while`** Loop Works

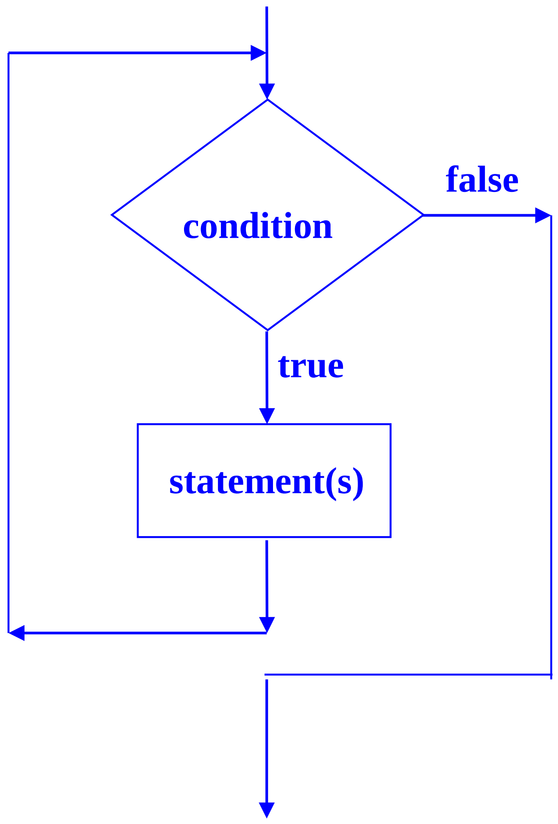
```
while (condition)
{   statement(s);
}
```

***condition*** is evaluated

- if it is true, the ***statement(s)*** are executed, and then ***condition*** is evaluated again
- if it is false, the loop is exited

An iteration is an execution of the loop body

# `while` Loop Flow of Control



# **while** Loop Example

```
int val = 5;
while (val >= 0)
{   cout << val << "  ";
    val = val - 1;
}
```

- produces output:

```
5  4  3  2  1  0
```

# **`while`** Loop is a Pretest Loop

- **`while`** is a pretest loop (***`condition`*** is evaluated before the loop executes)
- If the condition is initially false, the statement(s) in the body of the loop are never executed
- If the condition is initially true, the statement(s) in the body will continue to be executed until the condition becomes false

# Exiting the Loop

- The loop must contain code to allow ***`condition`*** to eventually become **`false`** so the loop can be exited
- Otherwise, you have an infinite loop (*i.e.*, a loop that does not stop)
- Example infinite loop:

```
x = 5;
while (x > 0)       // infinite loop because
    cout << x;      // x is always > 0
```

# Common Loop Errors

- Don’t put **;** immediately after ***(condition)***
- Don’t forget the { } :

```
int numEntries = 1;
while (numEntries <=3)
    cout << "Still working … ";
    numEntries++; // not in the loop body
```

- Don’t use = when you mean to use ==

```
while (numEntries = 3)  // always true
{
    cout << "Still working … ";
    numEntries++;
}
```

# `while` Loop Programming Style

- Loop body statements should be indented

- Align { and } with the loop header and place them on lines by themselves

Note: The conventions above make the program more understandable by someone who is reading it.  They have no effect on how the the program compiles or executes.

# 5.2 Using the `while` Loop for Input Validation

Loops are an appropriate structure for validating user input data

1. Prompt for and read in the data.
2. Use a **`while`** loop to test if data is valid.
3. Enter the loop only if data is not valid.
4. Inside the loop, display error message and prompt the user to re-enter the data.
5. The loop will not be exited until the user enters valid data.

# Input Validation Loop Example

```
cout << "Enter a number (1-100) and"
     << " I will guess it. ";
cin  >> number;

while (number < 1 || number > 100)
{  cout << "Number must be between 1 and 100."
        << " Re-enter your number. ";
   cin  >> number;
}
// Code to use the valid number goes here.
```

# 5.3 The Increment and Decrement Operators

- **Increment – increase value in variable**

  `++` adds one to a variable

  `val++;` is the same as `val = val + 1;`

- **Decrement – reduce value in variable**

  `--` subtracts one from a variable

  `val--;` is the same as `val = val – 1;`

- can be used in prefix mode (before) or postfix mode (after) a variable

# Prefix Mode

- **`++val`** and **`--val`** increment or decrement the variable, *then* return the new value of the variable.
- It is this returned new value of the variable that is used in any other operations within the same statement

# Prefix Mode Example

```
int x = 1, y = 1;

x = ++y;          // y is incremented to 2
                  // Then 2 is assigned to x
cout << x
  << "  " << y;  // Displays 2  2

x = --y;          // y is decremented to 1
                  // Then 1 is assigned to x
cout << x
  << "  " << y;  // Displays 1 1
```

# Postfix Mode

- **`val++`** and **`val--`** return the old value of the variable, *then* increment or decrement the variable
- It is this returned old value of the variable that is used in any other operations within the same statement

# Postfix Mode Example

```
int x = 1, y = 1;

x = y++;            // y++ returns a 1
                    // The 1 is assigned to x
                    // and y is incremented to 2
cout << x
  << "  " << y; // Displays 1  2

x = y--;            // y-- returns a 2
                    // The 2 is assigned to x
                    // and y is decremented to 1
cout << x
  << "  " << y; // Displays 2 1
```

# Increment & Decrement Notes

- Can be used in arithmetic expressions

```
result = num1++ + --num2;
```

- Must be applied to something that has a location in memory. Cannot have

```
result = (num1 + num2)++; // Illegal
```

- Can be used in relational expressions

```
if (++num > limit)
```

- Pre- and post-operations will cause different comparisons

# 5.4 Counters

- Counter: variable that is incremented or decremented each time a loop repeats
- Can be used to control execution of the loop (loop control variable)
- Must be initialized before entering loop
- May be incremented/decremented either inside the loop or in the loop test

# Letting the User Control the Loop

- Program can be written so that user input determines loop repetition
- Can be used when program processes a list of items, and user knows the number of items
- User is prompted before loop.  Their input is used to control number of repetitions

# User Controls the Loop Example

```
int num, limit;
cout << "Table of squares\n";
cout << "How high to go? ";
cin  >> limit;
cout << "\n\nnumber square\n";
num = 1;
while (num <= limit)
{  cout << setw(5) << num << setw(6)
        << num*num << endl;
   num++;
}
```

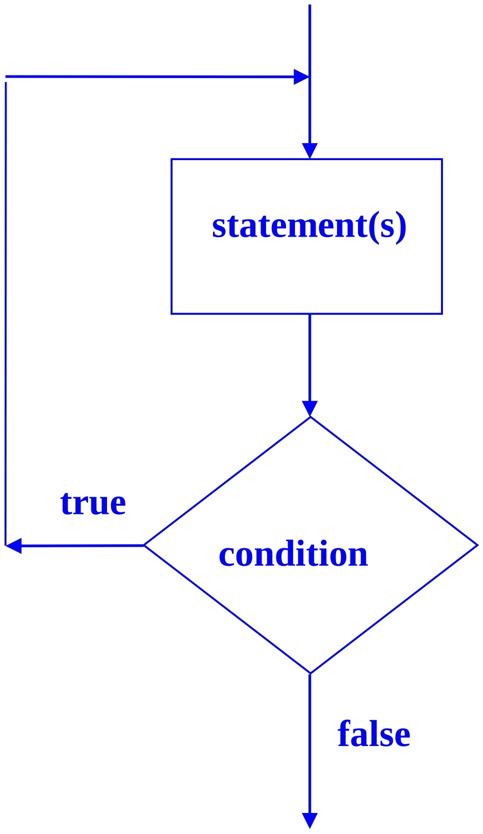
# 5.5 The **`do-while`** Loop

- **`do-while`**: a post test loop (***`condition`*** is evaluated <u>after</u> the loop executes)
- Format:

```
do
{   1 or more statements;
} while (condition);
```

Notice the required ;

# **`do-while`** Flow of Control



# **`do-while`** Loop Notes

- Loop always executes at least once
- Execution continues as long as ***`condition`*** is **`true`**; the loop is exited when ***`condition`*** becomes **`false`**
- { } are required, even if the body contains a single statement
- **`;`** after ***`(condition)`*** is also required

# **do-while** and Menu-Driven Programs

- do-while can be used in a menu-driven program to bring the user back to the menu to make another choice
- To simplify the processing of user input, use the **toupper** (‘to upper’) or **tolower** (to lower’) function

# Menu-Driven Program Example

```
do {
   // code to display menu
   // and perform actions
   cout << "Another choice? (Y/N) ";
} while (choice =='Y'||choice =='y');
```

The condition could be written as

```
    (toupper(choice) == 'Y');
```

or as

```
    (tolower(choice) == 'y');
```

# 5.6 The **`for`** Loop

- Pretest loop that executes zero or more times
- Useful for counter-controlled loop
- Format:

```
for( initialization; test; update )
{   1 or more statements;
}
```

Required ;

No ; goes here

# `for` Loop Mechanics

**Step 1:** Perform the initialization expression.

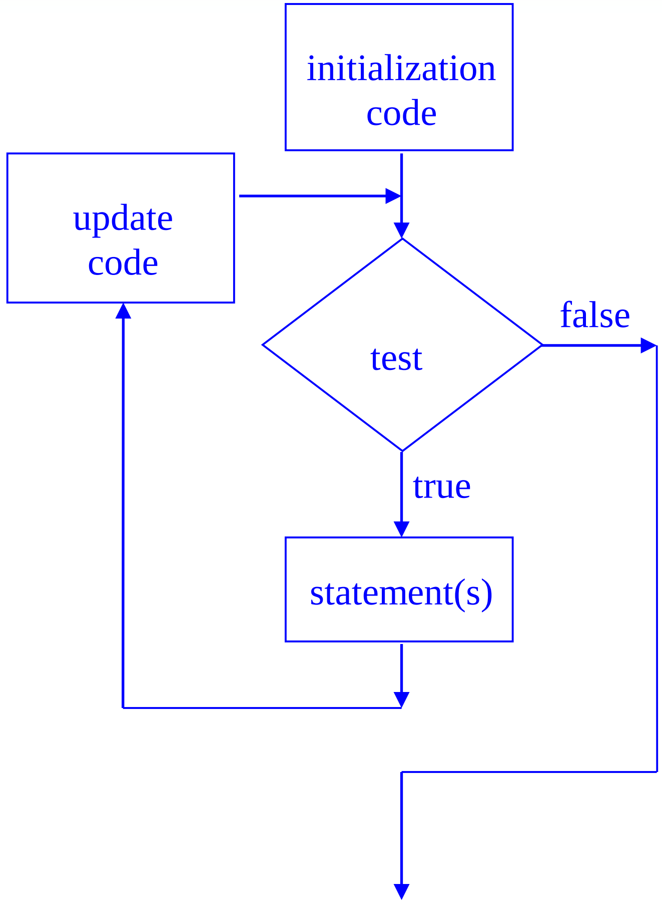
**Step 2:** Evaluate the test expression.
If it is true, go to step 3.
Otherwise, terminate the loop.

```
for (count = 1; count <= 5; count++)
{  cout << "Hello" << endl;
}
```

**Step 3:** Execute the body
of the loop.

**Step 4:** Perform the update expression.
Then go back to step 2.

# **`for`** Loop Flow of Control



# `for` Loop Example

```
int sum = 0, num;
for (num = 1; num <= 10; num++)
   sum += num;
cout << "Sum of numbers 1 – 10 is "
       << sum << endl;
```

# **`for`** Loop Notes

- If ***`test`*** is false the first time it is evaluated, the body of the loop will not be executed
- The update expression can increment or decrement by any amount
- Variables used in the initialization section should not be modified in the body of the loop

# **for** Loop Modifications

- Can define variables in initialization code
  - Their scope is the **for** loop
- Initialization and update code can contain more than one statement
  - Separate the statements with commas
- Example:

```
for (int sum = 0, num = 1; num <= 10; num++)
    sum += num;
```

# More **`for`** Loop Modifications
## (These are NOT Recommended)

- Can omit ***`initialization`*** if already done

```
int sum = 0, num = 1;
for (; num <= 10; num++)
   sum += num;
```

- Can omit ***`update`*** if done in loop

```
for (sum = 0, num = 1; num <= 10;)
   sum += num++;
```

- Can omit ***`test`*** – may cause an infinite loop

```
for (sum = 0, num = 1; ; num++)
    sum += num;
```

- Can omit loop body if all work is done in header

# 5.7 Keeping a Running Total

- running total: accumulated sum of numbers from each repetition of loop
- accumulator: variable that holds running total

```
int sum = 0, num = 1; // sum is the
while (num <= 10)     // accumulator
{   sum += num;
    num++;
}
cout << "Sum of numbers 1 – 10 is "
     << sum << endl;
```

# 5.8 Sentinels

- sentinel: value in a list of values that indicates end of the list
- Special value that cannot be confused with a valid value, *e.g.*, **`-999`** for a test score
- Used to terminate input when user may not know how many values will be entered

# Sentinel Example

```
int total = 0;
cout << "Enter points earned "
     << "(or -1 to quit): ";
cin  >> points;
while (points != -1) // -1 is the sentinel
{
   total += points;
   cout << "Enter points earned: ";
   cin  >> points;
}
```

# 5.9 Deciding Which Loop to Use

- **`while`**: pretest loop (loop body may not be executed at all)
- **`do-while`**: post test loop (loop body will always be executed at least once)
- **`for`**: pretest loop (loop body may not be executed at all); has initialization and update code; is useful with counters or if precise number of repetitions is known

# 5.10 Nested Loops

- A nested loop is a loop inside the body of another loop
- Example:

```
for (row=1; row<=3; row++)          outer loop
{
  for (col=1; col<=3; col++)        inner loop
  {
    cout << row * col << endl;
  }
}
```

# Notes on Nested Loops

- Inner loop goes through all its repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops.  In previous example, inner loop repeats 9 times

# 5.11 Breaking Out of a Loop

- Can use **`break`** to terminate execution of a loop
- Use sparingly if at all – makes code harder to understand
- When used in an inner loop, terminates that loop only and returns to the outer loop

# The **`continue`** Statement

- Can use **`continue`** to go to end of loop and prepare for next repetition
  - **`while`** and **`do-while`** loops go to test and repeat the loop if test condition is true
  - **`for`** loop goes to update step, then tests, and repeats loop if test condition is true
- Use sparingly – like **`break`**, can make program logic hard to follow

# 5.12 Using Files for Data Storage

- We can use a file instead of monitor screen for program output
- Files are stored on secondary storage media, such as disk
- Files allow data to be retained between program executions
- We can later use the file instead of a keyboard for program input

# File Types

- Text file – contains information encoded as text, such as letters, digits, and punctuation. Can be viewed with a text editor such as Notepad.
- Binary file – contains binary (0s and 1s) information that has not been encoded as text. It cannot be viewed with a text editor.

# File Access – Ways to Use the Data in a File

- Sequential access – read the 1st piece of data, read the 2nd piece of data, …, read the last piece of data.  To access the n-th piece of data, you have to retrieve the preceding n pieces first.
- Random (direct) access – retrieve any piece of data directly, without the need to retrieve preceding data items.

# What is Needed to Use Files

1. Include the **`fstream`** header file
2. Define a file stream object
   - **`ifstream`** for input from a file

     ```
     ifstream inFile;
     ```
   - **`ofstream`** for output to a file

     ```
     ofstream outFile;
     ```

# Open the File

3. Open the file

- Use the **open** member function

```
inFile.open("inventory.dat");
outFile.open("report.txt");
```

- Filename may include drive, path info.
- Output file will be created if necessary; existing output file will be erased first
- Input file must exist for **open** to work

# Use the File

4. Use the file

- Can use output file object and **<<** to send data to a file

```
outFile << "Inventory report";
```

- Can use input file object and **>>** to copy data from file to variables

```
inFile >> partNum;
inFile >> qtyInStock >> qtyOnOrder;
```

# Close the File

5. Close the file

- Use the **`close`** member function

```
inFile.close();
outFile.close();
```

Can use a file instead of keyboard for program input

- Don’t wait for operating system to close files at program end
  - There may be limit on number of open files
  - There may be buffered output data waiting to be sent to a file that could be lost

# Input File – the Read Position

- Read Position – location of the next piece of data in an input file

- Initially set to the first byte in the file

- Advances for each data item that is read. Successive reads will retrieve successive data items.

# Using  Loops to Process Files

- A loop can be used to read data from or write data to a file
- It is not necessary to know how much data is in the file or will be written to the file
- Several methods exist to test for the end of the file

# Using the **>>** Operator to Test for End of File (EOF) on an Input File

- The stream extraction operator (**>>**) returns a true or false value indicating if a read is successful
- This can be tested to find the end of file since the read “fails” when there is no more data
- Example:

```
while (inFile >> score)
   sum += score;
```

# File Open Errors

- An error will occur if an attempt to open a file for input fails:
  - File does not exist
  - Filename is misspelled
  - File exists, but is in a different place
- The file stream object is set to true if the open operation succeeded. It can be tested to see if the file can be used:

```
if (inFile)
{
  // process data from file
}  else
   cout << "Error on file open\n";
```

# User-Specified Filenames

- Program can prompt user to enter the names of input and/or output files. This makes the program more versatile.
- Filenames can be read into string objects. The C-string representation of the string object can then be passed to the open function:

```
cout << "Which input file? ";
cin >> inputFileName;
inFile.open(inputFileName.c_str());
```

# 5.13 Creating Good Test Data

- When testing a program, the quality of the test data is more important than the quantity.
- Test data should show how different parts of the program execute
- Test data should evaluate how program handles:
  - normal data
  - data that is at the limits the valid range
  - invalid data

See pr5-09.cpp

# Chapter 6:  Functions

# Topics

6.1 Modular Programming
6.2 Defining and Calling Functions
6.3 Function Prototypes
6.4 Sending Data into a Function
6.5 Passing Data by Value
6.6 The **`return`** Statement
6.7 Returning a Value from a Function
6.8 Returning a Boolean Value

# Topics (continued)

6.9 Using Functions in a Menu-Driven Program

6.10 Local and Global Variables

6.11 Static Local Variables

6.12 Default Arguments

6.13 Using Reference Variables as Parameters

6.14 Overloading Functions

6.15 The **`exit()`** Function

6.16 Stubs and Drivers

# 6.1 Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules.  Supports the divide-and-conquer approach to solving a problem.
- Function: a collection of statements to perform a specific task
- Motivation for modular programming
  - Simplifies the process of writing programs
  - Improves maintainability of programs

# 6.2 Defining and Calling Functions

- Function call: a statement that causes a function to execute
- Function definition: the statements that make up a function

# Function Definition

- Definition includes

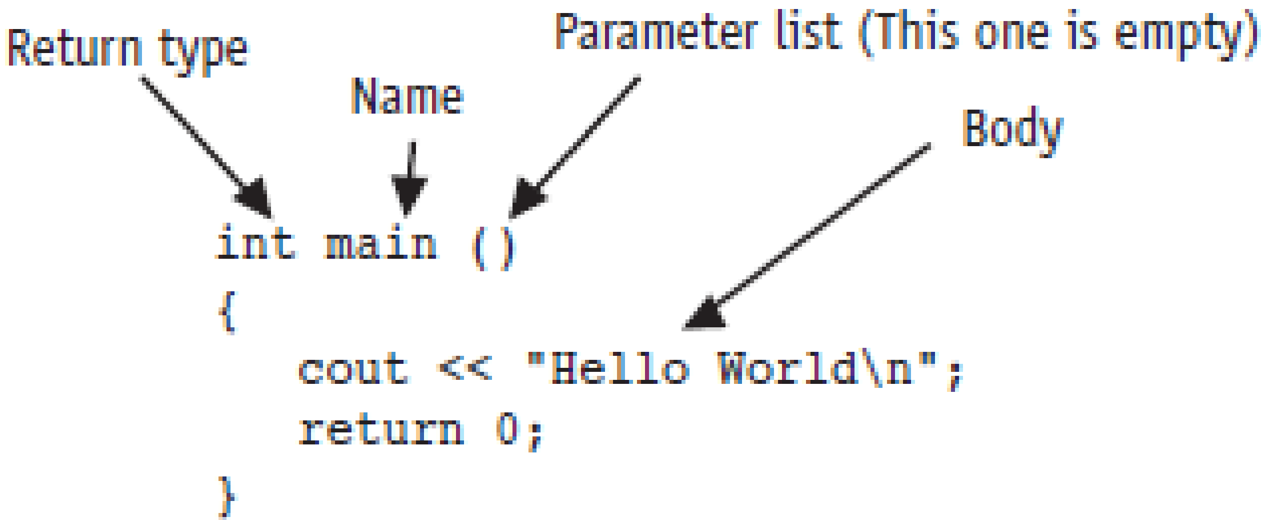
  name: name of the function. Function names follow same rules as variable names

  parameter list: variables that hold the values passed to the function

  body: statements that perform the function's task

  return type: data type of the value the function returns to the part of the program that called it

# Function Definition



# Function Header

- The function header consists of
  - the function *return type*
  - the function *name*
  - the function *parameter list*
- Example:

```
int main()
```

- Note: no `;` at the end of the header

# Function Return Type

- If a function returns a value, the type of the value must be indicated

```
int main()
```

- If a function does not return a value, its return type is **void**

```
void printHeading()
{
    cout << "\tMonthly Sales\n";
}
```

# Calling a Function

- To call a function, use the function name followed by `()` and `;`

```
printHeading();
```

- When a function is called, the program executes the body of the function
- After the function terminates, execution resumes in the calling module at the point of call

# Calling a Function

- **`main`** is automatically called when the program starts
- **`main`** can call any number of functions
- Functions can call other functions

# 6.3 Function Prototypes

The compiler must know the following about a function before it is called

- name
- return type
- number of parameters
- data type of each parameter

# Function Prototypes

Ways to notify the compiler about a function before a call to the function:

- Place function definition before calling function's definition
- Use a function prototype (similar to the heading of the function
  - Heading: `void printHeading()`
  - Prototype: `void printHeading();`
- Function prototype is also called a function declaration

# Prototype Notes

- Place prototypes near top of program
- Program must include either prototype or full function definition before any call to the function, otherwise a compiler error occurs
- When using prototypes, function definitions can be placed in any order in the source file.  Traditionally, **`main`** is placed first.

# 6.4 Sending Data into a Function

- Can pass values into a function at time of call

```
c = sqrt(a*a + b*b);
```

- Values passed to function are arguments
- Variables in function that hold values passed as arguments are parameters
- Alternate names:
  - argument: actual argument, actual parameter
  - parameter: formal argument, formal parameter

# Parameters, Prototypes, and Function Headings

- For each function argument,
  - the prototype must include the data type of each parameter in its **()**

```
void evenOrOdd(int);        //prototype
```

  - the heading must include a declaration, with variable type and name, for each parameter in its **()**

```
void evenOrOdd(int num)     //heading
```

- The function call for the above function would look like this: `evenOrOdd(val);  //call`

  Note: no data type on argument in call

# Function Call Notes

- Value of argument is copied into parameter when the function is called
- Function can have > 1 parameter
- There must be a data type listed in the prototype **( )** and an argument declaration in the function heading **( )** for each parameter
- Arguments will be promoted/demoted as necessary to match parameters.  Be careful!

# Calling Functions with Multiple Arguments

When calling a function with multiple arguments

- the number of arguments in the call must match the function prototype and definition
- the first argument will be copied into the first parameter, the second argument into the second parameter, etc.

# Calling Functions with Multiple Arguments Illustration

```
displayData(height, weight);  // call


void displayData(int h, int w)// heading
{
   cout << "Height = " << h << endl;
   cout << "Weight = " << w << endl;
}
```

# 6.5 Passing Data by Value

- Pass by value: when an argument is passed to a function, a copy of its value is placed in the parameter
- The function cannot access the original argument
- Changes to the parameter in the function do not affect the value of the argument in the calling function

# Passing Data to Parameters by Value

- Example: `int val = 5;`
  `evenOrOdd(val);`


- **evenOrOdd** can change variable **num**, but it will have no effect on variable **val**

# 6.6 The `return` Statement

- Used to end execution of a function
- Can be placed anywhere in a function
  - Statements that follow the `return` statement will not be executed
- Can be used to prevent abnormal termination of program
- Without a `return` statement, the function ends at its last `}`

# 6.7 Returning a Value from a Function

- **`return`** statement can be used to return a value from the function to the module that made the function call
- Prototype and definition must indicate data type of return value (not **`void`**)
- Calling function should use return value, *e.g.*,
  - assign it to a variable
  - send it to **`cout`**
  - use it in an arithmetic computation
  - use it in a relational expression

# Returning a Value – the `return` Statement

- Format: `return` ***`expression;`***
- ***`expression`*** may be a variable, a literal value, or an expression.
- ***`expression`*** should be of the same data type as the declared return type of the function (will be converted if not)

# 6.8 Returning a Boolean Value

- Function can return **`true`** or **`false`**
- Declare the return type in the function prototype and heading as **`bool`**
- The function body must contain **`return`** statement(s) that return **`true`** or **`false`**
- The calling function can use the return value in a relational expression

# Boolean **return** Example

```
bool isValid(int);            // prototype

bool isValid(int val)         // heading
{
   int min = 0, max = 100;
   if (val >= min && val <= max)
      return true;
   else
      return false;
}

if (isValid(score))           // call
   …
```

# 6.9 Using Functions in a Menu-Driven Program

Functions can be used

- to implement user choices from menu
- to implement general-purpose tasks
  - Higher-level functions can call general-purpose functions
  - This minimizes the total number of functions and speeds program development time

# 6.10 Local and Global Variables

- local variable: defined within a function or block; accessible only within the function or block
- Other functions and blocks can define variables with the same name
- When a function is called, local variables in the calling function are not accessible from within the called function

# Local Variable Lifetime

- A local variable only exists while its defining function is executing
- Local variables are destroyed when the function terminates
- Data cannot be retained in local variables between calls to the function in which they are defined

# Local and Global Variables

- global variable: a variable defined outside all functions; it is accessible to all functions within its scope
- Easy way to share large amounts of data between functions
- Scope of a global variable is from its point of definition to the program end
- Use sparingly

# Initializing Local and Global Variables

- Local variables must be initialized by the programmer
- Global variables are initialized to **`0`** (numeric) or **`NULL`** (character) when the variable is defined.  These can be overridden with explicit initial values.

# Global Variables – Why Use Sparingly?

Global variables make:

- Programs that are difficult to debug
- Functions that cannot easily be re-used in other programs
- Programs that are hard to understand

# Global Constants

- A global constant is a named constant that can be used by every function in a program
- It is useful if there are unchanging values that are used throughout the program
- They are safer to use than global variables, since the value of a constant cannot be modified during program execution

# Local and Global Variable Names

- Local variables can have same names as global variables
- When a function contains a local variable that has the same name as a global variable, the global variable is unavailable from within the function. The local definition "hides" or "shadows" the global definition.

# 6.11 Static Local Variables

- Local variables
  - Only exist while the function is executing
  - Are defined each time function is called
  - Lose their contents when function terminates
- **`static`** local variables
  - Are defined with key word **`static`**

    ```
    static int counter;
    ```
  - Are defined and initialized only the first time the function is executed
  - Retain their contents between function calls

# 6.12 Default Arguments

- Values passed automatically if arguments are missing from the function call
- Must be a constant declared in prototype or header (whichever occurs first)

```
void evenOrOdd(int = 0);
```

- Multi-parameter functions may have default arguments for some or all parameters

```
int getSum(int, int=0, int=0);
```

# Default Arguments

- If not all parameters to a function have default values, the ones without defaults must be declared first in the parameter list

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // wrong!
```

- When an argument is omitted from a function call, all arguments after it must also be omitted

```
sum = getSum(num1, num2);    // OK
sum = getSum(num1, , num3);  // wrong!
```

# 6.13 Using Reference Variables as Parameters

- Mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to ‘return’ more than 1 value

# Reference Variables

- A reference variable is an alias for another variable
- It is defined with an ampersand (**&**) in the prototype and in the header

```
void getDimensions(int&, int&);
```

- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters by reference

# Pass by Reference Example

```
void squareIt(int &); //prototype
void squareIt(int &num)
{
    num *= num;
}

int localVar = 5;
squareIt(localVar);  // localVar now
                     // contains 25
```

# Reference Variable Notes

- Each reference parameter must contain **&**
- Argument passed to reference parameter must be a variable.  It cannot be an expression or a constant.
- Use only when appropriate, such as when the function must input or change the value of the argument passed to it
- Files (*i.e.*, file stream objects) should be passed by reference

# 6.14 Overloading Functions

- Overloaded functions are two or more functions that have the same name, but different parameter lists
- Can be used to create functions that perform the same task, but take different parameter types or different number of parameters
- Compiler will determine which version of the function to call by the argument and parameter list

# Overloaded Functions Example

If a program has these overloaded functions,

```
void getDimensions(int);              // 1
void getDimensions(int, int);         // 2
void getDimensions(int, float);       // 3
void getDimensions(double, double);// 4
```

then the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);                // 1
getDimensions(length, width);         // 2
getDimensions(length, height);        // 3
getDimensions(height, base);          // 4
```

# 6.15 The `exit()` Function

- Terminates execution of a program
- Can be called from any function
- Can pass a value to operating system to indicate status of program execution
- Usually used for abnormal termination of program
- Requires **`cstdlib`** header file
- Use with care

# **exit()** – Passing Values to Operating System

- Use an integer value to indicate program status
- Often, 0 means successful completion, non-zero indicates a failure condition
- Can use named constants defined in **cstdlib**:
  - **EXIT_SUCCESS** and
  - **EXIT_FAILURE**

# 6.16 Stubs and Drivers

- Stub: dummy function in place of actual function
- Usually displays a message indicating it was called. May also display parameters
- Driver: function that tests a function by calling it
- Stubs and drivers are useful for testing and debugging program logic and design

# Chapter 7: Introduction to Classes and Objects

# Topics

7.1 Abstract Data Types
7.2 Object-Oriented Programming
7.3 Introduction to Classes
7.4 Creating and Using Objects
7.5 Defining Member Functions
7.6 Constructors
7.7 Destructors
7.8 Private Member Functions

# Topics (Continued)

7.9 Passing Objects to Functions
7.10 Object Composition
7.11 Separating Class Specification, Implementation, and Client Code
7.12 Structures
7.14 Introduction to Object-Oriented Analysis and Design
7.15 Screen Control

# 7.1 Abstract Data Types

- Programmer-created data types that specify
  - legal values that can be stored
  - operations that can be done on the values
- The user of an abstract data type (ADT) does not need to know any implementation details (*e.g.*, how the data is stored or how the operations on it are carried out)

# Abstraction in Software Development

- Abstraction allows a programmer to design a solution to a problem and to use data items without concern for how the data items are implemented
- This has already been encountered in the book:
  - To use the **pow** function, you need to know what inputs it expects and what kind of results it produces
  - You do not need to know how it works

# Abstraction and Data Types

- Abstraction: a definition that captures general characteristics without details

  ex: An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral

- Data Type: defines the kind of values that can be stored and the operations that can be performed on it

# 7.2 Object-Oriented Programming

- Procedural programming uses variables to store data, and focuses on the processes/ functions that occur in a program. Data and functions are separate and distinct.

- Object-oriented programming is based on objects that encapsulate the data and the functions that operate on it.

# Object-Oriented Programming Terminology

- **object**: software entity that combines data and functions that act on the data in a single unit
- **attributes**: the data items of an object, stored in **member variables**
- **member functions (methods)**: procedures/ functions that act on the attributes of the class

# More Object-Oriented Programming Terminology

- **data hiding**: restricting access to certain members of an object. The intent is to allow only member functions to directly access and modify the object's data
- **encapsulation**: the bundling of an object's data and procedures into a single entity

# Object Example

Square

Member variables (attributes)

```
int side;
```

Member functions

```
void setSide(int s)
{  side = s;    }

int getSide()
{  return side; }
```

Square object's data item: **`side`**

Square object's functions: **`setSide`** - set the size of the side of the square, **`getSide`** - return the size of the side of the square

# Why Hide Data?

- Protection – Member functions provide a layer of protection against inadvertent or deliberate data corruption
- Need-to-know – A programmer can use the data via the provided member functions.  As long as the member functions return correct information, the programmer needn’t worry about implementation details.

