# Object Oriented Programming

Binnur Kurt
*kurt@ce.itu.edu.tr*

Istanbul Technical University
*Computer Engineering Department*


Version 0.1.2

# About the Lecturer

- **BSc**

  İTÜ, Computer Engineering Department, 1995

- **MSc**

  İTÜ, Computer Engineering Department, 1997

- Areas of Interest
  - Digital Image and Video Analysis and Processing
  - Real-Time Computer Vision Systems
  - Multimedia: Indexing and Retrieval
  - Software Engineering
  - OO Analysis and Design

# Welcome to the Course

## Important Course Information

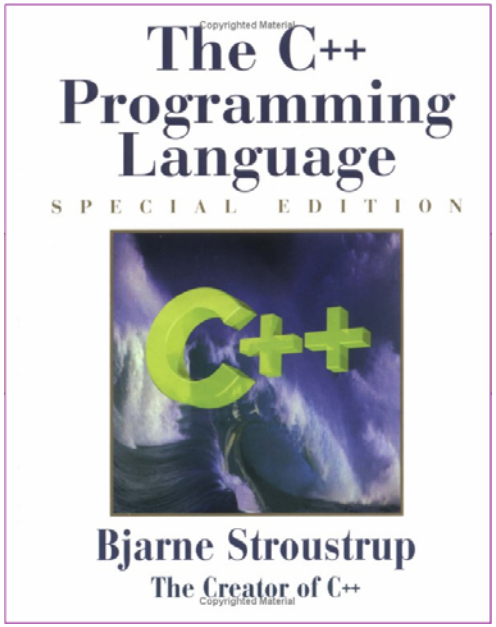
- Course Hours
  - 10:00-13:00 Thursday
- Course Web Page
  - http://www.cs.itu.edu.tr/~kurt/courses/blg252e
- Join to the group
  - http://groups.yahoo.com/group/blg252e
  - blg252e@yahoogroups.com
- E-mail Kurt@ce.itu.edu.tr

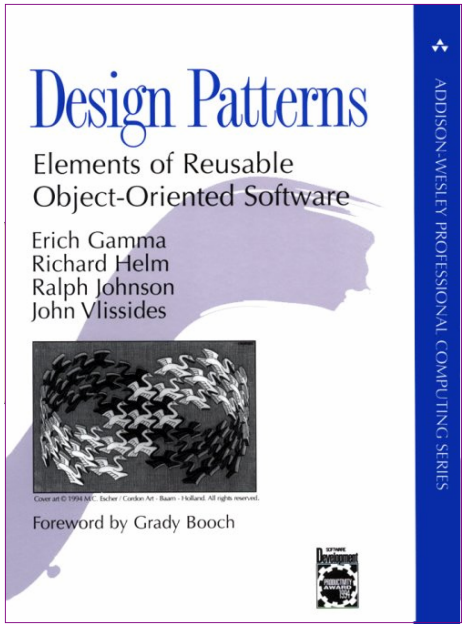
# Grading Scheme

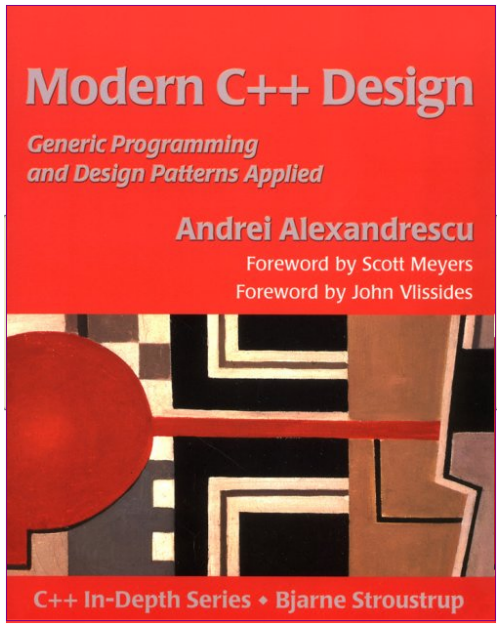
- 3 Homeworks (5% each)
- 2 Midterm Exams (20%,25%)
- A final exam (40%)
- You must follow the official Homework Guidelines

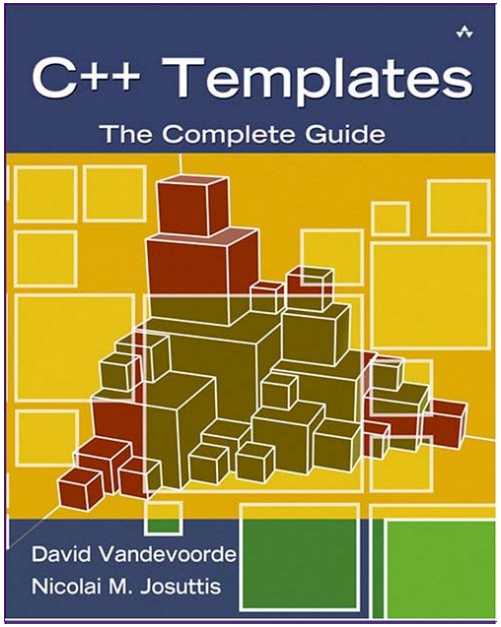
  (http://www.ce.itu.edu.tr/lisans/kilavuz.html).
- Academic dishonesty including but not limited to cheating, plagiarism, collaboration is unacceptable and subject to disciplinary actions. Any student found guilty will have grade F. Assignments are due in class on the due date. Late assignments will generally not be accepted. Any exception must be approved. Approved late assignments are subject to a grade penalty.

# References




The presentation is based on

Asst.Prof.Dr. Feza Buzlaca's Lecture Notes

*Tell me and I forget.*
*Show me and I remember.*
*Let me do and I understand.*

—Chinese Proverb

There is no time for lab sessions
On the course web page you will find lab files for each week. You should do the lab sessions on your own.
Just follow the instructions on these documents.

# Purpose of the Course

- To introduce several programming paradigms including **Object-Oriented Programming, Generic Programming, Design Patterns**
- To show how to use these programming schemes with the C++ programming language to build "**good**" programs.

