

**UNIT-1**  
**OBJECT ORIENTED**  
**THINKING**

# Programming Paradigms

- The most important aspect of C++ is its ability to support many different programming paradigms
  - procedural abstraction
  - modular abstraction
  - data abstraction
  - object oriented programming (this is discussed later, once we learn about the concept of inheritance)

# Procedural Abstraction

- This is where you build a “fence” around program segments, preventing some parts of the program from “seeing” how tasks are being accomplished.
- Any use of globals causes side effects that may not be predictable, reducing the viability of procedural abstraction

# Modular Abstraction

- With modular abstraction, we build a “screen” surrounding the internal structure of our program prohibiting programmers from accessing the data except through specified functions.
- Many times data structures (e.g., structures) common to a module are placed in a header files along with prototypes (allows external references)

# Modular Abstraction

- The corresponding functions that manipulate the data are then placed in an implementation file.
- Modules (files) can be compiled separately, allowing users access only to the object (.o) files
- We progress one small step toward OOP by thinking about the actions that need to take place on data...

# Modular Abstraction

- We implement modular abstraction by separating out various functions/structures/classes into multiple .cpp and .h files.
- .cpp files contain the implementation of our functions
- .h files contain the prototypes, class and structure definitions.

# Modular Abstraction

- We then include the .h files in modules that need access to the prototypes, structures, or class declarations:
  - `#include "myfile.h"`
  - (Notice the double quotes!)
- We then compile programs (on UNIX) by:
  - `g++ main.cpp myfile.cpp`
  - (Notice no .h file is listed on the above line)

# Data Abstraction

- Data Abstraction is one of the most powerful programming paradigms
- It allows us to create our own user defined data types (using the class construct) and
  - then define variables (i.e., objects) of those new data types.



# Data Abstraction

- With data abstraction we think about what operations can be performed on a particular type of data and not how it does it
- Here we are one step closer to object oriented programming

# Data Abstraction

- Data abstraction is used as a tool to increase the modularity of a program
- It is used to build walls between a program and its data structures
  - what is a data structure?
  - talk about some examples of data structures
- We use it to build new abstract data types

# Data Abstraction

- An abstract data type (ADT) is a data type that we create
  - consists of data and operations that can be performed on that data
- Think about a char type
  - it consists of 1 byte of memory and operations such as assignment, input, output, arithmetic operations can be performed on the data

# Data Abstraction

- An abstract data type is any type you want to add to the language over and above the fundamental types
- For example, you might want to add a new type called: `list`
  - which maintains a list of data
  - the data structure might be an array of structures
  - operations might be to add to, remove, display all, display some items in the list

# Data Abstraction

- Once defined, we can create lists without worrying about how the data is stored
- We “hide” the data structure used for the data within the data type -- so it is transparent to the program using the data type
- We call the program using this new data type: the client program (or client)

# Data Abstraction

- Once we have defined what data and operations make sense for a new data type, we can define them using the class construct in C++
- Once you have defined a class, you can create as many instances of that class as you want
- Each “instance” of the class is considered to be an “object” (variable)

# Data Abstraction

- Think of a class as similar to a data type
  - and an object as a variable
- And, just as we can have zero or more variables of any data type...
  - we can have zero or more objects of a class!
- Then, we can perform operations on an object in the same way that we can access members of a struct...

# Procedural versus Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program. The program starts at the beginning, does something, and ends.
- Object-Oriented programming is based on the data and the functions that operate on it. Objects are instances of abstract data types that represent the data and its functions



# Limitations of Procedural Programming

- If the data structures change, many functions must also be changed
- Programs that are based on complex function hierarchies are:
  - difficult to understand and maintain
  - difficult to modify and extend
  - easy to break

# What is OOPs?

**Object-oriented programming** – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

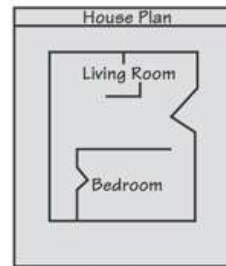
# Object-Oriented Programming Terminology

- class: like a `struct` (allows bundling of related variables), but variables and functions in the class can have different properties than in a `struct`
- object: an instance of a `class`, in the same way that a variable can be an instance of a `struct`

# Classes and Objects

- A Class is like a blueprint and objects are like houses built from the blueprint

Blueprint that describes a house.



Instances of the house described by the blueprint.



# Object-Oriented Programming Terminology

- attributes: members of a class
- methods or behaviors: member functions of a class

# More Object Terms

- data hiding: restricting access to certain members of an object
- public interface: members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption

# Creating a Class

- Objects are created from a `class`
- Format:

```
class ClassName
{
    declaration;
    declaration;
};
```

# Classic Class Example

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```



# Access Specifiers

- Used to control access to members of the class
- `public`: can be accessed by functions outside of the class
- `private`: can only be called by or accessed by functions that are members of the class
- In the example on the next slide, note that the functions are prototypes only (so far)

# Class Example

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```

# Access Specifiers

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```

Private Members

Public Members

# Access Specifiers (continued)

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

# Using `const` With Member Functions

- `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

```
double getWidth() const;  
double getLength() const;  
double getArea() const;
```

# Defining a Member Function

- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator ( :: )

```
int Rectangle::setWidth(double w)
{
    width = w;
}
```

# Global Functions

- Functions that are not part of a class, that is, do not have the `Class::name` notation, are global. This is what we have done up to this point.

# Accessors and Mutators

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way
- Accessor: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.



# Defining an Instance of a Class

- An object is an instance of a class
- Defined like structure variables:

```
Rectangle r;
```

- Access members using dot operator:

```
r.setWidth(5.2);
```

```
cout << r.getWidth();
```

- Compiler error if you attempt to access a `private` member using dot operator

# Derived Attributes

- Some data must be stored as an attribute.
- Other data should be computed. If we stored “area” as a field, its value would have to change whenever we changed length or width.
- In a class about a “person,” store birth date and compute age

# Pointers to Objects

- Can define a pointer to an object:

```
Rectangle *rPtr;
```

- Can access public members via pointer:

```
rPtr = &otherRectangle;
```

```
rPtr->setLength(12.5);
```

```
cout << rPtr->getLength() << endl;
```

# Dynamically Allocating Objects

```
Rectangle *r1;
```

```
r1 = new Rectangle();
```

- This allocates a rectangle and returns a pointer to it. Then:

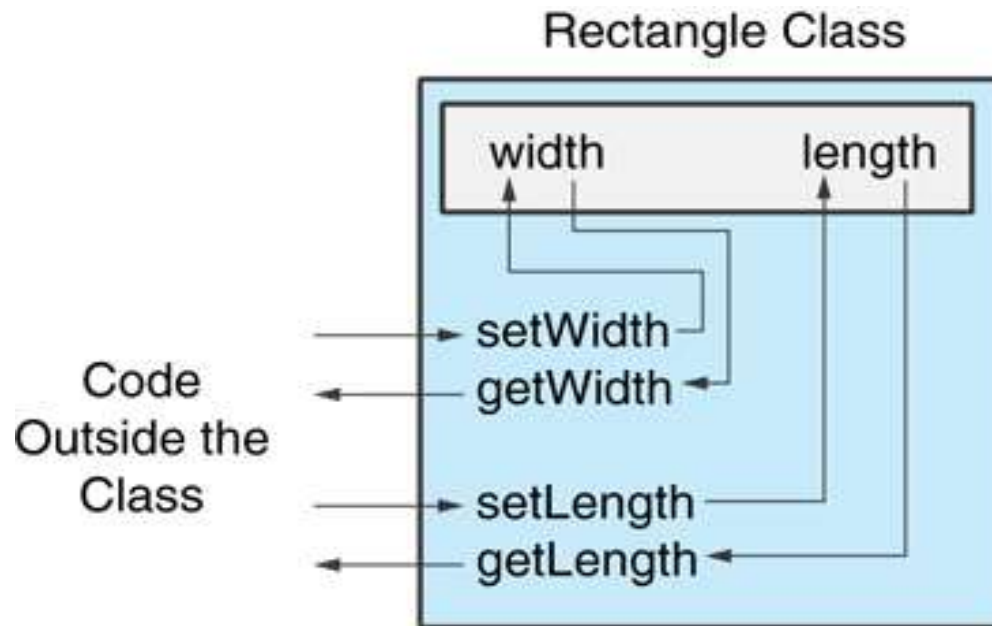
```
r1->setWidth(12.4);
```

# Private Members

- Making data members `private` provides data protection
- Data can be accessed only through `public` functions
- Public functions define the class's public interface

# Private Members

Code outside the class must use the class's public member functions to interact with the object.



# Separating Specification from Implementation

- Place class declaration in a header file that serves as the class specification file. Name the file *ClassName.h*, for example, `Rectangle.h`
- Place member function definitions in *ClassName.cpp*, for example, `Rectangle.cpp` File should `#include` the class specification file
- Programs that use the class must `#include` the class specification file, and be compiled and linked with the member function definitions

# Inline Member Functions

- Member functions can be defined
  - inline: in class declaration
  - after the class declaration
- Inline appropriate for short function bodies:

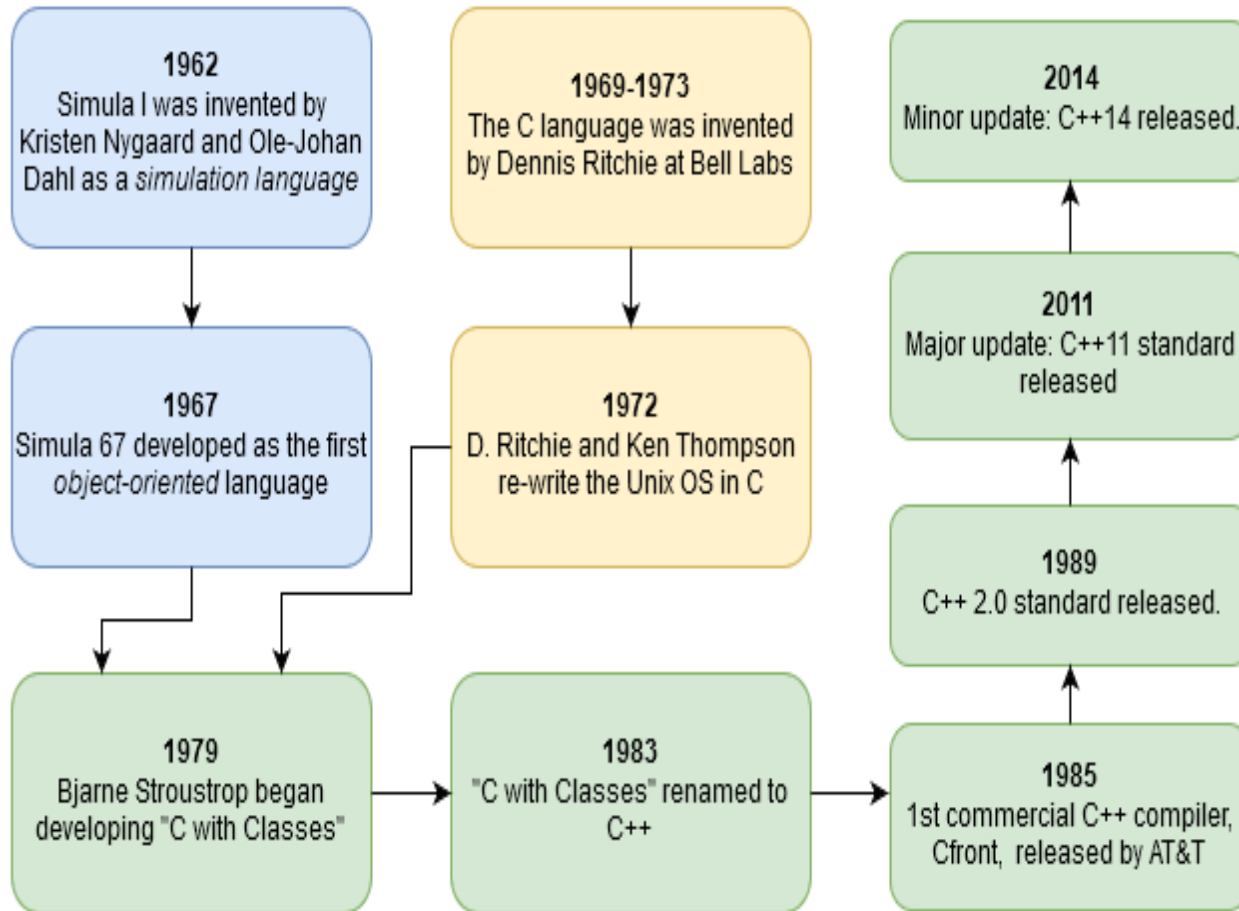
```
int getWidth() const
{ return width; }
```



# Tradeoffs – Inline vs. Regular Member Functions

- Regular functions – when called, compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence faster execution

# Very brief history of C++



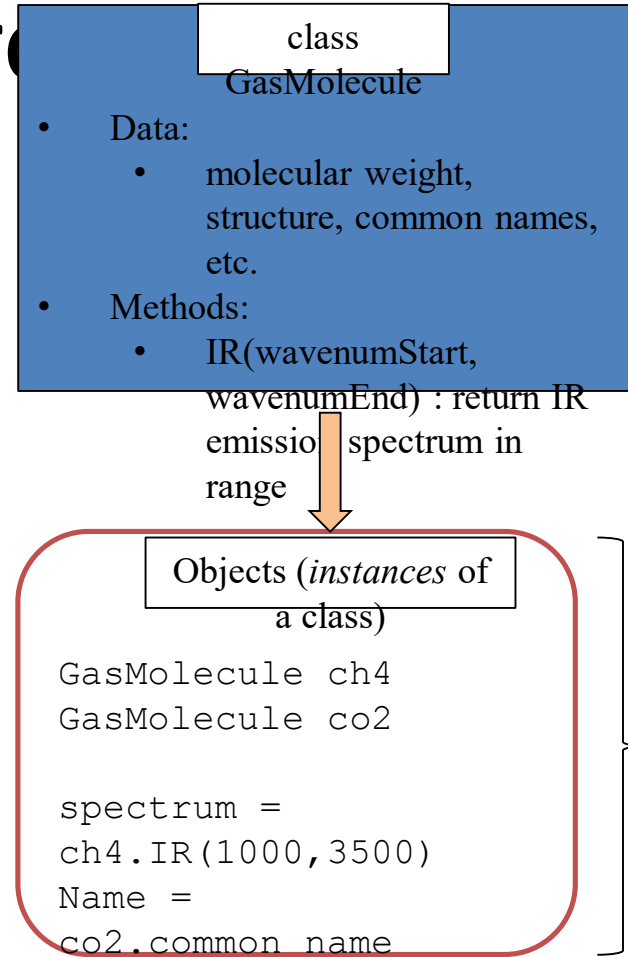
C



C++

# Object-oriented programming

- Object-oriented programming (OOP) seeks to define a program in terms of the *things* in the problem (files, molecules, buildings, cars, people, etc.), what they need, and what they can do.