# 7.3 Introduction to Classes

- **Class**: a programmer-defined data type used to define objects
- It is a pattern for creating objects

  ex:

  ```
  string fName, lName;
  ```

  creates two objects of the **`string`** class

# Introduction to Classes

- Class declaration format:

```
class className
{
    declaration;
    declaration;
};
```

Notice the required ;

# Access Specifiers

- Used to control access to members of the class.
- Each member is declared to be either

  **`public`**: can be accessed by functions outside of the class

  or

  **`private`**: can only be called by or accessed by functions that are members of the class

# Class Example


```
class Square
{
  private:
    int side;
  public:
    void setSide(int s)
    { side = s; }
    int getSide()
    { return side; }
};
```

# More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is **`private`**

# 7.4 Creating and Using Objects

- An object is an instance of a class
- It is defined just like other variables

```
Square sq1, sq2;
```

- It can access members using dot operator

```
sq1.setSide(5);
cout << sq1.getSide();
```

# Types of Member Functions

- **Acessor, get, getter function**: uses but does not modify a member variable

  ex: **`getSide`**

- **Mutator, set, setter function**: modifies a member variable

  ex: **`setSide`**

# 7.5 Defining Member Functions

- Member functions are part of a class declaration
- Can place entire function definition inside the class declaration

  or
- Can place just the prototype inside the class declaration and write the function definition after the class

# Defining Member Functions Inside the Class Declaration

- Member functions defined inside the class declaration are called inline functions
- Only very short functions, like the one below, should be inline functions

```
int getSide()
{ return side; }
```

# Inline Member Function Example

```
class Square
{
  private:
    int side;
  public:
    void setSide(int s)
     { side = s; }
    int getSide()
     { return side; }
};
```

**inline functions** (callout pointing to `setSide` and `getSide`)

# Defining Member Functions After the Class Declaration

- Put a function prototype in the class declaration
- In the function definition, precede the function name with the class name and scope resolution operator (::)

```
int Square::getSide()
{
  return side;
}
```

# Conventions and a Suggestion

Conventions:

- Member variables are usually **`private`**
- Accessor and mutator functions are usually **`public`**
- Use ‘get’ in the name of accessor functions, ‘set’ in the name of mutator functions

Suggestion: calculate values to be returned in accessor functions when possible, to minimize the potential for stale data

# Tradeoffs of Inline vs. Regular Member Functions

- When a regular function is called, control passes to the called function
  - the compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into the program in place of the call when the program is compiled
  - This makes alarger executable program, but
  - There is less function call overhead, and possibly faster execution

# 7.6 Constructors

- A **constructor** is a member function that is often used to initialize data members of a class
- Is called automatically when an object of the class is created
- It must be a **`public`** member function
- It must be named the same as the class
- It must have no return type

# Constructor – 2 Examples

**Inline:**

```
class Square
{
  . . .
  public:
    Square(int s)
    { side = s; }
  . . .
};
```

**Declaration outside the class:**

```
Square(int);   //prototype
               //in class

Square::Square(int s)
{
   side = s;
}
```

# Overloading Constructors

- A class can have more than 1 constructor
- Overloaded constructors in a class must have different parameter lists

```
Square();
Square(int);
```

# The Default Constructor

- Constructors can have any number of parameters, including none
- A **default constructor** is one that takes no arguments either due to
  - No parameters or
  - All parameters have default values
- If a class has any programmer-defined constructors, it must have a programmer-defined default constructor

# Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square()            // default
    { side = 1; }  // constructor

    // Other member
    // functions go here
};
```

Has no parameters

# Another Default Constructor Example

```
class Square
{
  private:
    int side;

  public:
    Square(int s = 1) // default
    { side = s; }     // constructor

    // Other member
    // functions go here
};
```

Has parameter but it has a default value

# Invoking a Constructor

- To create an object using the default constructor, use no argument list and no **( )**

```
Square square1;
```

- To create an object using a constructor that has parameters, include an argument list

```
Square square1(8);
```

# 7.7 Destructors

- Is a public member function automatically called when an object is destroyed
- The destructor name is *~className*, *e.g.*,

  **`~Square`**

- It has no return type
- It takes no arguments
- Only 1 destructor is allowed per class

  (*i.e.*, it cannot be overloaded)

# 7. 8 Private Member Functions

- A **`private`** member function can only be called by another member function of the same class
- It is used for internal processing by the class, not for use outside of the class

# 7.9 Passing Objects to Functions

- A class object can be passed as an argument to a function
- When passed by value, function makes a local copy of object. Original object in calling environment is unaffected by actions in function
- When passed by reference, function can use ‘set’ functions to modify the object.

# Notes on Passing Objects

- Using a value parameter for an object can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the parameter

# Notes on Passing Objects

- To save space and time, while protecting parameter data that should not be changed, use a **`const`** reference parameter

```
void showData(const Square &s)
                              // header
```

- In order to for the showData function to call **`Square`** member functions, those functions must use **`const`** in their prototype and header:

```
int Square::getSide() const;
```

# Returning an Object from a Function

- A function can return an object

```
Square initSquare();   // prototype
s1 = initSquare();     // call
```

- The function must define a object
  - for internal use
  - to use with **`return`** statement

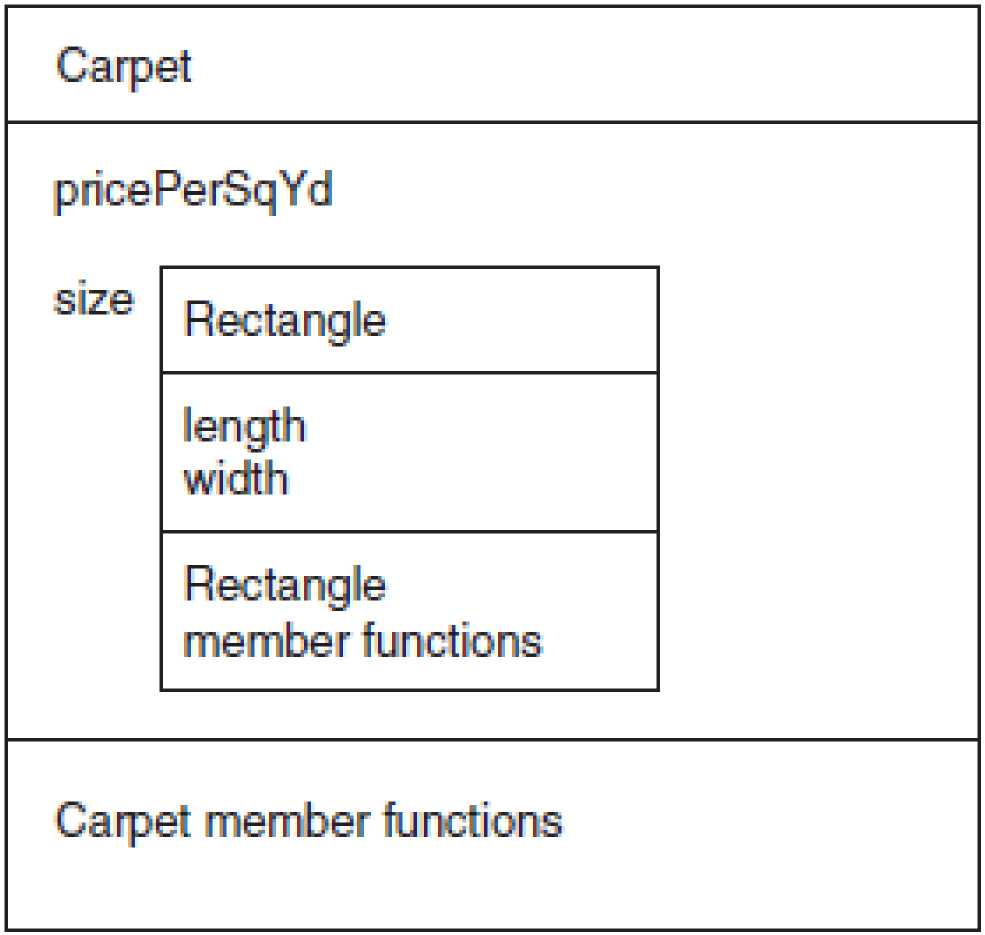
# Returning an Object Example

```
Square initSquare()
{
  Square s;    // local variable
  int inputSize;
  cout << "Enter the length of side: ";
  cin >> inputSize;
  s.setSide(inputSize);
  return s;
}
```

# 7.10 Object Composition

- Occurs when an object is a member variable of another object.
- It is often used to design complex objects whose members are simpler objects
- ex. (from book): Define a rectangle class. Then, define a carpet class and use a rectangle object as a member of a carpet object.

# Object Composition, cont.



# 7.11 Separating Class Specification, Implementation, and Client Code

Separating class declaration, member function definitions, and the program that uses the class into separate files is considered good design

# Using Separate Files

- Place class declaration in a header file that serves as the class specification file. Name the file ***classname*.h** (for example, **Square.h**)
- Place member function definitions in a class implementation file. Name the file ***classname.cpp*** (for example, **Square.cpp**)This file should **#include** the class specification file.
- A client program (client code) that uses the class must **#include** the class specification file and be compiled and linked with the class implementation file.

# Include Guards

- Used to prevent a header file from being included twice
- Format:

```
#ifndef symbol_name
#define symbol_name
. . .    (normal contents of header file)
#endif
```

- ***symbol_name*** is usually the name of the header file, in all capital letters:

```
#ifndef SQUARE_H
#define SQUARE_H
. . .
#endif
```

# What Should Be Done Inside *vs.* Outside the Class

- Class should be designed to provide functions to store and retrieve data
- In general, input and output (I/O) should be done by functions that use class objects, rather than by class member functions

# 7.12 Structures

- Structure: Programmer-defined data type that allows multiple variables to be grouped together
- Structure Declaration Format:

```
struct structure name
{
  type1 field1;
  type2 field2;
     …
  typen fieldn;
};
```

# Example **struct** Declaration

```
struct Student
{
   int studentID;
   string name;
   short year;
   double gpa;
};
```

structure name

structure members

Notice the required ;

# **`struct`** Declaration Notes

- **`struct`** names commonly begin with an uppercase letter
- The structure name is also called the tag
- Multiple fields of same type can be in a comma-separated list

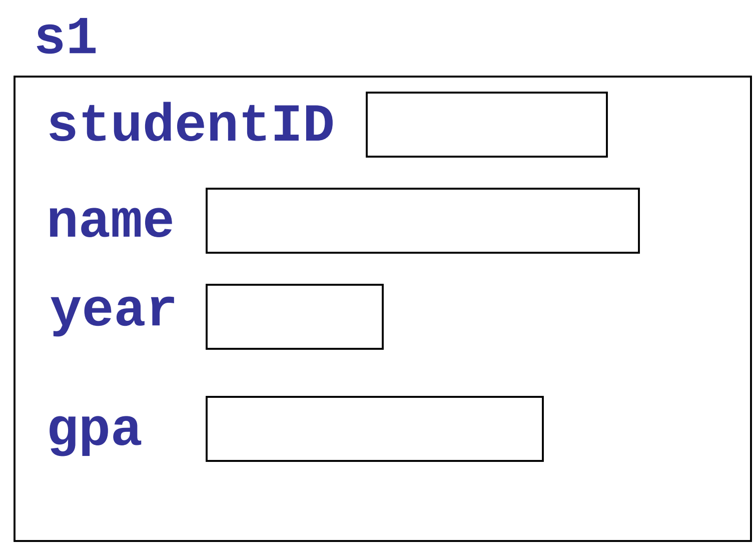
```
string name,
       address;
```

- Fields in a structure are all public by default

# Defining Structure Variables

- **`struct`** declaration does not allocate memory or create variables
- To define variables, use structure tag as type name

```
Student s1;
```



# Accessing Structure Members

- Use the dot **`(.)`** operator to refer to members of **`struct`** variables

```
getline(cin, s1.name);
cin >> s1.studentID;
s1.gpa = 3.75;
```

- Member variables can be used in any manner appropriate for their data type

# Displaying `struct` Members

To display the contents of a **`struct`** variable, you must display each field separately, using the dot operator

Wrong:

```
cout << s1; // won’t work!
```

Correct:

```
cout << s1.studentID << endl;
cout << s1.name << endl;
cout << s1.year << endl;
cout << s1.gpa;
```

# Comparing **`struct`** Members

- Similar to displaying a **`struct`**, you cannot compare two **`struct`** variables directly:

```
if (s1 >= s2) // won't work!
```

- Instead, compare member variables:

```
if (s1.gpa >= s2.gpa) // better
```

# Initializing a Structure

Cannot initialize members in the structure declaration, because no memory has been allocated yet

```
struct Student            // Illegal
{                         // initialization
   int studentID = 1145;
   string name = "Alex";
   short year = 1;
   float gpa = 2.95;
};
```

# Initializing a Structure (continued)

- Structure members are initialized at the time a structure variable is created
- Can initialize a structure variable's members with either
  - an initialization list
  - a constructor

# Using an Initialization List

An initialization list is an ordered set of values, separated by commas and contained in **{ }**, that provides initial values for a set of data members

```
{12, 6, 3}  // initialization list
            // with 3 values
```

# More on Initialization Lists

- Order of list elements matters: First value initializes first data member, second value initializes second data member, etc.
- Elements of an initialization list can be constants, variables, or expressions

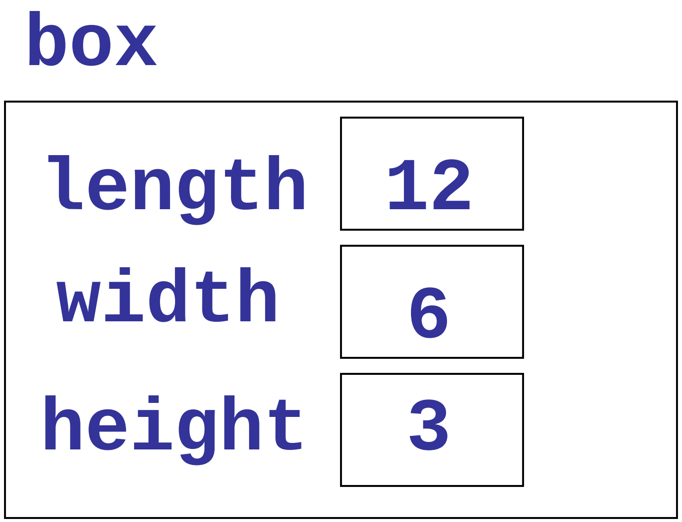
```
{12, W, L/W + 1} // initialization list
                 // with 3 items
```

# Initialization List Example

Structure Declaration

```
struct Dimensions
{ int length,
      width,
      height;
};
```

Structure Variable




```
Dimensions box = {12,6,3};
```

# Partial Initialization

- Can initialize just some members, but cannot skip over members

```
Dimensions box1 = {12,6};  //OK
Dimensions box2 = {12,,3}; //illegal
```

# Problems with Initialization List

- Can’t omit a value for a member without omitting values for all following members
- Does not work on most modern compilers if the structure contains any string objects
  - Will, however, work with C-string members

# Using a Constructor to Initialize Structure Members

- Similar to a constructor for a class:
  - name is the same as the name of the struct
  - no return type
  - used to initialize data members
- It is normally written inside the **`struct`** declaration

# A Structure with a Constructor

```
struct Dimensions
{
   int length,
       width,
       height;
   // Constructor
   Dimensions(int L, int W, int H)
   {length = L; width = W; height = H;}
};
```

# Nested Structures

A structure can have another structure as a member.

```
struct PersonInfo
{  string name,
          address,
          city;
};
struct Student
{  int            studentID;
   PersonInfo  pData;
   short          year;
   double         gpa;
};
```

# Members of Nested Structures

Use the dot operator multiple times to access fields of nested structures

```
Student s5;
s5.pData.name = "Joanne";
s5.pData.city = "Tulsa";
```

# Structures as Function Arguments

- May pass members of **`struct`** variables to functions

  ```
  computeGPA(s1.gpa);
  ```

- May pass entire **`struct`** variables to functions

  ```
  showData(s5);
  ```

- Can use reference parameter if function needs to modify contents of structure variable

# Notes on Passing Structures

- Using a value parameter for structure can slow down a program and waste space
- Using a reference parameter speeds up program, but allows the function to modify data in the structure
- To save space and time, while protecting structure data that should not be changed, use a **`const`** reference parameter

```
void showData(const Student &s)
                              // header
```

# Returning a Structure from a Function

- Function can return a **`struct`**

```
Student getStuData();  // prototype
s1 = getStuData();     // call
```

- Function must define a local structure variable
  - for internal use
  - to use with **`return`** statement

# Returning a Structure Example

```
Student getStuData()
{ Student s;    // local variable
  cin >> s.studentID;
  cin.ignore();
  getline(cin, s.pData.name);
  getline(cin, s.pData.address);
  getline(cin, s.pData.city);
  cin >> s.year;
  cin >> s.gpa;
  return s;
}
```

# Unions

- Similar to a **`struct`**, but
  - all members share a single memory location, which saves space
  - only 1 member of the union can be used at a time
- Declared using key word **`union`**
- Otherwise the same as **`struct`**
- Variables defined and accessed like **`struct`** variables

# Example **union** Declaration

```
union WageInfo
{
   double hourlyRate;
   float annualSalary;
};
```

union tag

union members

**Notice the required ;**

# 7.14 Introduction to Object-Oriented Analysis and Design

- Object-Oriented Analysis: that phase of program development when the program functionality is determined from the requirements
- It includes
  - identification of objects and classes
  - definition of each class's attributes
  - identification of each class's behaviors
  - definition of the relationship between classes

# Identify Objects and Classes

- Consider the major data elements and the operations on these elements
- Candidates include
  - user-interface components (menus, text boxes, *etc.*)
  - I/O devices
  - physical objects
  - historical data (employee records, transaction logs, *etc.*)
  - the roles of human participants

# Define Class Attributes

- Attributes are the data elements of an object of the class
- They are necessary for the object to work in its role in the program

# Define Class Behaviors

- For each class,
  - Identify what an object of a class should do in the program
- The behaviors determine some of the member functions of the class

# Relationships Between Classes

Possible relationships

- Access ("uses-a")
- Ownership/Composition ("has-a")
- Inheritance ("is-a")

# Finding the Classes

Technique:

- Write a description of the problem domain (objects, events, etc. related to the problem)
- List the nouns, noun phrases, and pronouns. These are all candidate objects
- Refine the list to include only those objects that are relevant to the problem

# Determine Class Responsibilities

Class responsibilities:

- What is the class responsible to know?
- What is the class responsible to do?

Use these to define some of the member functions

# Object Reuse

- A well-defined class can be used to create objects in multiple programs
- By re-using an object definition, program development time is shortened
- One goal of object-oriented programming is to support object reuse

# 7.15 Screen Control

- Programs to date have all displayed output starting at the upper left corner of computer screen or output window. Output is displayed left-to-right, line-by-line.
- Computer operating systems are designed to allow programs to access any part of the computer screen. Such access is operating system-specific.

# Screen Control – Concepts

- An output screen can be thought of as a grid of 25 rows and 80 columns. Row 0 is at the top of the screen. Column 0 is at the left edge of the screen.
- The intersection of a row and a column is a cell. It can display a single character.
- A cell is identified by its row and column number. These are its coordinates.

# Screen Control – Windows - Specifics

- **`#include <windows.h>`** to access the operating system from a program
- Create a handle to reference the output screen:

```
HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
```

- Create a COORD structure to hold the coordinates of a cell on the screen:

```
COORD position;
```

# Screen Control – Windows – More Specifics

- Assign coordinates where the output should appear:

```
position.X = 30;    // column
position.Y = 12;    // row
```

- Set the screen cursor to this cell:

```
SetConsoleCursorPosition(screen, position);
```

- Send output to the screen:

```
cout << "Look at me!" << endl;
```

  - be sure to end with **endl**, not **'\n'** or nothing

# Chapter 8: Arrays

# Topics

8.1 Arrays Hold Multiple Values
8.2 Accessing Array Elements
8.3 Inputting and Displaying Array Contents
8.4 Array Initialization
8.5 Processing Array Contents
8.6 Using Parallel Arrays

# Topics (continued)

8.7 The **`typedef`** Statement

8.8 Arrays as Function Arguments

8.9 Two-Dimensional Arrays

8.10 Arrays with Three or More Dimensions

8.11 Vectors

8.12 Arrays of Objects

# 8.1 Arrays Hold Multiple Values

- Array: variable that can store multiple values of the same type
- Values are stored in consecutive memory locations
- Declared using **[]** operator

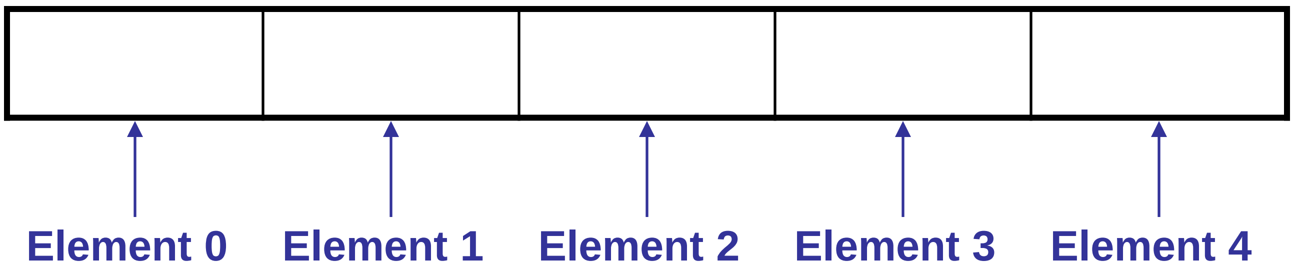
```
const int ISIZE = 5;
int tests[ISIZE];
```

# Array Storage in Memory

The definition

```
int tests[ISIZE];  // ISIZE is 5
```

allocates the following memory



# Array Terminology

In the definition `int tests[ISIZE];`

- **`int`** is the data type of the array elements
- **`tests`** is the name of the array
- **`ISIZE`**, in **`[ISIZE]`**, is the size declarator. It shows the number of elements in the array.
- The size of an array is the number of bytes allocated for it

*(number of elements) * (bytes needed for each element)*

# Array Terminology Examples

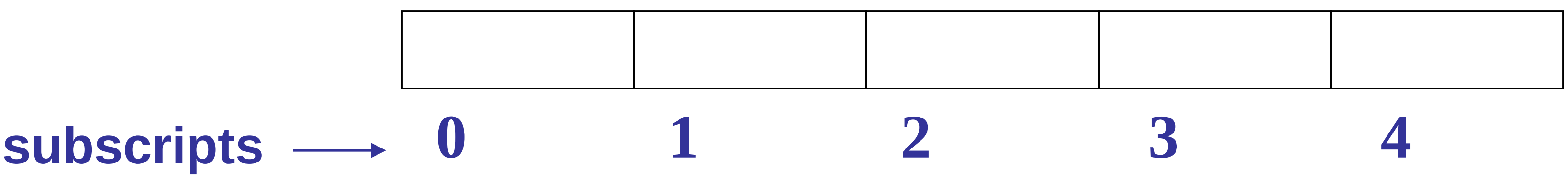
Examples:

Assumes **int** uses 4 bytes and **double** uses 8 bytes

```
const int ISIZE = 5, DSIZE = 10;
int tests[ISIZE];  // holds 5 ints, array
                   // occupies 20 bytes
double volumes[DSIZE];// holds 10 doubles,
                      // array occupies
                      // 80 bytes
```

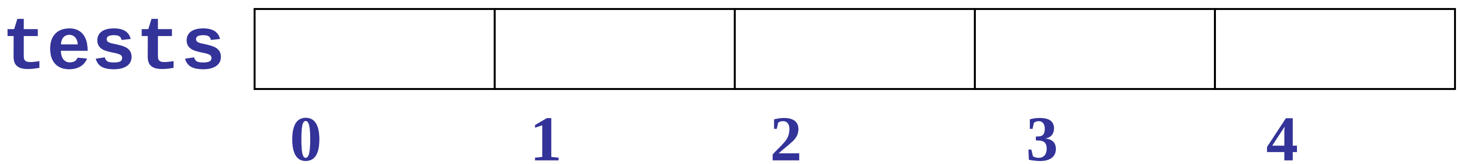
# 8.2 Accessing Array Elements

- Each array element has a subscript, used to access the element.
- Subscripts start at 0



# Accessing Array Elements

- Array elements (accessed by array name and subscript) can be used as regular variables




```
tests[0] = 79;
cout << tests[0];
cin  >> tests[1];
tests[4] = tests[0] + tests[1];
cout << tests; // illegal due to
               // missing subscript
```

# 8.3 Inputting and Displaying Array Contents

**`cout`** and **`cin`** can be used to display values from and store values into an array

```
const int ISIZE = 5;

int tests[ISIZE]; // Define 5-elt. array
cout << "Enter first test score ";
cin  >>  tests[0];
```

# Array Subscripts

- Array subscript can be an integer constant, integer variable, or integer expression
- Examples:

| | Subscript is |
|---|---|
| `cin  >> tests[3];` | int constant |
| `cout << tests[i];` | int variable |
| `cout << tests[i+j];` | int expression |

# Accessing All Array Elements

To access each element of an array

- Use a loop
- Let the loop control variable be the array subscript
- A different array element will be referenced each time through the loop

```
for (i = 0; i < 5; i++)
   cout << tests[i] << endl;
```

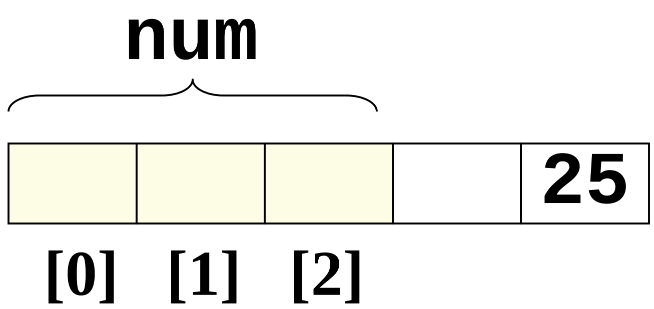
# Getting Array Data from a File

```
const int ISIZE = 5, sales[ISIZE];
ifstream dataFile;
datafile.open("sales.dat");
if (!dataFile)
   cout << "Error opening data file\n";
else
{  // Input daily sales
   for (int day = 0; day < ISIZE; day++)
      dataFile >> sales[day];
   dataFile.close();
}
```

# No Bounds Checking

- There are no checks in C++ that an array subscript is in range
- An invalid array subscript can cause program to overwrite other memory
- Example:

```
const int ISIZE = 3;
int i = 4;
int num[ISIZE];
num[i] = 25;
```



# Off-By-One Errors

- Most often occur when a program accesses data one position beyond the end of an array, or misses the first or last element of an array.
- Don't confuse the ordinal number of an array element (first, second, third) with its subscript (0, 1, 2)

# 8.4 Array Initialization

- Can be initialized during program execution with assignment statements

```
tests[0] = 79;
tests[1] = 82; // etc.
```

- Can be initialized at array definition with an initialization list

```
const int ISIZE = 5;
int tests[ISIZE] = {79,82,91,77,84};
```

# Start at element 0 or 1?

- You may choose to declare arrays to be one larger than needed. This allows you to use the element with subscript 1 as the 'first' element, etc., and may minimize off-by-one errors.
- The element with subscript 0 is not used.
- This is most often done when working with ordered data, *e.g.*, months of the year or days of the week

# Partial Array Initialization

- If array is initialized at definition with fewer values than the size declarator of the array, remaining elements will be set to **`0`** or the empty string

```
int tests[ISIZE] = {79, 82};
```

| | | | | |
|---|---|---|---|---|
| **79** | **82** | **0** | **0** | **0** |

- Initial values used in order; cannot skip over elements to initialize noncontiguous range
- Cannot have more values in initialization list than the declared size of the array

# Implicit Array Sizing

- Can determine array size by the size of the initialization list

```
short quizzes[]={12,17,15,11};
```

| | | | |
|---|---|---|---|
| 12 | 17 | 15 | 11 |

- Must use either array size declarator or initialization list when array is defined

# 8.5 Processing Array Contents

- Array elements can be
  - treated as ordinary variables of the same type as the array
  - used in arithmetic operations, in relational expressions, etc.
- Example:

```
if (principalAmt[3] >= 10000)
    interest = principalAmt[3] * intRate1;
else
    interest = principalAmt[3] * intRate2;
```

# Using Increment and Decrement Operators with Array Elements

When using **++** and **--** operators, don't confuse the element with the subscript

```
tests[i]++;  // adds 1 to tests[i]
tests[i++];  // increments i, but has
             // no effect on tests
```

# Copying One Array to Another

- Cannot copy with an assignment statement:

```
tests2 = tests;  //won't work
```

- Must instead use a loop to copy element-by-element:

```
for (int indx=0; indx < ISIZE; indx++)
   tests2[indx] = tests[indx];
```

# Are Two Arrays Equal?

- Like copying, cannot compare in a single expression:

```
if (tests2 == tests)
```

- Use a while loop with a boolean variable:

```
bool areEqual=true;
int indx=0;
while (areEqual && indx < ISIZE)
{
   if(tests[indx] != tests2[indx]
      areEqual = false;
}
```

# Sum, Average of Array Elements

- Use a simple loop to add together array elements

```
float average, sum = 0;
for (int tnum=0; tnum< ISIZE; tnum++)
    sum += tests[tnum];
```

- Once summed, average can be computed

```
average = sum/ISIZE;
```

# Largest Array Element

- Use a loop to examine each element and find the largest element (*i.e.*, one with the largest value)

```
int largest = tests[0];
for (int tnum = 1; tnum < ISIZE; tnum++)
{  if (tests[tnum] > largest)
      largest = tests[tnum];
}
cout << "Highest score is " << largest;
```

- A similar algorithm exists to find the smallest element

# Partially-Filled Arrays

- The exact amount of data (and, therefore, array size) may not be known when a program is written.
- Programmer makes best estimate for maximum amount of data, sizes arrays accordingly.  A sentinel value can be used to indicate end-of-data.
- Programmer must also keep track of how many array elements are actually used

# Using Arrays vs. Using Simple Variables

- An array is probably not needed if the input data is only processed once:
  - Find the sum or average of a set of numbers
  - Find the largest or smallest of a set of values
- If the input data must be processed more than once, an array is probably a good idea:
  - Calculate the average, then determine and display which values are above the average and which are below the average

# C-Strings and **`string`** Objects

Can be processed using array name

- Entire string at once, or
- One element at a time by using a subscript

```
string city;
cout << "Enter city name: ";
cin  >> city;
```

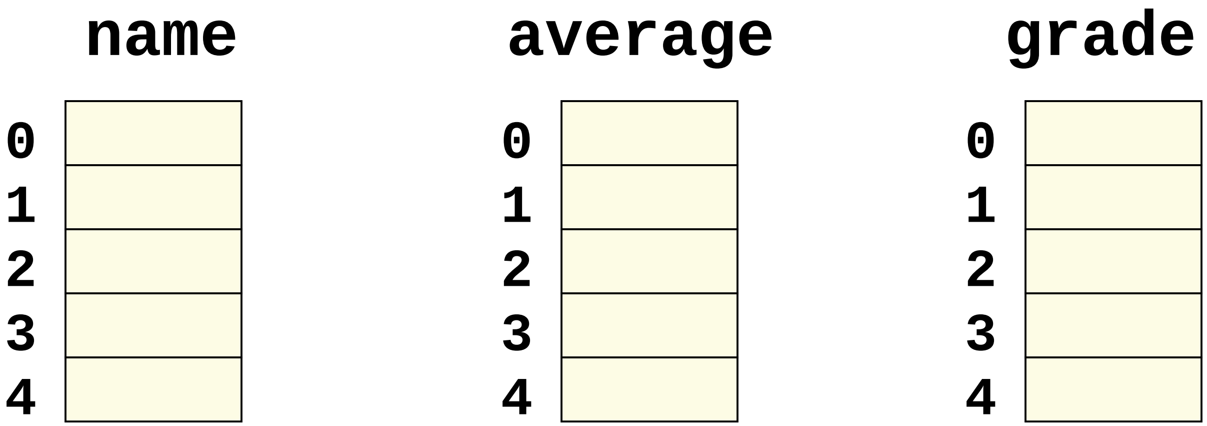
| 'S' | 'a' | 'l' | 'e' | 'm' |
|---|---|---|---|---|
| city[0] | city[1] | city[2] | city[3] | city[4] |

# 8.6 Using Parallel Arrays

- Parallel arrays: two or more arrays that contain related data
- Subscript is used to relate arrays
  - elements at same subscript are related
- The arrays do not have to hold data of the same type

# Parallel Array Example

```
const int ISIZE = 5;
string name[ISIZE];   // student name
float average[ISIZE]; // course average
char grade[ISIZE];    // course grade
```



# Parallel Array Processing

```
const int ISIZE = 5;
string name[ISIZE];   // student name
float average[ISIZE]; // course average
char grade[ISIZE];    // course grade
...
for (int i = 0; i < ISIZE; i++)
   cout << " Student: " << name[i]
        << " Average: " << average[i]
        << " Grade: "   << grade[i]
        << endl;
```

# 8.7 The **`typedef`** Statement

- Creates an alias for a simple or structured data type
- Format:

```
typedef existingType newName;
```

- Example:

```
typedef unsigned int Uint;
Uint tests[ISIZE]; // array of
                   // unsigned ints
```

# Uses of **`typedef`**

- Used to make code more readable
- Can be used to create alias for an array of a particular type

```
// Define yearArray as a data type
// that is an array of 12 ints
typedef int yearArray[MONTHS];

// Create two of these arrays
yearArray highTemps, lowTemps;
```

# 8.8 Arrays as Function Arguments

- Passing a single array element to a function is no different than passing a regular variable of that data type
- Function does not need to know that the value it receives is coming from an array

```
displayValue(score[i]);      // call
void displayValue(int item) // header
{  cout << item << endl;
}
```

# Passing an Entire Array

- To define a function that has an array parameter, use empty **[]** to indicate the array argument
- To pass an array to a function, just use the array name

```
// Function prototype
void showScores(int []);

// Function header
void showScores(int tests[])

// Function call
showScores(tests);
```

# Passing an Entire Array

- Use the array name, without any brackets, as the argument
- Can also pass the array size so the function knows how many elements to process

```
showScores(tests, 5);          // call

void showScores(int[], int); // prototype

void showScores(int A[],
                int size) // header
```

# Using **typedef** with a Passed Array

Can use **typedef** to simplify function prototype and heading

```
// Make intArray an integer array
// of unspecified size
typedef int intArray[];

// Function prototype
void showScores(intArray, int);