# Course Outline

1. Introduction to Object Oriented Programming.
2. C++: A Better C.
3. Classes and Objects
4. Constructors and Destructors
5. Operator Overloading
6. Inheritance
7. Pointers to Objects
8. Polymorphism
9. Exceptions

# Course Outline

10. Templates

11.The Standard Template Library - STL

# How to Use the Icons

## Demonstration

## Discussion

## Reference

## Exercise

# 1 INTRODUCTION

# Content

- Introduction to Software Engineering
- Object-Oriented Programming Paradigm

# Software

- Computer Software is the product that software engineers design and build.
- It encompasses
  - programs that execute within a computer of any size and architecture,
  - documents that encompass hard-copy and virtual forms,
  - data that combine numbers and text but also includes representations of pictorial, video and audio information.

# History

- Common problems:
  - Why does it take so long?
  - Why are development costs so high?
  - Why can't find all faults before delivery?
  - Why can't we measure the development?

# History

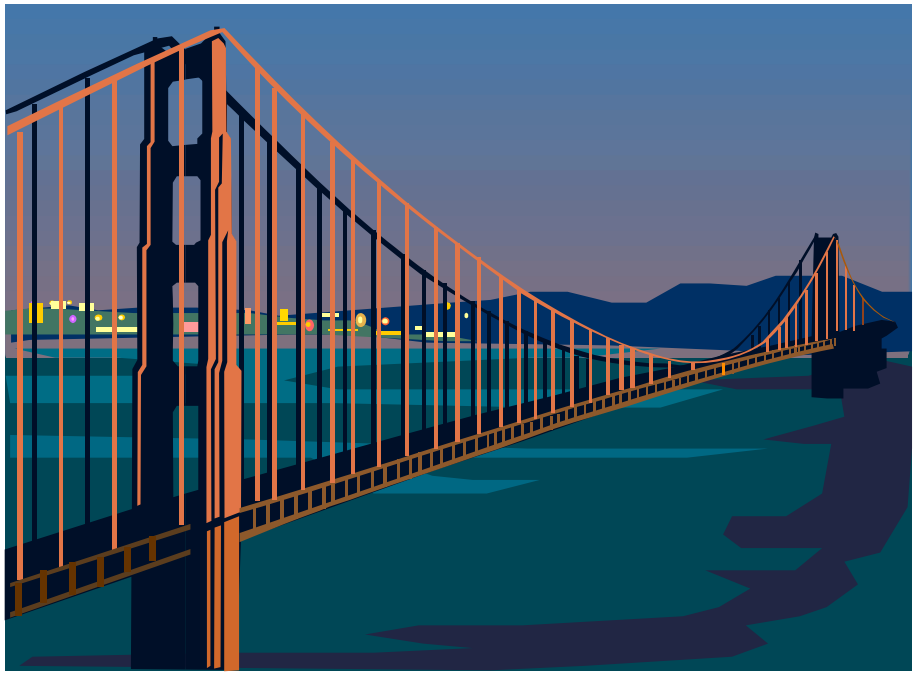
- Software Engineering: 1967, NATO Study Group, Garmisch/GERMANY
- 1968, NATO Software Engineering Conference: Software Crisis
  - Low quality
  - Not met deadlines and cost limits

# After 35 years

- Still softwares are
  - Late
  - Over budget
  - With residual faults
- Means
  - SW has own unique properties and problems
  - Crisis >>>>> Depression

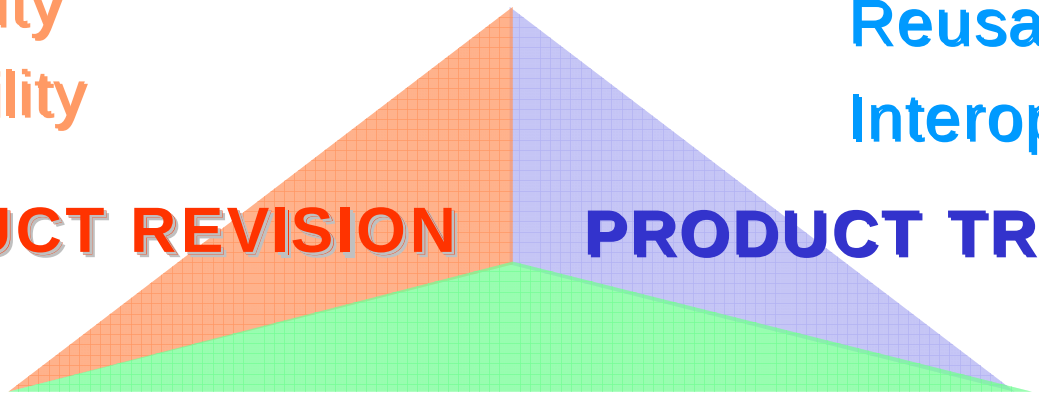
# Is SW An Engineering?