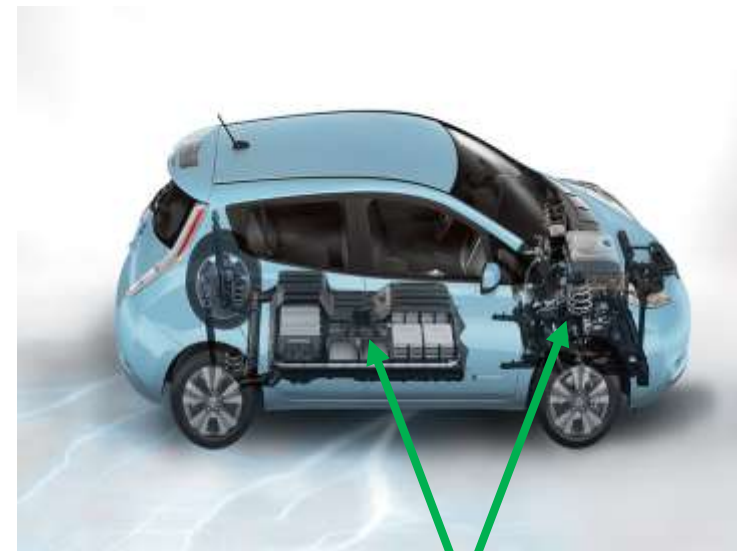


# Object-oriented programming

“Class Car”

- OOP defines *classes* to represent these things.
- Classes can contain data and methods (internal functions).
- Classes control access to internal data and methods. A public interface is used by external code when using the class.
- This is a highly effective way of modeling real world problems inside of a computer program.

public interface



private data and methods

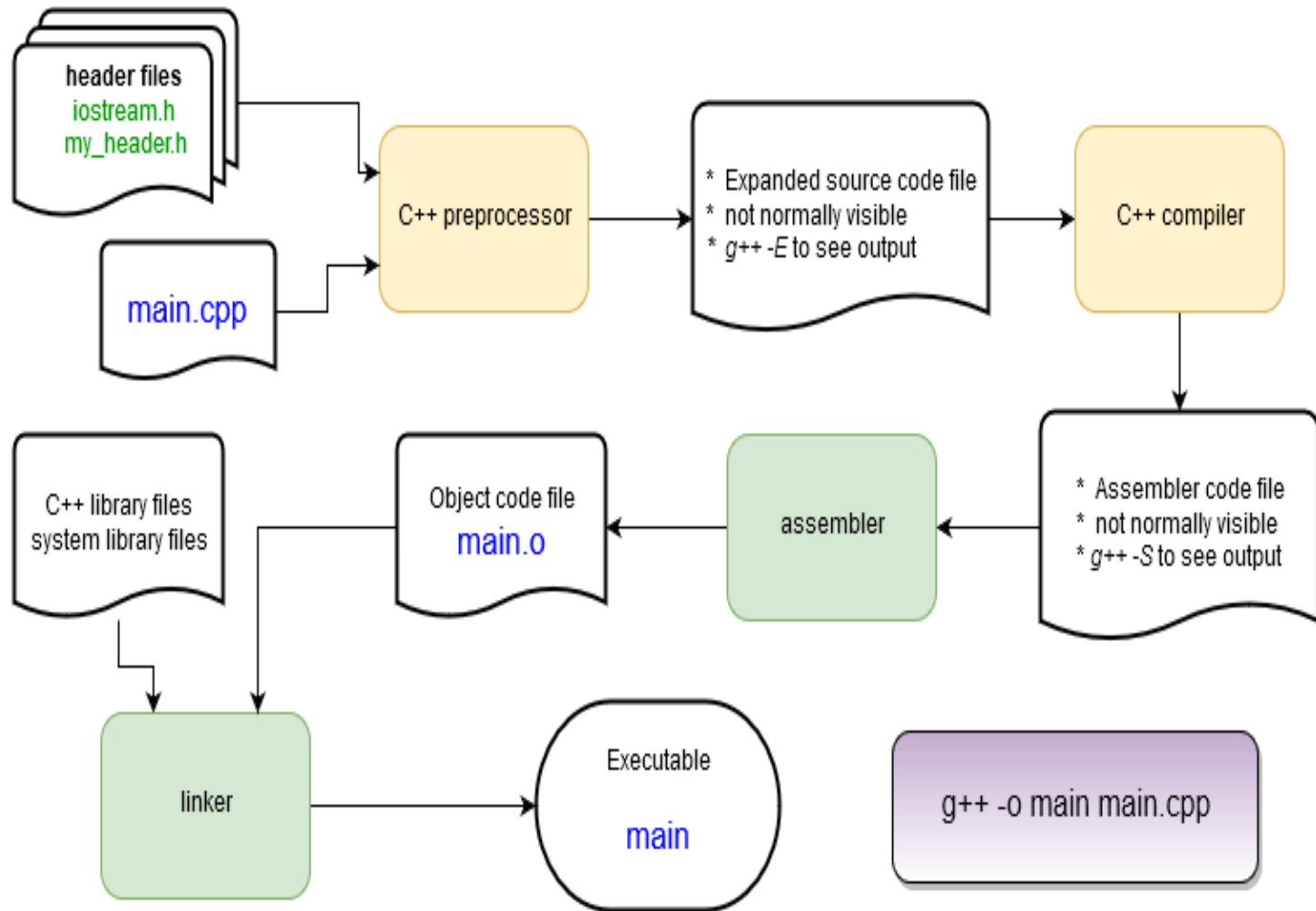
# Characteristics of C++

- C++ is...
  - Compiled.
    - A separate program, the compiler, is used to turn C++ source code into a form directly executed by the CPU.
  - Strongly typed and unsafe
    - Conversions between variable types must be made by the programmer (strong typing) but can be circumvented when needed (unsafe)
  - C compatible
    - call C libraries directly and C code is nearly 100% valid C++ code.
  - Capable of very high performance
    - The programmer has a very large amount of control over the program execution
  - Object oriented
    - With support for many programming styles (procedural, functional, etc.)
- No automatic memory management
  - The programmer is in control of memory usage

# When to choose C++

- Despite its many competitors C++ has remained popular for ~30 years and will continue to be so in the foreseeable future.
- Why?
  - Complex problems and programs can be effectively implemented
  - OOP works in the real world!
  - No other language quite matches C++'s combination of performance, expressiveness, and ability to handle complex programs.
- Choose C++ when:
  - Program performance matters
    - Dealing with large amounts of data, multiple CPUs, complex algorithms, etc.
  - Programmer productivity is less important
    - It is faster to produce working code in Python, R, Matlab or other scripting languages!
  - The programming language itself can help organize your code
    - Ex. In C++ your objects can closely model elements of your problem
  - Access to libraries
    - Ex. Nvidia's CUDA Thrust library for GPUs
  - Your group uses it already!

# Behind the Scenes: The Compilation Process



## Hello, World! explained

```
main.cpp X
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello world!" << endl;
8      return 0;
9  }
10
```

The *main* routine – the start of **every** C++ program! It returns an integer value to the operating system and (in this case) takes no arguments: `main()`

The **return** statement returns an integer value to the operating system after completion. 0 means “no error”. C++ programs **must** return an integer value.



# Hello, World! explained

```
main.cpp X
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello world!" << endl;
8      return 0;
9  }
10
```

loads a *header* file containing function and class definitions


Loads a *namespace* called *std*. Namespaces are used to separate sections of code for programmer convenience. To save typing we'll always use this line in this tutorial.

- *cout* is the *object* that writes to the stdout device, i.e. the console window.
- It is part of the C++ standard library.
- Without the “using namespace std;” line this would have been called as *std::cout*. It is defined in the *iostream* header file.
- << is the C++ *insertion operator*. It is used to pass characters from the right to the object on the left. *endl* is the C++ newline character.

# Header Files

C++ language headers aren't referred to with the .h suffix. `<iostream>` provides definitions for I/O functions, including the `cout` function.

- C++ (along with C) uses *header files* as to hold definitions for the compiler to use while compiling.
- A source file (file.cpp) contains the code that is compiled into an object file (file.o).
- The header (file.h) is used to tell the compiler what to expect when it assembles the program in the linking stage from the object files.
- Source files and header files can refer to any number of other header files.



```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " +
world ;
    cout << msg << endl;
    msg[0] = 'h';
    cout << msg << endl;
    return 0;
}
```

# Slight change

- Let's put the message into some variables of type *string* and print some numbers.
- Things to note:
  - Strings can be concatenated with a + operator.
  - No messing with null terminators or *strcat()* as in C
- Some string notes:
  - Access a string character by brackets or function:
    - `msg[0]` → "H" or `msg.at(0)` → "H"
    - C++ strings are *mutable* – they can be changed in place.
- Press F9 to recompile & run.

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " +
world ;
    cout << msg << endl;
    msg[0] = 'h';
    cout << msg << endl;
    return 0;
}
```



# A first C++ class: *string*

- *string* is not a basic type (more on those later), it is a class.
- `string hello` creates an *instance* of a string called "hello".
- `hello` is an object.
- Remember that a class defines some data and a set of functions (methods) that operate on that data.
- Let's use C::B to see what some of these methods are....

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " +
world ;
    cout << msg << endl;
    msg[0] = 'h';
    cout << msg << endl;
    return 0;
}
```

# A first C++ class: *string*

- Update the code as you see here.
- After the last character is entered C::B will display some info about the string class.
- If you click or type something else just delete and re-type the last character.
- Ctrl-space will force the list to appear.

```
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " +
world ;
    cout << msg << endl;
    msg[0] = 'h';
    cout << msg << endl;

    msg

    return 0;
}
```

# A first C++ class: *string*

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     string hello = "Hello";
8     string world = "world!";
9     string msg = hello + " " + world ;
10    cout << msg << endl;
11    msg[0] = 'h';
12    cout << msg << endl;
13
14    msg
15    hello: string
16    msg: string
17
18    world: string
19
20    (_gthrw_pthread_cond_signal, pthread_cond_signal, pthread_cond_signal)(): (_gthrw_pthread_cond_init, pthread_cond...
21    (_gthrw_pthread_key_create, pthread_key_create, pthread_key_create)(): (_gthrw_pthread_cond_timedwait, pthread_co...
22    (_gthrw_pthread_mutex_init, pthread_mutex_init, pthread_mutex_init)(): (_gthrw_pthread_mutex_timedlock, pthread_m...
23    (_gthrw_pthread_mutex_lock, pthread_mutex_lock, pthread_mutex_lock)(): (_gthrw_pthread_cancel, pthread_cancel, pt...
24    (_gthrw_pthread_self, pthread_self, pthread_self)(): (_gthrw_pthread_join, pthread_join, pthread_join)(_gthrw_pt...
25    (_gthrw_pthread_setspecific, pthread_setspecific, pthread_setspecific)(): (_gthrw_pthread_once, pthread_once, pth...
26    * _pthread_key_dest(): void
27    abort(): void
28    address(): const_pointer
```

List of other string objects

List of string methods

Shows this function (main) and the type of msg (string)

[main](#)

[string msg \(variable\)](#)

[Open declaration](#)

[Close Top](#)

- Next: let's find the size() method without scrolling for it.

# A first C++ class: *string*

- Start typing “msg.size()” until it appears in the list. Once it’s highlighted (or you scroll to it) press the Tab key to auto-enter it.
- On the right you can click “Open declaration” to see how the C++ compiler defines size(). This will open *basic\_string.h*, a built-in file.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     string hello = "Hello";
8     string world = "world!";
9     string msg = hello + " " + world;
10    cout << msg << endl;
11    msg[0] = 'h';
12    cout << msg << endl;
13
14    msg.size()
15
16
17
18 }
```

```
sig_atomic_t
# SIG_BLOCK
# SIG_DFL
# SIG_ERR
# SIG_GET
# SIG_IGN
# SIG_SETMASK
# SIG_SGE
# SIG_UNBLOCK
(0) size(): size_type
```

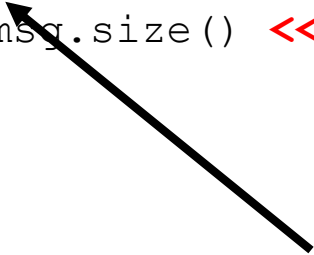
```
std:: cxx11::basic_string
public size_type size() const
(function)
Open declaration
Open implementation
Close Top
```

# A first C++ class: *string*

- Tweak the code to print the number of characters in the string, build, and run it.
- From the point of view of `main()`, the `msg` object has hidden away its means of tracking and retrieving the number of characters stored.
- Note: while the `string` class has a **huge** number of methods your typical C++ class has far fewer!

```
#include <iostream>
using namespace std;
int main()
{
    string hello = "Hello" ;
    string world = "world!" ;
    string msg = hello + " " +
world ;
    cout << msg << endl ;
    msg[0] = 'h';
    cout << msg << endl ;

    cout << msg.size() << endl
;
return 0;
}
```



- Note that `cout` prints integers without any modification!



# Break your code.

- Remove a semi-colon. Re-compile. What messages do you get from the compiler and C::B?
- Fix that and break something else. Capitalize *string* → *String*
- C++ can have elaborate error messages when compiling. Experience is the only way to learn to interpret them!
- Fix your code so it still compiles and then we'll move on...

# Basic Syntax

- C++ syntax is very similar to C, Java, or C#. Here's a few things up front and we'll cover more as we go along.

- Curly braces are used to denote a code block (like the main() function):

```
{ ... some code ... }
```

- Statements end with a semicolon:

```
int a ;  
a = 1 + 3 ;
```

- Comments are marked for a single line with a // or for multilines with a pair of /\* and \*/ :

```
// this is a  
comment.  
/* everything in  
here
```

- Variables can be declared in a code block.

```
comment */
```

```
void  
my_function() {  
    int a ;  
    a=1 ;  
    int b;  
}
```

- Functions are sections of code that are called from other code. Functions always have a return argument type, a function name, and then a list of arguments separated by commas:

```
int add(int x, int y)
{
    int z = x + y ;
    return z ;
}
```

```
// No arguments? Still need ():
void my_function() {
    /* do something...
       but a void value means
       the return statement can be
       skipped.*/
}
```

- Variables are declared with a type and name:

```
// Specify the type
int x = 100;
float y;
vector<string> vec ;
// Sometimes types can be
inferred
auto z = x;
```

- A sampling of arithmetic operators:
  - Arithmetic: + - \* / % ++ --
  - Logical: && (AND) || (OR) !(NOT)
  - Comparison: == > < >= <= !=
- Sometimes these can have special meanings beyond arithmetic, for example the “+” is used to concatenate strings.
- What happens when a syntax error is made?
  - The compiler will complain and refuse to compile the file.
  - The error message *usually* directs you to the error but sometimes the error occurs before the compiler discovers syntax errors so you hunt a little bit.

# Built-in (aka primitive or intrinsic) Types

- “primitive” or “intrinsic” means these types are not objects
- Here are the most commonly used types.
- Note: The exact bit ranges here are **platform and compiler dependent!**
  - Typical usage with PCs, Macs, Linux, etc. use these values
  - Variations from this table are found in specialized applications like embedded system processors.

Name	Name	Value
char	unsigned char	8-bit integer
short	unsigned short	16-bit integer
int	unsigned int	32-bit integer
long	unsigned long	64-bit integer
bool		true or false

Name	Value
float	32-bit floating point
double	64-bit floating point
long long	128-bit integer
long double	128-bit floating point

<http://www.cplusplus.com/doc/tutorial/variables/>

# Need to be sure of integer sizes?

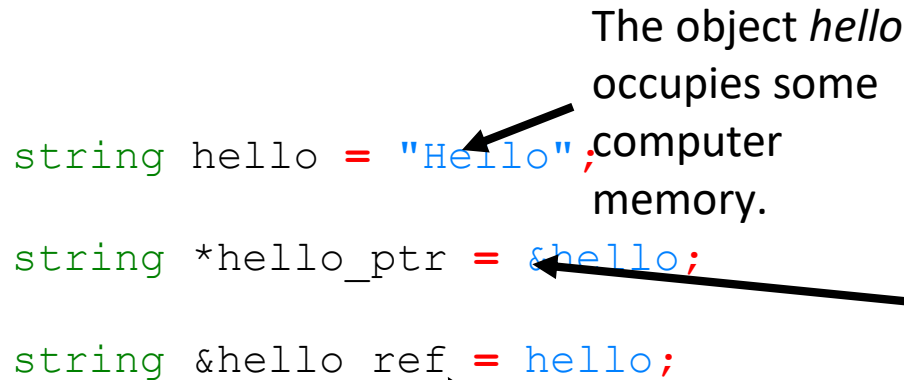
- In the same spirit as using *integer(kind=8)* type notation in Fortran, there are type definitions that exactly specify exactly the bits used. These were added in C++11.
- These can be useful if you are planning to port code across CPU architectures (ex. Intel 64-bit CPUs to a 32-bit ARM on an embedded board) or when doing particular types of integer math.
- For a full list and description see: <http://www.cplusplus.com/reference/cstdint/>

```
#include <cstdint>
```

Name	Name	Value
int8_t	uint8_t	8-bit integer
int16_t	uint16_t	16-bit integer
int32_t	uint32_t	32-bit integer
int64_t	uint64_t	64-bit integer

# Reference and Pointer Variables

```
string hello = "Hello";  
string *hello_ptr = &hello;  
string &hello_ref = hello;
```



The object *hello* occupies some computer memory.