// Function header
void showScores(intArray tests,
                        int size)
```

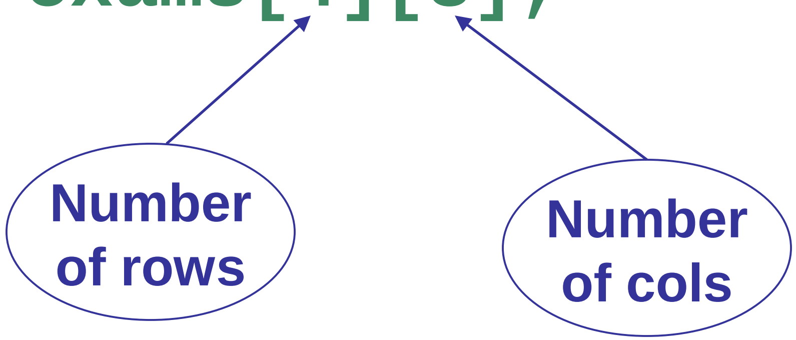
# Modifying Arrays in Functions

- Array parameters in functions are similar to reference variables
- Changes made to array in a function are made to the actual array in the calling function
- Must be careful that an array is not inadvertently changed by a function
- Can use const keyword in prototype and header to prevent changes

# 8.9 Two-Dimensional Arrays

- Can define one array for multiple sets of data
- Like a table in a spreadsheet
- Use two size declarators in definition

```
int exams[4][3];
```



# Two-Dimensional Array Representation

```
int exams[4][3];
```

**columns**

**rows**

| | | |
|---|---|---|
| `exams[0][0]` | `exams[0][1]` | `exams[0][2]` |
| `exams[1][0]` | `exams[1][1]` | `exams[1][2]` |
| `exams[2][0]` | `exams[2][1]` | `exams[2][2]` |
| `exams[3][0]` | `exams[3][1]` | `exams[3][2]` |

Use two subscripts to access element

```
exams[2][2] = 86;
```

# Initialization at Definition

- Two-dimensional arrays are initialized row-by-row

```
int exams[2][2] = { {84, 78},
                    {92, 97} };
```

| | |
|---|---|
| 84 | 78 |
| 92 | 97 |

- Can omit inner **{ }**

# Passing a Two-Dimensional Array to a Function

- Use array name and number of columns as arguments in function call

```
getExams(exams, 2);
```

- Use empty `[]` for row and a size declarator for col in the prototype and header

```
// Prototype, where NUM_COLS is 2
void getExams(int[][NUM_COLS], int);

// Header
void getExams
        (int exams[][NUM_COLS], int rows)
```

# Using **`typedef`** with a Two-Dimensional Array

Can use **`typedef`** for simpler notation

```
typedef int intExams[][2];
   ...
// Function prototype
void getExams(intExams, int);
// Function header
void getExams(intExams exams, int rows)
```

# 2D Array Traversal

- Use nested loops, one for row and one for column, to visit each array element.
- Accumulators can be used to sum the elements row-by-row, column-by-column, or over the entire array.

# 8.10 Arrays with Three or More Dimensions

- Can define arrays with any number of dimensions

```
short rectSolid(2,3,5);
double timeGrid(3,4,3,4);
```

- When used as parameter, specify size of all but 1st dimension

```
void getRectSolid(short [][3][5]);
```

# 8.11 Vectors

- Holds a set of elements, like an array
- Flexible number of elements - can grow and shrink
  - No need to specify size when defined
  - Automatically adds more space as needed
- Defined in the Standard Template Library (STL)
  - Covered in a later chapter
- Must include **`vector`** header file to use vectors

```
#include <vector>
```

# Vectors

- Can hold values of any type
  - Type is specified when a vector is defined

```
vector<int> scores;
vector<double> volumes;
```

- Can use **[ ]** to access elements

# Defining Vectors

- Define a vector of integers (starts with 0 elements)

```
vector<int> scores;
```

- Define **int** vector with initial size 30 elements

```
vector<int> scores(30);
```

- Define 20-element **int** vector and initialize all elements to 0

```
vector<int> scores(20, 0);
```

- Define **int** vector initialized to size and contents of vector **finals**

```
vector<int> scores(finals);
```

# Growing a Vector's Size

- Use **push_back** member function to add an element to a full array or to an array that had no defined size

```
// Add a new element holding a 75
scores.push_back(75);
```

- Use **size** member function to determine number of elements currently in a vector

```
howbig = scores.size();
```

# Removing Vector Elements

- Use **`pop_back`** member function to remove last element from vector

```
scores.pop_back();
```

- To remove all contents of vector, use **`clear`** member function

```
scores.clear();
```

- To determine if vector is empty, use **`empty`** member function

```
while (!scores.empty()) ...
```

# 8.14 Arrays of Objects

- Objects can also be used as array elements

```
class Square
{ private:
     int side;
  public:
     Square(int s = 1)
     { side = s; }
     int getSide()
     { return side; }
};
Square shapes[10];  // Create array of 10
                    // Square objects
```

# Arrays of Objects

- Like an array of structures, use an array subscript to access a specific object in the array
- Then use dot operator to access member methods of that object

```
for (i = 0; i < 10; i++)
  cout << shapes[i].getSide() << endl;
```

# Initializing Arrays of Objects

- Can use default constructor to perform same initialization for all objects
- Can use initialization list to supply specific initial values for each object

```
Square shapes[5] = {1,2,3,4,5};
```

- Default constructor is used for the remaining objects if initialization list is too short

```
Square boxes[5] = {1,2,3};
```

# Initializing Arrays of Objects

If an object is initialized with a constructor that takes > 1 argument, the initialization list must include a call to the constructor for that object

```
Rectangle spaces[3] =
{ Rectangle(2,5),
  Rectangle(1,3),
  Rectangle(7,7)  };
```

# Arrays of Structures

- Structures can be used as array elements

```
struct Student
{
   int studentID;
   string name;
   short year;
   double gpa;
};
const int CSIZE = 30;
Student class[CSIZE]; // Holds 30
                      // Student structures
```

# Arrays of Structures

- Use array subscript to access a specific structure in the array
- Then use dot operator to access members of that structure

```
cin  >> class[25].studentID;

cout << class[i].name << " has GPA "
     << class[i].gpa << endl;
```

# Chapter 9: Searching, Sorting, and Algorithm Analysis

# Topics

# 9.1 Introduction to Search Algorithms

- Search: to locate a specific item in a list (array, vector, etc.) of information
- Two algorithms (methods) considered here:
  - Linear search (also called Sequential Search)
  - Binary search

# Linear Search Algorithm

```
Set found to false
Set position to –1
Set index to 0
While index < number of elts and found is false
    If list [index] is equal to search value
        found = true
        position = index
    End If
    Add 1 to index
End While
Return position
```

# Linear Search Example

- Array **`numlist`** contains

| | | | | | | |
|---|---|---|---|---|---|---|
| 17 | 23 | 5 | 11 | 2 | 29 | 3 |

- Searching for the the value **`11`**, linear search examines **`17`**, **`23`**, **`5`**, and **`11`**
- Searching for the the value **`7`**, linear search examines **`17`**, **`23`**, **`5`**, **`11`**, **`2`**, **`29`**, and **`3`**

# Linear Search Tradeoffs

- Benefits
  - Easy algorithm to understand and to implement
  - Elements in array can be in any order
- Disadvantage
  - Inefficient (slow): for array of N elements, it examines N/2 elements on average for a value that is found in the array, N elements for a value that is not in the array

# Binary Search Algorithm

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

# Binary Search Example

- Array **`numlist2`** contains

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 11 | 17 | 23 | 29 |

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

# Binary Search Tradeoffs

- Benefit
  - Much more efficient than linear search. For an array of **N** elements, it performs at most $\log_2 N$ comparisons.
- Disadvantage
  - Requires that array elements be sorted

# 9.2 Searching an Array of Objects

- Search algorithms are not limited to arrays of integers
- When searching an array of objects or structures, the value being searched for is a member of an object or structure, not the entire object or structure
- Member in object/structure: key field
- Value used in search: search key

# 9.3 Introduction to Sorting Algorithms

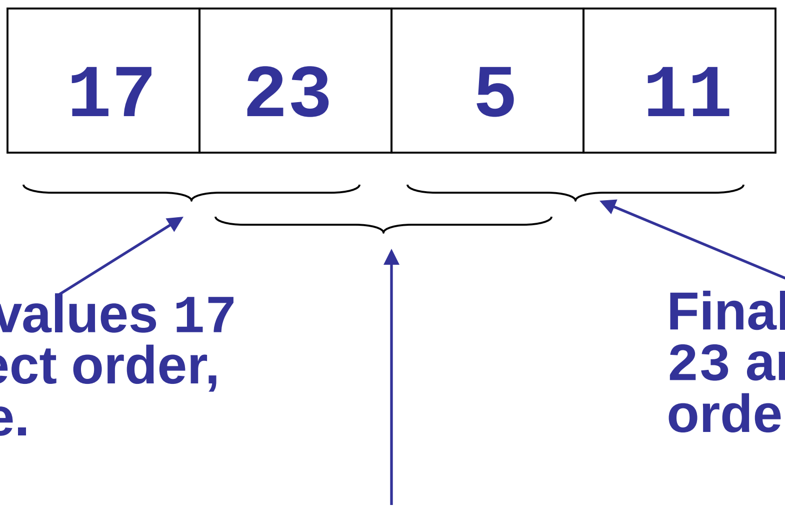
- Sort: arrange values into an order
  - Alphabetical
  - Ascending (smallest to largest) numeric
  - Descending (largest to smallest) numeric
- Two algorithms considered here
  - Bubble sort
  - Selection sort

# Bubble Sort Algorithm

1. Compare 1st two elements and exchange them if they are out of order.
2. Move down one element and compare 2nd and 3rd elements. Exchange if necessary. Continue until the end of the array.
3. Pass through the array again, repeating the process and exchanging as necessary.
4. Repeat until a pass is made with no exchanges.

# Bubble Sort Example

Array **`numlist3`** contains




**First, compare values 17 and 23. In correct order, so no exchange.**

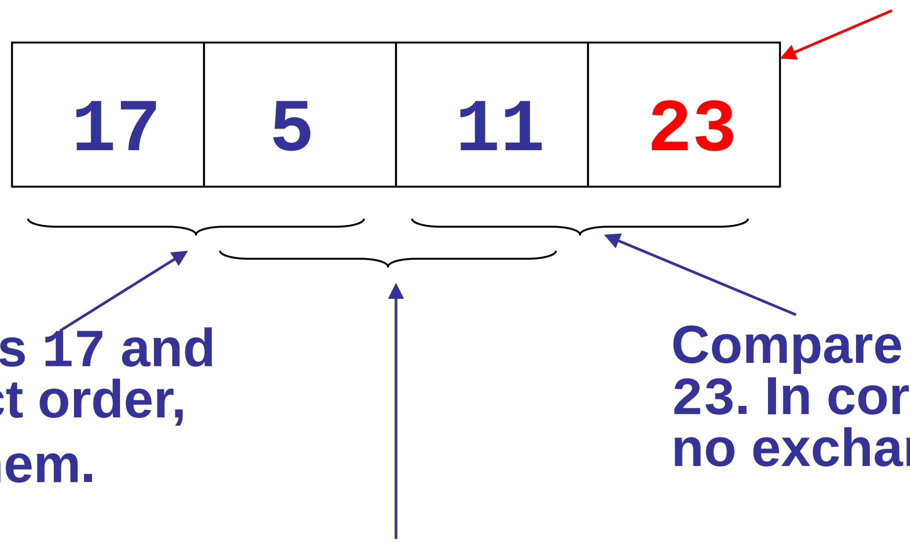
**Finally, compare values 23 and 11. Not in correct order, so exchange them.**

**Then, compare values 23 and  5. Not in correct order, so exchange them.**

# Bubble Sort Example (continued)

After first pass, array **`numlist3`** contains

**In order from previous pass**




**Compare values 17 and 5. Not in correct order, so exchange them.**

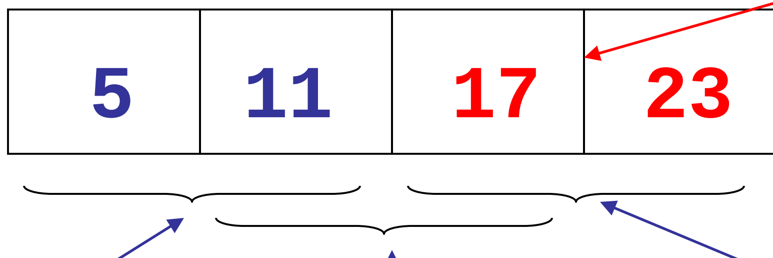
**Compare values 17 and 11. Not in correct order, so exchange them.**

**Compare values 17 and 23. In correct order, so no exchange.**

# Bubble Sort Example (continued)

After second pass, array **numlist3** contains

In order from previous passes




Compare values 5 and 11. In correct order, so no exchange.

Compare values 17 and 23. In correct order, so no exchange.

Compare values 11 and 17. In correct order, so no exchange.

**No exchanges, so array is in order**

# Bubble Sort Tradeoffs

- Benefit
  - Easy to understand and to implement
- Disadvantage
  - Inefficiency makes it slow for large arrays

# Selection Sort Algorithm

1. Locate smallest element in array and exchange it with element in position 0.
2. Locate next smallest element in array and exchange it with element in position 1.
3. Continue until all elements are in order.

# Selection Sort Example

Array **`numlist`** contains

| | | | |
|---|---|---|---|
| 11 | 2 | 29 | 3 |

Smallest element is **2**. Exchange 2 with element in 1st array position (*i.e.*, element 0).

Now in order →

| | | | |
|---|---|---|---|
| 2 | 11 | 29 | 3 |

# Selection Sort – Example (continued)

Next smallest element is **3**. Exchange **3** with element in $2^{nd}$ array position.

Now in order

| | | | |
|---|---|---|---|
| **2** | **3** | 29 | 11 |

Next smallest element is **11**. Exchange **11** with element in $3^{rd}$ array position.

Now in order

| | | | |
|---|---|---|---|
| **2** | **3** | **11** | 29 |

# Selection Sort Tradeoffs

- Benefit
  - More efficient than Bubble Sort, due to fewer exchanges
- Disadvantage
  - Considered harder than Bubble Sort to understand and implement

# 9.4 Sorting an Array of Objects

- As with searching, arrays to be sorted can contain objects or structures
- The key field determines how the structures or objects will be ordered
- When exchanging the contents of array elements, entire structures or objects must be exchanged, not just the key fields in the structures or objects

# 9.5 Sorting and Searching Vectors

- Sorting and searching algorithms can be applied to vectors as well as to arrays
- Need slight modifications to functions to use vector arguments
  - `vector <type> &` used in prototype
  - No need to indicate vector size, as functions can use **`size`** member function to calculate

# 9.6 Introduction to Analysis of Algorithms

- Given two algorithms to solve a problem, what makes one better than the other?
- Efficiency of an algorithm is measured by
  - space (computer memory used)
  - time (how long to execute the algorithm)
- Analysis of algorithms is a more effective way to find efficiency than by using empirical data

# Analysis of Algorithms: Terminology

- Computational Problem: a problem solved by an algorithm
- Basic step: an operation in the algorithm that executes in a constant amount of time
- Examples of basic steps:
  - exchange the contents of two variables
  - compare two values

# Analysis of Algorithms: Terminology

- Complexity of an algorithm: the number of basic steps required to execute the algorithm for an input of size N (N = number of input values)
- Worst-case complexity of an algorithm: the number of basic steps for input of size N that requires the most work
- Average case complexity function: the complexity for typical, average inputs of size N

# Complexity Example

Find the largest value in array A of size n

```
1. biggest = A[0]
2. indx = 0
3. while (indx < n) do
4.   if (A[n] > biggest)
5.   then
6.     biggest = A[n]
7.   end if
8. end while
```

Analysis:

Lines 1 and 2 execute once.

The test in line 3 executes n times.

The test in line 4 executes n times.

The assignment in line 6 executes at most n times.

Due to lines 3 and 4, the algorithm requires execution time proportional to n.

# Comparison of Algorithmic Complexity

Given algorithms F and G with complexity functions *f(n)* and *g(n)* for input of size n

- If the ratio $\frac{f(n)}{g(n)}$ approaches a constant value as n gets large, F and G have equivalent efficiency
- If the ratio $\frac{f(n)}{g(n)}$ gets larger as n gets large, algorithm G is more efficient than algorithm F
- If the ratio $\frac{f(n)}{g(n)}$ approaches 0 as n gets large, algorithm F is more efficient than algorithm G

# "Big O" Notation

- Function *f(n)* is *O(g(n))* ("f is big O of g") for some mathematical function *g(n)* if the ratio $\frac{f(n)}{g(n)}$ approaches a positive constant as n gets large
- *O(g(n))* defines a complexity class for the function f(n) and for the algorithm F
- Increasing complexity classes means faster rate of growth and less efficient algorithms

# Chapter 10: Pointers

# Topics

10.1 Pointers and the Address Operator

10.2 Pointer Variables

10.3 The Relationship Between Arrays and Pointers

10.4 Pointer Arithmetic

10.5 Initializing Pointers

10.6 Comparing Pointers

# Topics (continued)

10.7 Pointers as Function Parameters

10.8 Pointers to Constants and Constant Pointers

10.9 Dynamic Memory Allocation

10.10 Returning Pointers from Functions

10.11 Pointers to Class Objects and Structures

10.12 Selecting Members of Objects

# 10.1 Pointers and the Address Operator

- Each variable in a program is stored at a unique location in memory that has an address
- Use the address operator **&** to get the address of a variable:

```
int num = -23;
cout << &num; // prints address
              // in hexadecimal
```

- The address of a memory location is a pointer

# 10.2 Pointer Variables

- Pointer variable (pointer): a variable that holds an address
- Pointers provide an alternate way to access memory locations

# Pointer Variables

- Definition:

```
int  *intptr;
```

- Read as:
  "**intptr** can hold the address of an int" or "the variable that **intptr** points to has type int"

- The spacing in the definition does not matter:

```
int * intptr;
int*  intptr;
```

- ***** is called the indirection operator

# Pointer Variables

- Assignment:

```
int num = 25;
int *intptr;
intptr = &num;
```

- Memory layout:


**address of num: 0x4a00**

- Can access **num** using **intptr** and indirection operator *****:

```
cout << intptr;  // prints 0x4a00
cout << *intptr; // prints 25
*intptr = 20;    // puts 20 in num
```

# 10.3 The Relationship Between Arrays and Pointers

An array name is the starting address of the array

```
int vals[] = {4, 7, 11};
```

| | | |
|---|---|---|
| **4** | **7** | **11** |

**starting address of `vals`: `0x4a00`**

```
cout << vals;    // displays 0x4a00
cout << vals[0]; // displays 4
```

# The Relationship Between Arrays and Pointers

- An array name can be used as a pointer constant

```
int vals[] = {4, 7, 11};
cout << *vals;     // displays 4
```

- A pointer can be used as an array name

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

# Pointers in Expressions

- Given:

```
int vals[]={4,7,11};
int *valptr = vals;
```

- What is **`valptr + 1`**?
- It means (address in **`valptr`**) + (1 * size of an **`int`**)

```
cout << *(valptr+1); // displays 7
cout << *(valptr+2); // displays 11
```

- Must use **`( )`** in expression

# Array Access

Array elements can be accessed in many ways

| Array access method | Example |
|---|---|
| array name and [ ] | `vals[2] = 17;` |
| pointer to array and [ ] | `valptr[2] = 17;` |
| array name and subscript arithmetic | `*(vals+2) = 17;` |
| pointer to array and subscript arithmetic | `*(valptr+2) = 17;` |

# Array Access

- Array notation

```
vals[i]
```

is equivalent to the pointer notation

```
*(vals + i)
```

- No bounds checking is performed on array access

# 10.4 Pointer Arithmetic

Some arithmetic operators can be used with pointers:

- Increment and decrement operators **++**, **- -**
- Integers can be added to or subtracted from pointers using the operators **+**, **-**, **+=**, and **-=**
- One pointer can be subtracted from another by using the subtraction operator **-**

# Pointer Arithmetic

Assume the variable definitions

```
int vals[]={4,7,11};
int *valptr = vals;
```

Examples of use of **++** and **--**

```
valptr++; // points at 7
valptr--; // now points at 4
```

# More on Pointer Arithmetic

Assume the variable definitions:

```
int vals[]={4,7,11};
int *valptr = vals;
```

Example of the use of **+** to add an int to a pointer:

```
cout << *(valptr + 2)
```

This statement will print 11

# More on Pointer Arithmetic

Assume the variable definitions:

```
int vals[]={4,7,11};
int *valptr = vals;
```

Example of use of +=:

```
valptr = vals; // points at 4
valptr += 2;   // points at 11
```

# More on Pointer Arithmetic

Assume the variable definitions

```
int vals[] = {4,7,11};
int *valptr = vals;
```

Example of pointer subtraction

```
valptr += 2;
cout << valptr - val;
```

This statement prints **2**: the number of **`int`**s between **`valptr`** and **`val`**

# 10.5 Initializing Pointers

- Can initialize to NULL or 0 (zero)

```
int *ptr = NULL;
```

- Can initialize to addresses of other variables

```
int num, *numPtr = &num;
int val[ISIZE], *valptr = val;
```

- Initial value must have correct type

```
float cost;
int *ptr = &cost; // won't work
```

# 10.6 Comparing Pointers

- Relational operators can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)    // compares
                     // addresses
if (*ptr1 == *ptr2)  // compares
                     // contents
```

# 10.7 Pointers as Function Parameters

- A pointer can be a parameter
- It works like a reference parameter to allow changes to argument from within a function
- A pointer parameter must be explicitly dereferenced to access the contents at that address

# Pointers as Function Parameters

Requires:

1) asterisk * on parameter in prototype and heading

```
void getNum(int *ptr);
```

2) asterisk * in body to dereference the pointer

```
cin >> *ptr;
```

3) address as argument to the function in the call

```
getNum(&num);
```

# Pointers as Function Parameters

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int num1 = 2, num2 = -3;
swap(&num1, &num2);  //call
```

# 10.8 Ponters to Constants and Constant Pointers

- Pointer to a constant: cannot change the value that is pointed at

- Constant pointer: the address in the pointer cannot change after the pointer is initialized

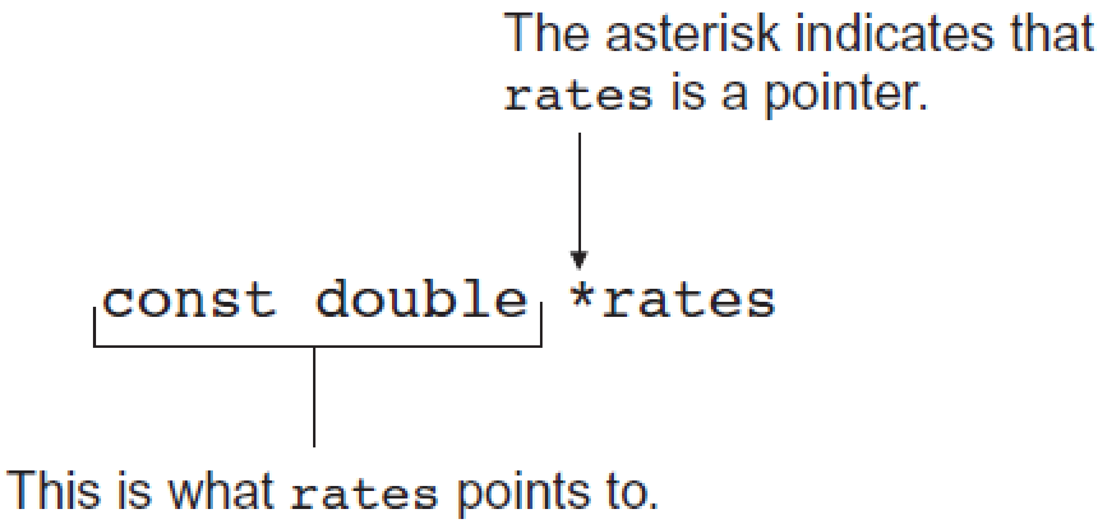
# Ponters to Constant

- Must use **const** keyword in pointer definition:

```
const double taxRates[] =
                   {0.65, 0.8, 0.75};
const double *ratePtr;
```

- Use **const** keyword for pointers in function headers to protect data from modification from within function

# Pointer to Constant – What does the Definition Mean?




Read as: “rates is a pointer to a constant that is a double.”

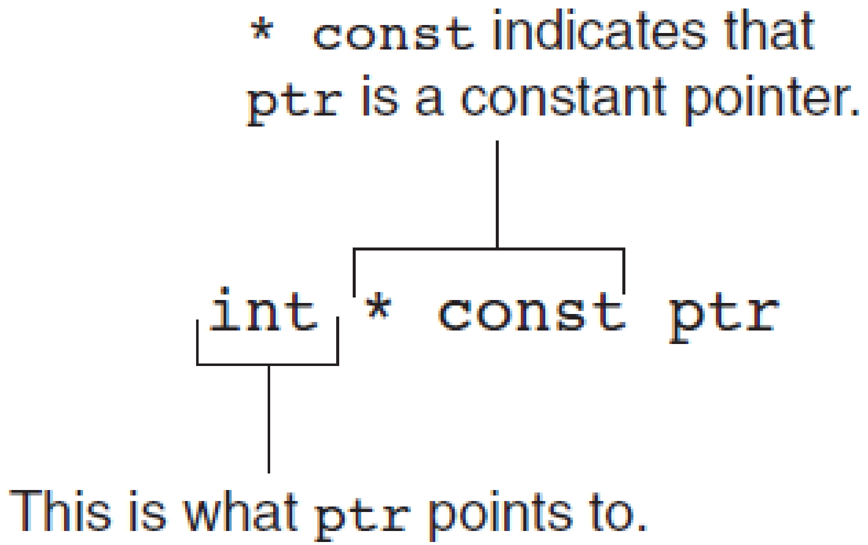
# Constant Pointers

- Defined with **`const`** keyword adjacent to variable name:

```
int classSize = 24;
int * const classPtr = &classSize;
```

- Must be initialized when defined
- Can be used without initialization as a function parameter
  - Initialized by argument when function is called
  - Function can receive different arguments on different calls
- While the <u>address</u> in the pointer cannot change, the <u>data</u> at that address may be changed

# Constant Pointer – What does the Definition Mean?



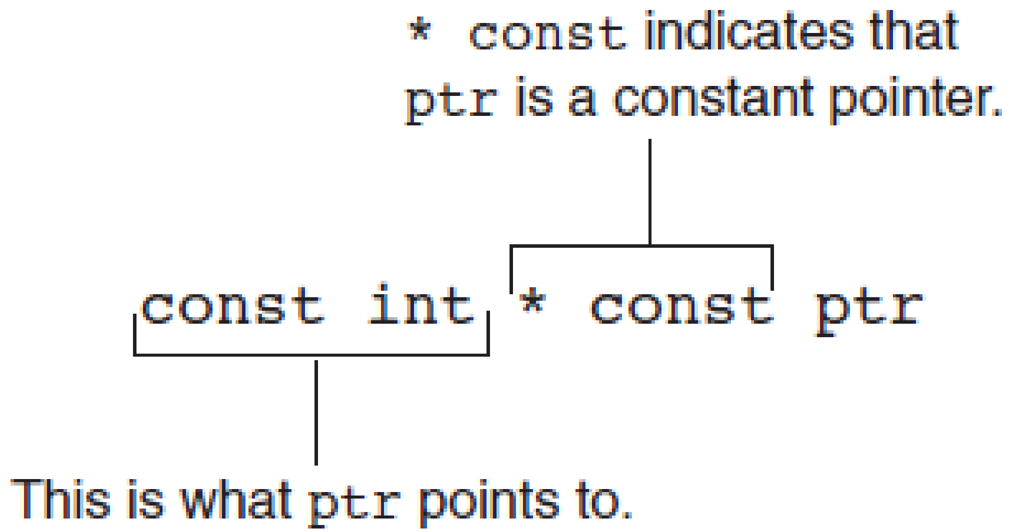

Read as: “pts is a constant pointer to an int.”

# Constant Pointer to Constant

- Can combine pointer to constants and constant pointers:

```
int size = 10;
const int * const ptr = &size;
```

- What does it mean?



# 10.9 Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Uses **new** operator to allocate memory

```
double *dptr;
dptr = new double;
```

- **new** returns address of memory location

# Dynamic Memory Allocation

- Can also use **new** to allocate array

```
arrayPtr = new double[25];
```

  - Program may terminate if there is not sufficient memory

- Can then use **[ ]** or pointer arithmetic to access array

# Dynamic Memory Example

```
int *count, *arrayptr;
count = new int;
cout <<"How many students? ";
cin >> *count;
arrayptr = new int[*count];

for (int i=0; i<*count; i++)
{
  cout << "Enter score " << i << ": ";
  cin >> arrayptr[i];
}
```

# Releasing Dynamic Memory

- Use **`delete`** to free dynamic memory

```
delete count;
```

- Use **`delete []`** to free dynamic array memory

```
delete [] arrayptr;
```

- Only use **`delete`** with dynamic memory!

# Dangling Pointers and Memory Leaks

- A pointer is dangling if it contains the address of memory that has been freed by a call to **`delete`**.
  - Solution: set such pointers to 0 as soon as memory is freed.
- A memory leak occurs if no-longer-needed dynamic memory is not freed. The memory is unavailable for reuse within the program.
  - Solution: free up dynamic memory after use

# 10.10 Returning Pointers from Functions

- Pointer can be return type of function

```
int* newNum();
```

- The function must not return a pointer to a local variable in the function
- The function should only return a pointer
  - to data that was passed to the function as an argument
  - to dynamically allocated memory

# 10.11 Pointers to Class Objects and Structures

- Can create pointers to objects and structure variables

```
struct Student {…};
class Square {…};
Student stu1;
Student *stuPtr = &stu1;
Square sq1[4];
Square *squarePtr = &sq1[0];
```

- Need to use **()** when using ***** and **.** operators

```
(*stuPtr).studentID = 12204;
```

# Structure Pointer Operator

- Simpler notation than **`(*ptr).member`**
- Use the form **`ptr->member:`**

```
stuPtr->studentID = 12204;
squarePtr->setSide(14);
```

in place of the form **`(*ptr).member:`**

```
(*stuPtr).studentID = 12204;
(*squarePtr).setSide(14);
```

# Dynamic Memory with Objects

- Can allocate dynamic structure variables and objects using pointers:

```
stuPtr = new Student;
```

- Can pass values to constructor:

```
squarePtr = new Square(17);
```

- **`delete`** causes destructor to be invoked:

```
delete squarePtr;
```

# Structure/Object Pointers as Function Parameters

- Pointers to structures or objects can be passed as parameters to functions
- Such pointers provide a pass-by-reference parameter mechanism
- Pointers must be dereferenced in the function to access the member fields

# Controlling Memory Leaks

- Memory that is allocated with **`new`** should be deallocated with a call to **`delete`** as soon as the memory is no longer needed. This is best done in the same function as the one that allocated the memory.
- For dynamically-created objects, **`new`** should be used in the constructor and **`delete`** should be used in the destructor

# 10.12 Selecting Members of Objects

Situation: A structure/object contains a pointer as a member. There is also a pointer to the structure/object.

Problem: How do we access the pointer member via the structure/object pointer?