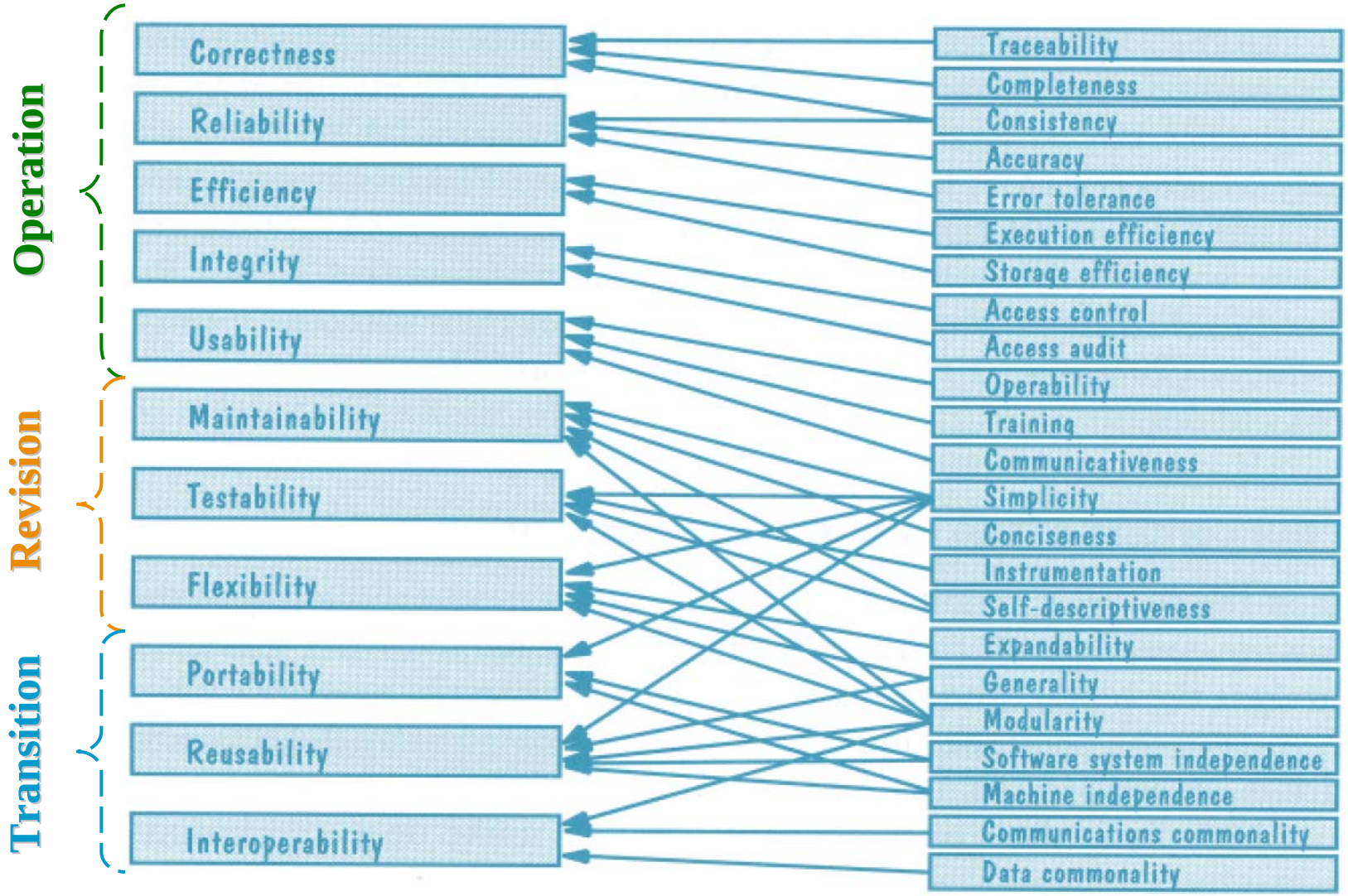
- May be?
- Bridge – Operating System
  - After collapse, redesign & rebuild
  - Inspect similar bridges
  - Perfectly engineered
  - Experience
  - Maintaining



# McCall Quality Triangle






Operation
Revision
Transition
Correctness
Reliability
Efficiency
Integrity
Usability
Maintainability
Testability
Flexibility
Portability
Reusability
Interoperability
Traceability
Completeness
Consistency
Accuracy
Error tolerance
Execution efficiency
Storage efficiency
Access control
Access audit
Operability
Training
Communicativeness
Simplicity
Conciseness
Instrumentation
Self-descriptiveness
Expandability
Generality
Modularity
Software system independence
Machine independence
Communications commonality
Data commonality