The asterisk indicates that *hello\_ptr* is a pointer to a string. *hello\_ptr* variable is assigned the memory address of object *hello* which is accessed with the “&” syntax.

The & here indicates that *hello\_ref* is a reference to a string. The *hello\_ref* variable is assigned the memory address of object *hello* automatically.

- Variable and object values are stored in particular locations in the computer’s memory.
- Reference and pointer variables **store the memory location of other variables.**
- Pointers are found in C. References are a C++ variation that makes pointers easier and safer to use.
- More on this topic later in the tutorial.

# Type Casting

- C++ is strongly typed. It will auto-convert a variable of one type to another in a limited fashion: if it will not change the value.

```
short x = 1 ;  
int y = x ; //  
OK  
short z = y ; //
```

- Conversions that don't change the value: increasing precision (float → double) or integer → floating point of at least the same precision.
- C++ allows for C-style type casting with the syntax: (new type) expression

```
double x = 1.0 ;  
int y = (int) x ;  
float z = (float) (x  
/ y) ;
```

- But since we're doing C++ we'll look at the 4 ways of doing this in C++ next...



# Type Casting

- `static_cast<new type>( expression )`

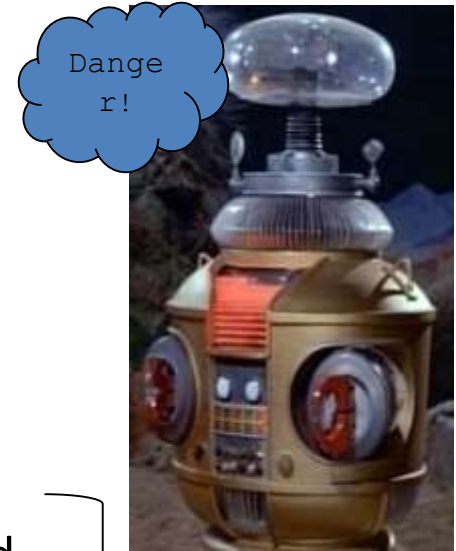
- This is exactly equivalent to the C style cast.
- This identifies a cast **at compile time**.
- This will allow casts that reduce precision (ex. double → float)
- ~99% of all your casts in C++ will be of this type.

```
double d = 1234.56 ;
float f =
static_cast<float>(d) ;
// same as
float g = (float) d ;
```

- `dynamic_cast<new type>( expression )`

- Special version where type casting is performed at runtime, only works on reference or pointer type variables.
- Usually handled automatically by the compiler where needed, rarely done by the programmer.

# Type Casting cont'd



- `const_cast<new type>( expression )`
  - Variables labeled as *const* can't have their value changed.
  - `const_cast` lets the programmer remove or add *const* to reference or pointer type variables.
  - If you need to do this, you probably want to re-think your code.
- `reinterpret_cast<new type>( expression )`
  - Takes the bits in the expression and re-uses them **unconverted** as a new type. Also only works on reference or pointer type variables.
  - Sometimes useful when reading in binary files and extracting parameters.

“unsafe”: the compiler will not protect you here!

The programmer must make sure everything is correct!

# Functions

The return type is *float*.

The function arguments L and W are sent as type *float*.

- Open the project “FunctionExample” in C::B files
  - Compile and run it!
- Open main.cpp
- 4 function calls are listed.
- The 1<sup>st</sup> and 2<sup>nd</sup> functions are identical in their behavior.
  - The values of L and W are sent to the function, multiplied, and the product is returned.
- RectangleArea2 uses *const* arguments
  - The compiler **will not** let you modify their values in the function.
  - Try it! Uncomment the line and see what happens when you recompile.
- The 3<sup>rd</sup> and 4<sup>th</sup> versions pass the arguments by *reference* with an added &

```
float RectangleArea1(float L, float W) {
    return L*W ;
}

float RectangleArea2(const float L, const float W)
{
    // L=2.0 ;
    return L*W ;
}

float RectangleArea3(const float& L, const float&
W) {
    return L*W ;
}

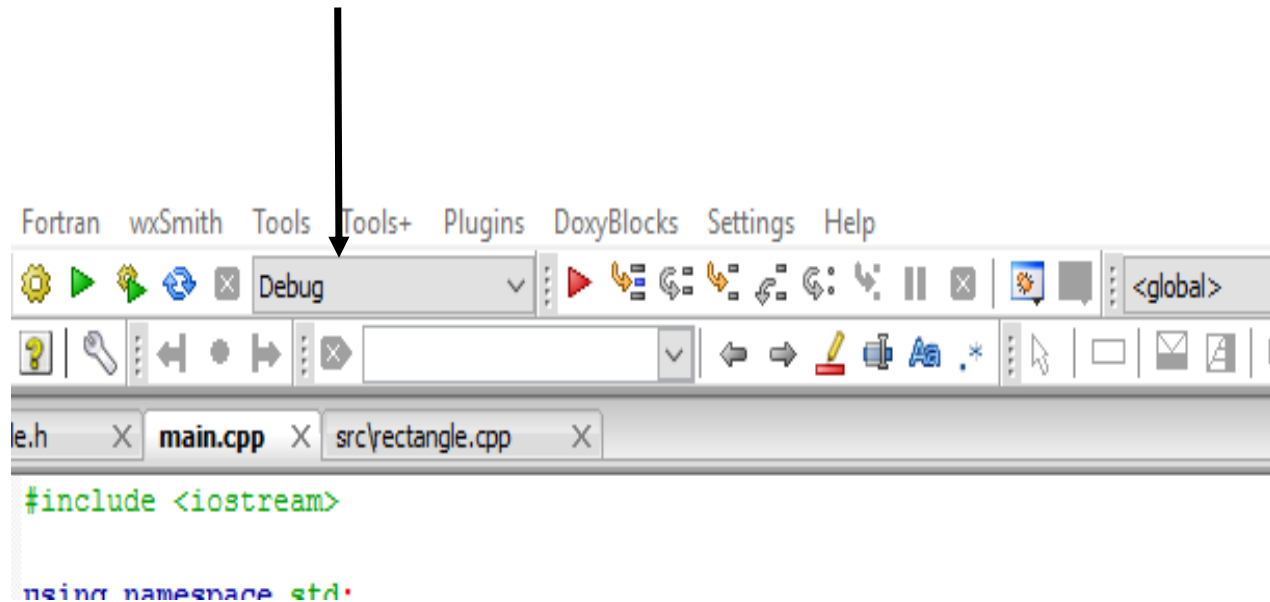
void RectangleArea4(const float& L, const float& W,
float& area) {
    area= L*W ;
}
```

Product is computed



# Using the C::B Debugger

- To show how this works we will use the C::B interactive debugger to step through the program line-by-line to follow the function calls.
- Make sure you are running in *Debug* mode. This turns off compiler optimizations and has the compiler include information in the compiled code for effective debugging.



# Add a Breakpoint

- Breakpoints tell the debugger to halt at a particular line so that the state of the program can be inspected.
- In main.cpp, double click to the left of the lines in the functions to set a pair of breakpoints. A red dot will appear.
- Click the red arrow to start the code in the debugger.



```
1 #include <iostream>
2
3 using namespace std;
4
5
6 float RectangleArea1(float L, float W)
7 {
8     return L*W ;
9 }
10
11
12
13 float RectangleArea2(const float L, const float W)
14 {
15     // L=2.0 ;
16     return L*W ;
17 }
18
19
20
21 float RectangleArea3(const float& L, const float& W)
22 {
23     return L*W ;
24 }
25
26
27
28 void RectangleArea4(const float& L, const float& W, float& a:
29 {
30     area= L*W ;
31 }
32
33
```



- The debugger will pause in the first function at the breakpoint.

```
5
6 float RectangleArea1(float L, float W)
7 {
8     return L*W ;
9 }
10
11
12
13 float RectangleArea2(const float L, const float W)
14 {
15     // L=2.0 ;
16     return L*W ;
17 }
18
19
20
21 float RectangleArea3(const float& L, const float& W)
22 {
23     return L*W ;
24 }
25
26
27
28 void RectangleArea4(const float& L, const float& W, float& area)
29 {
30     area= L*W ;
31 }
32
33
```

- Click the Debug menu, go to Debugging Windows, and choose *Call Stack*. Drag it to the right, then go back and choose *Watches*. Drag it to the right. Do the same for the *Breakpoints* option. Your screen will look something like this now...



Place the cursor in the function, click to run to the cursor

Run the next line

Step into a function call

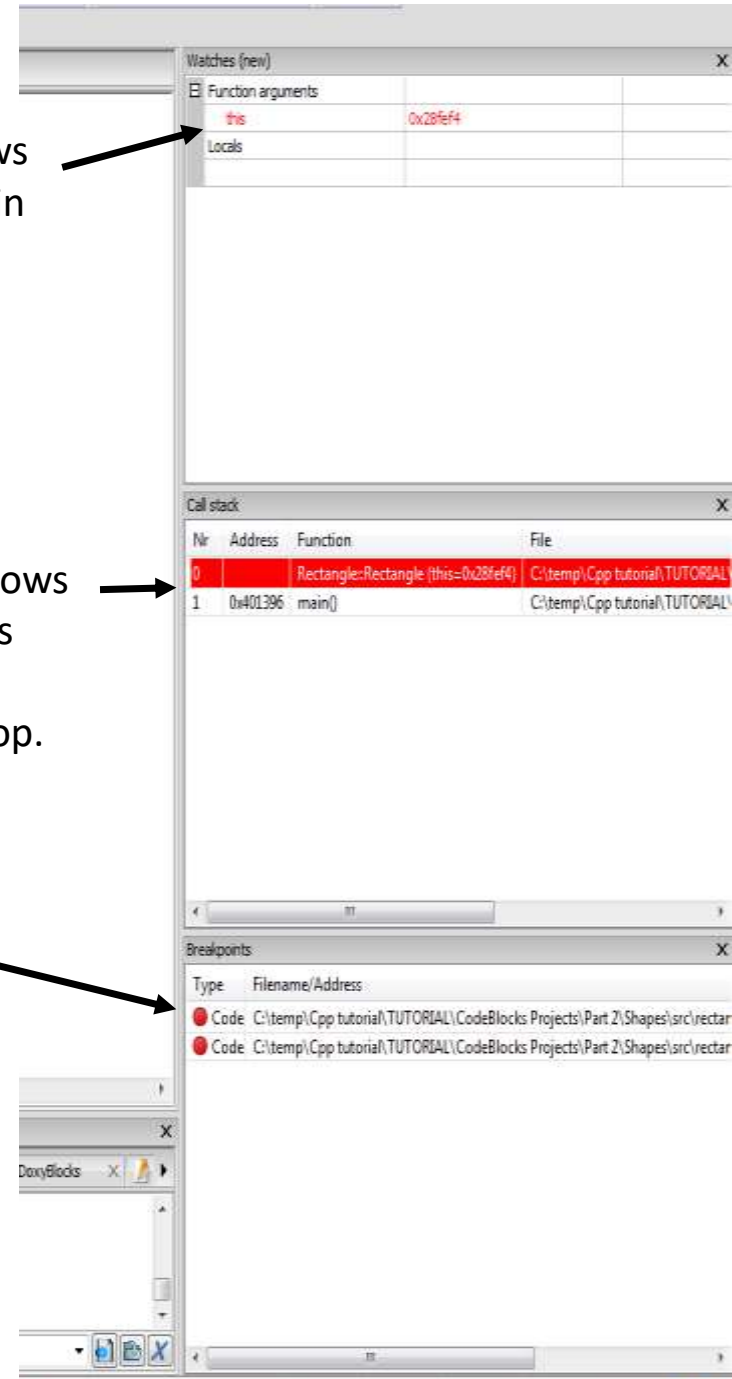
Step out of a function to the calling function.

Step by CPU instruction. Less useful, generally.

*Watches* shows the variables in use and their values

*Call Stack* shows the functions being called, newest on top.

*Breakpoints* lists the breakpoints you've created.



# UNIT-2

## C++ Classes and Data Abstraction



# Classes

- Object-oriented design (OOD): a problem solving methodology
- Objects: components of a solution
- Class: a collection of a fixed number of components
- Member: a component of a class

# Classes (cont'd.)

- Class definition:
  - Defines a data type; no memory is allocated
  - Don't forget the semicolon after the closing brace
- Syntax:

```
class classIdentifier
{
    classMembersList
};
```

# Classes (cont'd.)

- Class member can be a variable or a function
- If a member of a `class` is a variable
  - It is declared like any other variable
  - You cannot initialize a variable when you declare it
- If a member of a `class` is a function
  - Function prototype is listed
  - Function members can (directly) access any member of the `class`

# Classes (cont'd.)

- Three categories of class members:
  - `private` (default)
    - Member cannot be accessed outside the `class`
  - `public`
    - Member is accessible outside the class
  - `protected`

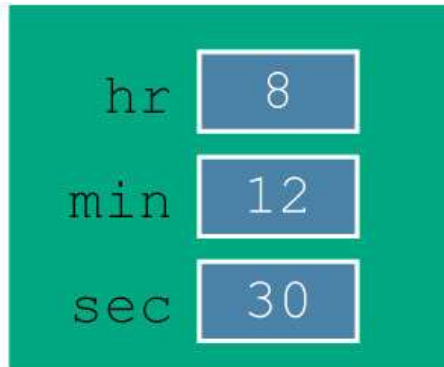
# Variable (Object) Declaration

- Once defined, you can declare variables of that `class` type

```
clockType myClock;
```

- A `class` variable is called a class object or class instance

myClock



yourClock

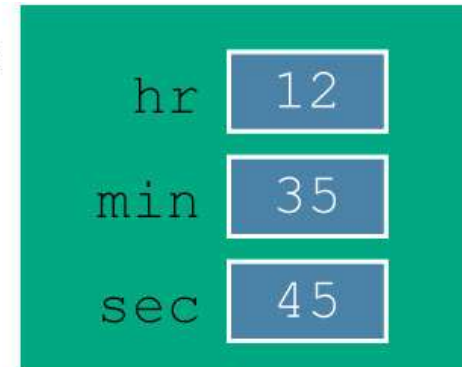


FIGURE 10-2 Objects myClock and yourClock

# Accessing Class Members

- Once an object is declared, it can access the `public` members of the class
- Syntax:

```
classObjectName.memberName
```

- The dot (.) is the **member access operator**
- If an object is declared in the definition of a member function of the `class`, it can access the `public` and `private` members

# Built-in Operations on Classes

- Most of C++'s built-in operations do not apply to classes
  - Arithmetic operators cannot be used on class objects unless the operators are overloaded
  - Cannot use relational operators to compare two class objects for equality
- Built-in operations that are valid for class objects:
  - Member access (.)
  - Assignment (=)

# Assignment Operator and Classes

myClock

hr	2
min	26
sec	47

yourClock

hr	14
min	39
sec	28

myClock

hr	14
min	39
sec	28

yourClock

hr	14
min	39
sec	28

(a) myClock and yourClock before executing `myClock = yourClock;`

(b) myClock and yourClock after executing `myClock = yourClock;`

**FIGURE 10-3** myClock and yourClock before and after executing the statement `myClock = yourClock;`



# Class Scope

- An object can be automatic or static
  - Automatic: created when the declaration is reached and destroyed when the surrounding block is exited
  - Static: created when the declaration is reached and destroyed when the program terminates
- Object has the same scope as other variables