```
struct GradeList
  { string courseNum;
    int * grades;
  }
GradeList test1, *testPtr = &test1;
```

# Selecting Members of Objects

| Expression | Meaning |
| --- | --- |
| **testPtr->grades** | Access the grades pointer in **test1**. This is the same as **(*testPtr).grades** |
| ***testPtr->grades** | Access the value pointed at by **testPtr->grades**. This is the same as ***(*testPtr).grades** |
| ***test1.grades** | Access the value pointed at by **test1.grades** |

# Chapter 11: More About Classes and Object-Oriented Programming

# Topics

11.1 The **`this`** Pointer and Constant Member Functions

11.2 Static Members

11.3 Friends of Classes

11.4 Memberwise Assignment

11.5 Copy Constructors

11.6 Operator Overloading

11.7 Type Conversion Operators

# Topics (continued)

11.8 Convert Constructors

11.9 Aggregation and Composition

11.10 Inheritance

11.11 Protected Members and Class Access

11.12 Constructors, Destructors, and Inheritance

11.13 Overriding Base Class Functions

# 11.1 The **`this`** Pointer and Constant Member Functions

- **`this`** pointer:
  - Implicit parameter passed to a member function
  - points to the object calling the function
- **`const`** member function:
  - does not modify its calling object

# Using the **`this`** Pointer

Can be used to access members that may be hidden by parameters with the same name:

```
class SomeClass
{
  private:
    int num;
  public:
    void setNum(int num)
    { this->num = num; }
};
```

# Constant Member Functions

- Declared with keyword **`const`**

- When **`const`** appears in the parameter list,

```
int setNum (const int num)
```

  the function is prevented from modifying the parameter. The parameter is read-only.

- When **`const`** follows the parameter list,

```
int getX()const
```

  the function is prevented from modifying the object.

# 11.2 Static Members

- Static member variable:
  - One instance of variable for the entire class
  - Shared by all objects of the class
- Static member function:
  - Can be used to access static member variables
  - Can be called before any class objects are created

# Static Member Variables

1) Must be declared in class with keyword **`static`**:

```
class IntVal
{
 public:
    intVal(int val = 0)
    { value = val; valCount++ }
    int getVal();
    void setVal(int);
  private:
    int value;
    static int valCount;
};
```

# Static Member Variables

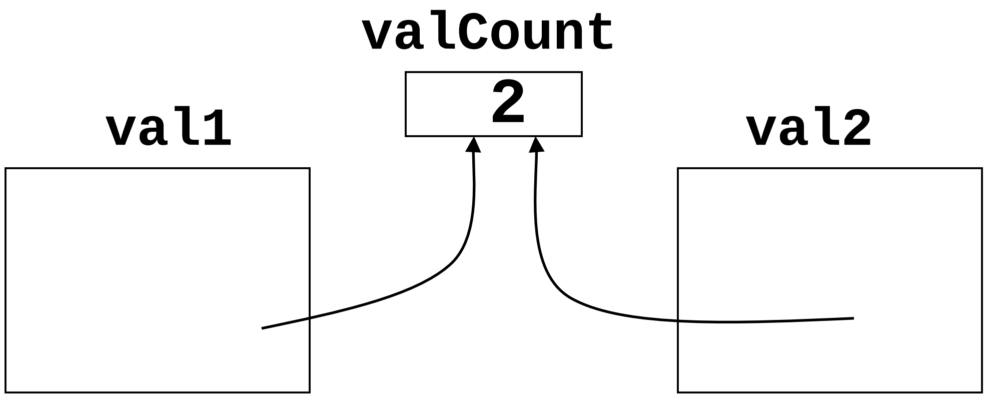
2) Must be defined outside of the class:

```
class IntVal
{
    //In-class declaration
    static int valCount;
    //Other members not shown
 };
 //Definition outside of class
 int IntVal::valCount = 0;
```

# Static Member Variables

3) Can be accessed or modified by any object of the class: Modifications by one object are visible to all objects of the class:

```
IntVal val1, val2;
```



# Static Member Functions

1)Declared with **`static`** before return type:

```
class IntVal
{ public:
      static int getValCount()
      { return valCount; }
    private:
      int value;
      static int valCount;
};
```

# Static Member Functions

2) Can be called independently of class objects, through the class name:

```
cout << IntVal::getValCount();
```

3) Because of item 2 above, the **`this`** pointer cannot be used

4) Can be called before any objects of the class have been created

5) Used primarily to manipulate static member variables of the class

# 11.3 Friends of Classes

- Friend function: a function that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with the **`friend`** keyword in the function prototype

# Friend Function Declarations

1) Friend function may be a stand-alone function:

```
class aClass
{
  private:
    int x;
    friend void fSet(aClass &c, int a);
};

void fSet(aClass &c, int a)
{
   c.x = a;
}
```

# Friend Function Declarations

2) Friend function may be a member of another class:

```
class aClass
{ private:
    int x;
    friend void OtherClass::fSet
                      (aClass &c, int a);
};
class OtherClass
{ public:
    void fSet(aClass &c, int a)
    { c.x = a; }
};
```

# Friend Class Declaration

3) An entire class can be declared a friend of a class:

```
class aClass
{private:
   int x;
   friend class frClass;
};
class frClass
{public:
   void fSet(aClass &c,int a){c.x = a;}
   int fGet(aClass c){return c.x;}
};
```

# Friend Class Declaration

- If **`frClass`** is a friend of **`aClass`**, then all member functions of **`frClass`** have unrestricted access to all members of **`aClass`**, including the private members.

- In general, restrict the property of Friendship to only those functions that must have access to the private members of a class.

# 11.4 Memberwise Assignment

- Can use **=** to assign one object to another, or to initialize an object with an object's data
- Examples (assuming class **V**):

```
V v1, v2;
… // statements that assign
… // values to members of v1
v2 = v1;     // assignment
V v3 = v2;  // initialization
```

# 11.5 Copy Constructors

- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor copies field-to-field, using memberwise assignment
- The default copy constructor works fine in most cases

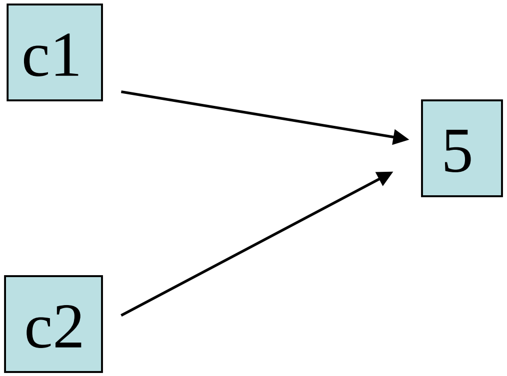
# Copy Constructors

Problems occur when objects contain pointers to dynamic storage:

```
class CpClass
{
 private:
   int *p;
 public:
   CpClass(int v=0)
      { p = new int; *p = v;}
   ~CpClass(){delete p;}
};
```

# Default Constructor Causes Sharing of Storage

```
CpClass c1(5);
if (true)
{
  CpClass c2=c1;
}
// c1 is corrupted
// when c2 goes
// out of scope and
// its destructor
// executes
```



# Problems of Sharing Dynamic Storage

- Destructor of one object deletes memory still in use by other objects

- Modification of memory by one object affects other objects sharing that memory

# Programmer-Defined Copy Constructors

- A copy constructor is one that takes a reference parameter to another object of the same class
- The copy constructor uses the data in the object passed as parameter to initialize the object being created
- Reference parameter should be **`const`** to avoid potential for data corruption

# Programmer-Defined Copy Constructors

- The copy constructor avoids problems caused by memory sharing
- Can allocate separate memory to hold new object's dynamic member data
- Can make new object's pointer point to this memory
- Copies the data, not the pointer, from the original object to the new object

# Copy Constructor Example

```
class CpClass
{
        int *p;
    public:
        CpClass(const CpClass &obj)
        { p = new int; *p = *obj.p; }
        CpClass(int v=0)
        { p = new int; *p = v; }
        ~CpClass(){delete p;}
};
```

# Copy Constructor – When Is It Used?

A copy constructor is called when

- An object is initialized from an object of the same class
- An object is passed by value to a function
- An object is returned using a **`return`** statement from a function

# 11.6 Operator Overloading

- Operators such as **=**, **+**, and others can be redefined for use with objects of a class

- The name of the function for the overloaded operator is **`operator`** followed by the operator symbol, *e.g.*,

  **`operator+`** is the overloaded **+** operator and

  **`operator=`** is the overloaded **=** operator

# Operator Overloading

- Operators can be overloaded as
  - instance member functions, or as
  - friend functions
- The overloaded operator must have the same number of parameters as the standard version. For example, **`operator=`** must have two parameters, since the standard = operator takes two parameters.

# Overloading Operators as Instance Members

A binary operator that is overloaded as an instance member needs only one parameter, which represents the operand on the right:

```
class OpClass
{
 private:
   int x;
 public:
   OpClass operator+(OpClass right);
};
```

# Overloading Operators as Instance Members

- The left operand of the overloaded binary operator is the calling object
- The implicit left parameter is accessed through the **`this`** pointer

```
OpClass OpClass::operator+(OpClass r)
{  OpClass sum;
   sum.x = this->x + r.x;
   return sum;
}
```

# Invoking an Overloaded Operator

- Operator can be invoked as a member function:

```
OpClass a, b, s;
s = a.operator+(b);
```

- It can also be invoked in the more conventional manner:

```
OpClass a, b, s;
s = a + b;
```

# Overloading Assignment

- Overloading the assignment operator solves problems with object assignment when an object contains pointer to dynamic memory.
- Assignment operator is most naturally overloaded as an instance member function
- It needs to return a value of the assigned object to allow cascaded assignments such as

```
a = b = c;
```

# Overloading Assignment

Assignment overloaded as a member function:

```
class CpClass
{
      int *p;
    public:
      CpClass(int v=0)
      { p = new int; *p = v;
      ~CpClass(){delete p;}
      CpClass operator=(CpClass);
};
```

# Overloading Assignment

Implementation returns a value:

```
CpClass CpClass::operator=(CpClass r)
{
  *p = *r.p;
  return *this;
};
```

Invoking the assignment operator:

```
CpClass a, x(45);
```

```
a.operator=(x); // either of these
a = x;          // lines can be used
```

# Notes on Overloaded Operators

- Overloading can change the entire meaning of an operator
- Most operators can be overloaded
- Cannot change the number of operands of the operator
- Cannot overload the following operators:

    **`?:   .   .*   sizeof`**

# Overloading Types of Operators

- **++**, **--** operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a **bool** value
- Overloaded stream operators **>>**, **<<** must return **istream**, **ostream** objects and take **istream**, **ostream** objects as parameters

# Overloaded **[ ]** Operator

- Can be used to create classes that behave like arrays, providing bounds-checking on subscripts
- Overloaded **[ ]** returns a reference to object, not an object itself

# 11.7 Type Conversion Operators

- Conversion Operators are member functions that tell the compiler how to convert an object of the class type to a value of another type
- The conversion information provided by the conversion operators is automatically used by the compiler in assignments, initializations, and parameter passing

# Syntax of Conversion Operators

- Conversion operator must be a member function of the class you are converting from
- The name of the operator is the name of the type you are converting to
- The operator does not specify a return type

# Conversion Operator Example

- To convert from a class **`IntVal`** to an integer:

```
class IntVal
{
    int x;
  public:
    IntVal(int a = 0){x = a;}
    operator int(){return x;}
};
```

- Automatic conversion during assignment:

```
IntVal obj(15); int i;
i = obj;  cout << i; // prints 15
```

# 11.8 Convert Constructors

Convert constructors are constructors that take a single parameter of a type other than the class in which they are defined

```
class CCClass
{  int x;
  public:
    CCClass()                  //default
    CCClass(int a, int b);
    CCClass(int a);       //convert
    CCClass(string s);  //convert
};
```

# Example of a Convert Constructor

The C++ **`string`** class has a convert constructor that converts from C-strings:

```
class string
{
  public:
    string(char *);  //convert
    …
};
```

# Uses of Convert Constructors

- They are automatically invoked by the compiler to create an object from the value passed as parameter:

```
string s("hello");  //convert C-string
CCClass obj(24);    //convert int
```

- The compiler allows convert constructors to be invoked with assignment-like notation:

```
string s = "hello"; //convert C-string
CCClass obj = 24;   //convert int
```

# Uses of Convert Constructors

- Convert constructors allow functions that take the class type as parameter to take parameters of other types:

```
void myFun(string s); // needs string
                      // object
myFun("hello");       // accepts C-string

void myFun(CCClass c);
myFun(34);            // accepts int
```

# 11.9 Aggregation and Composition

- Class aggregation: An object of one class owns an object of another class
- Class composition: A form of aggregation where the enclosing class controls the lifetime of the objects of the enclosed class
- Supports the modeling of ‘has-a’ relationship between classes – enclosing class ‘has a(n)’ instance of the enclosed class

# Object Composition

```
class StudentInfo
{
   private:
      string firstName, LastName;
      string address, city, state, zip;
    ...
};
class Student
{
   private:
      StudentInfo personalData;
    ...
};
```

# Member Initialization Lists

- Used in constructors for classes involved in aggregation.
- Allows constructor for enclosing class to pass arguments to the constructor of the enclosed class
- Notation:

```
owner_class(parameters):owned_class(parameters);
```

# Member Initialization Lists

Use:

```
class StudentInfo
{
     ...
};
class Student
{
   private:
       StudentInfo personalData;
   public:
       Student(string fname, lname):
       StudentInfo(fname, lname);
};
```

# Member Initialization Lists

- Member Initialization lists can be used to simplify the coding of constructors
- Should keep the entries in the initialization list in the same order as they are declared in the class

# Aggregation Through Pointers

- A ‘has-a’ relationship can be implemented by owning a pointer to an object
- Can be used when multiple objects of a class may ‘have’ the same attribute for a member
  - ex: students who may have the same city/state/ zipcode
- Using pointers minimizes data duplication and saves space

# Aggregation, Composition, and Object Lifetimes

- Aggregation represents the owner/owned relationship between objects.
- Composition is a form of aggregation in which the lifetime of the owned object is the same as that of the owner object
- Owned object is usually created as part of the owning object's constructor, destroyed as part of owning object's destructor

# 11.10 Inheritance

- Inheritance is a way of creating a new class by starting with an existing class and adding new members
- The new class can replace or extend the functionality of the existing class
- Inheritance models the 'is-a' relationship between classes

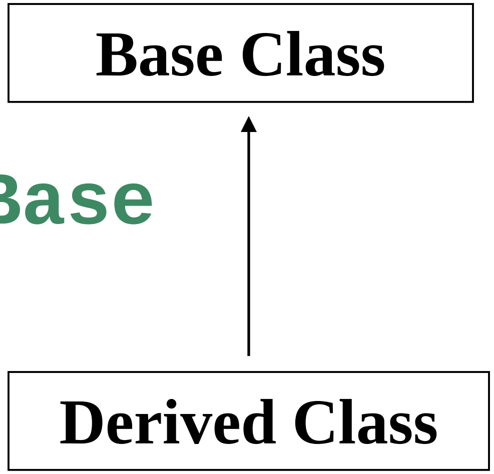
# Inheritance - Terminology

- The existing class is called the base class
  - Alternates: parent class, superclass
- The new class is called the derived class
  - Alternates: child class, subclass

# Inheritance Syntax and Notation

```
// Existing class
class Base
{
};
// Derived class
class Derived : public Base
{
};
```

**Inheritance Class Diagram**



# Inheritance of Members

```
class Parent
{
  int a;
  void bf();
};
class Child : public
          Parent
{
  int c;
  void df();
};
```

**Objects of Parent have members**

```
int a; void bf();
```

**Objects of Child have members**

```
int a; void bf();
int c; void df();
```

# 11.11 Protected Members and Class Access

- protected member access specification: A class member labeled **`protected`** is accessible to member functions of derived classes as well as to member functions of the same class
- Like **`private`**, except accessible to members functions of derived classes

# Base Class Access Specification

Base class access specification determines how **`private`**, **`protected`**, and **`public`** members of base class can be accessed by derived classes

# Base Class Access

C++ supports three inheritance modes, also called base class access modes:

- public inheritance

```
class Child : public Parent { };
```

- protected inheritance

```
class Child : protected Parent{ };
```

- private inheritance

```
class Child : private Parent{ };
```

# Base Class Access vs. Member Access Specification

Base class access is not the same as member access specification:

- Base class access: determine access for inherited members
- Member access specification: determine access for members defined in the class

# Member Access Specification

Specified using the keywords **`private`**, **`protected`**, **`public`**

```
class MyClass
{
  private: int a;
  protected: int b; void fun();
  public: void fun2();
};
```

# Base Class Access Specification

```
class Child : public Parent
{
   protected:
       int a;
   public:
       Child();
};
```

**base access**

**member access**

# Base Class Access Specifiers

1) **`public`** – object of derived class can be treated as object of base class (not vice-versa)
2) **`protected`** – more restrictive than **`public`**, but allows derived classes to know some of the details of parents
3) **`private`** – prevents objects of derived class from being treated as objects of base class.

# Effect of Base Access

| Base class members | | How base class members appear in derived class |
| --- | --- | --- |
| `private: x`<br>`protected: y`<br>`public: z` | `private` base class → | x inaccessible<br>`private: y`<br>`private: z` |
| `private: x`<br>`protected: y`<br>`public: z` | `protected` base class → | x inaccessible<br>`protected: y`<br>`protected: z` |
| `private: x`<br>`protected: y`<br>`public: z` | `public` base class → | x inaccessible<br>`protected: y`<br>`public: z` |

# 11.12 Constructors,Destructors and Inheritance

- By inheriting every member of the base class, a derived class object contains a base class object
- The derived class constructor can specify which base class constructor should be used to initialize the base class object

# Order of Execution

- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

# Order of Execution

```
// Student – base class
// UnderGrad – derived class
// Both have constructors, destructors
int main()
{
   UnderGrad u1;
   ...
   return 0;
}// end main
```

Execute **Student** constructor, then execute **UnderGrad** constructor

Execute **UnderGrad** destructor, then execute **Student** destructor

# Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading
- Can also be done with inline constructors
- Must be done if base class has no default constructor

# Passing Arguments to Base Class Constructor

```
class Parent {
    int x, y;
    public: Parent(int,int);
};
class Child : public Parent {
    int z
    public:
    Child(int a): Parent(a,a*a)
    {z = a;}
};
```

# 11.13 Overriding Base Class Functions

- Overriding: function in a derived class that has the *same name and parameter list* as a function in the base class
- Typically used to replace a function in base class with different actions in derived class
- Not the same as overloading – with overloading, the parameter lists must be different

# Access to Overridden Function

- When a function is overridden, all objects of derived class use the overriding function.
- If necessary to access the overridden version of the function, it can be done using the scope resolution operator with the name of the base class and the name of the function:

```
Student::getName();
```

# Chapter 12: More on C-Strings and the `string` Class

# Topics

12.1 C-Strings

12.2 Library Functions for Working with C-Strings

12.3 Conversions Between Numbers and Strings

12.4 Writing Your Own C-String Handling Functions

12.5 More About the C++ **`string`** Class

12.6 Creating Your Own String Class

# 12.1 C-Strings

- C-string: sequence of characters stored in adjacent memory locations and terminated by **`NULL`** character
- The C-string

**`"Hi there!"`**

would be stored in memory as shown:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| `H` | `i` | | `t` | `h` | `e` | `r` | `e` | `!` | `\0` |

# What is NULL?

- The null character is used to indicate the end of a string
- It can be specified as
  - the character **`'\0'`**
  - the **`int`** value 0
  - the named constant **`NULL`**

# Representation of C-strings

As a string literal

```
"Hi There!"
```

As a pointer to **char**

```
char *p;
```

As an array of characters

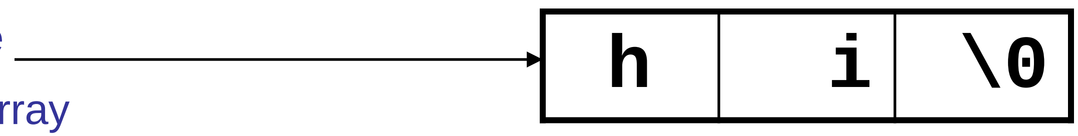
```
char str[20];
```

All three representations are pointers to char

# String Literals

- A string literal is stored as a null-terminated array of **char**
- Compiler uses the address of the first character of the array as the value of the string
- String literal is a pointer to char

value of "hi" is the
address of this array



# Array of **char**

- An array of char can be defined and initialized to a C-string

```
char str1[20] = "hi";
```

- An array of char can be defined and later have a string copied into it using **`strcpy`** or **`cin.getline`**

```
char str2[20], str3[20];
strcpy(str2, "hi");
cout << "Enter your name: ";
cin.getline(str3, 20);
```

# Array of **char**

- The name of an array of char is used as a pointer to char
- Unlike a string literal, a C-string defined as an array can be referred to in other parts of the program by using the array name

# Pointer to **char**

- Defined as

```
char *pStr;
```

- Does not itself allocate memory
- Useful in repeatedly referring to C-strings defined as a string literal

```
pStr = "Hi there";
cout << pStr << " "
     << pStr;
```

# Pointer to **char**

- Pointer to **char** can also refer to C-strings defined as arrays of char

```
char str[20] = "hi";
char *pStr = str;
cout << pStr;  // prints hi
```

- Can dynamically allocate memory to be used for C-string using **new**

# 12.2 Library Functions for Working with C-Strings

- Require **`cstring`** header file
- Functions take one or more C-strings as arguments. Argument can be:
  - Name of an array of char
  - pointer to char
  - string literal

# Library Functions for Working with C-Strings

```
int strlen(char *str)
```

Returns length of a C-string:

```
cout << strlen("hello");
```

Prints: **5**

Note: This is the number of characters in the string, NOT the size of the array that contains it

# strcat

```
strcat(char *dest, char *source)
```

- Takes two C-strings as input.  It adds the contents of the second string to the end of the first string:

```
char str1[15] = "Good ";
char str2[30] = "Morning!";
strcat(str1, str2);
cout << str1; // prints: Good Morning!
```

- No automatic bounds checking: programmer must ensure that 1st string has enough room for result

# strcpy

```
strcpy(char *dest, char *source)
```

- Copies a string from a source address to a destination address

```
char name[15];
strcpy(name, "Deborah");
cout << name; // prints Deborah
```

- Again, no automatic bounds checking

# strcmp

**`int strcmp(char *str1, char*str2)`**

- Compares strings stored at two addresses to determine their relative alphabetic order:
- Returns a value:

    less than 0 if **`str1`** precedes **`str2`**

    equal to 0 if **`str1`** equals **`str2`**

    greater than 0 if **`str1`** succeeds **`str2`**

# strcmp

- Often used to test for equality

```
if(strcmp(str1, str2) == 0)
    cout << "equal";
else
    cout << "not equal";
```

- Also used to determine ordering of C-strings in sorting applications
- Note:
  - Comparisons are case-sensitive: **"Hi" != "hi"**
  - C-strings cannot be compared using == (compares addresses of C-strings, not contents)

# strstr

**`char *strstr(char *str1,char *str2)`**

- Searches for the occurrence of **`str2`** within **`str1`**.
- Returns a pointer to the occurrence of **`str2`** within **`str1`** if found, and returns **`NULL`** otherwise

```
char s[15] = "Abracadabra";
char *found = strstr(s,"dab");
cout << found;      // prints dabra
```

# 12.3 Conversions Between Numbers and Strings

- **`"1416"`** is a string; **`1416`** without quotes is an **`int`**
- There are classes that can be used to convert between string and numeric forms of numbers
- Need to include **`sstream`** header file

# Conversion Classes

- **`istringstream:`**
  - contains a string to be converted to numeric values where necessary
  - Use **`str(s)`** to initialize string to contents of **`s`**
  - Use the stream extraction operator **`>>`** to read from the string
- **`ostringstream:`**
  - collects a string in which numeric data is converted as necessary
  - Use the stream insertion operator **`<<`** to add data onto the string
  - Use **`str()`** to retrieve converted string

# **atoi** and **atol**

- **atoi** converts **a**lphanumeric **to** **i**nt
- **atol** converts **a**lphanumeric **to** **l**ong

```
int atoi(char *numericStr)
long atol(char *numericStr)
```

- Examples:

```
int number; long lnumber;
number = atoi("57");
lnumber = atol("50000");
```

# atof

- **atof** converts a numeric string to a floating point number, actually a double

```
double atof(char *numericStr)
```

- Example:

```
double dnumber;
dnumber = atof("3.14159");
```

# **atoi**, **atol**, **atof**

- if C-string being converted contains non-digits, results are undefined
  - function may return result of conversion up to first non-digit
  - function may return 0
- All functions require **cstdlib**

# itoa

- **itoa** converts an **i**nt **to** an **a**lphanumeric string
- Allows user to specify the base of conversion

```
itoa(int num, char *numStr, int base)
```

- Example: To convert the number 1200 to a hexadecimal string

```
char numStr[10];
itoa(1200, numStr, 16);
```

- The function performs no bounds-checking on the array **numStr**

# 12.4 Character Testing

require **`cctype`** header file

| FUNCTION | MEANING |
|---|---|
| **`isalpha`** | **`true`** if arg. is a letter, **`false`** otherwise |
| **`isalnum`** | **`true`** if arg. is a letter or digit, **`false`** otherwise |
| **`isdigit`** | **`true`** if arg. is a digit 0-9, **`false`** otherwise |
| **`islower`** | **`true`** if arg. is lowercase letter, **`false`** otherwise |

# 12.4 Writing Your Own C-String Handling Functions

When writing C-String Handling Functions:

- can pass arrays or pointers to **char**
- can perform bounds checking to ensure enough space for results
- can anticipate unexpected user input

# 12.5 More About the C++ `string` Class

- The string class offers several advantages over C-style strings:
  - large body of member functions
  - overloaded operators to simplify expressions
- Need to include the `string` header file

# `string` class constructors

- Default constructor **`string()`**
- Copy constructor **`string(string&)`** initializes string objects with values of other string objects
- Convert constructor **`string(char *)`** initializes string objects with values of C-strings
- Various other constructors

# Overloaded `string` Operators

| OPERATOR | MEANING |
|---|---|
| `>>` | reads whitespace-delimited strings into string object |
| `<<` | inserts string object into a stream |
| `=` | assigns string on right to string object on left |
| `+=` | appends string on the right to the end of contents of string on left |

# Overloaded **`string`** Operators (continued)

| **OPERATOR** | **MEANING** |
|---|---|
| **`+`** | Returns concatenation of the two strings |
| **`[]`** | references character in string using array notation |
| **`>, >=,`**<br>**`<, <=,`**<br>**`==, !=`** | relational operators for string comparison. Return **`true`** or **`false`** |

# Overloaded **string** Operators

```
string word1, phrase;
string word2 = " Dog";
cin >> word1; // user enters "Hot"
              // word1 has "Hot"
phrase = word1 + word2; // phrase has
                        // "Hot Dog"
phrase += " on a bun";
for (int i = 0; i < 16; i++)
   cout << phrase[i];  // displays
                       // "Hot Dog on a bun"
```

# **string** Member Functions

Categories:

- conversion to C-strings: **c_str**, **data**
- modification: **append**, **assign**, **clear**, **copy**, **erase**, **insert**, **replace**, **swap**
- space management: **capacity**, **empty**, **length**, **resize**, **size**
- substrings: **find**, **substr**
- comparison: **compare**

# Conversion to C-strings

- **`data()`** and **`c_str()`** both return the C-string equivalent of a **`string`** object
- Useful when using a string object with a function that is expecting a C-string

```
char greeting[20] = "Have a ";
string str("nice day");
strcat(greeting, str.data());
```

# Modification of **`string`** objects

- **`str.append(string s)`**

  appends contents of **`s`** to end of **`str`**
- Convert constructor for **`string`** allows a C-string to be passed in place of **`s`**

```
string str("Have a ");
str.append("nice day");
```

- **`append`** is overloaded for flexibility

# Modification of **`string`** objects

- **`str.insert(int pos, string s)`** inserts **`s`** at position **`pos`** in **`str`**
- Convert constructor for **`string`** allows a C-string to be passed in place of **`s`**

```
string str("Have a day");
str.insert(7, "nice ");
```

- **`insert`** is overloaded for flexibility

# 12.6 Creating Your Own String Class

- A good way to put OOP skills into practice
- The class allocates dynamic memory, so has copy constructor, destructor, and overloaded assignment
- Overloads the stream insertion and extraction operators, and many other operators

# Chapter 13: Advanced File and I/O Operations

# Topics

13.1 Input and Output Streams

13.2 More Detailed Error Testing

13.3 Member Functions for Reading and Writing Files

13.4 Binary Files

13.5 Creating Records with Structures

13.6 Random-Access Files

13.7 Opening a File for Both Input and Output

# 13.1 Input and Output Streams

- Input Stream – data stream from which information can be read
  - Ex: **`cin`** and the keyboard
  - Use **`istream`**, **`ifstream`**, and **`istringstream`** objects to read data
- Output Stream – data stream to which information can be written
  - Ex: **`cout`** and monitor screen
  - Use **`ostream`**, **`ofstream`**, and **`ostringstream`** objects to write data
- Input/Output Stream – data stream that can be both read from and written to
  - Use **`fstream`** objects here

# File Stream Classes

- **`ifstream`** (open primarily for input), **`ofstream`** (open primarily for output), and **`fstream`** (open for either or both input and output)
- All have **`open`** member function to connect the program to an external file
- All have **`close`** member function to disconnect program from an external file when access is finished
  - Files should be open for as short a time as possible
  - Always close files before the program ends

# File Open Modes

- File open modes specify how a file is opened and what can be done with the file once it is open
- **`ios::in`** and **`ios::out`** are examples of file open modes, also called file mode flag
- File modes can be combined and passed as second argument of open member function

# The **`fstream`** Object

- **`fstream`** object can be used for either input or output

```
fstream file;
```

- To use **`fstream`** for input, specify **`ios::in`** as the second argument to open

```
file.open("myfile.dat",ios::in);
```

- To use **`fstream`** for output, specify **`ios::out`** as the second argument to open

```
file.open("myfile.dat",ios::out);
```

# File Mode Flags

| | |
|---|---|
| **`ios::app`** | create new file, or append to end of existing file |
| **`ios::ate`** | go to end of existing file; write anywhere |
| **`ios::binary`** | read/write in binary mode (not text mode) |
| **`ios::in`** | open for input |
| **`ios::out`** | open for output |

# Opening a File for Input and Output

- **`fstream`** object can be used for both input and output at the same time
- Create the fstream object and specify both **`ios::in`** and **`ios::out`** as the second argument to the open member function

```
fstream file;
file.open("myfile.dat",
          ios::in|ios::out);
```

# File Open Modes

- Not all combinations of file open modes make sense
- **`ifstream`** and **`ofstream`** have default file open modes defined for them, hence the second parameter to their **`open`** member function is optional

# Opening Files with Constructors

- Stream constructors have overloaded versions that take the same parameters as **open**
- These constructors open the file, eliminating the need for a separate call to **open**

```
fstream inFile("myfile.dat",
               ios::in);
```

# Default File Open Modes

- **`ofstream:`**
  - open for output only
  - file cannot be read from
  - file is created if no file exists
  - file contents erased if file exists

- **`ifstream`:**
  - open for input only
  - file cannot be written to
  - open fails if the file does not exist

# Output Formatting with I/O Manipulators

- Can format with I/O manipulators: they work with file objects just like they work with **`cout`**
- Can format with formatting member functions
- The **`ostringstream`** class allows in-memory formatting into a string object before writing to a file

# I/O Manipulators

| | |
|---|---|
| **`left, right`** | left or right justify output |
| **`oct, dec, hex`** | display output in octal, decimal, or hexadecimal |
| **`endl, flush`** | write newline (**`endl`** only) and flush output |
| **`showpos, noshowpos`** | do, do not show leading **+** with non-negative numbers |
| **`showpoint, noshowpoint`** | do, do not show decimal point and trailing zeroes |

# More I/O Manipulators

| | |
|---|---|
| **`fixed`, `scientific`** | use fixed or scientific notation for floating-point numbers |
| **`setw(n)`** | sets minimum field output width to **`n`** |
| **`setprecision(n)`** | sets floating-point precision to **`n`** |
| **`setfill(ch)`** | uses **`ch`** as fill character |

# **sstream** Formatting

1) To format output into an in-memory string object, include the **sstream** header file and create an **ostringstream** object

```
#include <sstream>
ostringstream outStr;
```

# **`sstream`** Formatting

2) Write to the **`ostringstream`** object using I/O manipulators, all other stream member functions:

```
outStr << showpoint << fixed
       << setprecision(2)
       << '$'<< amount;
```

# **sstream** Formatting

3) Access the C-string inside the **ostringstream** object by calling its **str** member function

```
cout << outStr.str();
```

# 13.2 More Detailed Error Testing

- Stream objects have error bits (flags) that are set by every operation to indicate success or failure of the operation, and the status of the stream
- Stream member functions report on the settings of the flags

# Error State Bits

Can examine error state bits to determine file stream status

| | |
|---|---|
| **`ios::eofbit`** | set when end of file detected |
| **`ios::failbit`** | set when operation failed |
| **`ios::hardfail`** | set when an irrecoverable error occurred |
| **`ios::badbit`** | set when invalid operation attempted |
| **`ios::goodbit`** | set when no other bits are set |

# Error Bit Reporting Functions

| | |
|---|---|
| **`eof()`** | true if **`eofbit`** set, false otherwise |
| **`fail()`** | true if **`failbit`** or **`hardfail`** set, false otherwise |
| **`bad()`** | true if **`badbit`** set, false otherwise |
| **`good()`** | true if **`goodbit`** set, false otherwise |
| **`clear()`** | clear all flags (no arguments), or clear a specific flag |

# Detecting File Operation Errors

- The file handle is set to true if a file operation succeeds. It is set to false when a file operation fails
- Test the status of the stream by testing the file handle:

```
inFile.open("myfile");
if (!inFile)
  { cout << "Can't open file";
    exit(1);
  }
```

# 13.3 Member Functions for Reading and Writing Files

Unlike the extraction operator **`>>`**, these reading functions do not skip whitespace:

- **`getline`**: read a line of input
- **`get`**: reads a single character
- **`seekg`**: goes to beginning of input file

# **`getline`** Member Function

```
getline(char s[ ],
          int max, char stop ='\n')
```

- **`char s[ ]:`** Character array to hold input
- **`int max`** : 1 more than the maximum number of characters to read
- **`char stop`**: Terminator to stop at if encountered before **`max`** number of characters is read . Optional, default is **`'\n'`**

# Single Character Input

```
get(char &ch)
```

Read a single character from the input stream and put it in **ch**. Does not skip whitespace.

```
ifstream inFile;  char ch;
inFile.open("myFile");
inFile.get(ch);
cout << "Got " << ch;
```

# Single Character Input, Again

**`get()`**

Read a single character from the input stream and return the character. Does not skip whitespace.

```
ifstream inFile;  char ch;
inFile.open("myFile");
ch = inFile.get();
cout << "Got " << ch;
```

# Single Character Input, with a Difference

**`peek()`**

Read a single character from the input stream but do not remove the character from the input stream. Does not skip whitespace.