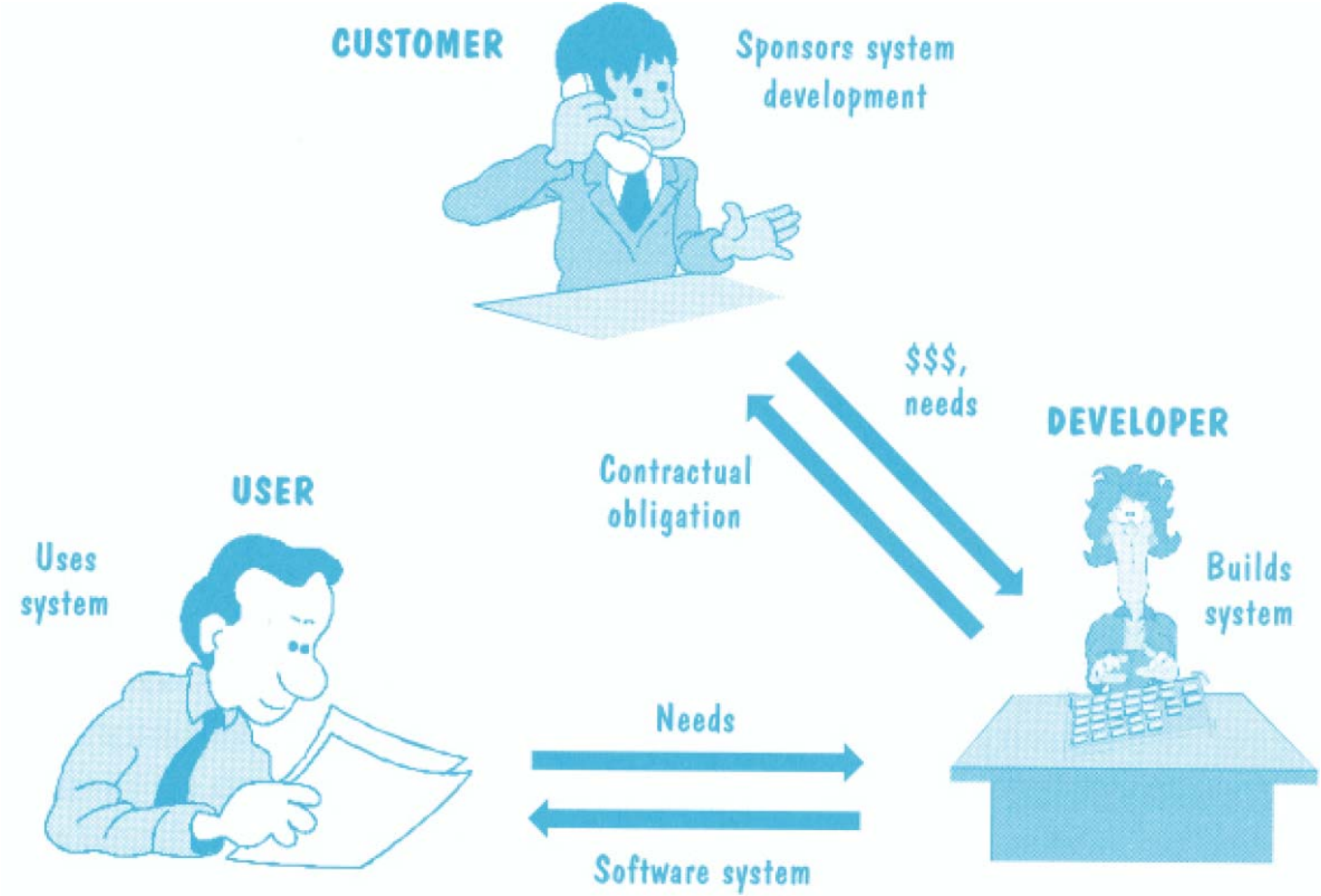
# McCall Quality Triangle

- Correctness: The extent to which a program satisfies its specification and fulfills the customer's mission objectives
- Reliability: The extent to which a program can be expected to perform its intended function with required precision
- Efficiency: The amount of computing resources and code required by a program to perform its function
- Integrity: Extent to which access to software or data by unauthorized persons can be controlled
- Usability: Effort required to learn, operate, prepare input and interpret output of a program
- Maintainability: Effort required to locate and fix an error in a program

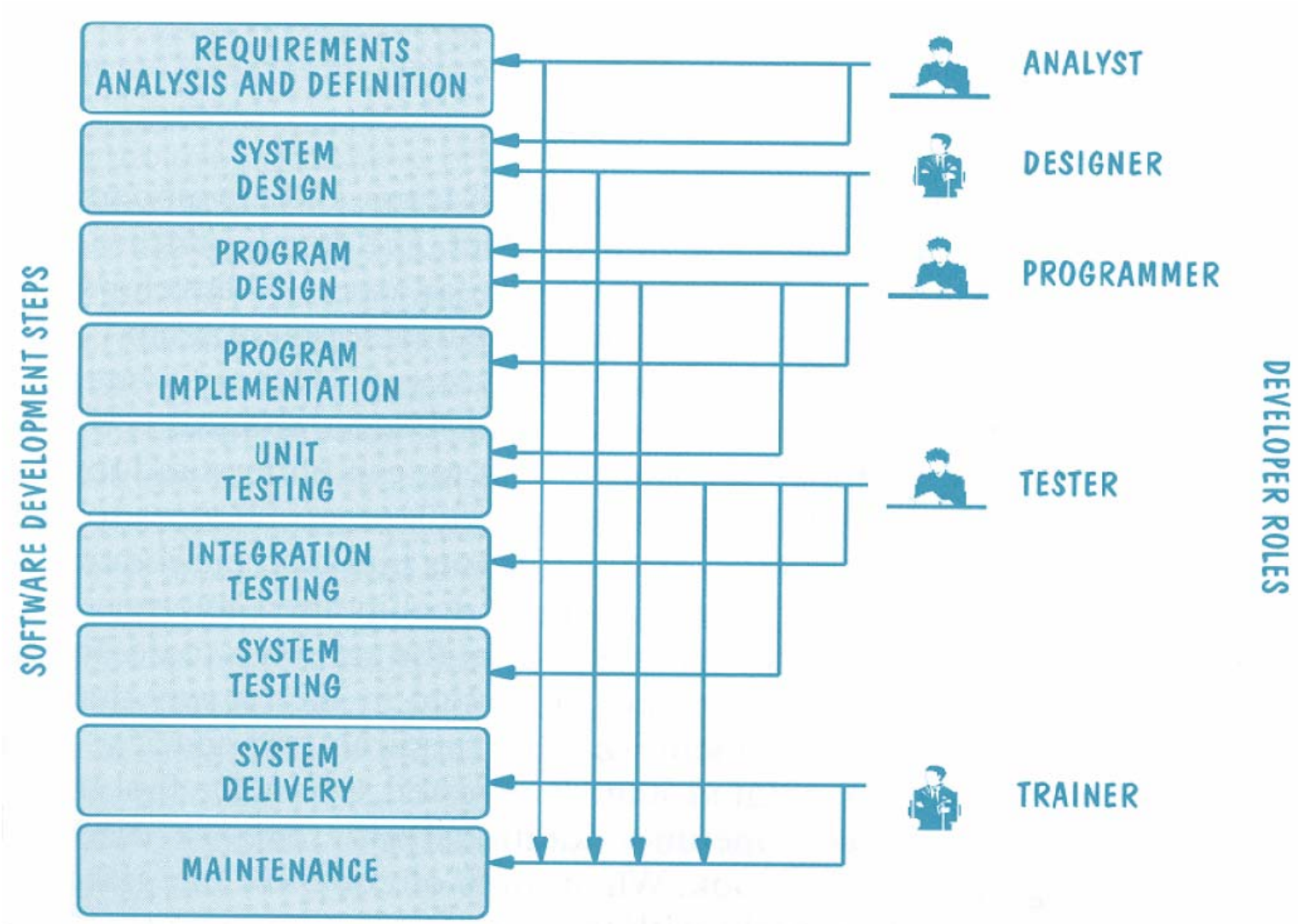
# McCall Quality Triangle

- Flexibility: Effort required to modify an operational program
- Testability: Effort required to test a program to ensure that it performs its intended function
- Portability: Effort required to transfer the program from one hardware and/or software system environment to another
- Reusability: Extent to which a program can be reused in other applications
- Interoperability: Effort required to couple one system to another

# Customer-User-Developer



# Development Team



# Software Life Cycle

- Requirements Phase
- Specification Phase
- Design Phase
- Implementation Phase
- Integration Phase
- Maintenance Phase
- Retirement Phase

# Requirements Phase

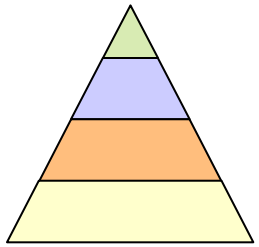
- Defining constraints
  - Functions
  - Due dates
  - Costs
  - Reliability
  - Size
- Types
  - Functional
  - Non-Functional

# Specification Phase

- Documentation of requirements
  - Inputs & Outputs
  - Formal
  - Understandable for user & developer
  - Usually functional requirements. (what to do)
  - Base for testing & maintenance
- The contract between customer & developer ?