# Class Scope (cont'd.)

- A member of the `class` is local to the `class`
- Can access a `class` member outside the `class` by using the `class` object name and the member access operator (`.`)

# Functions and Classes

- Objects can be passed as parameters to functions and returned as function values
- As parameters to functions
  - Objects can be passed by value or by reference
- If an object is passed by value
  - Contents of data members of the actual parameter are copied into the corresponding data members of the formal parameter

# Reference Parameters and Class Objects (Variables)

- Passing by value might require a large amount of storage space and a considerable amount of computer time to copy the value of the actual parameter into the formal parameter
- If a variable is passed by reference
  - The formal parameter receives only the address of the actual parameter

# Reference Parameters and Class Objects (Variables) (cont'd.)

- Pass by reference is an efficient way to pass a variable as a parameter
  - Problem: when passing by reference, the actual parameter changes when formal parameter changes
  - Solution: use `const` in the formal parameter declaration

# Implementation of Member Functions

- Must write the code for functions defined as function prototypes
- Prototypes are left in the class to keep the class smaller and to hide the implementation
- To access identifiers local to the class, use the scope resolution operator ::

# Implementation of Member Functions (cont'd.)

myClock



(a) myClock before executing  
`myClock.setTime(3, 48, 52);`

myClock



(b) myClock after executing  
`myClock.setTime(3, 48, 52);`

**FIGURE 10-4** myClock before and after executing the statement `myClock.setTime(3, 48, 52);`

# Implementation of Member Functions (cont'd.)



FIGURE 10-5 Objects `myClock` and `yourClock`

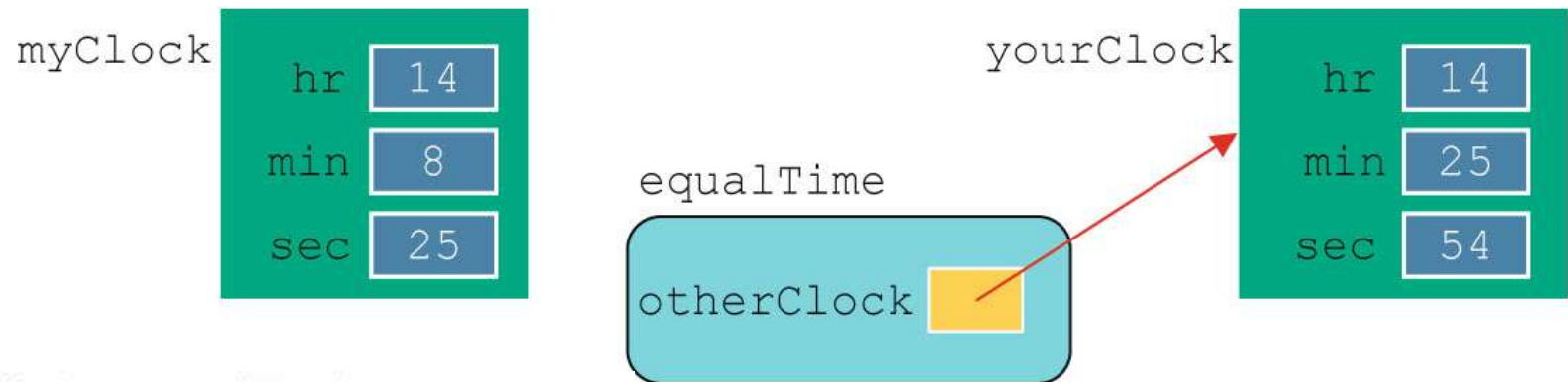


FIGURE 10-6 Object `myClock` and parameter `otherClock`



# Implementation of Member Functions (cont'd.)

- Once a `class` is properly defined and implemented, it can be used in a program
  - A program that uses/manipulates objects of a class is called a client of that class
- When you declare objects of the `class` `clockType`, each object has its own copy of the member variables (`hr`, `min`, and `sec`)
  - Called instance variables of the class
    - Every object has its own instance of the data

# Accessor and Mutator Functions

- Accessor function: member function that only accesses the value(s) of member variable(s)
- Mutator function: member function that modifies the value(s) of member variable(s)
- Constant function:
  - Member function that cannot modify member variables
  - Use `const` in function heading

# Order of `public` and `private` Members of a Class

- C++ has no fixed order in which to declare `public` and `private` members
- By default, all members of a class are `private`
- Use the member access specifier `public` to make a member available for `public` access

# Constructors

- Use constructors to guarantee that member variables of a class are initialized
- Two types of constructors:
  - With parameters
  - Without parameters (default constructor)
  - Name of a constructor = name of the class
  - A constructor has no type

# Constructors (cont'd.)

- A class can have more than one constructor
  - Each must have a different formal parameter list
- Constructors execute automatically when a class object enters its scope
- They cannot be called like other functions
- Which constructor executes depends on the types of values passed to the class object when the class object is declared

# Invoking a Constructor

- A constructor is automatically executed when a class variable is declared
- Because a class may have more than one constructor, you can invoke a specific constructor

# Invoking the Default Constructor

- To invoke the default constructor:

```
className classObjectName;
```

- Example:

```
clockType yourClock;
```

# Invoking a Constructor with Parameters

- Syntax:

```
className classObjectName(argument1, argument2, ...);
```

- Number and type of arguments should match the formal parameters (in the order given) of one of the constructors
  - Otherwise, C++ uses type conversion and looks for the best match
  - Any ambiguity causes a compile-time error



# Constructors and Default Parameters

- A constructor can have default parameters
  - Rules for declaring formal parameters are the same as for declaring default formal parameters in a function
  - Actual parameters are passed according to same rules for functions
- Default constructor: a constructor with no parameters or with all default parameters

# Classes and Constructors: A Precaution

- If a class has no constructor(s), C++ provides the default constructor
  - However, object declared is still uninitialized
- If a class includes constructor(s) with parameter(s), but not the default constructor
  - C++ does not provide the default constructor

# Arrays of Class Objects (Variables) and Constructors

- If you declare an array of class objects, the class should have the default constructor

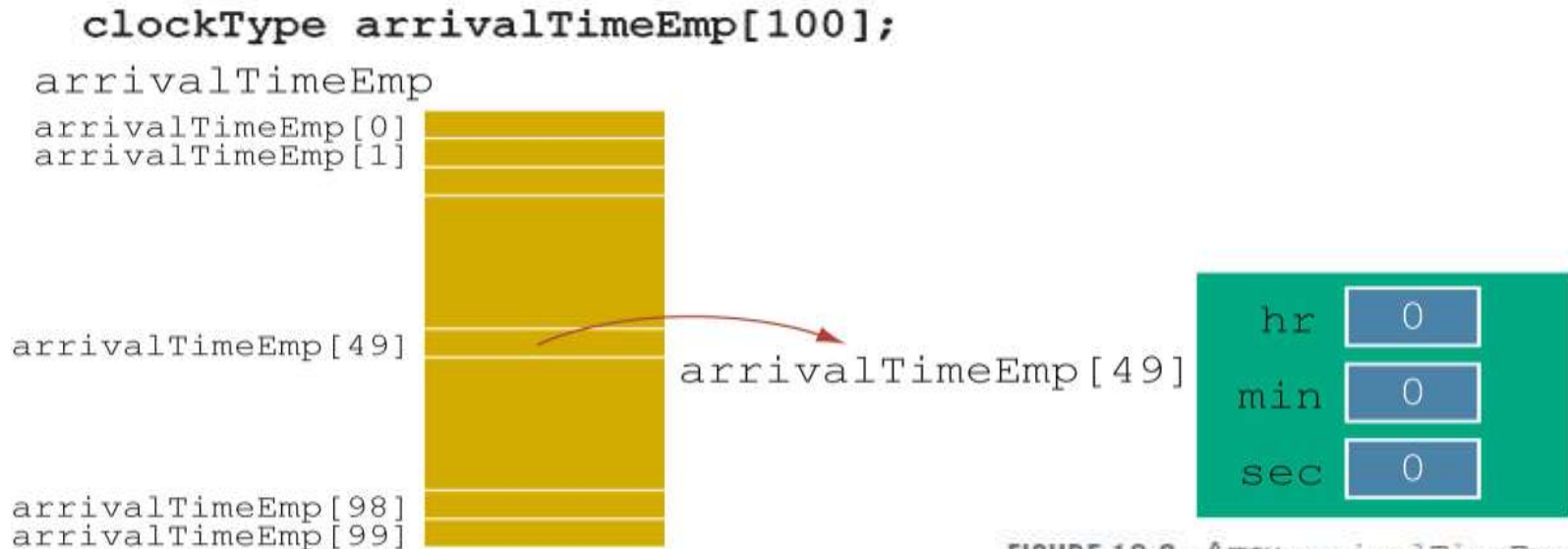


FIGURE 10-8 Array `arrivalTimeEmp`

# Destructors

- Destructors are functions without any type
- The name of a destructor is the character '~' followed by class name
  - For example:  

```
~clockType ();
```
- A class can have only one destructor
  - The destructor has no parameters
- Destructor automatically executes when the class object goes out of scope

# Data Abstract, Classes, and Abstract Data Types

- Abstraction
  - Separating design details from usage
  - Separating the logical properties from the implementation details
- Abstraction can also be applied to data
- Abstract data type (ADT): data type that separates the logical properties from the implementation details

# A struct Versus a class

- By default, members of a struct are public
  - private specifier can be used in a struct to make a member private
- By default, the members of a class are private
- classes and structs have the same capabilities

# A struct Versus a class (cont'd.)

- In C++, the definition of a `struct` was expanded to include member functions, constructors, and destructors
- If all member variables of a `class` are `public` and there are no member functions
  - Use a `struct`

# Information Hiding

- Information hiding: hiding the details of the operations on the data
- Interface (header) file: contains the specification details
  - File extension is `.h`
- Implementation file: contains the implementation details
  - File extension is `.cpp`
- In header file, include function prototypes and comments that briefly describe the functions
  - Specify preconditions and/or postconditions



# Information Hiding (cont'd.)

- Implementation file must include header file via `include` statement
- In `include` statement:
  - User-defined header files are enclosed in double quotes
  - System-provided header files are enclosed between angular brackets

# Information Hiding (cont'd.)

- Precondition: A statement specifying the condition(s) that must be true before the function is called
- Postcondition: A statement specifying what is true after the function call is completed

# Executable Code

- To use an object in a program
  - The program must be able to access the implementation
- Visual C++, Visual Studio .NET, C++ Builder, and CodeWarrior put the editor, compiler, and linker into a package
  - One command (build, rebuild, or make) compiles program and links it with the other necessary files
  - These systems also manage multiple file programs in the form of a project

# Static Members of a Class

- Use the keyword `static` to declare a function or variable of a class as `static`
- A `public static` function or member of a class can be accessed using the class name and the scope resolution operator
- `static` member variables of a class exist even if no object of that `class` type exists

# Static Members of a Class (cont'd.)

- Multiple objects of a class each have their own copy of non-static member variables
- All objects of a class share any static member of the class

# Summary

- Class: collection of a fixed number of components
- Members: components of a class
  - Accessed by name
  - Classified into one of three categories:
    - `private`, `protected`, and `public`
- Class variables are called class objects or, simply, objects

# Summary (cont'd.)

- The only built-in operations on classes are assignment and member selection
- Constructors guarantee that data members are initialized when an object is declared
  - Default constructor has no parameters
- Destructor automatically executes when a class object goes out of scope
  - A class can have only one destructor
  - The destructor has no parameters

# Summary (cont'd.)

- Abstract data type (ADT): data type that separates the logical properties from the implementation details
- A `public static` member, function or data, of a class can be accessed using the class name and the scope resolution operator
- Static data members of a class exist even when no object of the class type exists
- Instance variables: non-static data members



# **UNIT-3**

## **C++ Inheritance**

# C++ Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

# C++ Inheritance

Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes.

The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

# C++ Inheritance

General Format for implementing the concept of Inheritance:

***class derived\_classname: access specifier baseclassname***

For example, if the *base* class is *MyClass* and the derived class is *sample* it is specified as:

**class sample: public MyClass**

The above makes *sample* have access to both *public* and *protected* variables of base class *MyClass*

# C++ Inheritance

## **public, private and protected access specifiers:**

1 If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.

2 Public members and variables are accessible from outside the class.

3 Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

# C++ Inheritance

## Inheritance Example:

```
class MyClass
```

```
{    public:  
    MyClass(void) { x=0; }  
    void f(int n1)  
    { x= n1*5;}  
    void output(void) { cout<<x; }  
    private:  
    int x;  
};
```

# C++ Inheritance

## Inheritance Example:

```
class sample: public MyClass
{ public:
  sample(void) { s1=0; }
  void f1(int n1)
      { s1=n1*10;}
  void output(void)
  { MyClass::output(); cout << s1; }
private:
  int s1;
};
```

# C++ Inheritance

## Inheritance Example:

```
int main(void)
{
    sample s;
    s.f(10);
    s.output();
    s.f1(20);
    s.output();
}
```

The output of the above program is

50

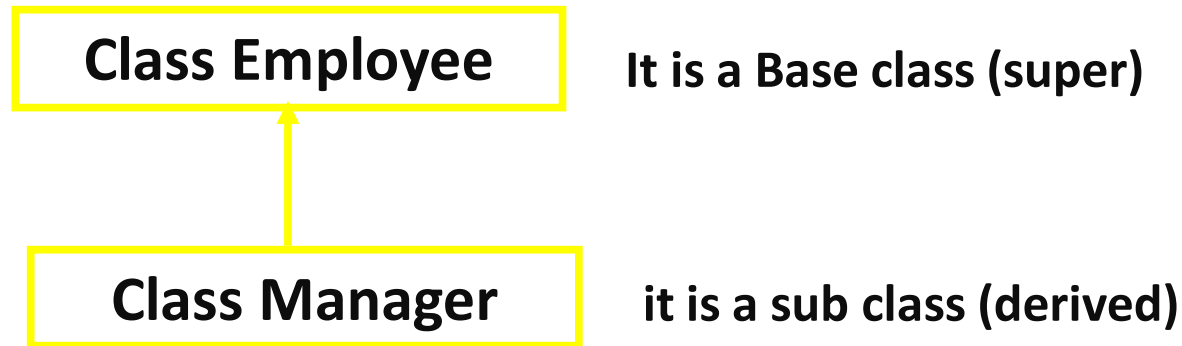
200



# Types of Inheritance

## 1. Single class Inheritance:

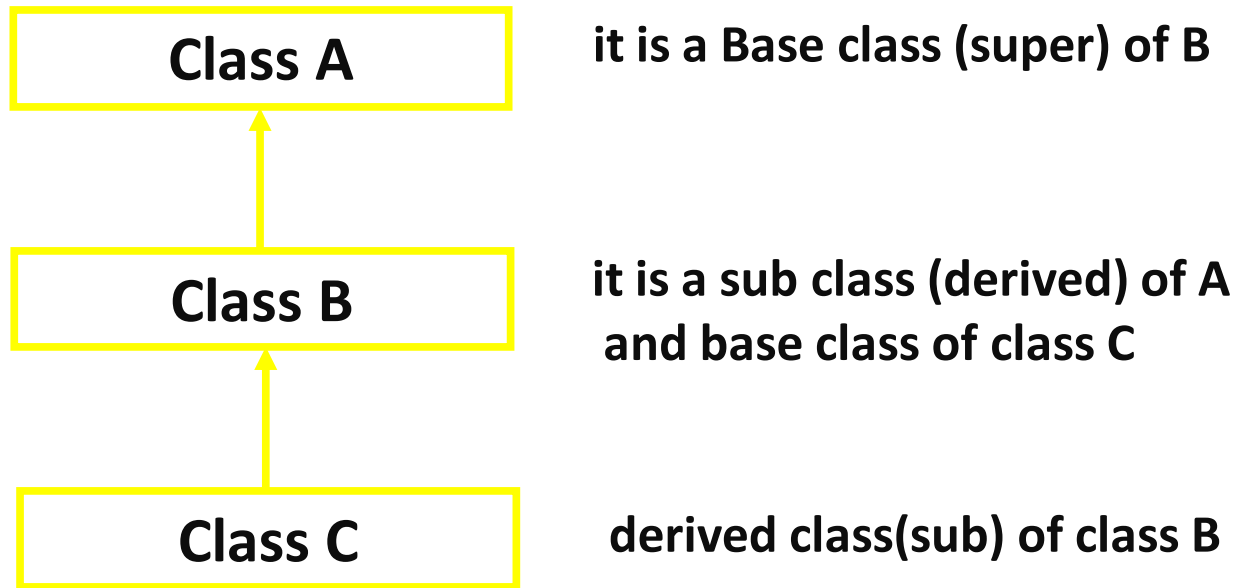
Single inheritance is the one where you have a single base class and a single derived class.