```
ifstream inFile;  char ch;
inFile.open("myFile");
ch = inFile.peek();
cout << "Got " << ch;
ch = inFile.peek();
cout << "Got " << ch;//same output
```

# Single Character Output

- **`put(char ch)`**

  Output a character to a file
- Example

```
ofstream outFile;
outFile.open("myfile");
outFile.put('G');
```

# Moving About in Input Files

**`seekg(offset, place)`**

Move to a given **`offset`** relative to a given **`place`** in the file

- **`offset`**: number of bytes from **`place`**, specified as a **`long`**
- **`place`**: location in file from which to compute offset

  **`ios::beg`**: beginning of file

  **`ios::end`**: end of the file

  **`ios::cur`**: current position in file

# Example of Single Character I/O

To copy an input file to an output file

```
char ch; infile.get(ch);
while (!infile.fail())
{ outfile.put(ch);
  infile.get(ch);
}
infile.close();
outfile.close();
```

# Rewinding a File

- To move to the beginning of file, seek to an offset of zero from beginning of file

```
inFile.seekg(0L, ios::beg);
```

- Error or eof bits will block seeking to the beginning of file. Clear bits first:

```
inFile.clear();
inFile.seekg(0L, ios::beg);
```

# 13.4 Binary Files

- Binary files store data in the same format that a computer has in main memory
- Text files store data in which numeric values have been converted into strings of ASCII characters
- Files are opened in text mode (as text files) by default

# Using Binary Files

- Pass the **`ios::binary`** flag to the **`open`** member function to open a file in binary mode

```
infile.open("myfile.dat",ios::binary);
```

- Reading and writing of binary files requires special **`read`** and **`write`** member functions

```
read(char *buffer, int numberBytes)
write(char *buffer, int numberBytes)
```

# Using **read** and **write**

```
read(char *buffer, int numberBytes)
write(char *buffer, int numberBytes)
```

- **buffer:** holds an array of bytes to transfer between memory and the file
- **numberBytes**: the number of bytes to transfer

Address of the buffer needs to be cast to **char *** using **reinterpret_cast <char *>**

# Using **`write`**

To write an array of 2 doubles to a binary file

```
ofstream outFile("myfile",ios:binary);
double d[2] = {12.3, 34.5};
outFile.write(
reinterpret_cast<char *>(d),sizeof(d));
```

# Using **read**

To read two 2 doubles from a binary file into an array

```
ifstream inFile("myfile", ios:binary);
const int DSIZE = 10;
double data[DSIZE];
inFile.read(
    reinterpret_cast<char *>(data),
    2*sizeof(double));
// only data[0] and data[1] contain
// values
```

# 13.5 Creating Records with Structures

- Can write structures to, read structures from files
- To work with structures and files,
  - use **`binary`** file flag upon open
  - use **`read`**, **`write`** member functions

# Creating Records with Structures

```
struct TestScore
{ int studentId;
  float score;
  char grade;
};
TestScore test1[20];
...
// write out test1 array to a file
gradeFile.write(
  reinterpret_cast<char*>(test1),
  sizeof(test1));
```

# Notes on Structures Written to Files

- Structures to be written to a file must not contain pointers
- Since string objects use pointers and dynamic memory internally, structures to be written to a file must not contain any string objects

# 13.6 Random-Access Files

- Sequential access: start at beginning of file and go through data the in file, in order, to the end of the file
  - to access 100th entry in file, go through 99 preceding entries first
- Random access: access data in a file in any order
  - can access 100th entry directly

# Random Access Member Functions

- **seekg** (seek get): used with input files
- **seekp** (seek put): used with output files

Both are used to go to a specific position in a file

# Random Access Member Functions

**`seekg(offset,place)`**
**`seekp(offset,place)`**

**`offset`**: long integer specifying number of bytes to move

**`place`**: starting point for the move, specified by **`ios:beg`**, **`ios::cur`** or **`ios:end`**

# Random-Access Member Functions

- Examples:

```
// Set read position 25 bytes
// after beginning of file
inData.seekg(25L, ios::beg);

// Set write position 10 bytes
// before current position
outData.seekp(-10L, ios::cur);
```

# Random Access Information

- **`tellg`** member function: return current byte position in input file, as a **`long`**

```
long whereAmI;
whereAmI = inFile.tellg();
```

- **`tellp`** member function: return current byte position in output file, as a **`long`**

```
whereAmI = outFile.tellp();
```

# 13.7 Opening a File for Both Input and Output

- A file can be open for input and output simultaneously
- Supports updating a file:
  - read data from file into memory
  - update data
  - write data back to file
- Use **`fstream`** for file object definition:

```
fstream gradeList("grades.dat",
                  ios::in | ios::out);
```

# Chapter 14: Recursion

# Topics

14.1 Introduction to Recursion
14.2 The Recursive Factorial Function
14.3 The Recursive gcd Function
14.4 Solving Recursively Defined Problems
14.5 A Recursive Binary Search Function
14.6 The QuickSort Algorithm
14.7 The Towers of Hanoi
14.8 Exhaustive and Enumeration Algorithms
14.9 Recursion Versus Iteration

# 14.1 Introduction to Recursion

- A recursive function is a function that calls itself.
- Recursive functions can be useful in solving problems that can be broken down into smaller or simpler subproblems of the same type. A base case should eventually be reached, at which time the breaking down (recursion) will stop.

# Recursive Functions

Consider a function for solving the count-down problem from some number `num` down to `0`:

– The base case is when `num` is already `0`: the problem is solved and we "blast off!"

– If `num` is greater than `0`, we count off `num` and then recursively count down from `num-1`

# Recursive Functions

A recursive function for counting down to 0:

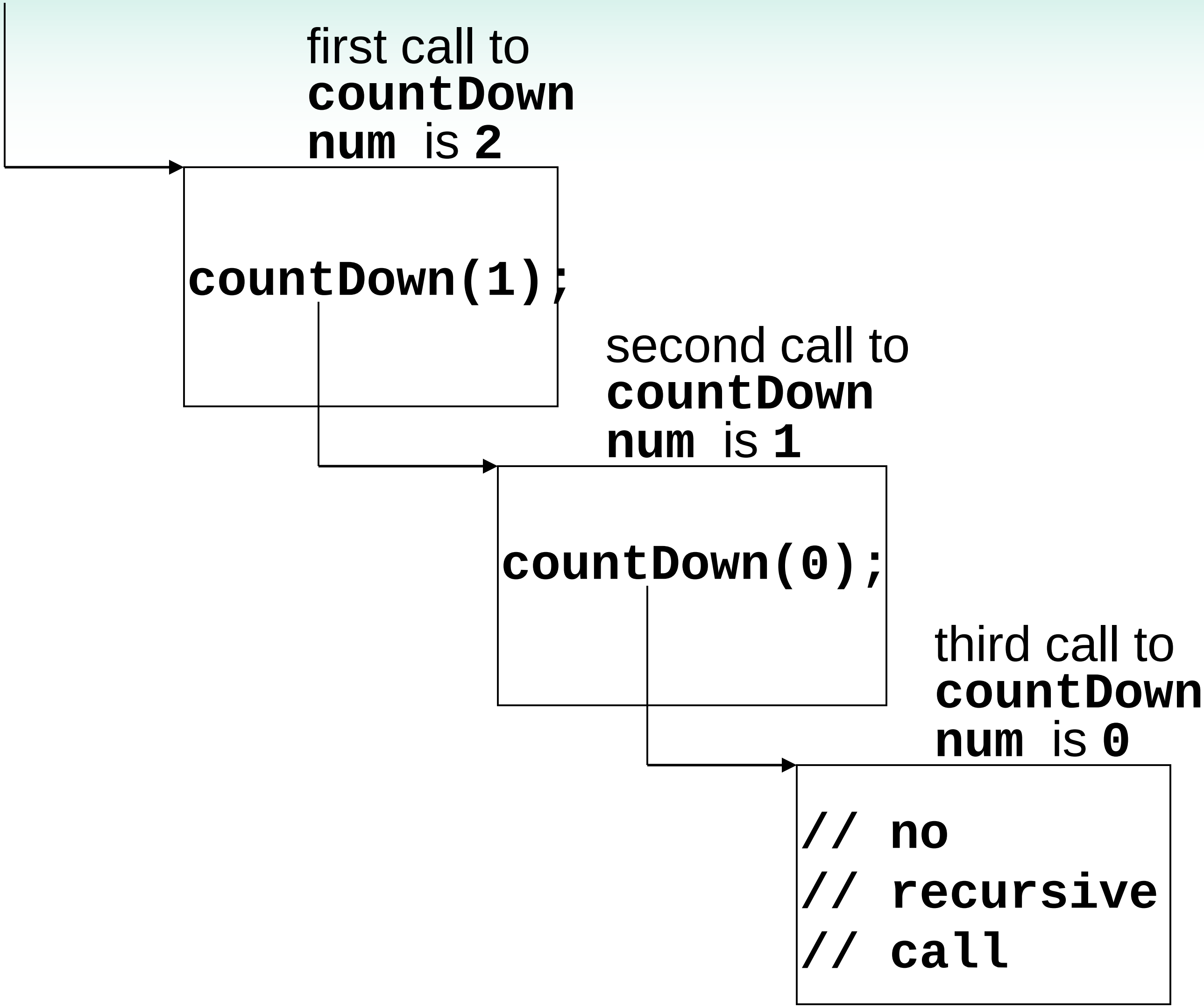
```
void countDown(int num)
{
   if (num == 0)
       cout << "Blastoff!";
    else
    {
       cout << num << ". . .";
       countDown(num-1); // recursive
    }                    // call
}
```

# What Happens When Called?

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

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

# What Happens When Called?




OUTPUT:

**2...**

**1...**

**Blastoff!**

# Stopping the Recursion

- A recursive function should include a test for the base cases
- In the sample program, the test is:

```
if (num == 0)
```

# Stopping the Recursion

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

# Stopping the Recursion

- With each recursive call, the parameter controlling the recursion should move closer to the base case
- Eventually, the parameter reaches the base case and the chain of recursive calls terminates

# Stopping the Recursion

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

Value passed to recursive call is closer to base case of num = 0.

# What Happens When Called?

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

# Types of Recursion

- Direct recursion
  - a function calls itself
- Indirect recursion
  - function A calls function B, and function B calls function A. Or,
  - function A calls function B, which calls …, which then calls function A

# 14.2 The Recursive Factorial Function

- The factorial of a nonnegative integer $n$ is the product of all positive integers less or equal to $n$
- The factorial of $n$ is denoted by $n!$
- The factorial of 0 is 1

$$0\,! = 1$$

$$n\,! = n \times (n-1) \times \ldots \times 2 \times 1 \quad \text{if} \quad n > 0$$

# Recursive Factorial Function

- Factorial of $n$ can be expressed in terms of the factorial of $n$-1

  $$0\,! = 1$$

  $$n\,! = n \times (n-1)\,!$$

- Recursive function

```
int factorial(int n)
{ if (n == 0) return 1;
  else
      return n *factorial(n-1);
}
```

# 14.3 The Recursive gcd Function

- Greatest common divisor (gcd) of two integers $x$ and $y$ is the largest number that divides both $x$ and $y$
- The Greek mathematician Euclid discovered that
  - If $y$ divides $x$, then gcd($x$, $y$) is just $y$
  - Otherwise, the gcd($x$, $y$) is the gcd of $y$ and the remainder of dividing $x$ by $y$

# The Recursive gcd Function

```
int gcd(int x, int y)
{
    if (x % y == 0) //base case
        return y;
    else
        return gcd(y, x % y);
}
```

# 14.4 Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:

  **`0, 1, 1, 2, 3, 5, 8, 13, 21, ...`**
- After the initial **`0`** and **`1`**, each term is the sum of the two preceding terms
- Recursive calculation of the nth Fibonacci number:

  **`fib(n) = fib(n – 1) + fib(n – 2);`**
- Base cases: **`n == 0, n == 1`**

# Recursive Fibonacci Function

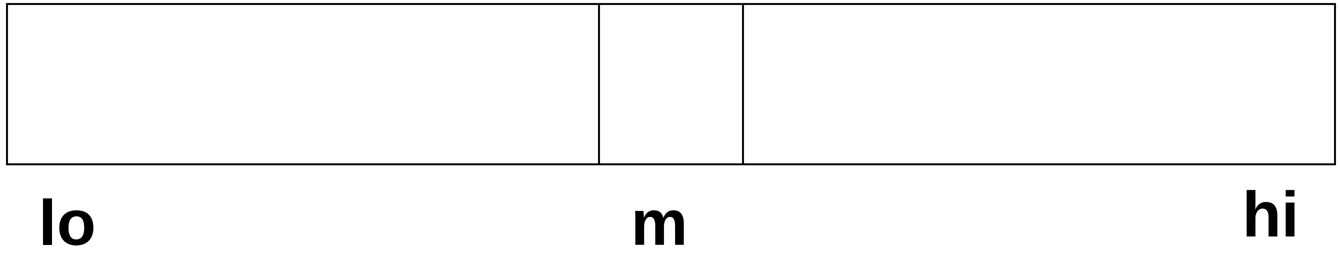
```
int fib(int n)
{
   if (n <= 0)            // base case
       return 0;
   else if (n == 1)       // base case
       return 1;
   else
         return fib(n – 1) + fib(n – 2);
}
```

# 14.5 A Recursive Binary Search Function

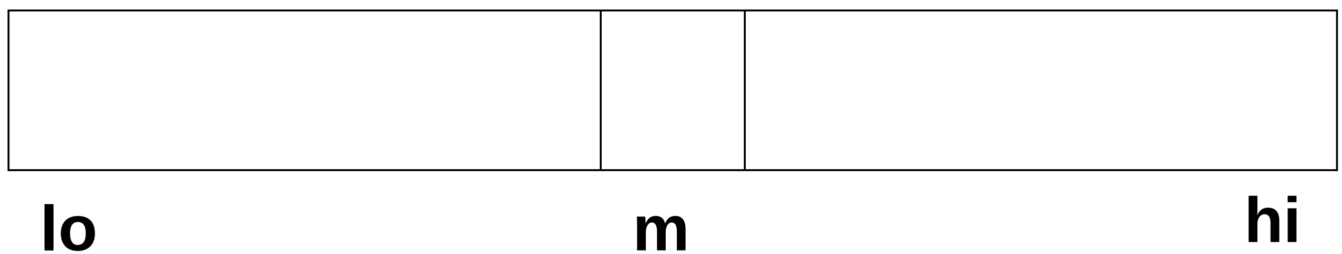
- Assume an array **`a`** that is sorted in ascending order, and an item **`X`** to search for
- We want to write a function that searches for **`X`** within the array **`a`**, returning the index of **`X`** if it is found, and returning **`-1`** if **`X`** is not in the array

# Recursive Binary Search

A recursive strategy for searching a portion of the array from index `lo` to index `hi` is to set `m` to the index of the middle element of the array:



# Recursive Binary Search




If `a[m] == X`, we found `X`, so return `m`

If `a[m] > X`, recursively search `a[lo..m-1]`

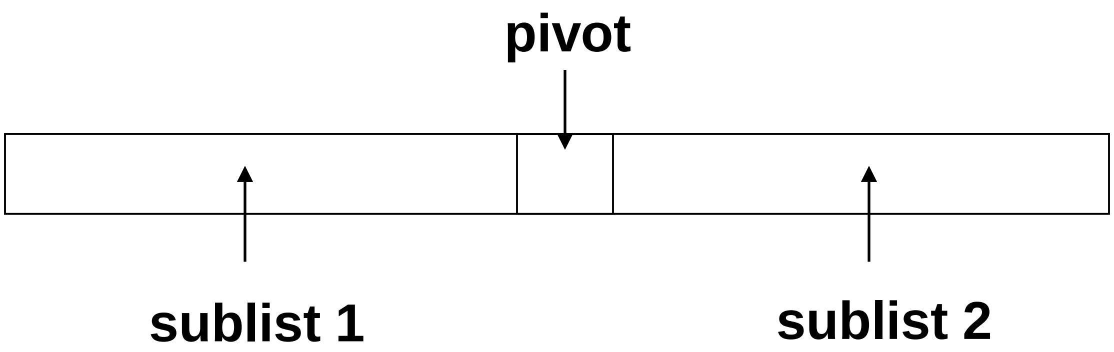
If `a[m] < X`, recursively search `a[m+1..hi]`

# Recursive Binary Search

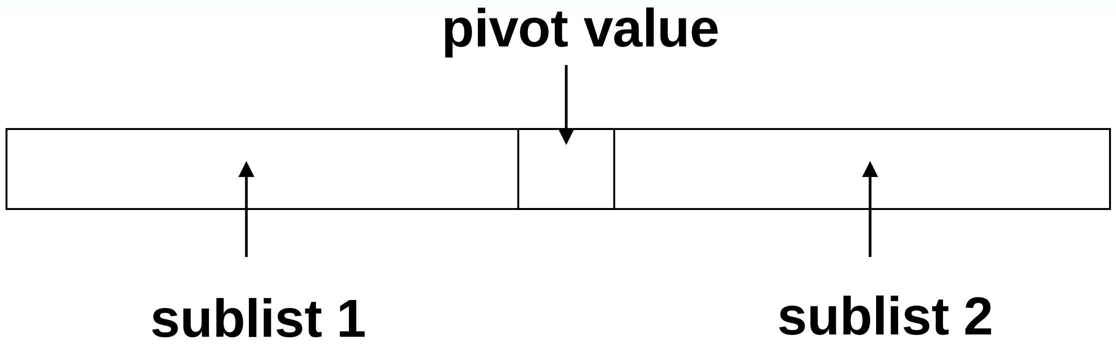
```
int bSearch(int a[],int lo,int hi,int X)
{
  int m = (lo + hi) /2;
  if(lo > hi) return -1;     // base
  if(a[m] == X) return m;    // base
  if(a[m] > X)
    return bsearch(a,lo,m-1,X);
  else
    return bsearch(a,m+1,hi,X);
}
```

# 14.6 The QuickSort Algorithm

- Recursive algorithm that can sort an array
- First, determine an element to use as pivot_value:



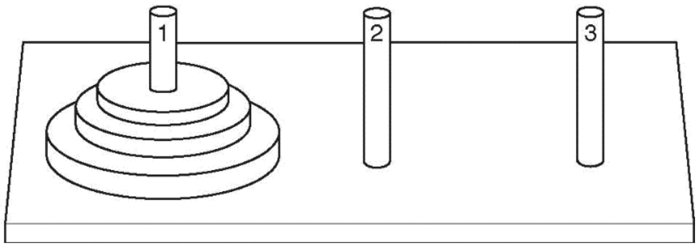
# The QuickSort Algorithm




- Then, values are shifted so that elements in sublist1 are < pivot and elements in sublist2 are >= pivot
- Algorithm then recursively sorts sublist1 and sublist2
- Base case: a sublist has size <=1

# 14.7 The Towers of Hanoi

- Setup: 3 pegs, one has n disks on it, the other two pegs empty. The disks are arranged in increasing diameter, top→ bottom
- Objective: move the disks from peg 1 to peg 3, observing these rules:
  - only one disk moves at a time
  - all remain on pegs except the one being moved
  - a larger disk cannot be placed on top of a smaller disk at any time



# The Towers of Hanoi

How it works:

| | |
|---|---|
| n=1 | Move disk from peg 1 to peg 3.<br>Done. |
| n=2 | Move top disk from peg 1 to peg 2.<br>Move remaining disk from peg 1 to peg 3.<br>Move disk from peg 2 to peg 3.<br>Done. |

# Outline of Recursive Algorithm

If n==0, do nothing (base case)

If n>0, then

a. Move the topmost n-1 disks from peg1 to peg2
b. Move the $n^{th}$ disk from peg1 to peg3
c. Move the n-1 disks from peg2 to peg3

end if

# 14.8 Exhaustive and Enumeration Algorithms

- Enumeration algorithm: generate all possible combinations

  Example: all possible ways to make change for a certain amount of money

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

  Example: change for a certain amount of money that uses the fewest coins

# 14.9 Recursion vs. Iteration

- Benefits (+), disadvantages(-) for recursion:
  - + Natural formulation of solution to certain problems
  - + Results in shorter, simpler functions
  - – May not execute very efficiently
- Benefits (+), disadvantages(-) for iteration:
  - + Executes more efficiently than recursion
  - – May not be as natural a method of solution as recursion for some problems

# Chapter 15: Polymorphism and Virtual Functions

# Topics

15.1 Type Compatibility in Inheritance Hierarchies

15.2 Polymorphism and Virtual Member Functions

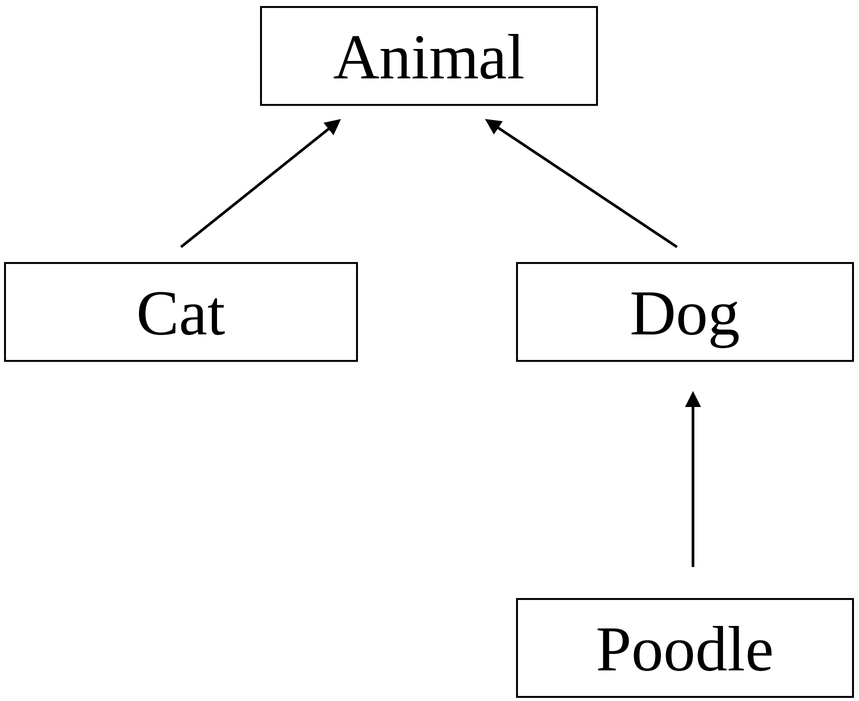
15.3 Abstract Base Classes and Pure Virtual Functions

15.4 Composition Versus Inheritance

# 15.1 Type Compatibility in Inheritance Hierarchies

- Classes in a program may be part of an inheritance hierarchy

- Classes lower in the hierarchy are special cases of those above



# Type Compatibility in Inheritance

- A pointer to a derived class can be assigned to a pointer to a base class. Another way to say this is:
- A base class pointer can point to derived class objects

```
Animal *pA = new Cat;
```

# Type Compatibility in Inheritance

- Assigning a base class pointer to a derived class pointer requires a cast

```
Animal *pA = new Cat;
Cat *pC;
pC = static_cast<Cat *>(pA);
```

- The base class pointer must already point to a derived class object for this to work

# Using Type Casts with Base Class Pointers

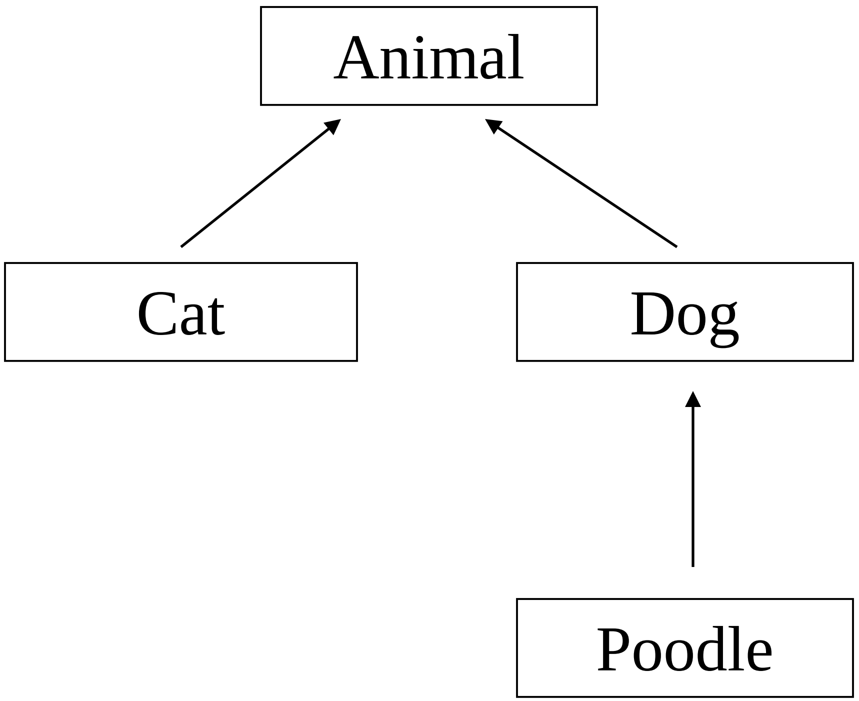
- C++ uses the declared type of a pointer to determine access to the members of the pointed-to object
- If an object of a derived class is pointed to by a base class pointer, all members of the derived class may not be accessible
- Type cast the base class pointer to the derived class (via **`static_cast`**) in order to access members that are specific to the derived class

# 15.2 Polymorphism and Virtual Member Functions

- Polymorphic code: Code that behaves differently when it acts on objects of different types
- Virtual Member Function: The C++ mechanism for achieving polymorphism

# Polymorphism

Consider the Animal, Cat, Dog hierarchy where each class has its own version of the member function id( )



# Polymorphism

```
class Animal{
 public: void id(){cout << "animal";}
}
class Cat : public Animal{
 public: void id(){cout << "cat";}
}
class Dog : public Animal{
 public: void id(){cout << "dog";}
}
```

# Polymorphism

- Consider the collection of different Animal objects

```
Animal *pA[] = {new Animal, new Dog,
                       new Cat};
```

and accompanying code

```
for(int k=0; k<3; k++)
    pA[k]->id();
```

- Prints: **`animal animal animal`**, ignoring the more specific versions of **`id()`** in **`Dog`** and **`Cat`**

# Polymorphism

- The preceding code is not polymorphic: it behaves the same way even though **`Animal`**, **`Dog`** and **`Cat`** have different types and different **`id()`** member functions
- Polymorphic code would have printed "**`animal dog cat"`** instead of "**`animal animal animal"`**

# Polymorphism

- The code is not polymorphic because in the expression

  **`pA[k]->id()`**

  the compiler sees only the type of the pointer **`pA[k]`**, which is pointer to **`Animal`**
- Compiler does not see type of actual object pointed to, which may be **`Animal`**, or **`Dog`**, or **`Cat`**

# Virtual Functions

Declaring a function **`virtual`** will make the compiler check the type of each object to see if it defines a more specific version of the virtual function

# Virtual Functions

If the member functions **`id()`** are declared virtual, then the code

```
Animal *pA[] = {new Animal,
                new Dog, new Cat};
for(int k=0; k<3; k++)
    pA[k]->id();
```

will print **`animal dog cat`**

# Virtual Functions

How to declare a member function virtual:

```
class Animal{
 public: virtual void id(){cout << "animal";}
}
class Cat : public Animal{
 public: virtual void id(){cout << "cat";}
}
class Dog : public Animal{
 public: virtual void id(){cout << "dog";}
}
```

# Function Binding

- In **`pA[k]->id()`**, Compiler must choose which version of **`id()`** to use: There are different versions in the **`Animal`**, **`Dog`**, and **`Cat`** classes
- Function binding is the process of determining which function definition to use for a particular function call
- The alternatives are *static* and *dynamic* binding

# Static Binding

- Static binding chooses the function in the class of the base class pointer, ignoring any versions in the class of the object actually pointed to
- Static binding is done at compile time

# Dynamic Binding

- Dynamic Binding determines the function to be invoked at execution time
- Can look at the actual class of the object pointed to and choose the most specific version of the function
- Dynamic binding is used to bind virtual functions

# 15.3 Abstract Base Classes and Pure Virtual Functions

- An abstract class is a class that contains no objects that are not members of subclasses (derived classes)
- For example, in real life, Animal is an abstract class: there are no animals that are not dogs, or cats, or lions…

# Abstract Base Classes and Pure Virtual Functions

- Abstract classes are an organizational tool. They are useful in organizing inheritance hierarchies
- Abstract classes can be used to specify an interface that must be implemented by all subclasses

# Abstract Functions

- The member functions specified in an abstract class do not have to be implemented
- The implementation is left to the subclasses
- In C++, an abstract class is a class with at least one abstract member function

# Pure Virtual Functions

- In C++, a member function of a class is declared to be an abstract function by making it virtual and replacing its body with **`= 0;`**

```
class Animal{
  public:
   virtual void id()=0;
};
```

- A virtual function with its body omitted and replaced with **`=0`** is called a pure virtual function, or an abstract function

# Abstract Classes

- An abstract class can not be instantiated
- An abstract class can only be inherited from; that is, you can derive classes from it
- Classes derived from abstract classes must override all pure virtual functions with a concrete member functions before they can be instantiated.

# 15.4 Composition vs. Inheritance

- Inheritance models an 'is a' relation between classes.  An object of a derived class 'is a(n)' object of the base class
- Example:
  - an **`UnderGrad`** is a **`Student`**
  - a **`Mammal`** is an **`Animal`**
  - a **`Poodle`** is a **`Dog`**

# Composition vs. Inheritance

- When defining a new class:
- Composition is appropriate when the new class needs to use an object of an existing class
- Inheritance is appropriate when
  - objects of the new class are a subset of the objects of the existing class, or
  - objects of the new class will be used in the same ways as the objects of the existing class

# Chapter 16: Exceptions, Templates, and the Standard Template Library (STL)

# Topics

16.1 Exceptions

16.2 Function Templates

16.3 Class Templates

16.4 Class Templates and Inheritance

16.5 Introduction to the Standard Template Library

# 16.1 Exceptions

- An exception is a value or an object that indicates that an error has occurred
- When an exception occurs, the program must either terminate or jump to special code for handling the exception.
- The special code for handling the exception is called an exception handler

# Exceptions – Key Words

- `throw` – followed by an argument, is used to signal an exception
- `try` – followed by a block `{ }`, is used to invoke code that throws an exception
- `catch` – followed by a block `{ }`, is used to process exceptions thrown in a preceding `try` block.  It takes a parameter that matches the type of exception thrown

# Throwing an Exception

- Code that detects the exception must pass information to the exception handler. This is done using a **`throw`** statement:

```
throw "Emergency!"
throw 12;
```

- In C++, information thrown by the **`throw`** statement may be a value of any type

# Catching an Exception

- Block of code that handles the exception is said to catch the exception and is called an exception handler
- An exception handler is written to catch exceptions of a given type: For example, the code

```
catch(char *str)
{
   cout << str;
}
```

can only catch exceptions of type C-string

# Catching an Exception

Another example of a handler:

```
catch(int x)
{
    cerr << "Error: " << x;
}
```

This can catch exceptions of type **int**

# Connecting to the Handler

Every catch block is attached to a try block of code and is responsible for handling exceptions thrown from that block

```
try
{

}
catch(char e1)
{
    // This code handles exceptions
    // of type char that are thrown
    // in this block
}
```

# Execution of Catch Blocks

- The catch block syntax is similar to a that of a function
- A catch block has a formal parameter that is initialized to the value of the thrown exception before the block is executed

# Exception Example

- An example of exception handling is code that computes the square root of a number.
- It throws an exception in the form of a C-string if the user enters a negative number

# Example

```
int main( )
{
  try
  {
    double x;
    cout << "Enter a number: ";
    cin >> x;
    if (x < 0) throw "Bad argument!";
    cout << "Square root of " << x << " is " << sqrt(x);
  }
  catch(char *str)
  {
       cout << str;
  }
  return 0;
}
```

# Flow of Control

1. Computer encounters a **`throw`** statement in a **`try`** block
2. The computer evaluates the **`throw`** expression, and immediately exits the **`try`** block
3. The computer selects an attached **`catch`** block that matches the type of the thrown value, places the value in the catch block's formal parameter, and executes the catch block

# Uncaught Exception

- An exception may be uncaught if
  - there is no **`catch`** block with a data type that matches the exception that was thrown, or
  - it was not thrown from within a **`try`** block
- The program will terminate in either case

# Handling Multiple Exceptions

Multiple catch blocks can be attached to the same block of code. The catch blocks should handle exceptions of different types

```
try{...}
catch(int iEx){ }
catch(char *strEx){ }
catch(double dEx){ }
```

# Throwing an Exception Class

- An exception class can be defined and thrown
- A catch block must be designed to catch an object of the exception class
- The exception class object can pass data to exception handler via data members

# Exception When Calling **new**

- If **new** cannot allocate memory, it throws an exception of type **bad_alloc**
- Must **#include <new>** to use **bad_alloc**
- Can invoke **new** from within a **try** block, and use a **catch** block to detect that memory was not allocated.

# Nested Exception Handling

**`try`** blocks can be nested in other **`try`** blocks and even in catch blocks

```
try
{
    try{ } catch(int i){ }
}
catch(char *s)
{ }
```

# Where to Find an Exception Handler?

- The compiler looks for a suitable handler attached to an enclosing **`try`** block in the same function
- If there is no matching handler in the function, it terminates execution of the function, and continues the search for a handler starting at the point of the call in the calling function.

# Unwinding the Stack

- An unhandled exception propagates backwards into the calling function and appears to be thrown at the point of the call
- The computer will keep terminating function calls and tracing backwards along the call chain until it finds an enclosing **`try`** block with a matching handler, or until the exception propagates out of **`main`** (terminating the program).
- This process is called unwinding the call stack

# Rethrowing an Exception

- Sometimes an exception handler may need to do some tasks, then pass the exception to a handler in the calling environment.
- The statement