# Design Phase

- Defining Internal structure (how to do)
- Has some levels (or types of docs)
  - Architectural design
  - Detailed design
  - ...



- Important
  - To backtrack the aims of decisions
  - To easily maintain

# Implementation Phase

- Simply coding
- Unit tests
  - For verification

# Integration Phase

- Combining modules
- System tests
  - For validation
- Quality tests

# Maintenance Phase

- Corrective
- Enhancement
  - Perfective
  - Adaptive
- Usually maintainers are not the same people with developers.
- The only input is (in general) the source code of the software?!?

# Retirement Phase

- When the cost of maintenance is not effective.
  - Changes are so drastic, that the software should be redesigned.
  - So many changes may have been made.
  - The update frequency of docs is not enough.
  - The hardware (or OS) will be changed.

# Why Object Technology?

- Expectations are,
- Reducing the effort, complexity, and cost of development and maintenance of software systems.
- Reducing the time to adapt an existing system (quicker reaction to changes in the business environment). Flexibility, reusability.
- Increasing the reliability of the system.

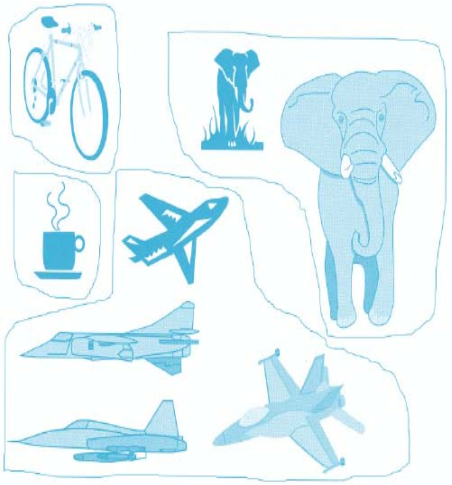
# Why C++

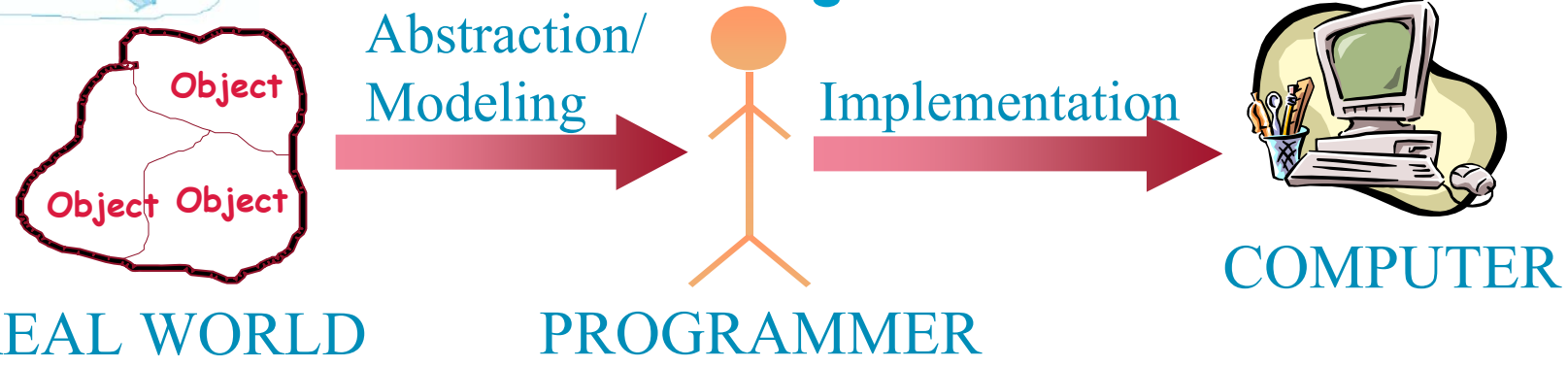
- C++ supports writing high quality programs (supports OO)
- C++ is used by hundreds of thousands of programmers in every application domain.
  - This use is supported by hundreds of libraries, hundreds of textbooks, several technical journals, many conferences.
- Application domain:
  - Systems programming: Operating systems, device drivers. Here, direct manipulation of hardware under real-time constraints are important.
  - Banking, trading, insurance: Maintainability, ease of extension, ease of testing and reliability is important.
  - Graphics and user interface programs
  - Computer Communication Programs

# What is Programming?

- Like any human language, a programming language provides a way to express concepts.
- Program development involves creating models of real world situations and building computer programs based on these models.
- Computer programs describe the method of implementing the model.
- Computer programs may contain computer world representations of the things that constitute the solutions of real world problems.

# What is Programming? (Con't)






- If successful, this medium of expression (the object-oriented way) will be significantly easier, more flexible, and efficient than the alternatives as problems grow larger and more complex.

# Learning C++

- Like human languages, programming languages also have many syntax and grammar rules.
- Knowledge about grammar rules of a programming language is not enough to write “good” programs.
- The most important thing to do when learning C++ is to focus on concepts and not get lost in language-technical details.
- Design techniques is far more important than an understanding of details; that understanding comes with time and practice.
- Before the rules of the programming language, the programming scheme must be understood.
- Your purpose in learning C++ must not be simply to learn a new syntax for doing things the way you used to, but to learn new and better ways of building systems

# Software Quality Metrics

- A program must do its job correctly. It must be useful and usable.
- A program must perform as fast as necessary (Real-time constraints).
- A program must not waste system resources (processor time, memory, disk capacity, network capacity) too much.
- It must be reliable.
- It must be easy to update the program.
- A good software must have sufficient documentation (users manual).