# Types of Inheritance

## 2. Multilevel Inheritance:

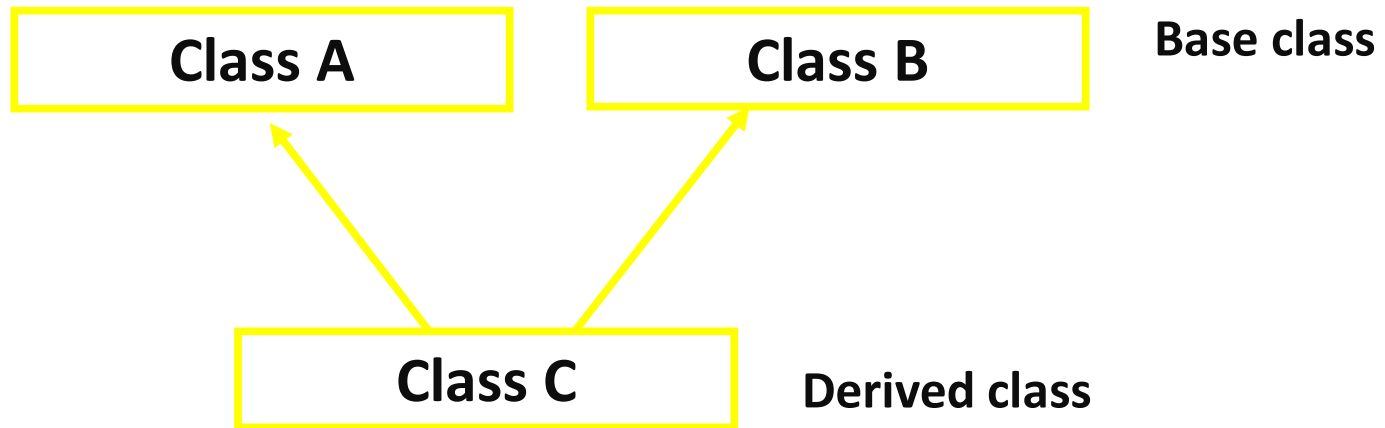
In Multi level inheritance, a class inherits its properties from another derived class.



# Types of Inheritance

## 3. Multiple Inheritances:

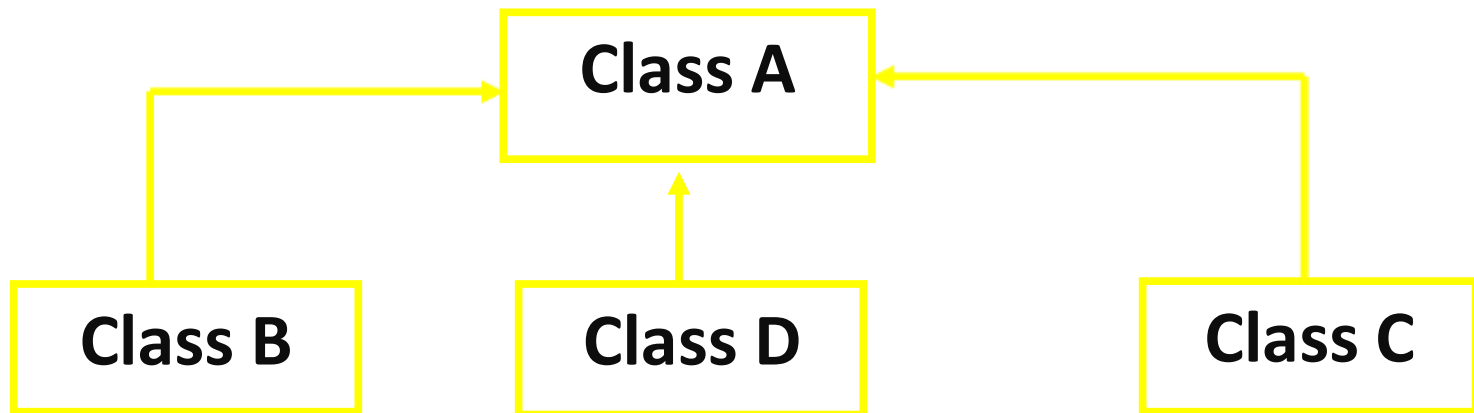
In Multiple inheritances, a derived class inherits from multiple base classes. It has properties of both the base classes.



# Types of Inheritance

## 4. Hierarchical Inheritance:

In hierarchical Inheritance, it's like an inverted tree. So multiple classes inherit from a single base class. It's quite analogous to the File system in a unix based system.



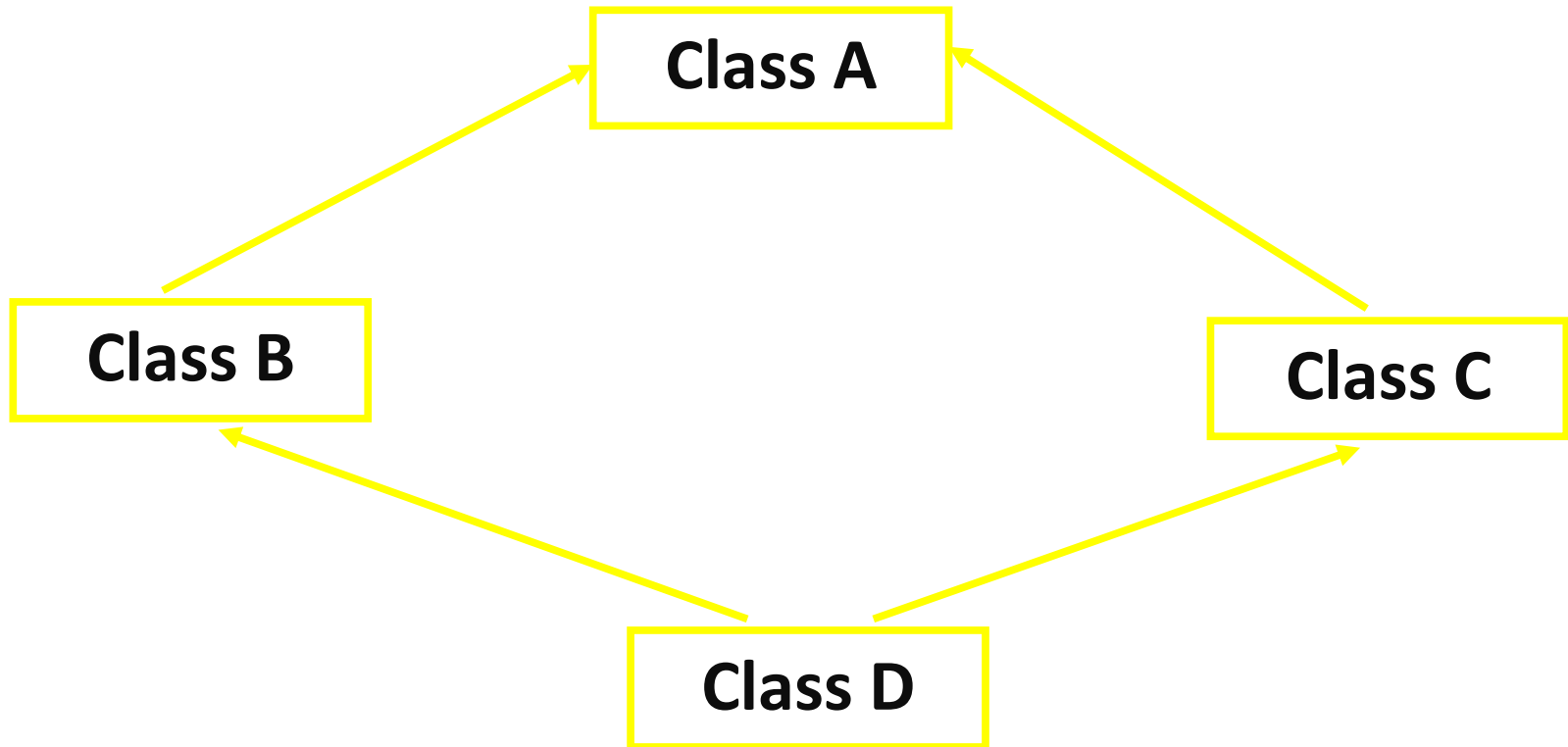
# Types of Inheritance

## 5. Hybrid Inheritance:

- ✓ In this type of inheritance, we can have mixture of number of inheritances but this can generate an error of using same name function from no of classes, which will bother the compiler to how to use the functions.
- ✓ Therefore, it will generate errors in the program. This has known as ambiguity or duplicity.
- ✓ Ambiguity problem can be solved by using **virtual base classes**

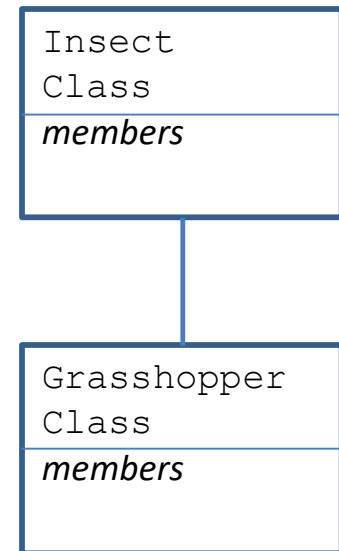
# Types of Inheritance

## 5. Hybrid Inheritance:

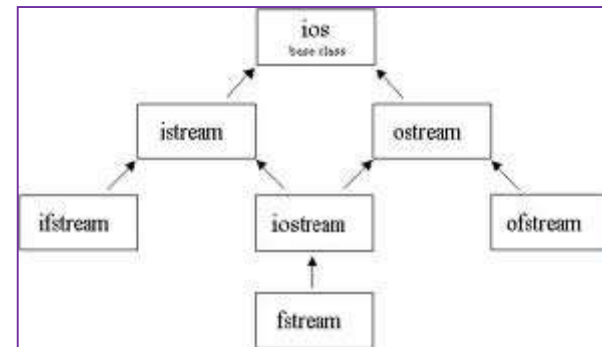
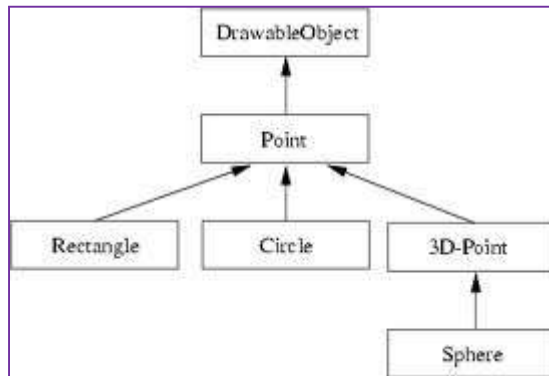
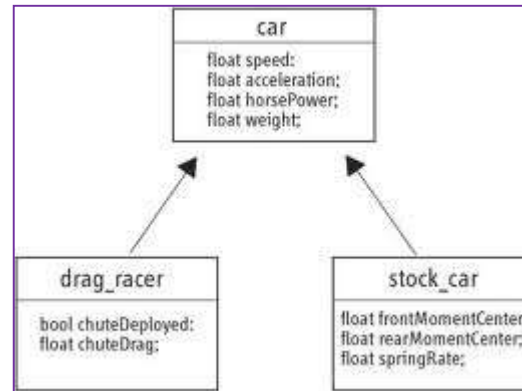
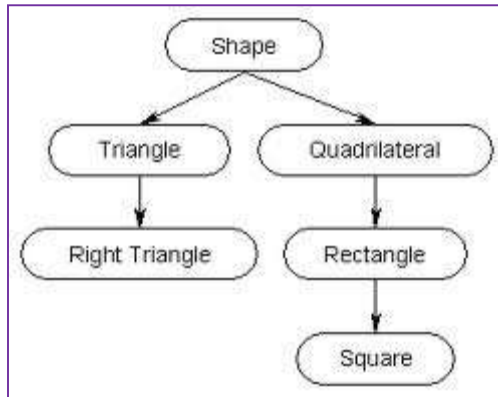


# C++ Inheritance

- Inheritance = the “**Is a**” Relationship
- A poodle **is a** dog
- A car **is a** vehicle
- A tree **is a** plant
- A rectangle **is a** shape
- A football player **is a** an athlete
- Base Class is the *General Class*
- Derived Class is the *Specialized Class*



# C++ Inheritance





# C++ Inheritance

## • Syntax

```
class B {  
    int I;  
public:  
    void Set_I(int X){I=X;}  
    int Get_I() {return I;}  
};
```

**Base Class  
Access  
Specification**

B  
Class //Base  
members

D  
Class //Derived  
members

```
class D : public B {  
    int J;  
public:  
    void Set_J(int X)  
        {J = X;}  
    int Mul()  
        {return J * Get_I();}  
        // J * I → Compile error!  
};
```

```
int main() {  
    D ob;  
    ob.Set_J(10);  
    ob.Set_I(4);  
    // ob.I = 8; Compile error!  
    cout << ob.Mul() << endl;  
    return 0;  
} // end main
```

### Access Specification: **Public**

- Public members of Base are public members of Derived
- Private members of Base remain private members, but are inherited by the Derived class.

i.e. "They are invisible to the Derived class"

# C++ Inheritance

- A base class is not exclusively “owned” by a derived class. A base class can be inherited by any number of different classes.
- There may be times when you want to keep a member of a base class private but still permit a derived class access to it.  
SOLUTION: Designate the data as **protected**.

# C++ Inheritance

*Private members of the base class are always private to the derived class regardless of the access specifier.*

- Protected Data Inherited as Public

```
class Base {  
    protected:  
    int a, b;  
    public:  
    void Setab(int n, int m)  
        { a = n; b = m; }  
};
```

```
class Derived: public Base {  
    int c;  
    public:  
    void Setc(int x) { c = x; }  
    void Showabc() {  
        cout << a << " " << b << " " << c << endl;  
    }  
};
```

```
int main() {  
    Derived ob;  
  
    ob.Setab(1,2);  
    ob.Setc(3);  
    ob.Showabc();  
    //ob.a = 5 NO! NO!  
  
    return 0;  
} // end main
```

# C++ Inheritance

- **Public Access Specifier**

- *Private members of Base remain private members and are inaccessible to the derived class.*
- *Public members of Base are public members of Derived*

*BUT*

- *Protected members of a base class are accessible to members of any class derived from that base.*  
*Protected members, like private members, are not accessible outside the base or derived classes.*

# C++ Inheritance

*Private members of the base class are always private to the derived class regardless of the access specifier*

- But when a base class is inherited as **protected**, **public and protected** members of the base class become protected members of the derived class.

```
class Base {  
    protected:  
    int a, b;  
    public:  
    void Setab(int n, int m)  
        { a = n; b = m; }  
};
```

```
int main() {  
    Derived ob;  
  
    //ob.Setab(1,2); ERROR  
    //ob.a = 5;      NO! NO!  
  
    ob.Setc(3);  
    ob.Showabc();  
  
    return 0;  
} // end main
```

```
class Derived: protected Base {  
    int c;  
    public:  
    void Setc(int x) { c = x; }  
    void Showabc() {  
        cout << a << " " << b << " " << c << endl;  
    }  
};
```

*Private members of the base class are always private to the derived class regardless of the access specifier*

# C++ Inheritance

- **Protected Access Specifier**

- Private members of the base class are inaccessible to the derived class.
- Public members of the base class become protected members of the derived class.
- Protected members of the base class become protected members of the derived class.

i.e. only the public members of the derived class are accessible by the user application.