```
throw;
```

with no parameters can be used within a **`catch`** block to pass the exception to a handler in the outer block

# 16.2 Function Templates

- Function template: A pattern for creating definitions of functions that differ only in the type of data they manipulate. It is a generic function

- They are better than overloaded functions, since the code defining the algorithm of the function is only written once

# Example

Two functions that differ only in the type of the data they manipulate

```
void swap(int &x, int &y)
 { int temp = x; x = y;
   y = temp;
 }

void swap(char &x, char &y)
 { char temp = x; x = y;
   y = temp;
 }
```

# A **swap** Template

The logic of both functions can be captured with one template function definition

```
template<class T>
void swap(T &x, T &y)
{ T temp = x; x = y;
  y = temp;
}
```

# Using a Template Function

- When a function defined by a template is called, the compiler creates the actual definition from the template by inferring the type of the type parameters from the arguments in the call:

```
int i = 1, j = 2;
swap(i,j);
```

- This code makes the compiler instantiate the template with type `int` in place of the type parameter `T`

# Function Template Notes

- A function template is a pattern
- No actual code is generated until the function named in the template is called
- A function template uses no memory
- When passing a class object to a function template, ensure that all operators referred to in the template are defined or overloaded in the class definition

# Function Template Notes

- All data types specified in template prefix must be used in template definition
- Function calls must pass parameters for all data types specified in the template prefix
- Function templates can be overloaded – need different parameter lists
- Like regular functions, function templates must be defined before being called

# Where to Start When Defining Templates

- Templates are often appropriate for multiple functions that perform the same task with different parameter data types
- Develop function using usual data types first, then convert to a template:
  - add template prefix
  - convert data type names in the function to a type parameter (*i.e.*, a T type) in the template

# 16.3 Class Templates

- It is possible to define templates for classes. Such classes define abstract data types
- Unlike functions, a class template is instantiated by supplying the type name (**`int`**, **`float`**, **`string`**, etc.) at object definition

# Class Template

Consider the following classes

1. Class used to join two integers by adding them:

```
class Joiner
{ public:
     int combine(int x, int y)
     {return x + y;}
};
```

2. Class used to join two strings by concatenating them:

```
class Joiner
{ public:
     string combine(string x, string y)
     {return x + y;}
};
```

# Example class Template

A single class template can capture the logic of both classes: it is written with a template prefix that specifies the data type parameters:

```
template <class T>
class Joiner
{
public:
   T combine(T x, T y)
      {return x + y;}
};
```

# Using Class Templates

To create an object of a class defined by a template, specify the actual parameters for the formal data types

```
Joiner<double> jd;
Joiner<string> sd;
cout << jd.combine(3.0, 5.0);
cout << sd.combine("Hi ", "Ho");
```

Prints **`8.0`** and **`Hi Ho`**

# 16.4 Class Templates and Inheritance

- Templates can be combined with inheritance
- You can derive
  - Non template classes from a template class: instantiate the base class template and then inherit from it
  - Template class from a template class
- Other combinations are possible

# 16.5 Introduction to the Standard Template Library

- Standard Template Library (STL): a library containing templates for frequently used data structures and algorithms
- Programs can be developed faster and are more portable if they use templates from the STL

# Standard Template Library

Two important types of data structures in the STL:

- containers: classes that store data and impose some organization on it
- iterators: like pointers; provides mechanisms for accessing elements in a container

# Containers

Two types of container classes in STL:

– sequential containers: organize and access data sequentially, as in an array. These include **`vector`**, **`dequeue`**, and **`list`** containers.

– associative containers: use keys to allow data elements to be quickly accessed. These include **`set`**, **`multiset`**, **`map`**, and **`multimap`** containers.

# Creating Container Objects

- To create a list of **`int`**, write

```
list<int> mylist;
```

- To create a vector of **`string`** objects, write

```
vector<string> myvector;
```

- Requires the **`vector`** header file

# Iterators

- Generalization of pointers, used to access information in containers
- Many types:
  - forward (uses **`++`**)
  - bidirectional (uses **`++`** and **`--`** )
  - random-access
  - input (can be used with **`cin`** and **`istream`** objects)
  - output (can be used with **`cout`** and **`ostream`** objects)

# Containers and Iterators

- Each container class defines an iterator type, used to access its contents
- The type of an iterator is determined by the type of the container:

```
list<int>::iterator x;
list<string>::iterator y;
```

**`x`** is an iterator for a container of type **`list<int>`**

# Containers and Iterators

Each container class defines functions that return iterators:

**`begin()`:** returns iterator to item at start

**`end()`:** returns iterator denoting end of container

# Containers and Iterators

- Iterators support pointer-like operations. If **`iter`** is an iterator, then
  - **`*iter`** is the item it points to: this dereferences the iterator
  - **`iter++`** advances to the next item in the container
  - **`iter--`** backs up in the container
- The **`end()`** iterator points to past the end: it should never be dereferenced

# Traversing a Container

Given a vector:

```
vector<int> v;
for  (int k=1; k<= 5; k++)
  v.push_back(k*k);
```

Traverse it using iterators:

```
vector<int>::iterator iter = v.begin();
while (iter != v.end())
  { cout << *iter << " "; iter++}
```

Prints **1 4 9 16 25**

# Some `vector` Class Member Functions

| **Function** | **Description** |
|---|---|
| front(), back() | Returns a reference to the first, last element in a vector |
| size() | Returns the number of elements in a vector |
| capacity() | Returns the number of elements that a vector can hold |
| clear() | Removes all elements from a vector |
| push_back(value) | Adds element containing value as the last element in the vector |
| pop_back() | Removes the last element from the vector |
| insert(iter, value) | Inserts new element containing value just before element pointed at by iter |

# Algorithms

- STL contains algorithms that are implemented as function templates to perform operations on containers.
- Requires **`algorithm`** header file
- Collection of algorithms includes

| | |
|---|---|
| **`binary_search`** | **`count`** |
| **`for_each`** | **`find`** |
| **`max_element`** | **`min_element`** |
| **`random_shuffle`** | **`sort`** |

and others

# Using STL algorithms

- Many STL algorithms manipulate portions of STL containers specified by a begin and end iterator
- **`max_element(iter1, iter2)`** finds max element in the portion of a container delimited by **`iter1`**, **`iter2`**
- **`min_element(iter1, iter2)`** is similar to above

# More STL algorithms

- **`random_shuffle(iter1, iter2)`** randomly reorders the portion of the container in the given range
- **`sort(iter1, iter2)`** sorts the portion of the container specified by the given range

# `random-shuffle` Example

The following example stores the squares 1, 4, 9, 16, 25 in a vector, shuffles the vector, and then prints it out

# **`random_shuffle`** example

```
int main()
{
   vector<int> vec;
   for (int k = 1; k <= 5; k++)
     vec.push_back(k*k);
   random_shuffle(vec.begin(),vec.end());
   vector<int>::iterator p = vec.begin();
   while (p != vec.end())
   { cout << *p << "  "; p++;
   }
   return 0;
}
```

# Chapter 17: Linked Lists

# Topics

17.1 Introduction to the Linked List ADT

17.2 Linked List Operations

17.3 A Linked List Template

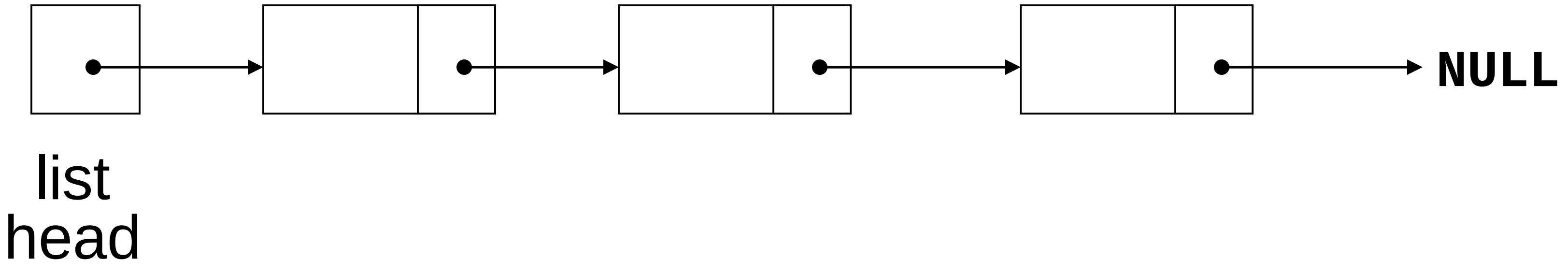
17.4 Recursive Linked List Operations

17.5 Variations of the Linked List

17.6 The STL **`list`** Container

# 17.1 Introduction to the Linked List ADT

- Linked list: a sequence of data structures (nodes) with each node containing a pointer to its successor
- The last node in the list has its successor pointer set to NULL

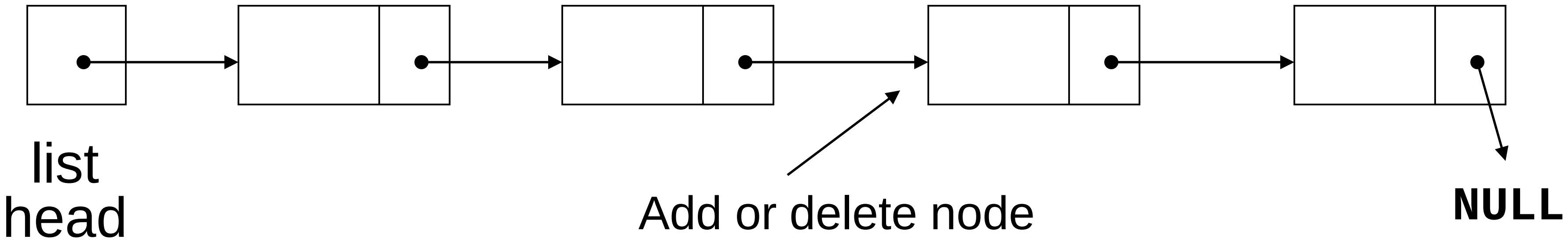


# Linked List Terminology

- The node at the beginning is called the head of the list
- The entire list is identified by the pointer to the head node.  This pointer is called the list head.

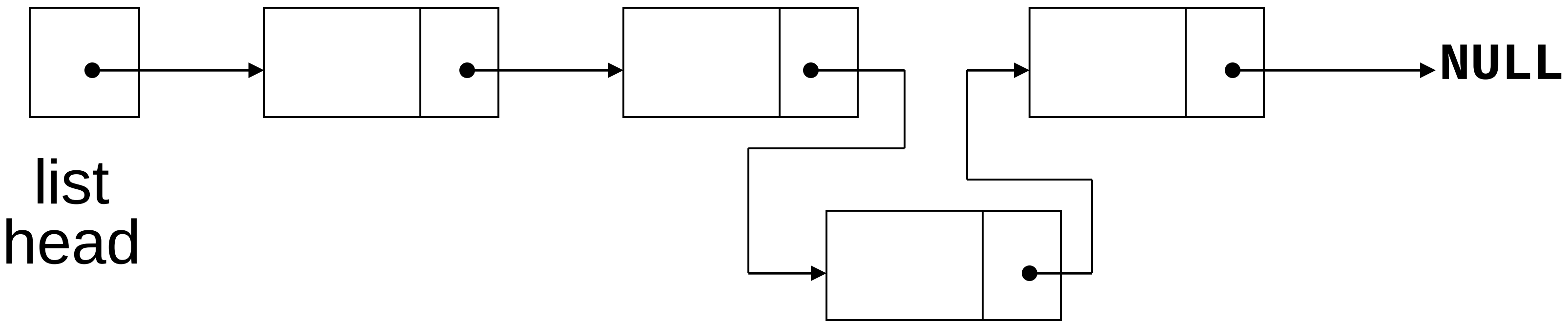
# Linked Lists

- Nodes can be added or removed from the linked list during execution
- Addition or removal of nodes can take place at beginning, end, or middle of the list



# Linked Lists vs. Arrays and Vectors

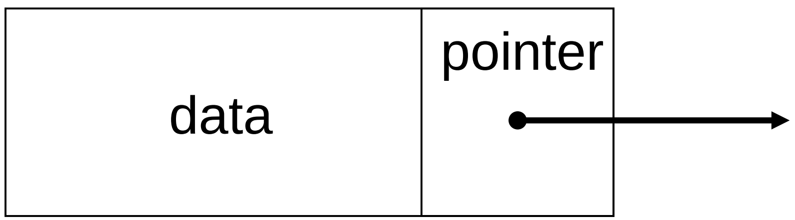
- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Unlike vectors, insertion or removal of a node in the middle of the list is very efficient



# Node Organization

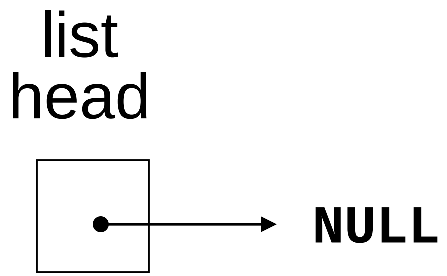
A node contains:

- data: one or more data fields – may be organized as structure, object, etc.
- a pointer that can point to another node



# Empty List

- A list with no nodes is called the empty list
- In this case the list head is set to **`NULL`**

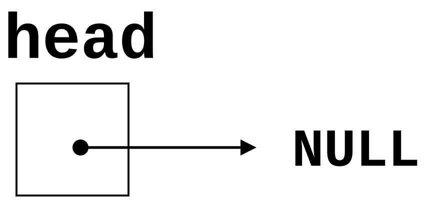


# Creating an Empty List

- Define a pointer for the head of the list:

```
ListNode *head = NULL;
```

- Head pointer initialized to **`NULL`** to indicate an empty list



# C++ Implementation

Implementation of nodes requires a structure containing a pointer to a structure of the same type (a self-referential data structure):

```
struct ListNode
{
    int data;
    ListNode *next;
};
```

# C++ Implementation

Nodes can be equipped with constructors:

```
struct ListNode
{
   int data;
   ListNode *next;
   ListNode(int d, ListNode* p=NULL)
    {data = d; next = p;}
};
```

# Building a List from a File of Numbers

```
ListNode *head = NULL;
int val;
while (inFile >> val)
{
  // add new nodes at the head
  head = new ListNode(val, head);
};
```

# Traversing a Linked List

- List traversals visit each node in a linked list to display contents, validate data, etc.
- Basic process of traversal:

*set a pointer to the head pointer*

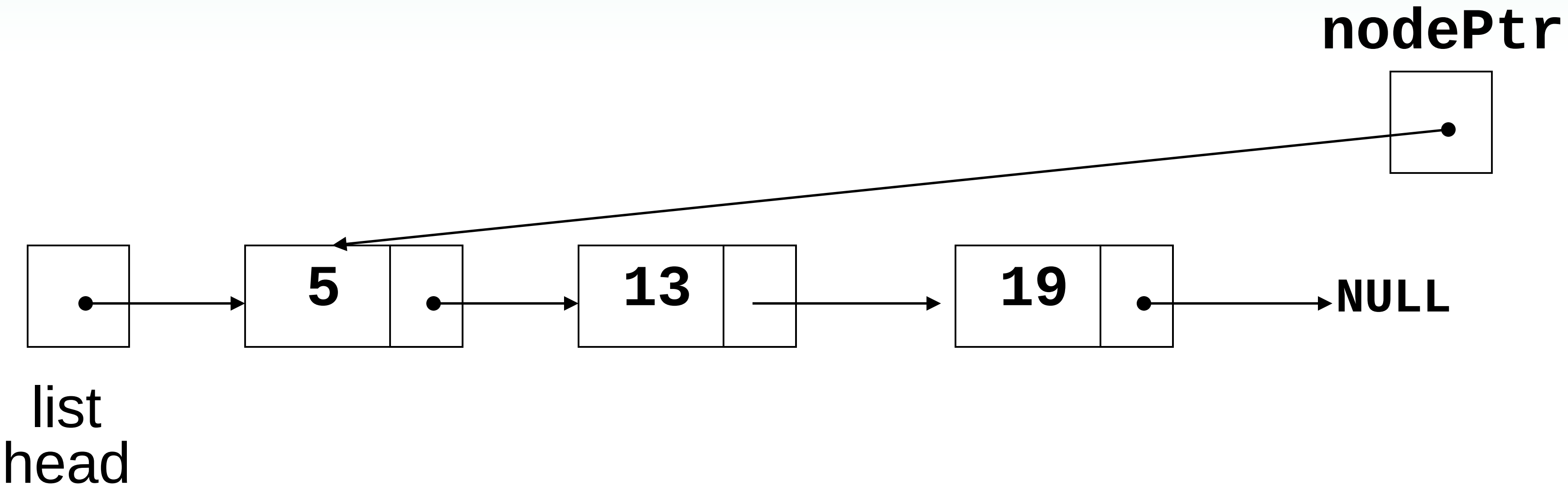
*while pointer is not* **`NULL`**

    *process data*

    *set pointer to the successor of the current node*

*end while*

# Traversing a Linked List




**nodePtr** points to the node containing **5**, then the node containing **13**, then the node containing **19**, then points to **NULL**, and the list traversal stops

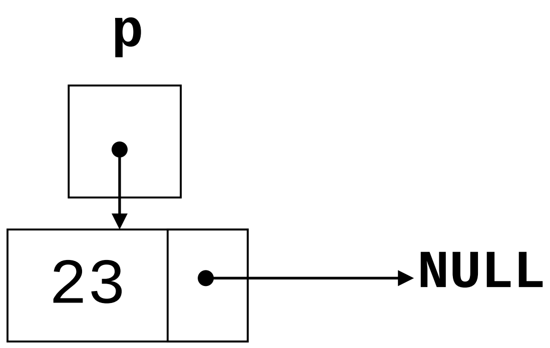
# 17.2 Linked List Operations

Basic operations:

- add a node to the end of the list
- insert a node within the list
- traverse the linked list
- Delete/remove a node from the list
- delete/destroy the list

# Creating a Node

```
ListNode *p;
int num = 23;
p = new ListNode(num);
```



# Appending an Item

To add an item to the end of the list:

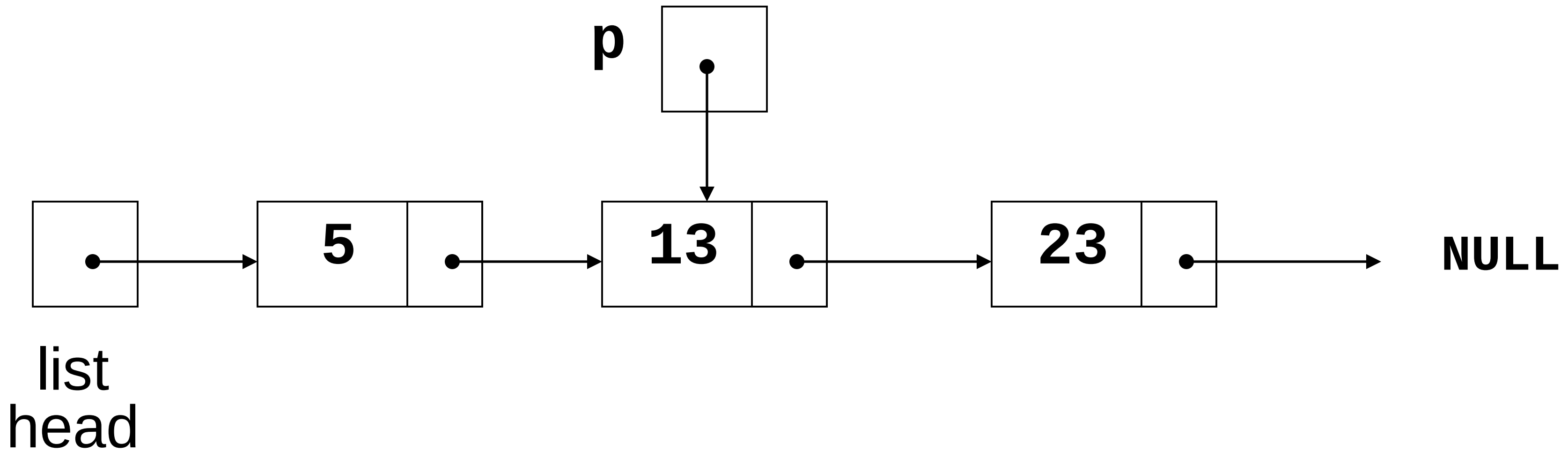
- If the list is empty, set **head** to a new node containing the item

```
head = new ListNode(num);
```

- If the list is not empty, move a pointer **p** to the last node, then add a new node containing the item

```
p->next = new ListNode(num);
```

# Appending an Item




List originally has nodes with 5 and 13.

**p** locates the last node, then a node with a new item, 23, is added

# Destroying a Linked List

- Must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - Free the node's memory
- Finally, set the list head to **`NULL`**

# Inserting a Node

- Used to insert an item into a sorted list, keeping the list sorted.
- Two possibilities:
  - Insertion is at the head of the list (because item at head is already greater than item being inserted, or because list is empty
  - Insertion is after an existing node in a non-empty list

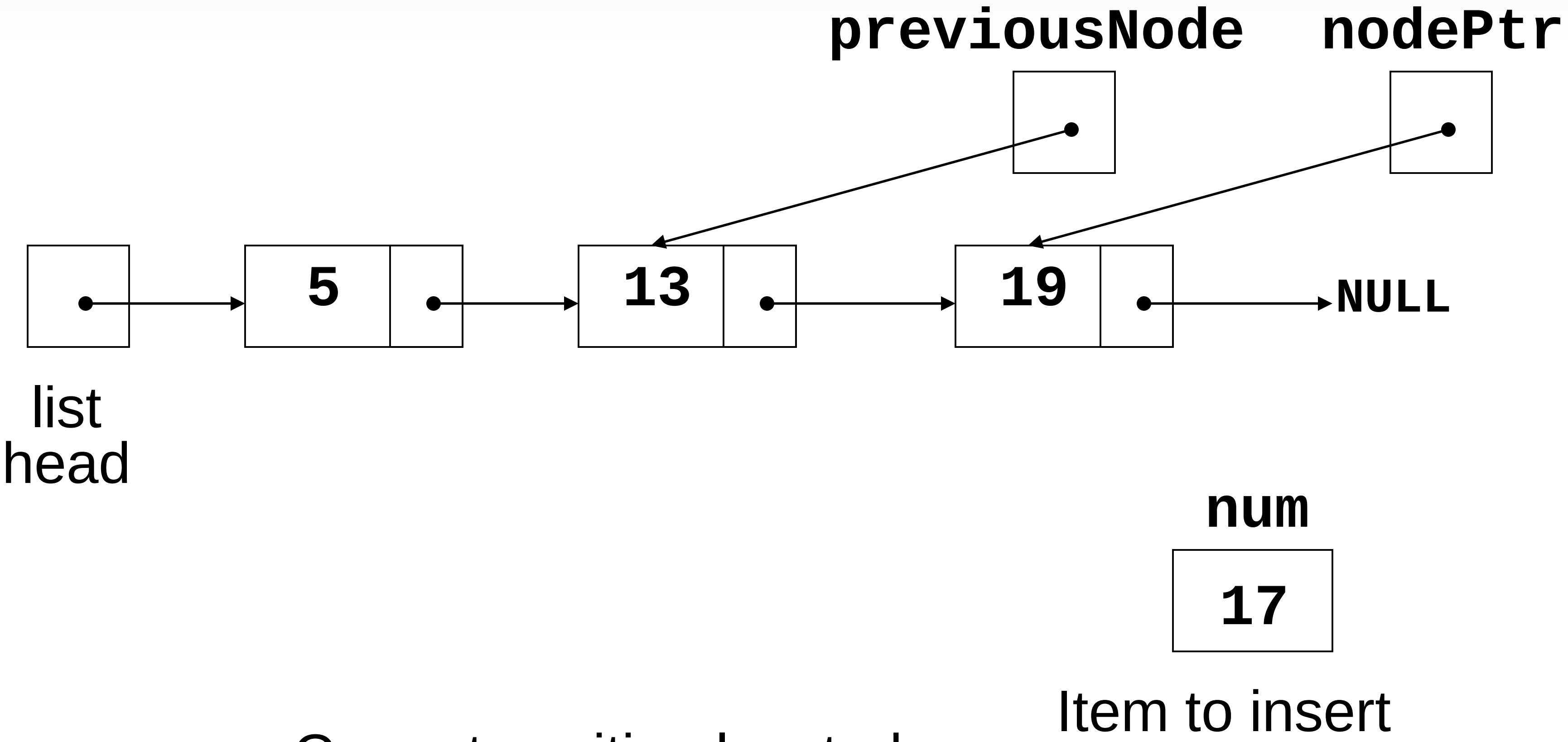
# Inserting a Node at Head of a List

- Test to see if
  - head pointer is **`NULL`**, or
  - node value pointed at by head is greater than value to be inserted
- Must test in this order: unpredictable results if second test is attempted on an empty list
- Create new node, set its next pointer to head, then point head to it

# Inserting a Node in Body of a List

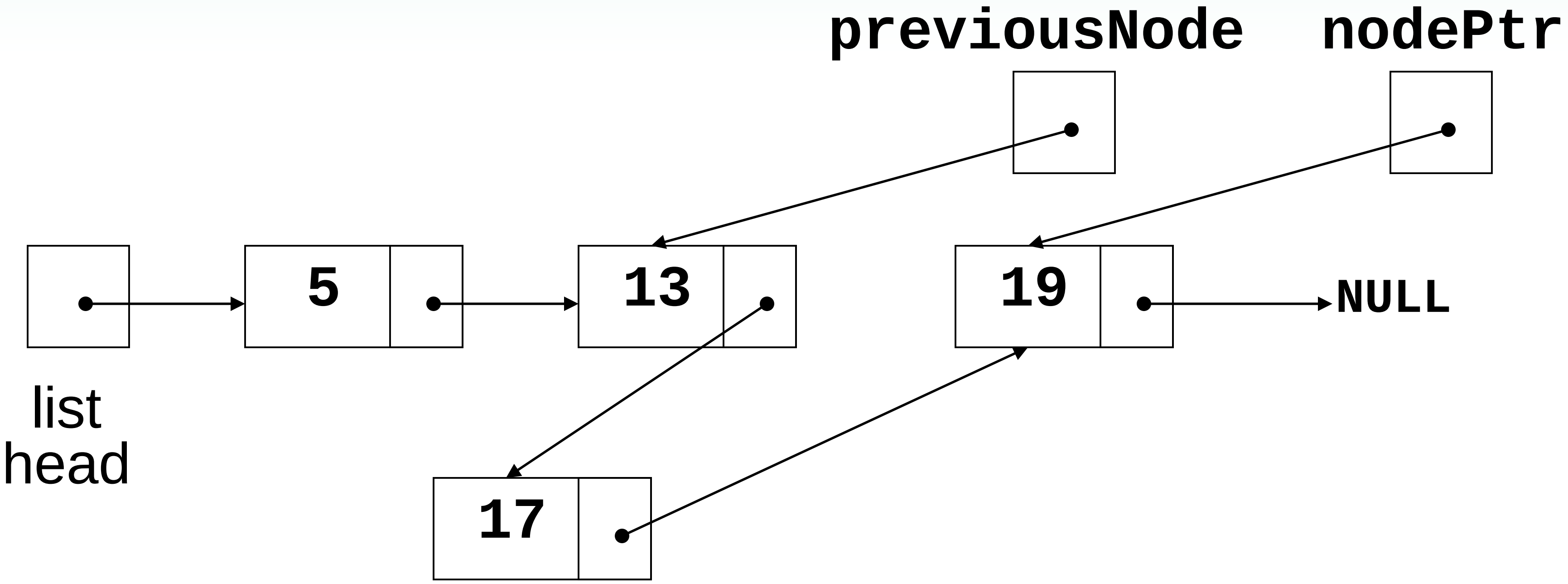
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

# Inserting a Node into a Linked List




Correct position located

# Inserting a Node into a Linked List



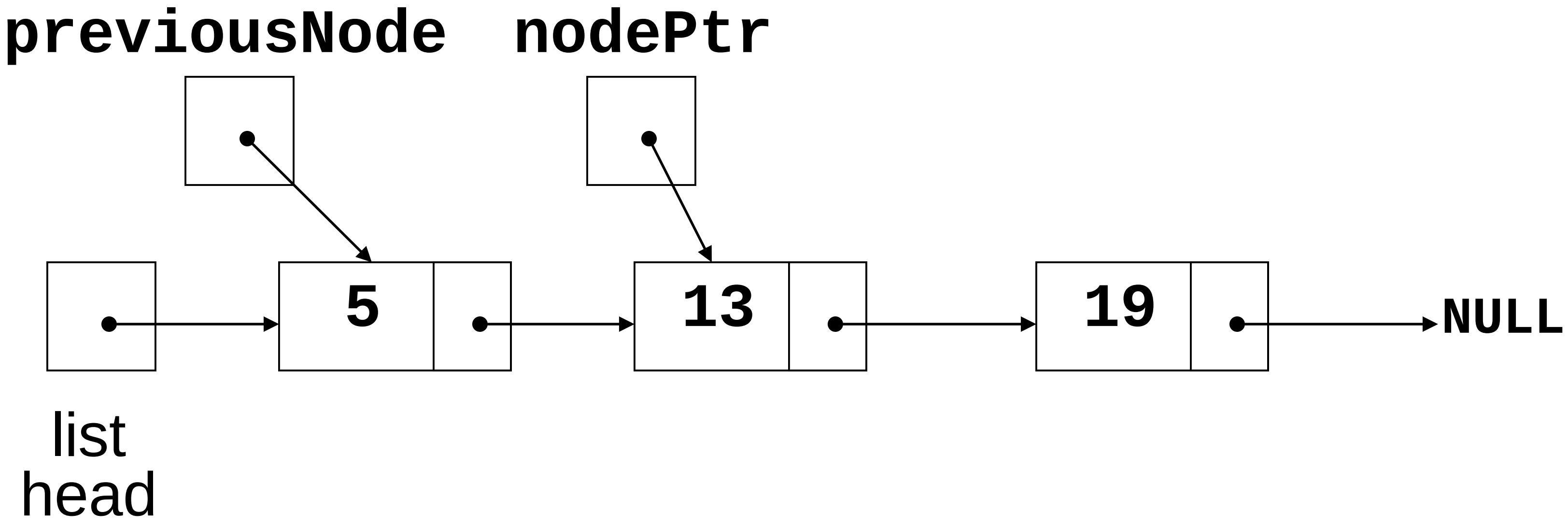

New node created and inserted in order in the linked list

# Removing an Element

- Used to remove a node from a linked list
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

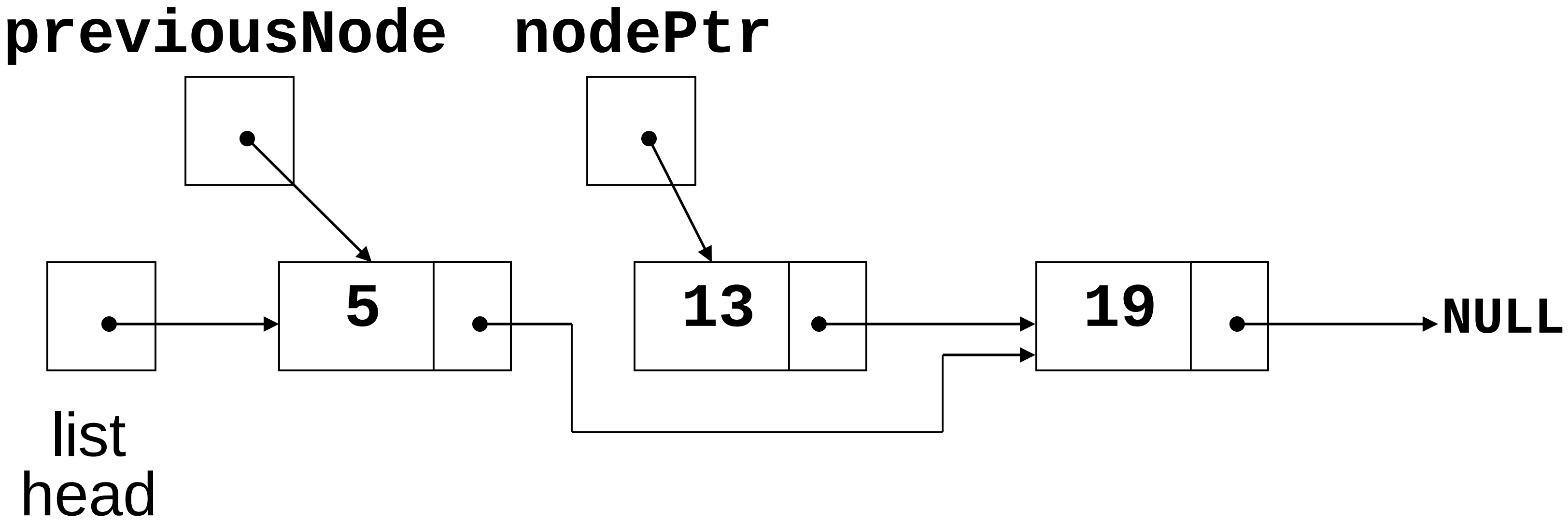
# Deleting a Node

Contents of node to
be deleted: 13



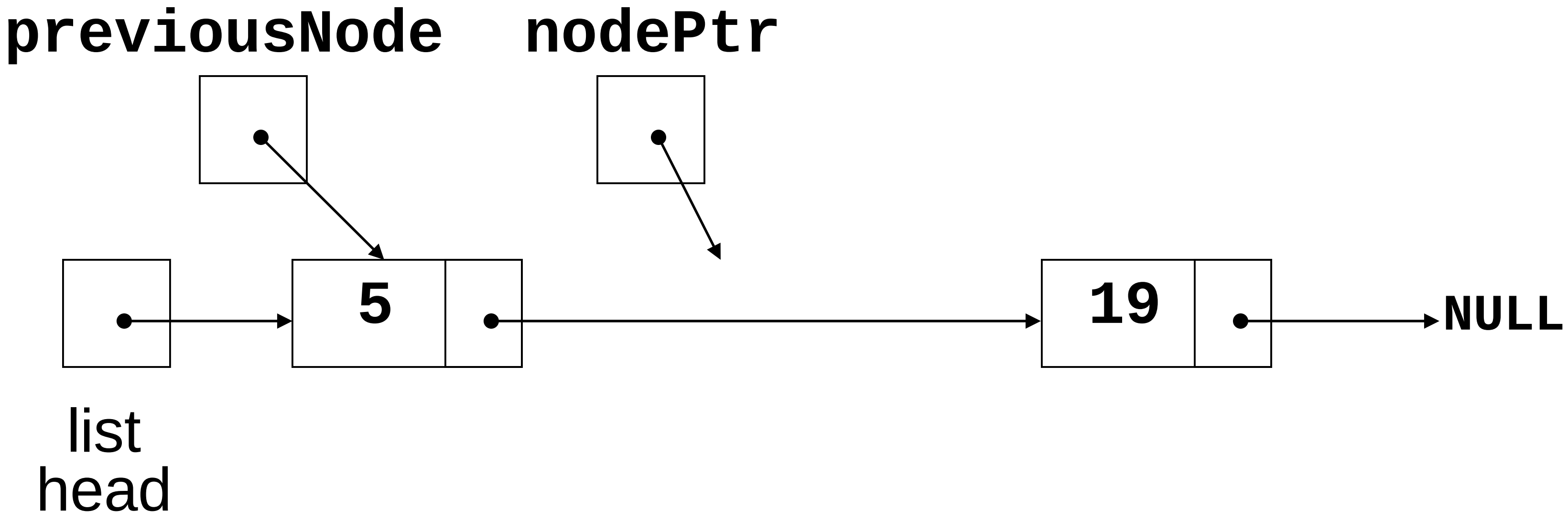

Locating the node containing **13**

# Deleting a Node




Adjusting pointer around the node to be deleted

# Deleting a Node




Linked list after deleting the node containing **13**

# 17.3 A Linked List Template

- A linked list template can be written by replacing the type of the data in the node with a type parameter, say T.
- If defining the linked list as a class template, then all member functions must be function templates
- Implementation assumes use with data types that support comparison: **==** and **<=**

# 17.4 Recursive Linked List Operations

- A non-empty linked list consists of a head node followed by the rest of the nodes
- The rest of the nodes form a linked list that is called the tail of the original list

# Recursive Linked List Operations

Many linked list operations can be broken down into the smaller problems of processing the head of the list and then recursively operating on the tail of the list

# Recursive Linked List Operations

To find the length (number of elements) of a list

- If the list is empty, the length is 0 (base case)
- If the list is not empty, find the length of the tail and then add 1 to obtain the length of the original list

# Recursive Linked List Operations

To find the length of a list:

```
int length(ListNode *myList)
{
    if (myList == NULL) return 0;
    else
    return 1 + length(myList->next);
}
```

# Recursive Linked List Operations

Using recursion to display a list:

```
void displayList(ListNode *myList)
{
  if (myList != NULL)
  {
    cout << myList->data << " ";
    displayList(myList->next);
  }
}
```

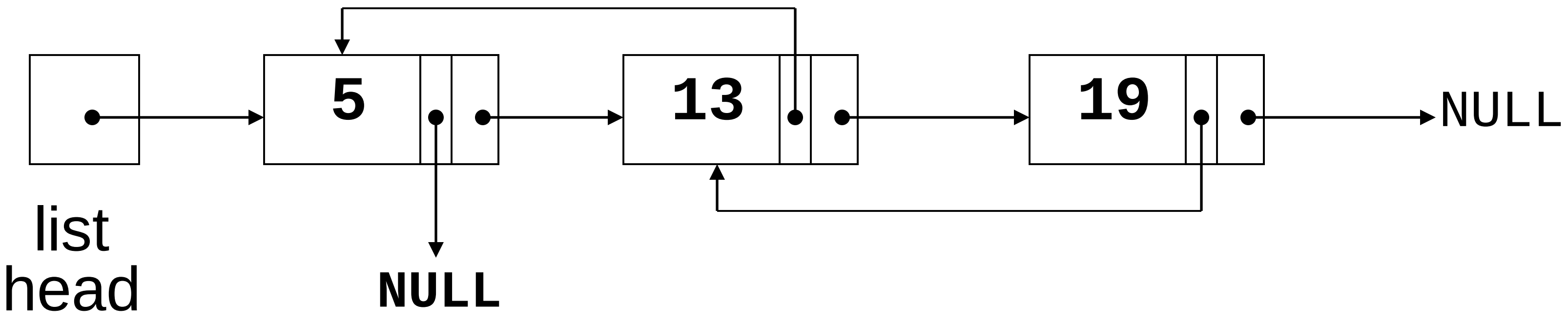
# Other Recursive Linked List Operations

- Insert and remove operations can be written to use recursion
- General design considerations:
  - Base case is often when the list is empty
  - Recursive case often involves the use of the tail of the list (*i.e.,* the list without the head). Since the tail has one fewer entry than the list that was passed in to this call, the recursion eventually stops.

# 17.5 Variations of the Linked List

Other linked list organizations:

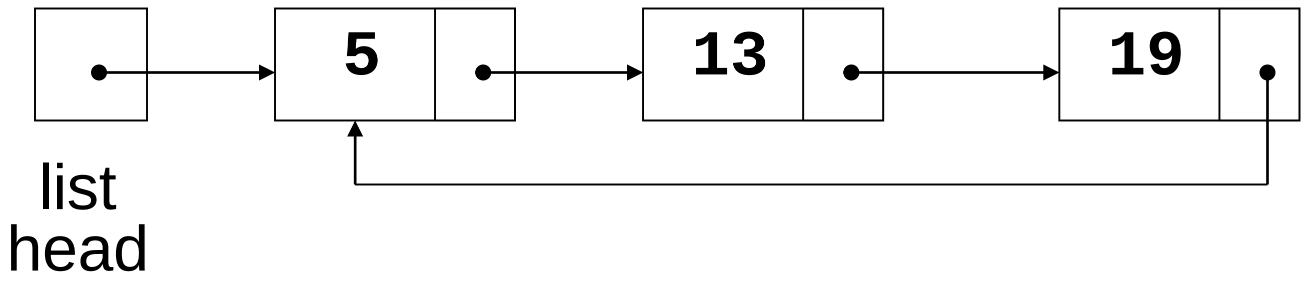
– doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list



# Variations of the Linked List

Other linked list organizations:

– circular linked list: the last node in the list points back to the first node in the list, not to **`NULL`**



# 17.6 The STL `list` Container

- Template for a doubly linked list
- Member functions for
  - locating beginning, end of list: **`front`**, **`back`**, **`end`**
  - adding elements to the list: **`insert`**, **`merge`**, **`push_back`**, **`push_front`**
  - removing elements from the list: **`erase`**, **`pop_back`**, **`pop_front`**, **`unique`**

# Chapter 18: Stacks and Queues

# Topics

18.1 Introduction to the Stack ADT

18.2 Dynamic Stacks

18.3 The STL **`stack`** Container

18.4 Introduction to the Queue ADT

18.5 Dynamic Queues

18.6 The STL **`deque`** and **`queue`** Containers

18.7 Eliminating Recursion

# 18.1 Introduction to the Stack ADT

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria serving area
  - return addresses for function calls

# Stack Basics

- Stack is usually implemented as a list, with additions and removals taking place at one end of the list
- The active end of the list implementing the stack is the top of the stack
- Stack types:
  - Static – fixed size, often implemented using an array
  - Dynamic – size varies as needed, often implemented using a linked list

# Stack Operations and Functions

Operations:

- push: add a value at the top of the stack
- pop: remove a value from the top of the stack

Functions:

- isEmpty: true if the stack currently contains no elements
- isFull: true if the stack is full; only useful for static stacks

# Static Stack Implementation

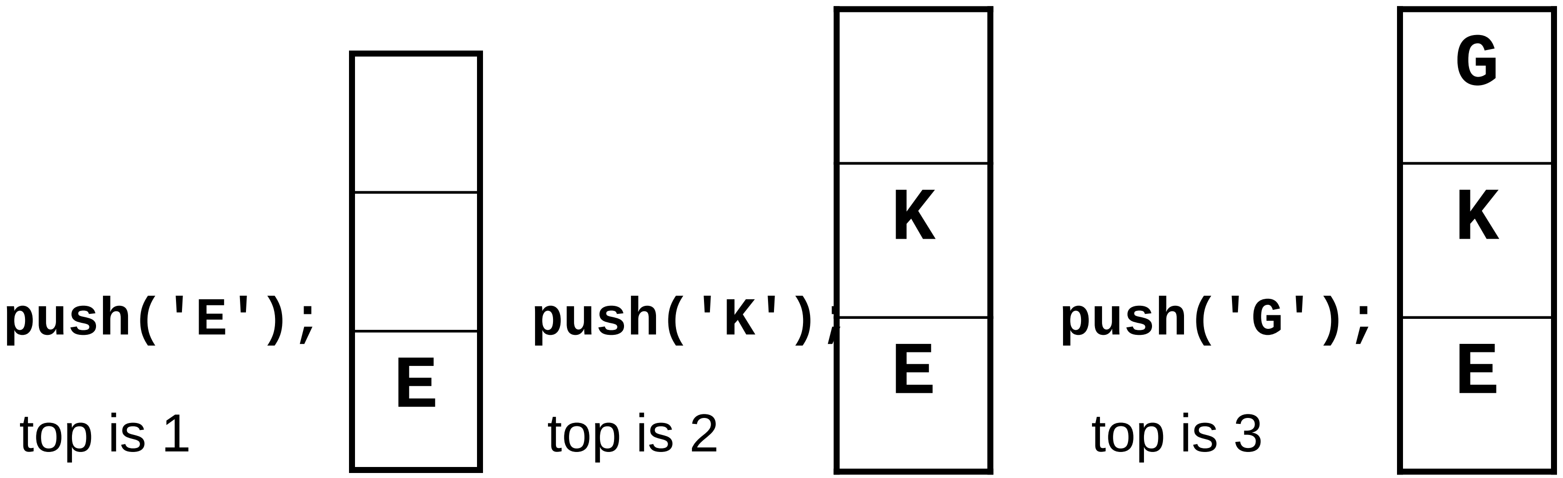
- Uses an array of a fixed size
- Bottom of stack is at index 0.  A variable called top tracks the current top of the stack

```
const int STACK_SIZE = 3;
char s[STACK_SIZE];
int top = 0;
```

top is where the next item will be added

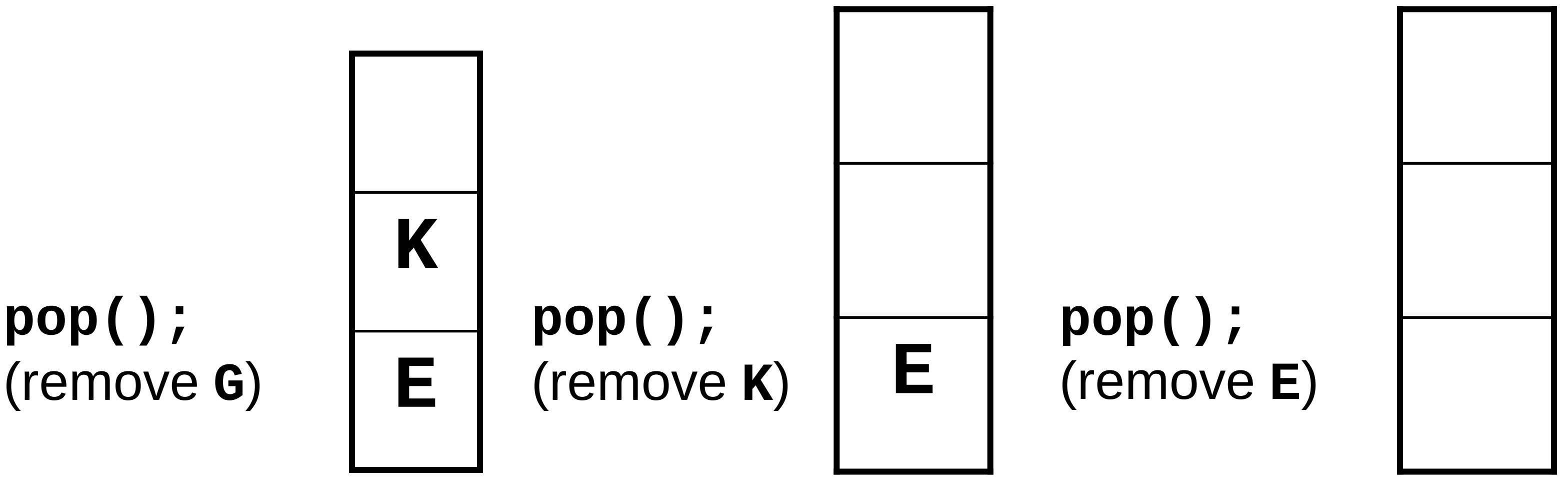
# Array Implementation Example

This stack has max capacity 3, initially top = 0 and stack is empty.



# Stack Operations Example

After three pops, **`top`** is **`0`** and the stack is empty



# Array Implementation

```
char s[STACK_SIZE];
int top=0;
```

To check if stack is empty:

```
bool isEmpty()
{
  if (top == 0)
      return true;
  else return false;
}
```

# Array Implementation

```
char s[STACK_SIZE];
int top=0;
```

To check if stack is full:

```
bool isFull()
{
  if (top == STACK_SIZE)
        return true;
  else return false;
}
```

# Array Implementation

To add an item to the stack

```
void push(char x)
{
  if (isFull())
     {error(); exit(1);}
  // or could throw an exception
  s[top] = x;
  top++;
}
```

# Array Implementation

To remove an item from the stack

```
void pop(char &x)
{
  if (isEmpty())
     {error(); exit(1);}
  // or could throw an exception
  top--;
  x = s[top];
}
```

# Class Implementation

```
class STACK
 {
private:
    char *s;
    int capacity, top;
 public:
    void push(char x);
    void pop(char &x);
    bool isFull(); bool isEmpty();
    STACK(int stackSize);
    ~STACK()
 };
```

# Exceptions from Stack Operations

- Exception classes can be added to the stack object definition to handle cases where an attempt is made to push onto a full stack (overflow) or to pop from an empty stack (underflow)
- Programs that use **`push`** and **`pop`** operations should do so from within a **`try`** block.
- **`catch`** block(s) should follow the **`try`** block, interpret what occurred, and inform the user.

# 18.2 Dynamic Stacks

- Implemented as a linked list
- Can grow and shrink as necessary
- Can't ever be full as long as memory is available

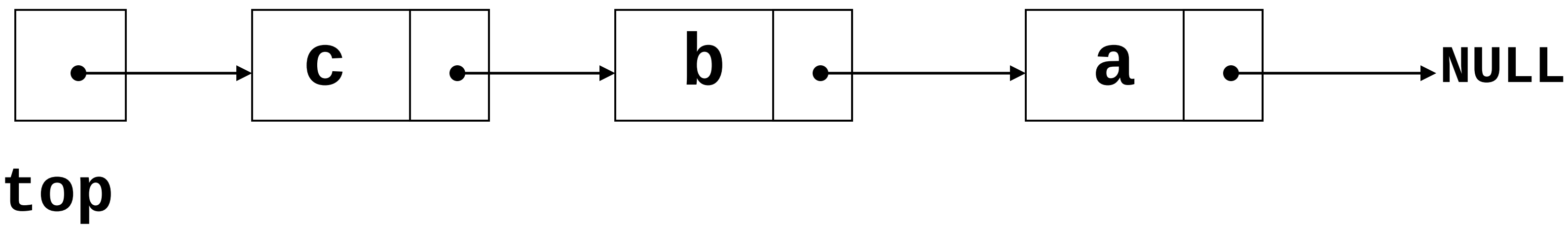
# Dynamic Linked List Implementation

- Define a class for a dynamic linked list
- Within the class, define a private member class for dynamic nodes in the list
- Define a pointer to the beginning of the linked list, which will serve as the top of the stack

# Linked List Implementation

A linked stack after three push operations:

```
push('a'); push('b'); push('c');
```



# Operations on a Linked Stack

Check if stack is empty:

```
bool isEmpty()
{
   if (top == NULL)
      return true;
   else
      return false;
}
```

# Operations on a Linked Stack

Add a new item to the stack

```
void push(char x)
{
  top = new LNode(x, top);
}
```

# Operations on a Linked Stack

Remove an item from the stack

```
void pop(char &x)
{
  if (isEmpty())
  { error(); exit(1);}
  x = top->value;
  LNode *oldTop = top;
  top = top->next;
  delete oldTop;
}
```

•

# 18.3 The STL `stack` Container

- Stack template can be implemented as a **`vector`**, **`list`**, or a **`deque`**
- Implements **`push`**, **`pop`**, and **`empty`** member functions
- Implements other member functions:
  - **`size`**: number of elements on the stack
  - **`top`**: reference to element on top of the stack (must be used with **`pop`** to remove and retrieve top element)

# Defining an STL-based Stack

- Defining a stack of **`char`**, named **`cstack`**, implemented using a **`vector`**:

```
stack< char, vector<char> > cstack;
```

- Implemented using a list:

```
stack< char, list<char> > cstack;
```

- Implemented using a **`deque`** (default):

```
stack< char > cstack;
```

- Spaces are required between consecutive **`> >`** symbols to distinguish from stream extraction

# 18.4 Introduction to the Queue ADT

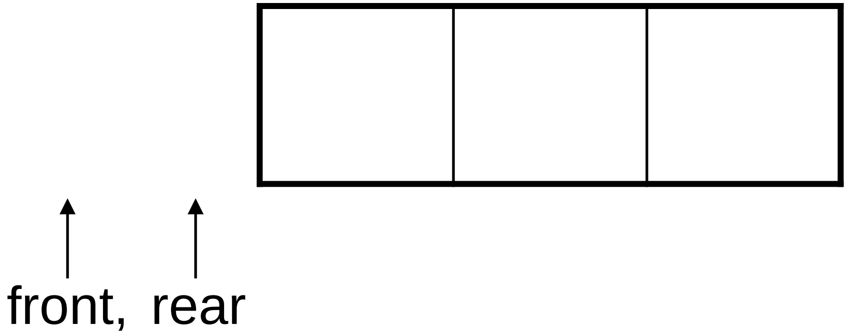
- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print requests sent by users to a network printer
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

# Queue Locations and Operations

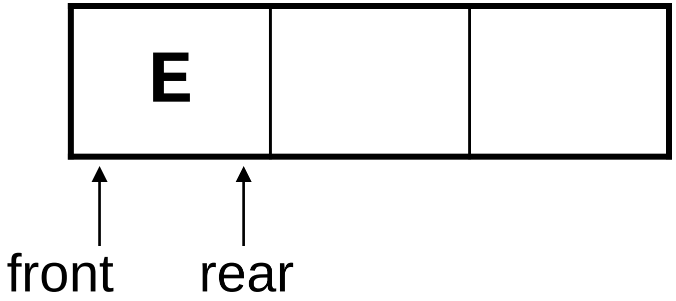
- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

# Array Implementation of Queue

An empty queue that can hold **`char`** values:

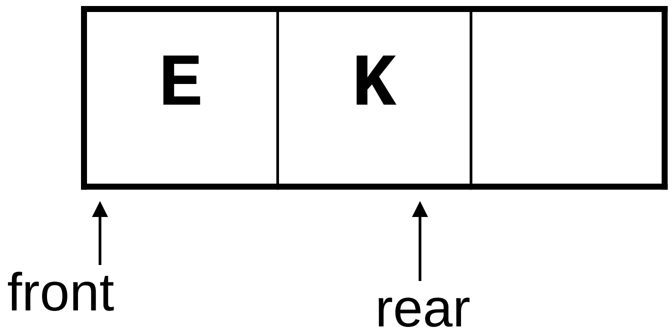



**`enqueue('E');`**

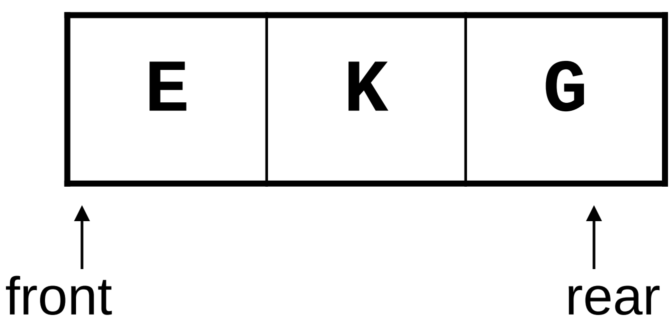


# Queue Operations - Example

**`enqueue('K');`**

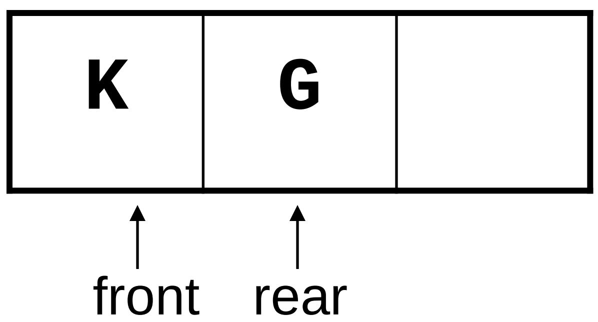



**`enqueue('G');`**

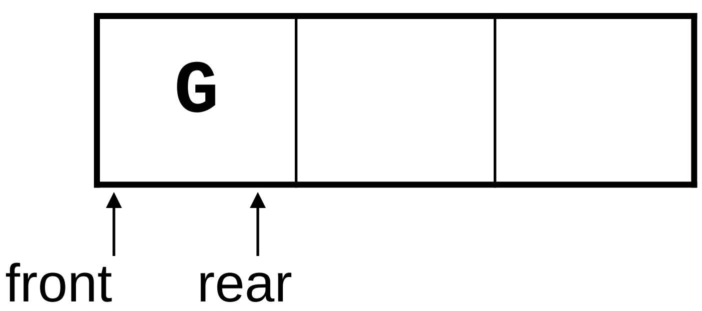


# Queue Operations - Example

```
dequeue(); // remove E
```




```
dequeue(); // remove K
```



# Array Implementation Issues

- In the preceding example, Front never moves.
- Whenever **dequeue** is called, all remaining queue entries move up one position.  This takes time.
- Alternate approach:
  - Circular array:  **`front`** and **`rear`** both move when items are added and removed.  Both can ‘wrap around’ from the end of the array to the front if warranted.
- Other conventions are possible

# Array Implementation Issues

- Variables needed
  - `const int QSIZE = 100;`
  - `char q[QSIZE];`
  - `int front = -1;`
  - `int rear = -1;`
  - `int number = 0; //how many in queue`
- Could make these members of a queue class, and queue operations would be member functions

# **`isEmpty`** Member Function

Check if queue is empty

```
bool isEmpty()
{
   if (number > 0)
      return false;
   else
      return true;
}
```

# `isFull` Member Function

Check if queue is full

```
bool isFull()
{
  if (number < QSIZE)
    return false;
  else
    return true;
}
```

# **enqueue** and **dequeue**

- To enqueue, we need to add an item **`x`** to the rear of the queue
- Queue convention says **`q[rear]`** is already occupied. Execute

```
if(!isFull)
{ rear = (rear + 1) % QSIZE;
// mod operator for wrap-around
   q[rear] = x;
  number ++;
}
```

# **enqueue** and **dequeue**

- To dequeue, we need to remove an item `x` from the front of the queue
- Queue convention says `q[front]` has already been removed. Execute

```
if(!isEmpty)
{  front = (front + 1) % QSIZE;
   x = q[front];
   number--;
}
```

# **enqueue** and **dequeue**

- **`enqueue`** moves **`rear`** to the right as it fills positions in the array
- **`dequeue`** moves **`front`** to the right as it empties positions in the array
- When **`enqueue`** gets to the end, it wraps around to the beginning to use those positions that have been emptied
- When **`dequeue`** gets to the end, it wraps around to the beginning use those positions that have been filled

# **enqueue** and **dequeue**

- Enqueue wraps around by executing

```
rear = (rear + 1) % QSIZE;
```

- Dequeue wraps around by executing

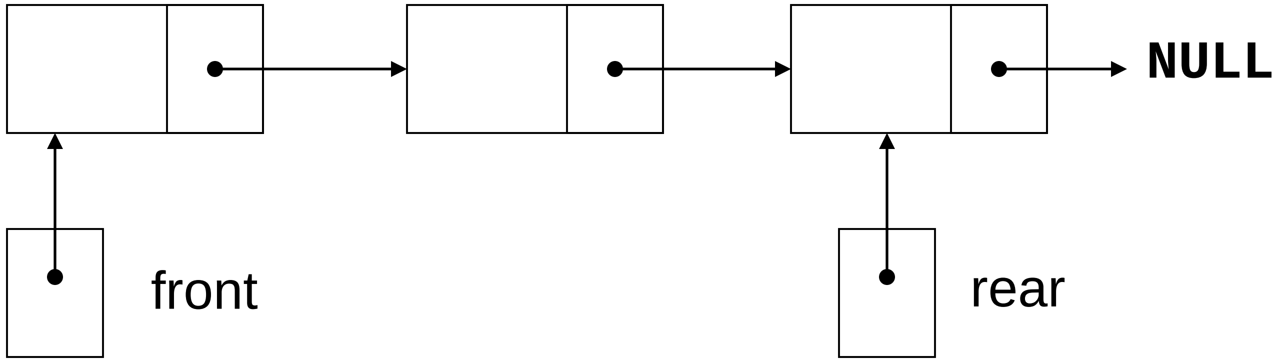
```
front = (front + 1) % QSIZE;
```

# Exception Handling in Static Queues

- As presented, the static queue class will encounter an error if an attempt is made to enqueue an element to a full queue, or to dequeue an element from an empty queue
- A better design is to throw an underflow or an overflow exception and allow the programmer to determine how to proceed
- Remember to throw exceptions from within a **`try`** block, and to follow the **`try`** block with a **`catch`** block

# 18.5 Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- This allows dynamic sizing and avoids the issue of wrapping indices



# Dynamic Queue Implementation Data Structures

- Define a class for the dynamic queue
- Within the dynamic queue, define a private member class for a dynamic node in the queue
- Define pointers to the front and rear of the queue

# isEmpty Member Function

To check if queue is empty:

```
bool isEmpty()
{
  if (front == NULL)
      return true;
  else
      return false;
}
```

# enqueue Member Function Details

To add item at rear of queue

```
if (isEmpty())
    {
       front = new QNode(x);
       rear = front;
    }
 else
    {
       rear->next = new QNode(x);
       rear = rear->next;
    }
```

# dequeue Member Function

To remove item from front of queue

```
if (isEmpty())
{
    error(); exit(1);
}
x = front->value;
QNode *oldfront = front;
front = front->next;
delete oldfront;
```

# 18.6 The STL **deque** and **queue** Containers

- **deque**: a double-ended queue (DEC). Has member functions to enqueue (**push_back**) and dequeue (**pop_front**)

- **queue**: container ADT that can be used to provide a queue based on a **vector**, **list**, or **deque**. Has member functions to enqueue (**push**) and dequeue (**pop**)

# Defining a Queue

- Defining a queue of **`char`**, named cQueue, based on a **deque**:

```
deque<char> cQueue;
```

- Defining a **queue** with the default base container

```
queue<char> cQueue;
```

- Defining a queue based on a **`list`**:

```
queue<char, list<char> > cQueue;
```

- Spaces are required between consecutive **`> >`** symbols to distinguish from stream extraction

# 18.7 Eliminating Recursion

- Recursive solutions to problems are often elegant but inefficient
- A solution that does not use recursion is more efficient for larger sizes of inputs
- Eliminating the recursion: re-writing a recursive algorithm so that it uses other programming constructs (stacks, loops) rather than recursive calls

# Chapter 19: Binary Trees

# Topics

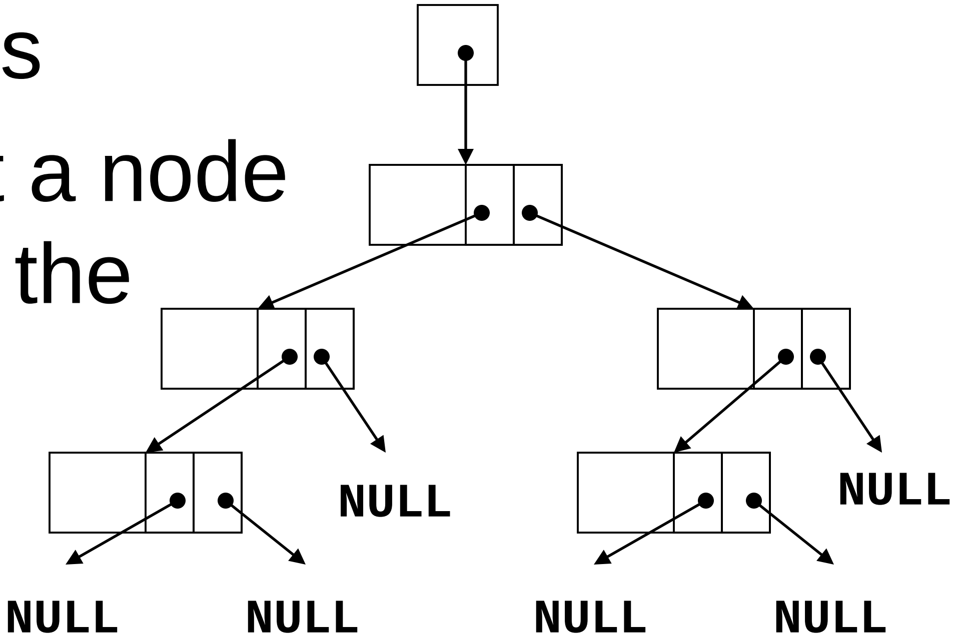
19.1 Definition and Application of Binary Trees

19.2 Binary Search Tree Operations

19.3 Template Considerations for Binary Search Trees

# 19.1 Definition and Application of Binary Trees

- Binary tree: a nonlinear data structure in which each node may point to 0, 1, or two other nodes
- The nodes that a node $N$ points to are the (left or right) children of $N$

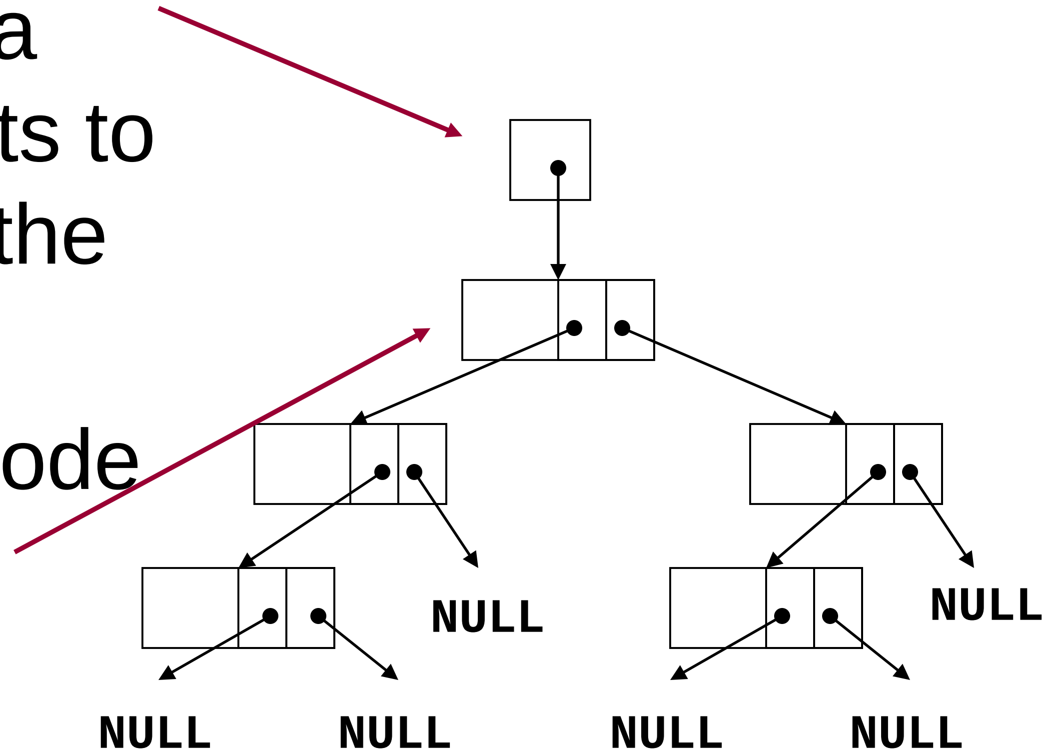


# Terminology

- If a node ***N*** is a child of another node ***P***, then ***P*** is called the parent of ***N***
- A node that has no children is called a leaf node
- In a binary tree there is a unique node with no parent. This is the root of the tree

# Binary Tree Terminology

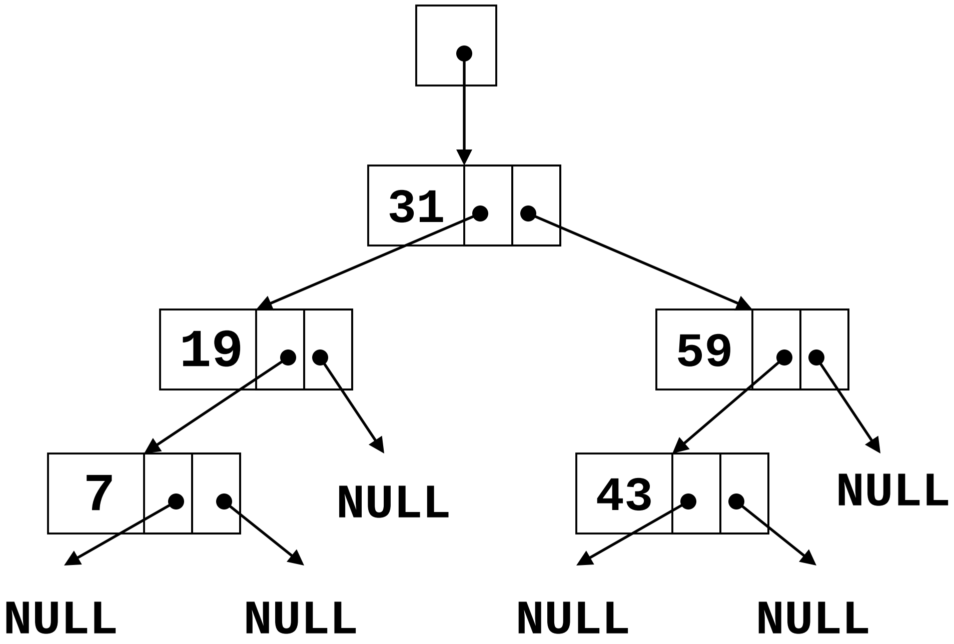
- Root pointer: like a head pointer for a linked list, it points to the root node of the binary tree
- Root node: the node with no parent



# Binary Tree Terminology

Leaf nodes: nodes that have no children

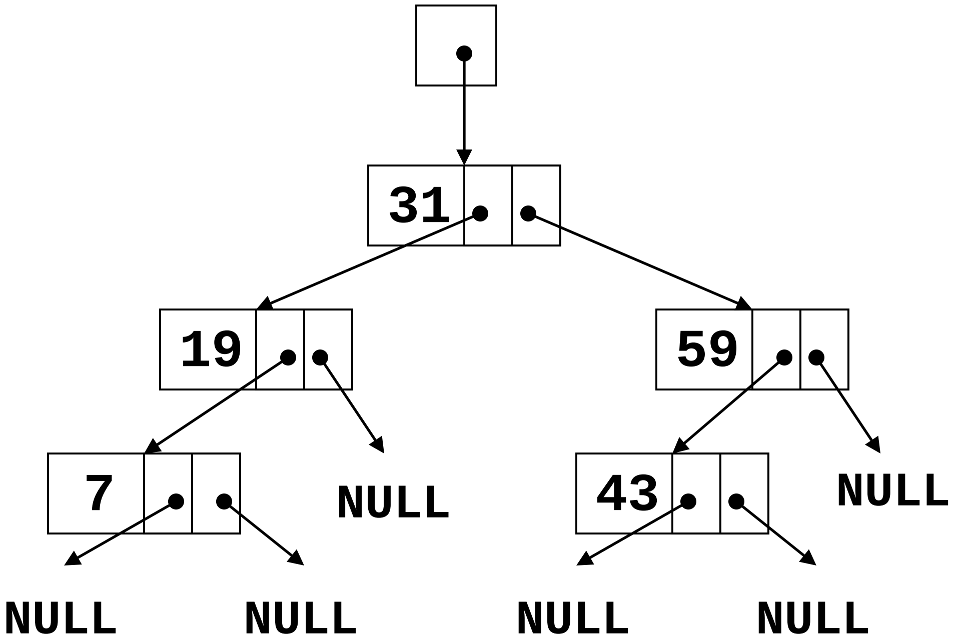
The nodes containing **7** and **43** are leaf nodes



# Binary Tree Terminology

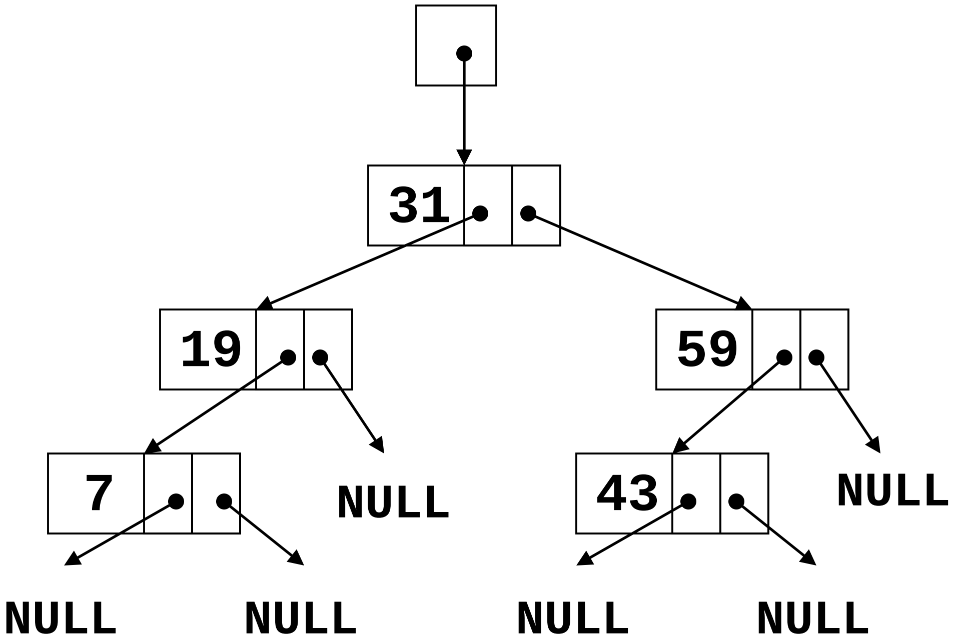
Child nodes, children:

The children of the node containing **31** are the nodes containing **19** and **59**



# Binary Tree Terminology

The parent of the node containing **43** is the node containing **59**



# Binary Tree Terminology

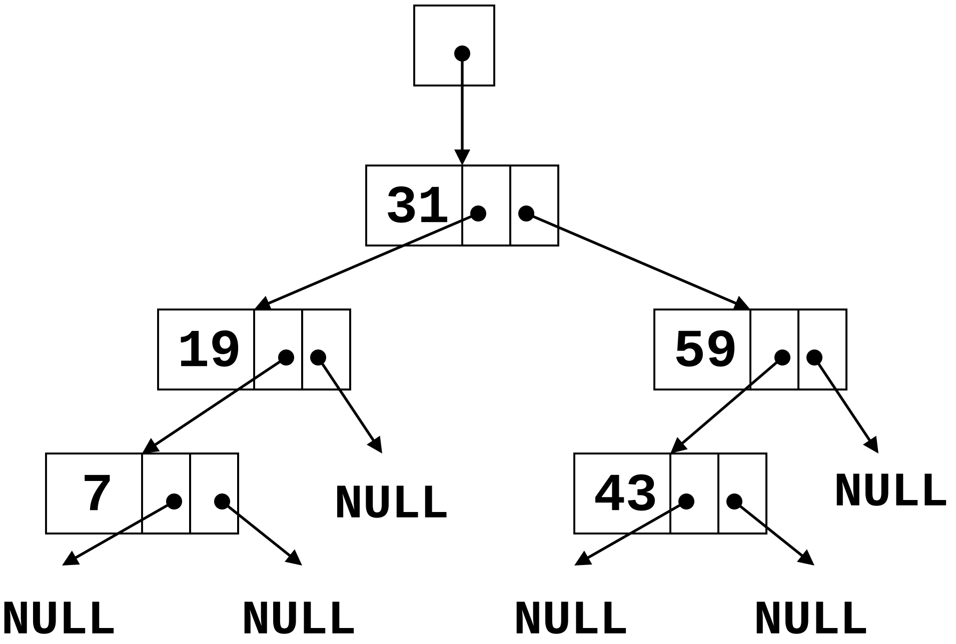
- A subtree of a binary tree is a part of the tree from a node $N$ down to the leaf nodes
- Such a subtree is said to be rooted at $N$, and $N$ is called the root of the subtree

# Subtrees of Binary Trees

- A subtree of a binary tree is itself a binary tree
- A nonempty binary tree consists of a root node, with the rest of its nodes forming two subtrees, called the left and right subtree

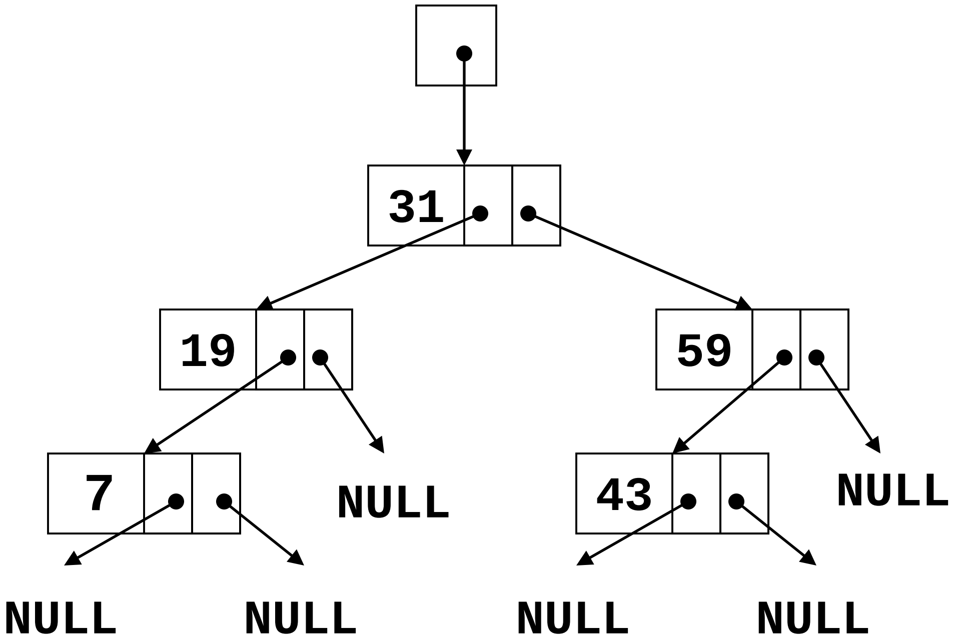
# Binary Tree Terminology

- The node containing **31** is the root
- The nodes containing **19** and **7** form the left subtree
- The nodes containing **59** and **43** form the right subtree



# Uses of Binary Trees

- Binary search tree: a binary tree whose data is organized to simplify searches
- Left subtree at each node contains data values less than the data in the node
- Right subtree at each node contains values greater than the data in the node
-



# 19.2 Binary Search Tree Operations

- **Create** a binary search tree
- **Insert a node** into a binary tree – put node into tree in its correct position to maintain order
- **Find a node** in a binary tree – locate a node with particular data value
- **Delete a node** from a binary tree – remove a node and adjust links to preserve the binary tree and the order

# Binary Search Tree Node

- A node in a binary tree is like a node in a linked list, except that it has two node pointer fields:

```
class TreeNode
{
   int value;
   TreeNode *left;
   TreeNode *right;
};
```

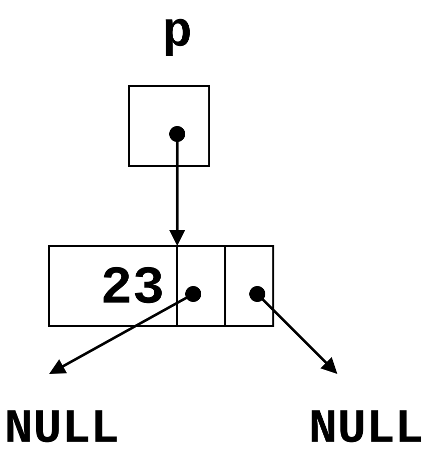
- Define the nodes as class objects.  A constructor can aid in the creation of nodes

# **TreeNode** Constructor

```
TreeNode::TreeNode(int val,
                   TreeNode *l1=NULL,
                   TreeNode *r1=NULL)
{
   value = val;
   left = l1;
   right = r1;
}
```

# Creating a New Node

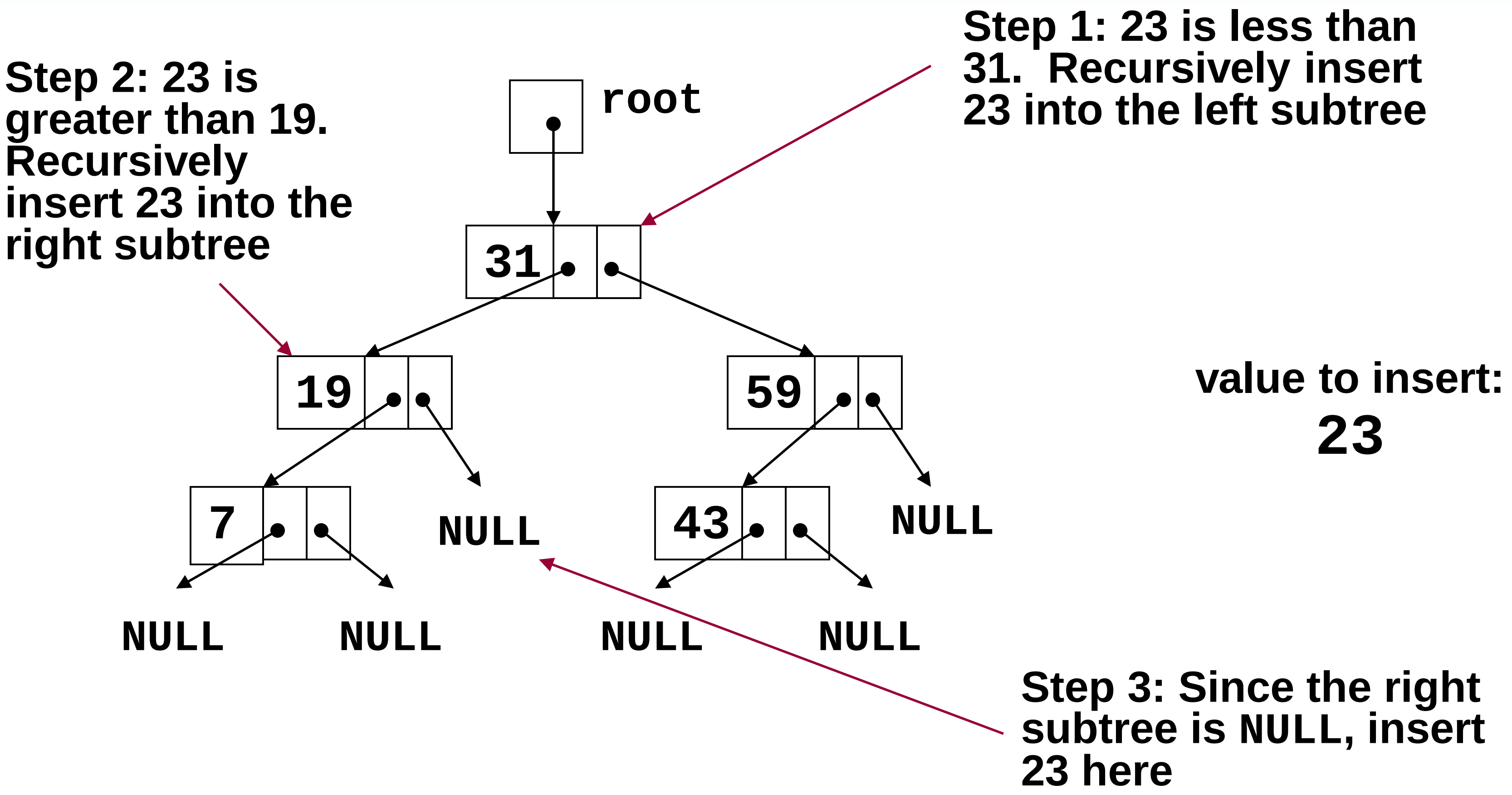
```
TreeNode *p;
int num = 23;
p = new TreeNode(num);
```



# Inserting an item into a Binary Search Tree

1) If the tree is empty, replace the empty tree with a new binary tree consisting of the new node as root, with empty left and right subtrees
2) Otherwise, if the item is less than the root, recursively insert the item in the left subtree. If the item is greater than the root, recursively insert the item into the right subtree

# Inserting an item into a Binary Search Tree

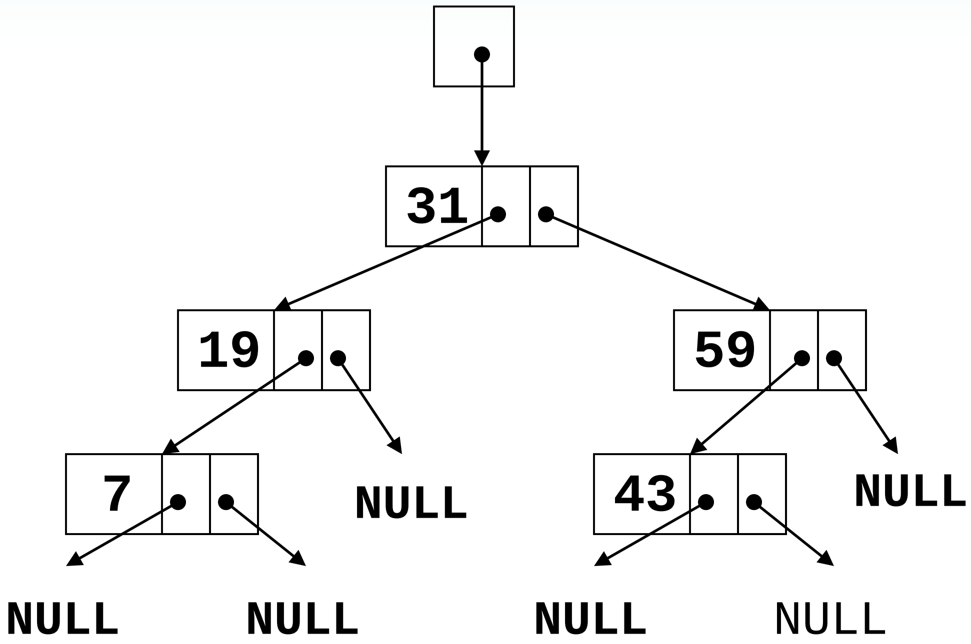


# Traversing a Binary Tree

Three traversal methods:

1) Inorder:
   a) Traverse left subtree of node
   b) Process data in node
   c) Traverse right subtree of node
2) Preorder:
   a) Process data in node
   b) Traverse left subtree of node
   c) Traverse right subtree of node
3) Postorder:
   a) Traverse left subtree of node
   b) Traverse right subtree of node
   c) Process data in node

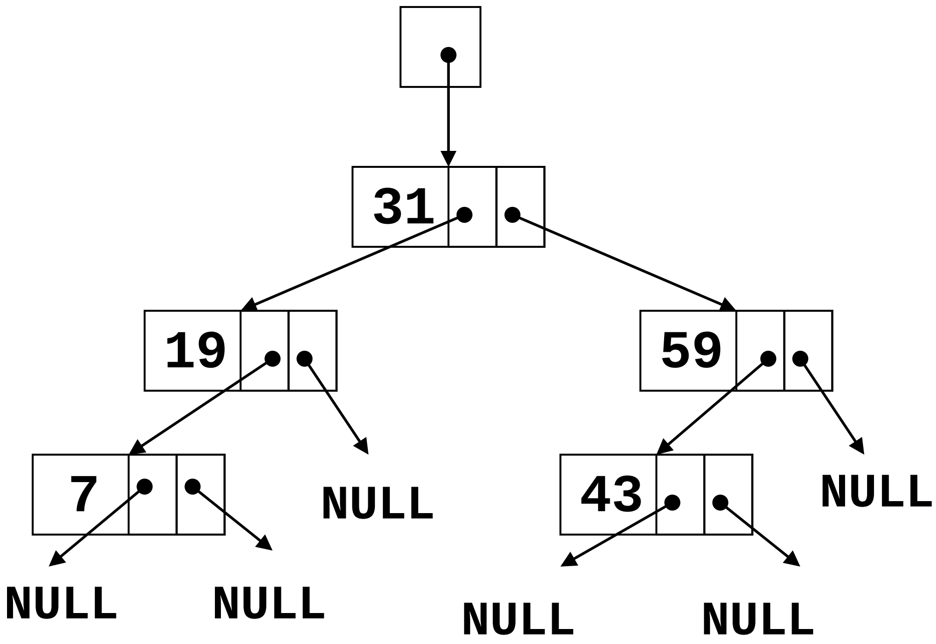
# Traversing a Binary Tree




| TRAVERSAL METHOD | NODES ARE VISITED IN THIS ORDER |
|---|---|
| Inorder | **7, 19, 31, 43, 59** |
| Preorder | **31, 19, 7, 59, 43** |
| Postorder | **7, 19, 43, 59, 31** |

# Searching in a Binary Tree

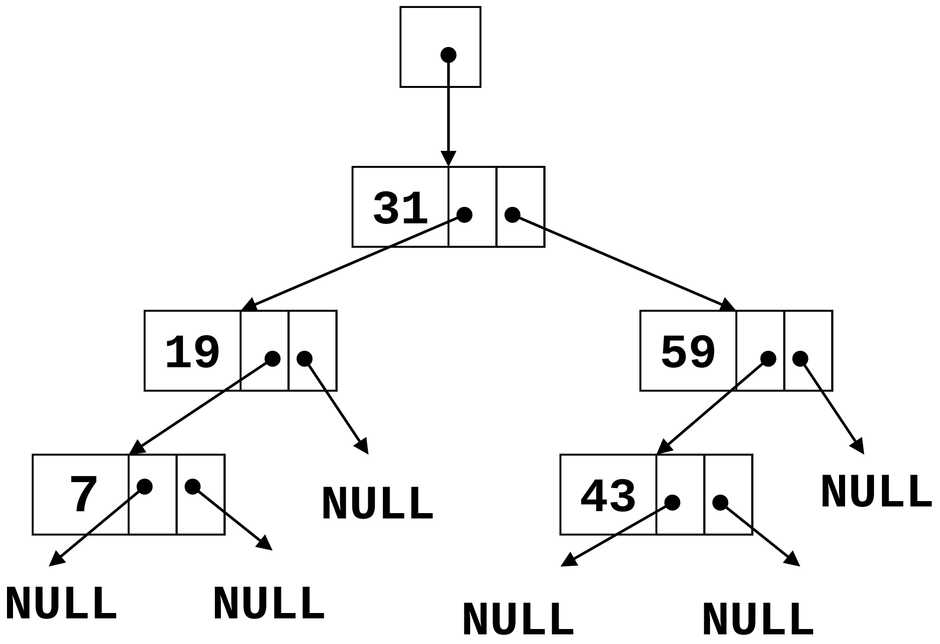
1) Start at root node
2) Examine node data:
   a) Is it desired value? Done
   b) Else, is desired data < node data? Repeat step 2 with left subtree
   c) Else, is desired data > node data? Repeat step 2 with right subtree
3) Continue until desired value found or **`NULL`** pointer reached



# Searching in a Binary Tree

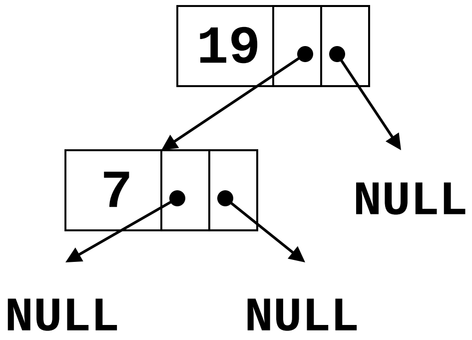
To locate the node containing **43**,

1. Examine the root node (**31**)
2. Since **43 > 31**, examine the right child of the node containing **31**, (**59**)
3. Since **43 < 59**, examine the left child of the node containing **59**, (**43**)
4. The node containing 43 has been found

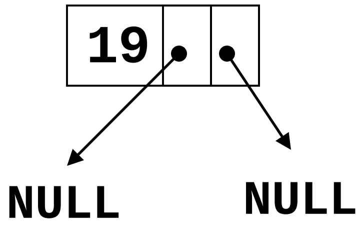


# Deleting a Node from a Binary Tree – Leaf Node

If node to be deleted is a leaf node, replace parent node's pointer to it with a **`NULL`** pointer, then delete the node




**Deleting node with 7 – before deletion**

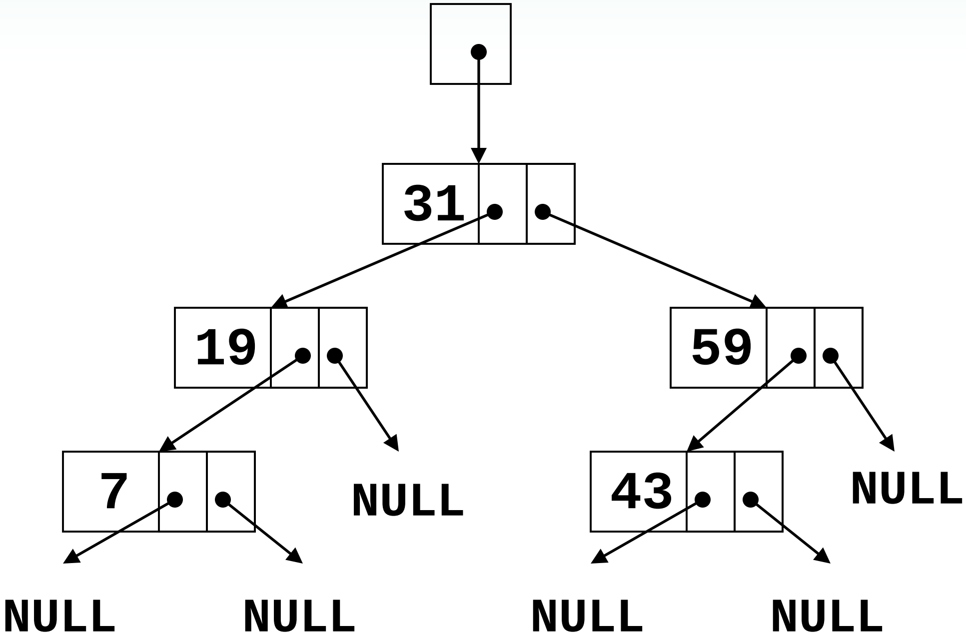



**Deleting node with 7 – after deletion**

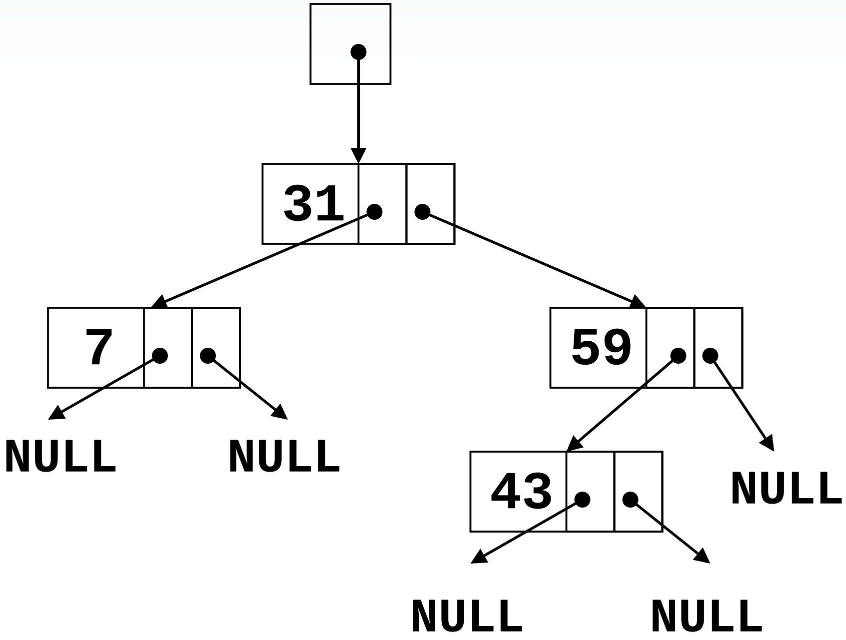
# Deleting a Node from a Binary Tree – One Child

If node to be deleted has one child node, adjust pointers so that parent of node to be deleted points to child of node to be deleted, then delete the node

# Deleting a Node from a Binary Tree – One Child




Deleting node containing 19 – before deletion

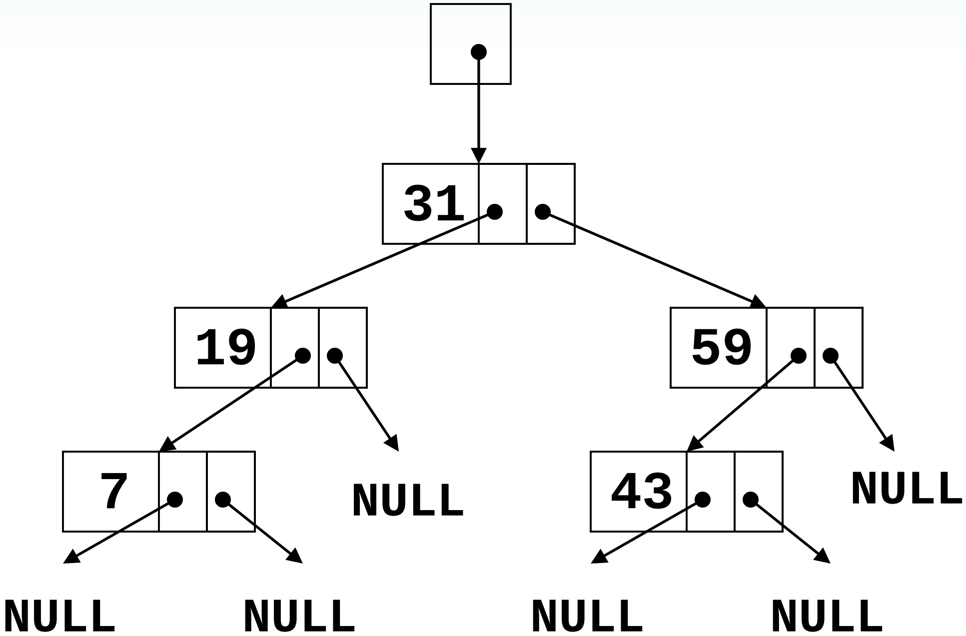



Deleting node containing 19 – after deletion

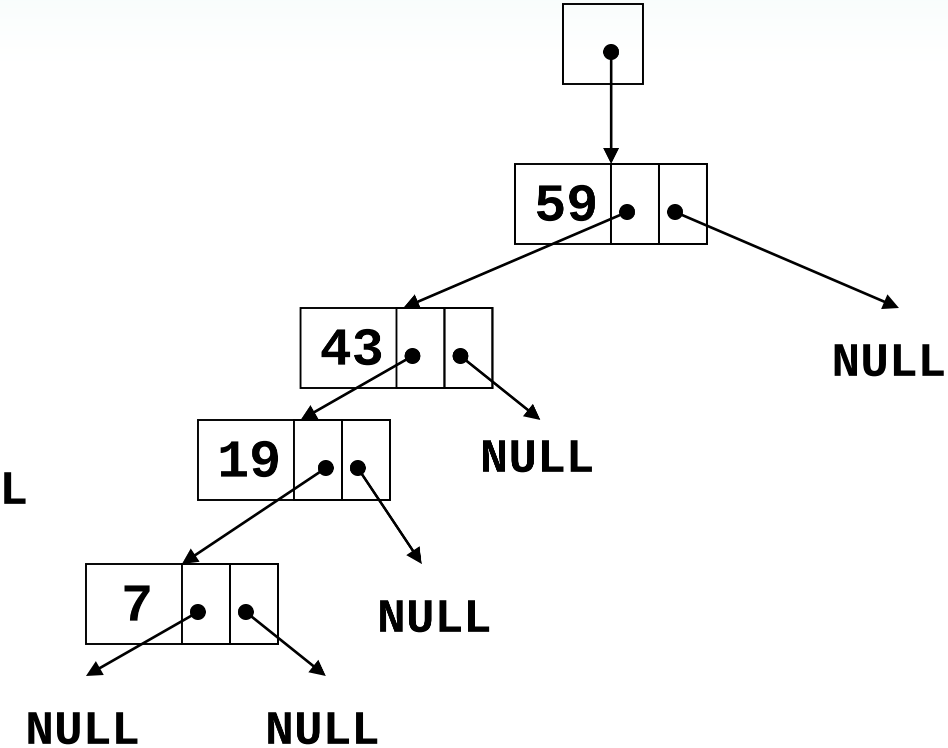
# Deleting a Node from a Binary Tree – Two Children

- If node to be deleted has left and right children,
  - ‘Promote’ one child to take the place of the deleted node
  - Locate correct position for other child in subtree of promoted child
- Convention in text: “attach" the right subtree to its parent, then position the left subtree at the appropriate point in the right subtree

# Deleting a Node from a Binary Tree – Two Children




**Deleting node with 31 – before deletion**




**Deleting node with 31 – after deletion**

# 19.3 Template Considerations for Binary Search Trees

- Binary tree can be implemented as a template, allowing flexibility in determining type of data stored
- Implementation must support relational operators **>**, **<**, and **==** to allow comparison of nodes