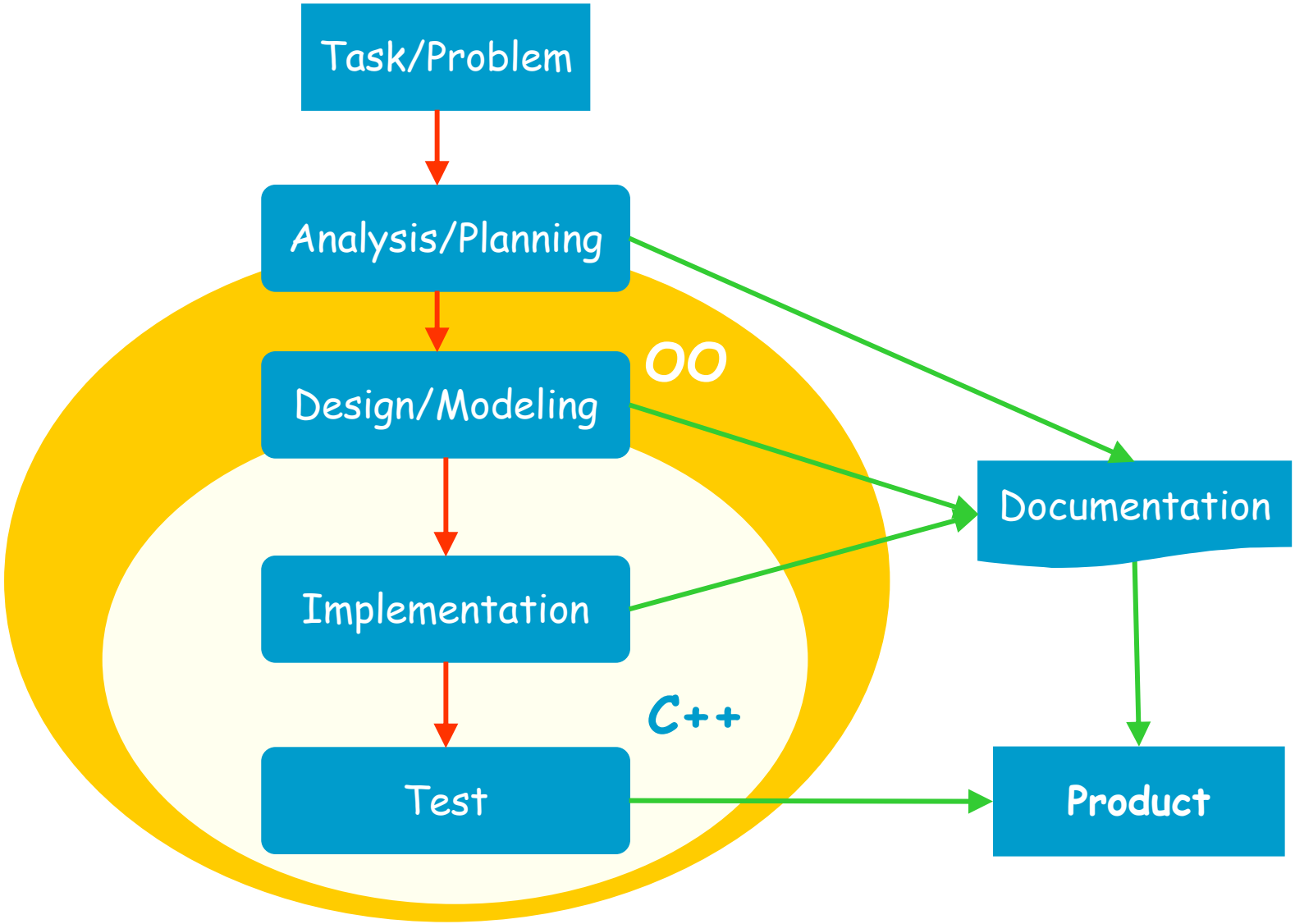
user

- Source code must be readable and understandable.
- It must be easy to maintain and update (change) the program.
- A program must consist of independent modules, with limited interaction.
- An error may not affect other parts of a program (Locality of errors).
- Modules of the program must be reusable in further projects.
- A software project must be finished before its deadline.
- A good software must have sufficient documentation (about development).

Software developer

Object-oriented programming technique enables programmers to build high-quality programs. While designing and coding a program, these quality metrics must be kept always in mind.

# Software Development Process



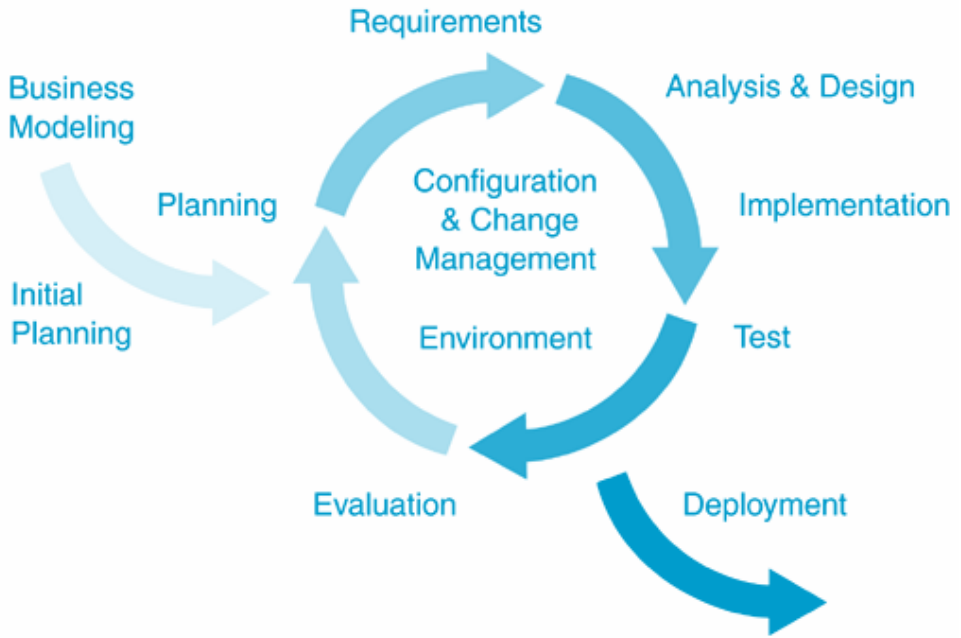
- Analysis: Gaining a clear understanding of the problem. Understanding requirements. They may change during (or after) development of the system!
- Building the programming team.
- Design: Identifying the key concepts involved in a solution. Models of the key concepts are created. This stage has a strong effect on the quality of the software. Therefore, before the coding, verification of the created model must be done.
- Design process is connected with the programming scheme. Here, our design style is object-oriented.
- Coding: The solution (model) is expressed in a program.
- Coding is connected with the programming language. In this course we will use C++.
- Documentation: Each phase of a software project must be clearly explained. A users manual should be also written.
- Test: the behavior of the program for possible inputs must be examined.