# C++ Inheritance

- **Constructors & Destructors**
  - When a base class and a derived class both have constructor and destructor functions
    - Constructor functions are executed in order of derivation – base class before derived class.
    - Destructor functions are executed in reverse order – the derived class's destructor is executed before the base class's destructor.
  - A derived class does not inherit the constructors of its base class.

# C++ Inheritance

```
class Base {
    public:
        Base() { cout << "Constructor Base Class\n";}
        ~Base() {cout << "Destructing Base Class\n";}
};
class Derived : public Base {
    public:
        Derived() { cout << Constructor Derived Class\n";}
        ~Derived(){ cout << Destructing Derived Class\n";}
};
```

```
int main() {
    Derived ob;
    return 0;
}
```

```
---- OUTPUT ----
Constructor Base Class
Constructor Derived Class
Destructing Derived Class
Destructing Base Class
```



# C++ Inheritance

- **Passing an argument to a derived class's constructor**

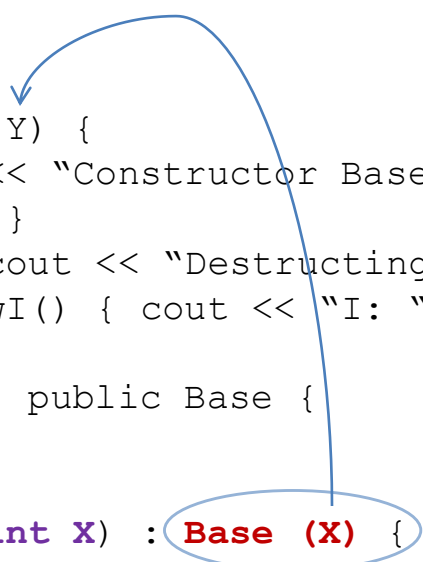
```
Class Base {
    public:
        Base() {cout << "Constructor Base Class\n";}
        ~Base() {cout << "Destructing Base Class\n";}
};
Class Derived : public Base {
    int J;
    public:
        Derived(int X) {
            cout << Constructor Derived Class\n";
            J = X;
        }
        ~Derived() { cout << Destructing Derived Class\n";}
        void ShowJ() { cout << "J: " << J << "\n"; }
};
```

```
int main() {
    Derived Ob(10);
    Ob.ShowJ();
    return 0;
} // end main
```

# C++ Inheritance

- Arguments to both Derived and Base Constructors

```
Class Base {
    int I;
public:
    Base(int Y) {
        cout << "Constructor Base Class\n";
        I = Y;}
    ~Base() {cout << "Destructing Base Class\n";}
    void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
    int J;
public:
    Derived(int X) : Base (X) {
        cout << Constructor Derived Class\n";
        J = X;
    }
    ~Derived() { cout << Destructing Derived Class\n";}
    void ShowJ() { cout << << "J:" << J << "\n"; }
};
```



```
int main() {
    Derived Ob(10);

    Ob.ShowI();
    Ob.ShowJ();
    return 0;
} // end main
```

# C++ Inheritance

- **Different arguments to the Base – All arguments to the Derived.**

```
Class Base {
    int I;
public:
    Base(int Y) {
        cout << "Constructor Base Class\n";
        I = Y;}
    ~Base(){cout << "Destructing Base Class\n";}
    void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
    int J;
public:
    Derived(int X, int Y) : Base (Y) {
        cout << Constructor Derived Class\n";
        J = X;
    }
    ~Derived(){ cout << Destructing Derived Class\n";}
    void ShowJ() { cout << << "J:" << J << "\n"; }
};
```

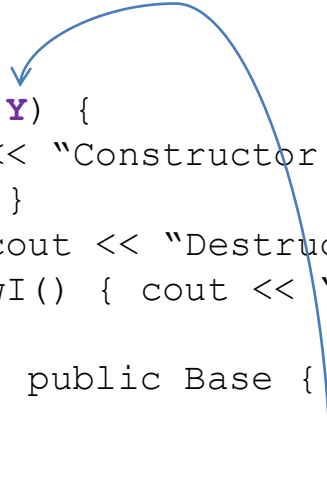
```
int main() {
    Derived Ob(5,8);

    Ob.ShowI();
    Ob.ShowJ();
    return 0;
} // end main
```

# C++ Inheritance

- OK – If Only Base has Argument

```
Class Base {
    int I;
    public:
        Base(int Y) {
            cout << "Constructor Base Class\n";
            I = Y; }
        ~Base() {cout << "Destructing Base Class\n";}
        void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
    int J;
    public:
        Derived(int X) : Base (X) {
            cout << Constructor Derived Class\n";
            J = 0;           // X not used here
        }
        ~Derived() { cout << Destructing Derived Class\n";}
        void ShowJ() { cout << "J:" << J << "\n"; }
};
```



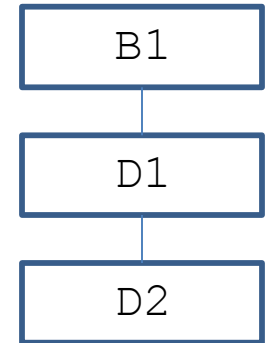
```
int main() {
    Derived Ob(10);

    Ob.ShowI();
    Ob.ShowJ();
    return 0;
} // end main
```

# C++ Inheritance

- **Multiple Inheritance – Inheriting more than one base class**
  1. Derived class can be used as a base class for another derived class  
(*multilevel class hierarchy*)
  2. A derived class can directly inherit more than one base class. 2 or more base classes are combined to help create the derived class

# C++ Inheritance



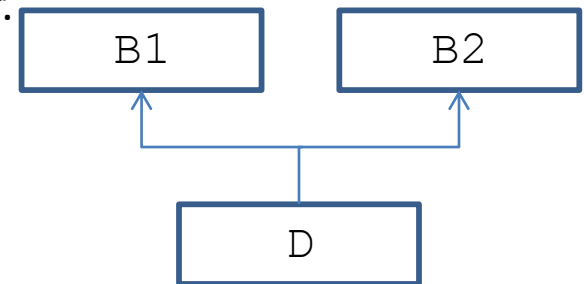
- **Multiple Inheritance**

1. **Multilevel Class Hierarchy**

- Constructor functions of all classes are called in order of derivation: B1, D1, D2
- Destructor functions are called in reverse order

2. **When a derived class directly inherits multiple base classes...**

- Access\_Specifiers { public, private, protected } can be different
- Constructors are executed in the order left to right, that the base classes are specified.
- Destructors are executed in the opposite order.



```
class Derived_Class_Name: access Base1,  
                        access Base2,... access BaseN  
{  
    //.. body of class  
} end Derived_Class_Name
```

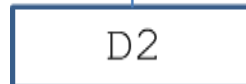
# C++ Inheritance

- Derived class inherits a class derived from another class.

```
class B1 {  
    int A;  
    public:  
        B1(int Z) { A = Z; }  
        int GetA() { return A; }  
};
```

```
class D1 : public B1 {  
    int B;  
    public:  
        D1(int Y, int Z) : B1 (Z) { B = Y; }  
        void GetB() { return B; }  
};
```

```
Class D2 : public D1 {  
    int C;  
    public:  
        D2 (int X, int Y, int Z) : D1 ( Y, Z) { C = X; }  
        void ShowAll () {  
            cout << GetA() << " " << GetB() << " " << C << endl; }  
};
```



```
int main() {  
    D2 Ob(5,7,9);  
  
    Ob.ShowAll();  
  
    // GetA & GetB are still public here  
    cout << Ob.GetA() << " "  
         << Ob.GetB() << endl;  
  
    return 0;  
} // end main
```

*Because bases are inherited as public,  
D2 has access to public elements of both B1 and D1*

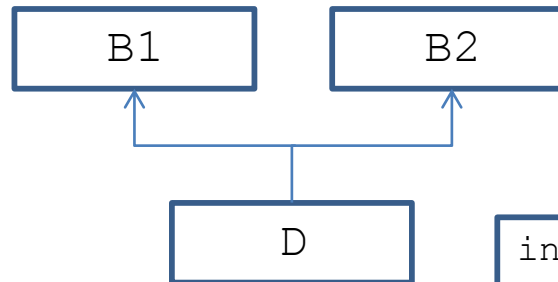
# C++ Inheritance

## Derived Class Inherits Two Base Classes

```
class B1 {  
    int A;  
    public:  
        B1(int Z) { A = Z; }  
        int GetA() { return A; }  
};
```

```
class B2 {  
    int B;  
    public:  
        B2 (int Y) { B = Y; }  
        void GetB() { return B; }  
};
```

```
class D : public B1, public B2 {  
    int C;  
    public:  
        D (int X, int Y, int Z) : B1(Z), B2 (Y) { C = X; }  
        void ShowAll () {  
            cout << GetA() << " " << GetB() << " " << C << endl; }  
};
```



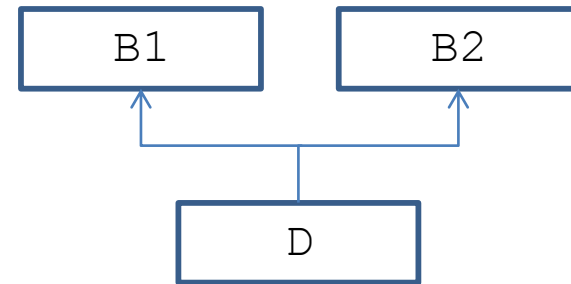
```
int main() {  
    D Ob(5,7,9);  
  
    Ob.ShowAll();  
  
    return 0;  
} // end main
```



# C++ Inheritance

- Inheritance Multiple Base Classes  
(constructor and destructor)

```
class B1 {  
public:  
    B1() {cout << "Constructing B1\n"; }  
    ~B1() {cout << "Destructing B1\n"; }  
};  
class B2 {  
public:  
    B2() {cout << "Constructing B2\n"; }  
    ~B2() {cout << "Destructing B2\n"; }  
};  
class D : public B1, public B2 {  
public:  
    D() {cout << "Constructing D\n"; }  
    ~D() {cout << "Destructing D\n"; }  
};
```



```
int main () {  
    D ob;  
    return 0;  
} // end main
```

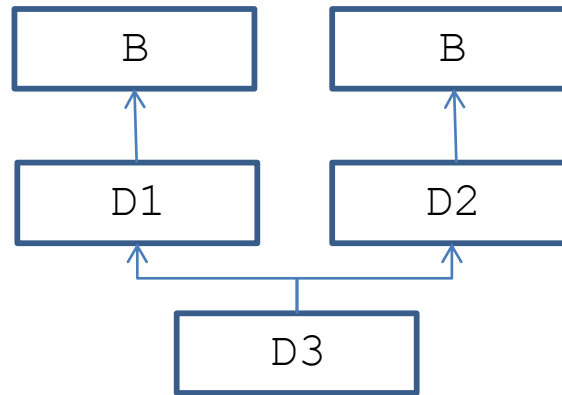
```
----OUTPUT----  
Constructing B1  
Constructing B2  
Constructing D  
Destructing D  
Destructing B2  
Destructing B1
```

# C++ Inheritance

- **Virtual Base Class**

- **Problem:**

- The Base B is inherited twice by D3.



- There is ambiguity!

- Solution: mechanism by which only one copy of B will be included in D3.

# C++ Inheritance

```
class B {
    public:
        int I;
};
class D1 : virtual public B {
    public:
        int J;
};
class D2 : virtual public B {
    public:
        int K;
};
class D3 : public D1, public D2 {
    public:
        int product {return I * J * K; }
};
```

```
int main() {
    D3 ob;

    ob.I = 15; //must be virtual
              // else compile
              // time error

    ob.J = 21;
    ob.K = 26;

    cout << "Product: "
          << ob.product() << endl;
    return 0;
} // end main
```

# C++ Inheritance

- A Derived class does not inherit the constructors of its base class.
- Good Advice: You can and should include a call to one of the base class constructors when you define a constructor for a derived class.
- If you do not include a call to a base class constructor, then the default (zero argument) constructor of the base class is called automatically.
- If there is no default constructor for the base class, an error occurs.

# C++ Inheritance

- If the programmer does not define a *copy constructor* in a derived class (or any class), C++ will auto-generate a copy constructor for you. (Bit-wise copy)
- Overloaded assignment operators are not inherited, but can be used.
- When the destructor for the derived class is invoked, it auto-invokes the destructor of the base class. No need to explicitly call the base class destructor.

# C++ Inheritance

- A derived class inherits all the member functions (and member variables) that belong to the base class – except for the constructor.
- If a derived class requires a different implementation for an inherited member function, the function may be redefined in the derived class. (not the same *overloading*)
  - List its declaration in the definition of the derived class (even though it is the same as the base class).
  - Redefined function will have the same number and types of parameters. I.e. signature is the same.
  - Ok to use both (must use the base class qualifier to distinguish between the 2)

# C++ Inheritance

- **Virtual Functions**

- Background:

- A pointer declared as a pointer to a base class can also be used to point to any class derived from that base.
    - We can use a base pointer to point to a derived object, but you can access only those members of the derived object that were inherited from the base. The base pointer has knowledge only of the base class; it knows nothing about the members added by the derived class.
    - A pointer of the derived type cannot (should not) be used to access an object of the base class.

# C++ Inheritance

- **Virtual Functions-  
Background**

```
class Base {
    int X;
public:
    void SetX(int I) { X = I;}
    int  GetX()      { return X;}
};
class Derived : public Base {
    int Y;
public:
    void SetY(int I) { Y = I;}
    int  GetY()      { return Y;}
};
```

```
int main() {
    Base      *ptr;
    Base      BaseOb;
    Derived   DerivedOb;

    ptr = &BaseOb;
    ptr→SetX(15);
    cout <<"Base X: "
         << ptr→GetX() << endl;

    ptr = &DerivedOb;
    ptr→SetX(29);

    DerivedOb.SetY(42); // cannot use ptr
    cout << "Derived Object X: "
         << ptr→GetX()      << endl;
    cout << "Derived Object Y: "
         << DerivedOb.GetY() << endl;

    return 0;
} // end main
```



# C++ Inheritance

- **Virtual Functions**

- When the programmer codes “virtual” for a function, the programmer is saying, “I do not know how this function is implemented”.
- Technique of waiting *until runtime* to determine the implementation of a procedure is called **late binding** or **dynamic binding**.
- A virtual function is a member function that is declared within a base class and redefined by a derived class.
- Demonstrates “One interface, multiple methods” philosophy that is polymorphism.
- “Run-time polymorphism”- when a virtual function is called through a pointer.
- When a virtual function is redefined by a derived class, the keyword `virtual` is not needed.
- “A base pointer points to a derived object that contains a virtual function and that virtual function is called through that pointer, C++ determines which version of that function will be executed based upon the type of object being pointed to by the pointer.” Schildt

# C++ Inheritance

- **Virtual Functions**

- Exact same **prototype** (*Override* not Overload)  
Signature + return type
- Can only be class members
- Destructors can be virtual; constructors cannot.
- Done at runtime!
- *Late Binding*: refers to events that must occur at run time.
- *Early Binding*: refers to those events that can be known at compile time.

# C++ Inheritance

## • Virtual Functions

*Polymorphic class*  
contains a virtual  
function.

```
class Base {
public:
    int I;
    base(int X) { I = X;}
    virtual void func() {
        cout << "Using Base version of func(): ";
        cout << I << endl;
    }
};

class D1 : public Base {
public:
    D1(int X) : base(X) {}
    void func() {
        cout << "Using D1's version of func(): ";
        cout << I*I << endl;
    }
};

class D2 : public Base {
public:
    D2(int X) : base(X) {}
    void func() {
        cout << "Using D2's version of func(): ";
        cout << I+I << endl;
    }
};
```

```
int main() {
    Base *ptr;
    Base BaseOb(10);
    D1 D1Ob(10);
    D2 D2Ob(10);

    ptr = &BaseOb;
    ptr->func(); // use Base's func()

    ptr = &D1Ob;
    ptr->func(); // use D1's func()

    ptr = &D2Ob;
    ptr->func(); // use D2's func()

    return 0;
}
```

-----OUTPUT-----

```
Using Base version of func(): 10
Using D1's version of func(): 100
Using D2's version of func(): 20
```

*If the derived class does not override a virtual function,  
the function defined within its base class is used.*

# C++ Inheritance

```
class Area {
    double dim1, dim2;
public :
    void SetArea(double d1, double d2) {
        dim1 = d1;
        dim2 = d2;
    }
    void GetDim (double &d1, double &d2) {
        d1 = dim1;
        d2 = dim2;
    }
    {
        virtual double GetArea() {
            cout << "DUMMY DUMMY OVERRIDE function";
            return 0.0;
        }
    }
};

class Rectangle : public Area {
public :
    double GetArea() {
        double temp1, temp2
        GetDim (temp1, temp2);
        return temp1 * temp2;
    }
};

class Triangle : public Area {
public :
    double GetArea() {
        double temp1, temp2
        GetDim (temp1, temp2);
        return 0.5 temp1 * temp2;
    }
};
```

```
int main () {
    Area *ptr;
    Rectangle R;
    Triangle T;

    R.SetArea(3.3, 4.5);
    T.SetArea(4.0, 5.0);

    ptr = &R;
    cout << "RECTANGLE_AREA: "
         << ptr->GetArea() << endl;

    ptr = &T;
    cout << "TRIANGLE_AREA: "
         << ptr->GetArea() << endl;

    return 0;
} // end main
```

*When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class **MUST** override this function. C++ supports **pure virtual functions** to do this.*

```
Virtual double GetArea() = 0; // pure virtual
```

# C++ Inheritance

- **Virtual Functions**

- When a class contains at least one *pure* virtual function, it is referred to as an *abstract class*.
  - An abstract class contains at least one function for which no body exists, so an abstract class exists mainly to be inherited.
  - Abstract classes do not stand alone.
  - If Class B has a virtual function called f(), and D1 inherits B and D2 inherits D1, both D1 and D2 can override f() relative to their respective classes.
-

# UNIT-4

## C++ Input/output

# Introduction

- Many C++ I/O features are object-oriented
  - Use references, function overloading and operator overloading
- C++ uses type safe I/O
  - Each I/O operation is automatically performed in a manner sensitive to the data type
- Extensibility
  - Users may specify I/O of user-defined types as well as standard types

# Streams

- Stream
  - A transfer of information in the form of a sequence of bytes
- I/O Operations:
  - Input: A stream that flows from an input device ( i.e.: keyboard, disk drive, network connection) to main memory
  - Output: A stream that flows from main memory to an output device ( i.e.: screen, printer, disk drive, network connection)



# Streams

- I/O operations are a bottleneck
  - The time for a stream to flow is many times larger than the time it takes the CPU to process the data in the stream
- Low-level I/O
  - Unformatted
  - Individual byte unit of interest
  - High speed, high volume, but inconvenient for people
- High-level I/O
  - Formatted
  - Bytes grouped into meaningful units: integers, characters, etc.
  - Good for all I/O except high-volume file processing

# iostream Library Header Files

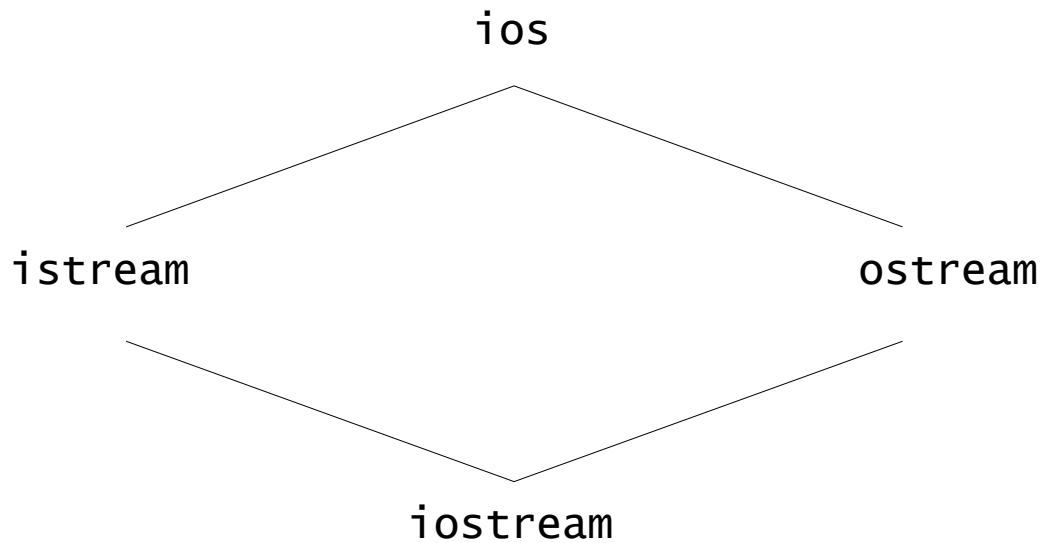
- `iostream` library:
  - `<iostream.h>`: Contains `cin`, `cout`, `cerr` and `clog` objects
  - `<iomanip.h>`: Contains *parameterized stream manipulators*

# Stream Input/Output Classes and Objects

- `ios`:
  - `istream` and `ostream` inherit from `ios`
    - `iostream` inherits from `istream` and `ostream`.
- `<<` (left-shift operator)
  - Overloaded as *stream insertion operator*
- `>>` (right-shift operator)
  - Overloaded as *stream extraction operator*
  - Both operators used with `cin`, `cout`, `cerr`, `clog`, and with user-defined stream objects

# Stream Input/Output Classes and Objects

Figure 21.1 Portion of the stream I/O class hierarchy.



# Stream Input/Output Classes and Objects

- `istream`: input streams

```
cin >> grade;
```

- `cin` knows what type of data is to be assigned to `grade` (based on the type of `grade`).

- `ostream`: output streams

```
– cout << grade;
```

- `cout` knows the type of data to output

```
– cerr << errorMessage;
```

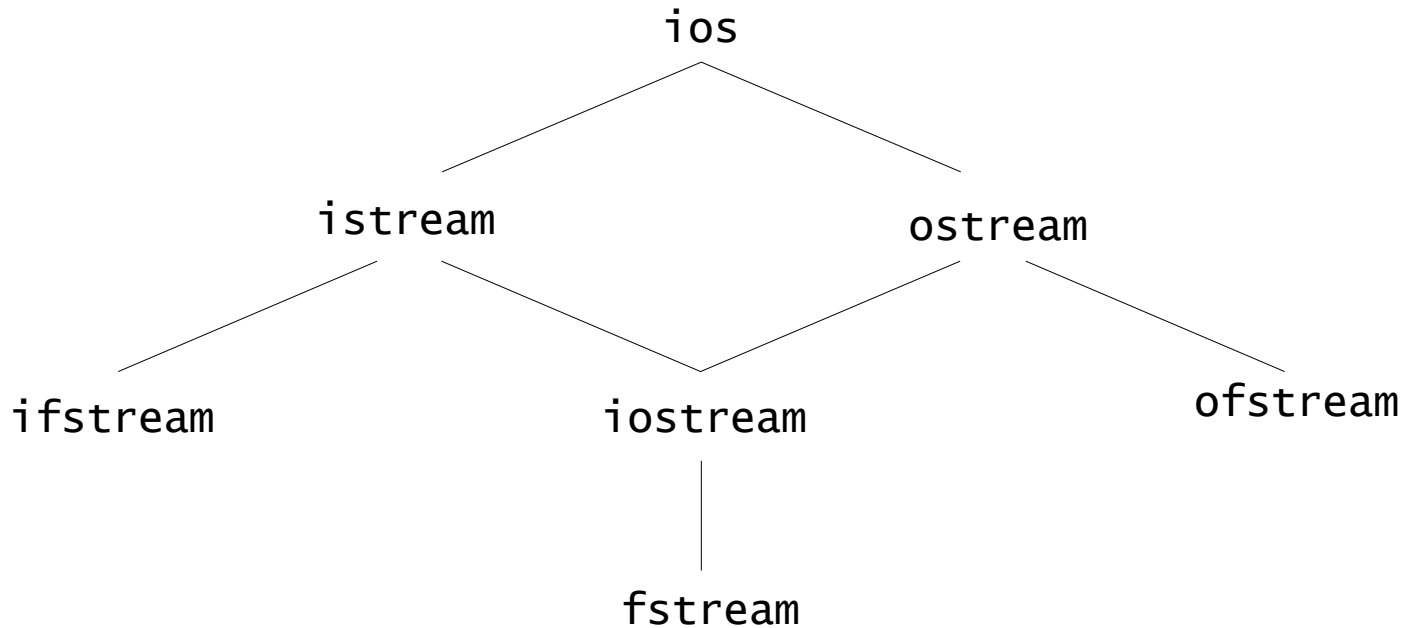
- Unbuffered - prints `errorMessage` immediately.

```
– clog << errorMessage;
```

- Buffered - prints `errorMessage` as soon as output buffer is full or flushed

# Stream Input/Output Classes and Objects

Figure 21.2 Portion of stream-I/O class hierarchy with key file-processing classes.



# Stream Output

- `ostream`: performs formatted and unformatted output
  - Uses `put` for characters and `write` for unformatted output
  - Output of integers in decimal, octal and hexadecimal
  - Varying precision for floating points
  - Formatted text outputs

# Stream-Insertion Operator

- `<<` is overloaded to output built-in types
  - Can also be used to output user-defined types
  - `cout << '\n';`
    - Prints newline character
  - `cout << endl;`
    - `endl` is a stream manipulator that issues a newline character and flushes the output buffer
  - `cout << flush;`
    - `flush` flushes the output buffer



```
1 // Fig. 21.3: fig21_03.cpp
2 // Outputting a string using stream insertion.
3 #include <iostream>
4
5 using std::cout;
6
7 int main()
8 {
9     cout << "welcome to C++!\n";
10
11     return 0;
12 } // end function main
```



Outline

fig21\_03.cpp

Program Output

Welcome to C++!

```
1 // Fig. 21.4: fig21_04.cpp
2 // Outputting a string using two stream insertions.
3 #include <iostream>
4
5 using std::cout;
6
7 int main()
8 {
9     cout << "Welcome to ";
10    cout << "C++!\n";
11
12    return 0;
13 } // end function main
```



Outline



fig21\_04.cpp

Welcome to C++!

Program Output

```
1 // Fig. 21.5: fig21_05.cpp
2 // Using the endl stream manipulator.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     cout << "welcome to ";
11     cout << "C++!";
12     cout << endl; // end line stream manipulator
13
14     return 0;
15 } // end function main
```



Outline



fig21\_05.cpp

```
Welcome to C++!
```

Program Output

```
1 // Fig. 21.6: fig21_06.cpp
2 // Outputting expression values.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     cout << "47 plus 53 is ";
11
12     // parentheses not needed; used for clarity
13     cout << ( 47 + 53 ); // expression
14     cout << endl;
15
16     return 0;
17 } // end function main
```



Outline



fig21\_06.cpp

```
47 plus 53 is 100
```

Program Output

# Cascading Stream-Insertion/Extraction Operators

- `<<` : Associates from left to right, and returns a reference to its left-operand object (i.e. `cout`).
  - This enables cascading  
`cout << "How" << " are" << " you?";`

Make sure to use parenthesis:

```
cout << "1 + 2 = " << (1 + 2);
```

NOT

```
cout << "1 + 2 = " << 1 + 2;
```

```
1 // Fig. 21.7: fig21_07.cpp
2 // Cascading the overloaded << operator.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     cout << "47 plus 53 is " << ( 47 + 53 ) << endl;
11
12     return 0;
13 } // end function main
```



Outline



fig21\_07.cpp

```
47 plus 53 is 100
```

Program Output

## Output of char \* Variables

- << will output a variable of type char \* as a string
- To output the address of the first character of that string, cast the variable as type void \*

```
1 // Fig. 21.8: fig21_08.cpp
2 // Printing the address stored in a char* variable
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     const char *string = "test";
11
12     cout << "value of string is: " << string
13         << "\nvalue of static_cast< void * >( string ) is: "
14         << static_cast< void * >( string ) << endl;
15     return 0;
16 } // end function main
```



Outline

fig21\_08.cpp

Program Output

```
Value of string is: test
Value of static_cast< void *>( string ) is:
0046C070
```



# Character Output with Member Function `put`; Cascading `puts`

- `put` member function
  - Outputs one character to specified stream  
`cout.put( 'A' );`
  - Returns a reference to the object that called it, so may be cascaded  
`cout.put( 'A' ).put( '\n' );`
  - May be called with an ASCII-valued expression  
`cout.put( 65 );`
    - Outputs A

# Stream Input

- `>>` (stream-extraction)
  - Used to perform stream input
  - Normally ignores whitespaces (spaces, tabs, newlines)
  - Returns zero (`false`) when EOF is encountered, otherwise returns reference to the object from which it was invoked (i.e. `cin`)
- `>>` controls the state bits of the stream
  - `failbit` set if wrong type of data input
  - `badbit` set if the operation fails

# Stream-Extraction Operator

- `>>` and `<<` have relatively high precedence
  - Conditional and arithmetic expressions must be contained in parentheses

- Popular way to perform loops

```
while (cin >> grade)
```

- Extraction returns 0 (false) when EOF encountered, and loop ends

```
1 // Fig. 21.9: fig21_09.cpp
2 // Calculating the sum of two integers input from the keyboard
3 // with cin and the stream-extraction operator.
4 #include <iostream>
5
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 int main()
11 {
12     int x, y;
13
14     cout << "Enter two integers: ";
15     cin >> x >> y;
16     cout << "Sum of " << x << " and " << y << " is: "
17         << ( x + y ) << endl;
18
19     return 0;
20 } // end function main
```



Outline



fig21\_09.cpp

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

Program Output

```
1 // Fig. 21.10: fig21_10.cpp
2 // Avoiding a precedence problem between the stream-insertion
3 // operator and the conditional operator.
4 // Need parentheses around the conditional expression.
5 #include <iostream>
6
7 using std::cout;
8 using std::cin;
9 using std::endl;
10
11 int main()
12 {
13     int x, y;
14
15     cout << "Enter two integers: ";
16     cin >> x >> y;
17     cout << x << ( x == y ? " is" : " is not" )
18         << " equal to " << y << endl;
19
20     return 0;
21 } // end function main
```



Outline



fig21\_10.cpp

```
Enter two integers: 7 5
7 is not equal to 5
Enter two integers: 8 8
8 is equal to 8
```

Program Output



## Outline



fig21\_11.cpp

```
1 // Fig. 21.11: fig21_11.cpp
2 // Stream-extraction operator returning false on end-of-file.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 {
11     int grade, highestGrade = -1;
12
13     cout << "Enter grade (enter end-of-file to end): ";
14     while ( cin >> grade ) {
15         if ( grade > highestGrade )
16             highestGrade = grade;
17
18         cout << "Enter grade (enter end-of-file to end): ";
19     } // end while
20
21     cout << "\n\nHighest grade is: " << highestGrade << endl;
22     return 0;
23 } // end function main
```

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z
Highest grade is: 99
```



Outline



**Program Output**

## 21.4.2 get and getLine Member Functions

- `cin.eof()`: returns true if end-of-file has occurred on `cin`
- `cin.get()`: inputs a character from stream (even white spaces) and returns it
- `cin.get( c )`: inputs a character from stream and stores it in `c`