# UML

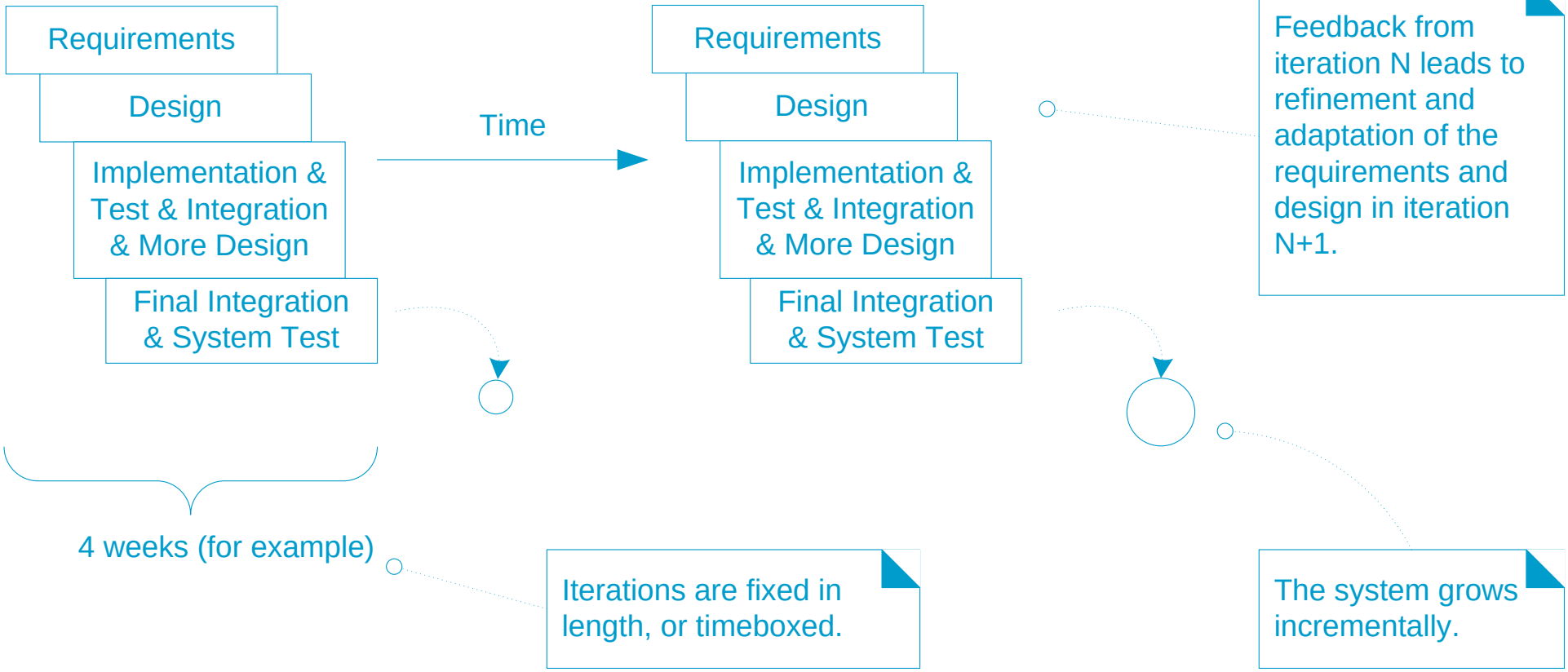
- They are important **design principles** and **design patterns**, which help us developing high-quality software. The Unified Modeling Language (**UML**) is useful to express the model.

# Unified Process (UP)

- The UP promotes several best practices.
- Iterative
- Incremental
- Risk-driven