```
1 // Fig. 21.12: fig21_12.cpp
2 // Using member functions get, put and eof.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 {
11     char c;
12
13     cout << "Before input, cin.eof() is " << cin.eof()
14         << "\nEnter a sentence followed by end-of-file:\n";
15
16     while ( ( c = cin.get() ) != EOF )
17         cout.put( c );
18
19     cout << "\nEOF in this system is: " << c;
20     cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
21     return 0;
22 } // end function main
```



Outline



fig21\_12.cpp

# Stream Manipulators

- Stream manipulator capabilities
  - Setting field widths
  - Setting precisions
  - Setting and unsetting format flags
  - Setting the fill character in fields
  - Flushing streams
  - Inserting a newline in the output stream and flushing the stream
  - Inserting a null character in the output stream and skipping whitespace in the input stream

# Integral Stream Base: dec, oct, hex and setbase

- oct, hex or dec:

- Change base of which integers are interpreted from the stream.

Example:

```
int n = 15;  
cout << hex << n;
```

- Prints "F"

- setbase:

- Changes base of integer output
- Load <iomanip>
- Accepts an integer argument (10, 8, or 16)

```
cout << setbase(16) << n;
```

- Parameterized stream manipulator - takes an argument

```
1 // Fig. 21.16: fig21_16.cpp
2 // Using hex, oct, dec and setbase stream manipulators.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::hex;
12 using std::dec;
13 using std::oct;
14 using std::setbase;
15
16 int main()
17 {
18     int n;
19
20     cout << "Enter a decimal number: ";
21     cin >> n;
22
```



## Outline



fig21\_16.cpp (Part 1  
of 2)

```
23     cout << n << " in hexadecimal is: "  
24         << hex << n << '\n'  
25         << dec << n << " in octal is: "  
26         << oct << n << '\n'  
27         << setbase( 10 ) << n << " in decimal is: "  
28         << n << endl;  
29  
30     return 0;  
31 } // end function main
```



Outline



fig21\_16.cpp (Part 2  
of 2)

```
Enter a decimal number: 20  
20 in hexadecimal is: 14  
20 in octal is: 24  
20 in decimal is: 20
```

Program Output

# Floating-Point Precision (`precision`, `setprecision`)

- `precision`
  - Member function
  - Sets number of digits to the right of decimal point  
`cout.precision(2);`
  - `cout.precision()` returns current precision setting
- `setprecision`
  - Parameterized stream manipulator
  - Like all parameterized stream manipulators, `<iomanip>` required
  - Specify precision:  
`cout << setprecision(2) << x;`
- For both methods, changes last until a different value is set

```
1 // Fig. 21.17: fig21_17.cpp
2 // Controlling precision of floating-point values
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::ios;
12 using std::setiosflags;
13 using std::setprecision;
14
15 #include <cmath>
16
17 int main()
18 {
19     double root2 = sqrt( 2.0 );
20     int places;
21
22     cout << setiosflags( ios::fixed )
23         << "Square root of 2 with precisions 0-9.\n"
24         << "Precision set by the "
25         << "precision member function:" << endl;
26
```



## Outline



fig21\_17.cpp (Part 1  
of 2)

```
27 for ( places = 0; places <= 9; places++ ) {
28     cout.precision( places );
29     cout << root2 << '\n';
30 } // end for
31
32 cout << "\nPrecision set by the "
33     << "setprecision manipulator:\n";
34
35 for ( places = 0; places <= 9; places++ )
36     cout << setprecision( places ) << root2 << '\n';
37
38 return 0;
39 } // end function main
```



Outline



fig21\_17.cpp (Part 2  
of 2)





Outline



**Program Output**

Square root of 2 with precisions 0-9.

Precision set by the precision member function:

```
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Precision set by the setprecision manipulator:

```
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

# Field Width(`setw`, `width`)

- `ios width` member function
  - Sets field width (number of character positions a value should be output or number of characters that should be input)
  - Returns previous width
  - If values processed are smaller than width, fill characters inserted as padding
  - Values are not truncated - full number printed
  - `cin.width(5);`
- `setw` stream manipulator  
`cin >> setw(5) >> string;`
- Remember to reserve one space for the null character

```
1 // fig21_18.cpp
2 // Demonstrating the width member function
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 int main()
10 {
11     int w = 4;
12     char string[ 10 ];
13
14     cout << "Enter a sentence:\n";
15     cin.width( 5 );
16
17     while ( cin >> string ) {
18         cout.width( w++ );
19         cout << string << endl;
20         cin.width( 5 );
21     } // end while
22
23     return 0;
24 } // end function main
```



Outline



fig21\_18.cpp

Enter a sentence:

This is a test of the width member function

This

is

a

test

of

the

widt

h

memb

er

func

tion



Outline



**Program Output**

# User-Defined Manipulators

- We can create our own stream manipulators
  - bell
  - ret (carriage return)
  - tab
  - endl
- Parameterized stream manipulators
  - Consult installation manuals



## Outline



fig21\_19.cpp (Part 1 of 2)

```
1 // Fig. 21.19: fig21_19.cpp
2 // Creating and testing user-defined, nonparameterized
3 // stream manipulators.
4 #include <iostream>
5
6 using std::ostream;
7 using std::cout;
8 using std::flush;
9
10 // bell manipulator (using escape sequence \a)
11 ostream& bell( ostream& output ) { return output << '\a'; }
12
13 // ret manipulator (using escape sequence \r)
14 ostream& ret( ostream& output ) { return output << '\r'; }
15
16 // tab manipulator (using escape sequence \t)
17 ostream& tab( ostream& output ) { return output << '\t'; }
18
19 // endLine manipulator (using escape sequence \n
20 // and the flush member function)
21 ostream& endLine( ostream& output )
22 {
23     return output << '\n' << flush;
24 } // end function endLine
25
```

```
26 int main()
27 {
28     cout << "Testing the tab manipulator:" << endl
29         << 'a' << tab << 'b' << tab << 'c' << endl
30         << "Testing the ret and bell manipulators:"
31         << endl << ".....";
32     cout << bell;
33     cout << ret << "-----" << endl;
34     return 0;
35 } // end function main
```



## Outline

fig21\_19.cpp (Part 2 of 2)

## Program Output

```
Testing the tab manipulator:
a      b      c
Testing the ret and bell manipulators:
-----.....
```

## 21.7 Stream Format States

- Format flags
  - Specify formatting to be performed during stream I/O operations
- `setf`, `unsetf` and `flags`
  - Member functions that control the flag settings



# Stream Error States

- `eofbit`
  - Set for an input stream after end-of-file encountered
  - `cin.eof()` returns `true` if end-of-file has been encountered on `cin`
  
- `failbit`
  - Set for a stream when a format error occurs
  - `cin.fail()` - returns `true` if a stream operation has failed
  - Normally possible to recover from these errors

# Stream Error States

- `badbit`
  - Set when an error occurs that results in data loss
  - `cin.bad()` returns `true` if stream operation failed
  - normally nonrecoverable
- `goodbit`
  - Set for a stream if neither `eofbit`, `failbit` or `badbit` are set
  - `cin.good()` returns `true` if the `bad`, `fail` and `eof` functions would all return `false`.
  - I/O operations should only be performed on “good” streams
- `rdstate`
  - Returns the state of the stream
  - Stream can be tested with a `switch` statement that examines all of the state bits
  - Easier to use `eof`, `bad`, `fail`, and `good` to determine state

# Stream Error States

- `clear`
  - Used to restore a stream's state to "good"
  - `cin.clear()` clears `cin` and sets `goodbit` for the stream
  - `cin.clear( ios::failbit )` actually sets the `failbit`
    - Might do this when encountering a problem with a user-defined type
- Other operators
  - `operator!`
    - Returns `true` if `badbit` or `failbit` set
  - `operator void*`
    - Returns `false` if `badbit` or `failbit` set
  - Useful for file processing

```

1 // Fig. 21.29: fig21_29.cpp
2 // Testing error states.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::cin;
8
9 int main()
10 {
11     int x;
12     cout << "Before a bad input operation:"
13         << "\ncin.rdstate(): " << cin.rdstate()
14         << "\n    cin.eof(): " << cin.eof()
15         << "\n    cin.fail(): " << cin.fail()
16         << "\n    cin.bad(): " << cin.bad()
17         << "\n    cin.good(): " << cin.good()
18         << "\n\nExpects an integer, but enter a character: ";
19     cin >> x;
20
21     cout << "\nAfter a bad input operation:"
22         << "\ncin.rdstate(): " << cin.rdstate()
23         << "\n    cin.eof(): " << cin.eof()
24         << "\n    cin.fail(): " << cin.fail()
25         << "\n    cin.bad(): " << cin.bad()
26         << "\n    cin.good(): " << cin.good() << "\n\n";
27

```



## Outline



fig21\_29.cpp (Part 1 of 2)

```
28  cin.clear();
29
30  cout << "After cin.clear()"
31      << "\ncin.fail(): " << cin.fail()
32      << "\ncin.good(): " << cin.good() << endl;
33  return 0;
34 } // end function main
```



## Outline

fig21\_29.cpp (Part 2 of 2)

## Program Output

Before a bad input operation:

```
cin.rdstate(): 0
  cin.eof(): 0
cin.fail(): 0
  cin.bad(): 0
cin.good(): 1
```

Expects an integer, but enter a character: A

After a bad input operation:

```
cin.rdstate(): 2
  cin.eof(): 0
cin.fail(): 1
  cin.bad(): 0
cin.good(): 0
```

After cin.clear()

```
cin.fail(): 0
cin.good(): 1
```

# UNIT-5

## Exception Handling

# Introduction

- Errors can be dealt with at place error occurs
  - Easy to see if proper error checking implemented
  - Harder to read application itself and see how code works
- Exception handling
  - Makes clear, robust, fault-tolerant programs
  - C++ removes error handling code from "main line" of program
- Common failures
  - new not allocating memory
  - Out of bounds array subscript
  - Division by zero
  - Invalid function parameters

# Introduction

- Exception handling - catch errors before they occur
  - Deals with synchronous errors (i.e., Divide by zero)
  - Does not deal with asynchronous errors - disk I/O completions, mouse clicks - use interrupt processing
  - Used when system can recover from error
    - Exception handler - recovery procedure
  - Typically used when error dealt with in different place than where it occurred
  - Useful when program cannot recover but must shut down cleanly
- Exception handling should not be used for program control
  - Not optimized, can harm program performance



# Introduction

- Exception handling improves fault-tolerance
  - Easier to write error-processing code
  - Specify what type of exceptions are to be caught
- Most programs support only single threads
  - Techniques in this chapter apply for multithreaded OS as well (windows NT, OS/2, some UNIX)
- Exception handling another way to return control from a function or block of code

# When Exception Handling Should Be Used

- Error handling should be used for
  - Processing exceptional situations
  - Processing exceptions for components that cannot handle them directly
  - Processing exceptions for widely used components (libraries, classes, functions) that should not process their own exceptions
  - Large projects that require uniform error processing

# Other Error-Handling Techniques

- Use assert
  - If assertion false, the program terminates
- Ignore exceptions
  - Use this "technique" on casual, personal programs - not commercial!
- Abort the program
  - Appropriate for nonfatal errors give appearance that program functioned correctly
  - Inappropriate for mission-critical programs, can cause resource leaks
- Set some error indicator
  - Program may not check indicator at all points where error could occur

## Other Error-Handling Techniques (II)

- Test for the error condition
  - Issue an error message and call `exit`
  - Pass error code to environment
- `setjump` and `longjump`
  - In `<cssetjmp>`
  - Jump out of deeply nested function calls back to an error handler.
  - Dangerous - unwinds the stack without calling destructors for automatic objects (more later)
- Specific errors
  - Some have dedicated capabilities for handling them
  - If `new` fails to allocate memory `new_handler` function executes to deal with problem

# Basics of C++ Exception Handling: try, throw, catch

- A function can throw an exception object if it detects an error
  - Object typically a character string (error message) or class object
  - If exception handler exists, exception caught and handled
  - Otherwise, program terminates

# Basics of C++ Exception Handling: try, throw, catch (II)

- Format
  - Enclose code that may have an error in try block
  - Follow with one or more catch blocks
    - Each catch block has an exception handler
  - If exception occurs and matches parameter in catch block, code in catch block executed
  - If no exception thrown, exception handlers skipped and control resumes after catch blocks
  - throw point - place where exception occurred
    - Control cannot return to throw point

# Throwing an Exception

- `throw` – indicates an exception has occurred
  - Usually has one operand (sometimes zero) of any type
    - If operand an object, called an exception object
    - Conditional expression can be thrown
  - Code referenced in a `try` block can throw an exception
  - Exception caught by closest exception handler
  - Control exits current `try` block and goes to catch handler (if it exists)
  - Example (inside function definition)

```
if ( denominator == 0 )  
    throw DivideByZeroException();
```

    - Throws a `dividebyzeroexception` object

## Throwing an Exception (II)

- Exception not required to terminate program
  - However, terminates block where exception occurred



# Catching an Exception

- Exception handlers are in catch blocks
  - Format: `catch( exceptionType parameterName){`  
    exception handling code  
    }
  - Caught if argument type matches throw type
  - If not caught then terminate called which (by default) calls abort
  - Example:

```
catch ( DivideByZeroException ex) {  
    cout << "Exception occurred: " << ex.what()  
    <<'\n'  
}
```
  - Catches exceptions of type `DivideByZeroException`

## Catching an Exception

- catch parameter matches thrown object when
  - They are of the same type
    - Exact match required - no promotions/conversions allowed
  - The catch parameter is a public base class of the thrown object
  - The catch parameter is a base-class pointer/ reference type and the thrown object is a derived-class pointer/ reference type
  - The catch handler is `catch( ... )`
  - Thrown const objects have `const` in the parameter type

# Exception Specifications

- Exception specification (throw list)

- Lists exceptions that can be thrown by a function

Example:

```
int g( double h ) throw ( a, b, c )
{
    // function body
}
```

- Function can throw listed exceptions or derived types
- If other type thrown, function unexpected called
- throw() (i.e., no throw list) states that function will not throw any exceptions
  - In reality, function can still throw exceptions, but calls unexpected (more later)
- If no throw list specified, function can throw any exception

## Processing Unexpected Exceptions

- **Function unexpected**
  - Calls the function specified with `set_unexpected`
    - Default: `terminate`
- **Function terminate**
  - Calls function specified with `set_terminate`
    - Default: `abort`
- **`set_terminate` and `set_unexpected`**
  - Prototypes in `<exception>`
  - Take pointers to functions (i.e., Function name)
    - Function must return `void` and take no arguments
  - Returns pointer to last function called by `terminate` or `unexpected`

# Stack Unwinding

- Function-call stack unwound when exception thrown and not caught in a particular scope
  - Tries to catch exception in next outer try/catch block
  - Function in which exception was not caught terminates
    - Local variables destroyed
    - Control returns to place where function was called
  - If control returns to a try block, attempt made to catch exception
    - Otherwise, further unwinds stack
  - If exception not caught, terminate called

# Rethrowing an Exception

- Rethrowing exceptions
  - Used when an exception handler cannot process an exception
  - Rethrow exception with the statement:  
`throw;`
    - No arguments
    - If no exception thrown in first place, calls `terminate`
  - Handler can always rethrow exception, even if it performed some processing
  - Rethrown exception detected by next enclosing try block

# Catching an Exception

- Catch all exceptions

`catch(...)` - catches all exceptions

- You do not know what type of exception occurred
- There is no parameter name - cannot reference the object

- If no handler matches thrown object

- Searches next enclosing try block
  - If none found, terminate called
- If found, control resumes after last catch block
- If several handlers match thrown object, first one found is executed

## Catching an Exception

- Unreleased resources
  - Resources may have been allocated when exception thrown
  - catch handler should delete space allocated by new and close any opened files
- catch handlers can throw exceptions
  - Exceptions can only be processed by outer try blocks