# Unified Process (UP)

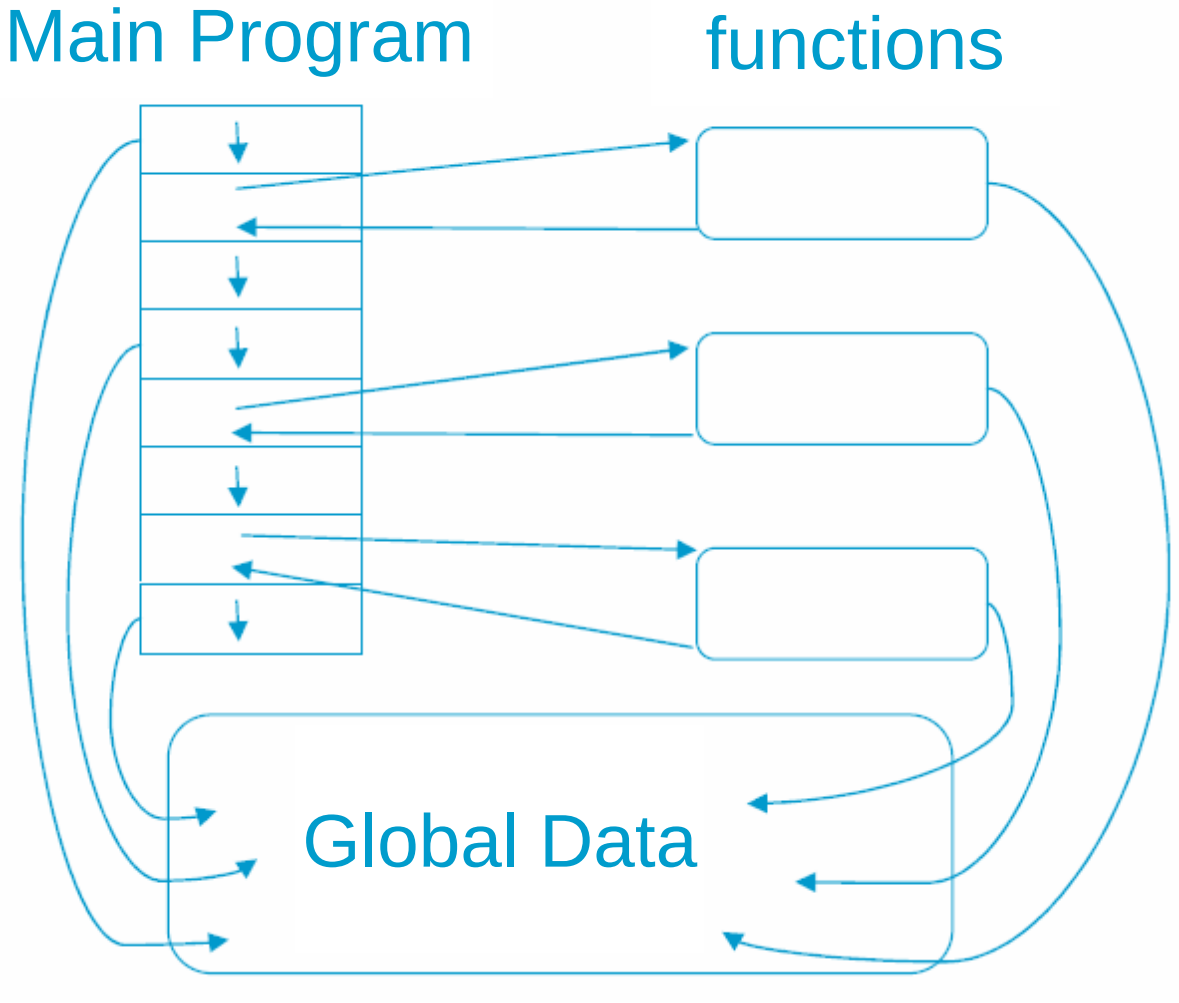


# Procedural Programming

- Pascal, C, BASIC, Fortran, and similar traditional programming languages are procedural languages. That is, each statement in the language tells the computer to do something.
- In a procedural language, the emphasis is on doing things (functions).
- A program is divided into functions and—ideally, at least—each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.

# Procedural Programming





# Problems with Procedural Programming

- Data Is Undervalued
- Data is, after all, the reason for a program's existence. The important parts of a program about a school for example, are not functions that display the data or functions that checks for correct input; they are student, teacher data.
- Procedural programs (functions and data structures) don't model the real world very well. The real world does not consist of functions.
- Global data can be corrupted by functions that have no business changing it.
- To add new data items, all the functions that access the data must be modified so that they can also access these new items.
- Creating new data types is difficult.

# Besides...

- It is also possible to write good programs by using procedural programming (C programs).
- But object-oriented programming offers programmers many advantages, to enable them to write high-quality programs.

# Object Oriented Programming



The fundamental idea behind object-oriented programming is:

• The real world consists of objects. Computer programs may contain computer world representations of the things (objects) that constitute the solutions of real world problems.

• Real world objects have two parts:

  •*Properties* (or *state* :characteristics that can change),

  •*Behavior* (or *abilities* :things they can do).

•To solve a programming problem in an object-oriented language, the programmer no longer asks *how the problem will be divided into functions*, but **how it will be divided into objects**.

•The emphasis is on **data**

# Object Oriented Programming

- What kinds of things become objects in object-oriented programs?
  - Human entities: Employees, customers, salespeople, worker, manager
  - Graphics program: Point, line, square, circle, ...
  - Mathematics: Complex numbers, matrix
  - Computer user environment: Windows, menus, buttons
  - Data-storage constructs: Customized arrays, stacks, linked lists

# OOP : Encapsulation and Data Hiding



Thinking in terms of objects rather than functions has a helpful effect on design process of programs. This results from the close match between objects in the programming sense and objects in the real world.

To create software models of real world objects both *data* and the *functions* that operate on that data are combined into a single program entity. Data represent the properties (state), and functions represent the behavior of an object. Data and its functions are said to be *encapsulated* into a single entity.

An object's functions, called *member functions* in C++ typically provide the only way to access its data. The data is *hidden*, so it is safe from accidental alteration.

# OOP : Encapsulation and Data Hiding *Con't*

- ***Encapsulation*** and ***data hiding*** are key terms in the description of object-oriented languages.
- If you want to modify the data in an object, you know exactly what functions interact with it: the member functions in the object. No other functions can access the data. This simplifies writing, debugging, and maintaining the program.

# Example: A Point on the plane



A Point on a plane has two properties; x-y coordinates.

Abilities (behavior) of a Point are, moving on the plane, appearing on the screen and disappearing.

A model for 2 dimensional points with the following parts:

Two integer variables (**x** , **y**) to represent x and y coordinates

A function to move the point: **move** ,

A function to print the point on the screen: **print** ,

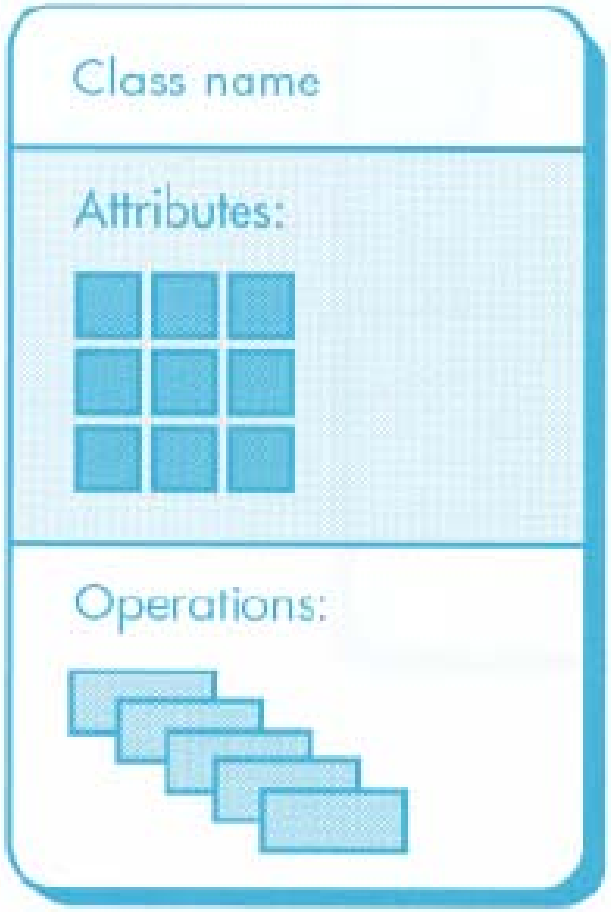
A function to hide the point: **hide** .

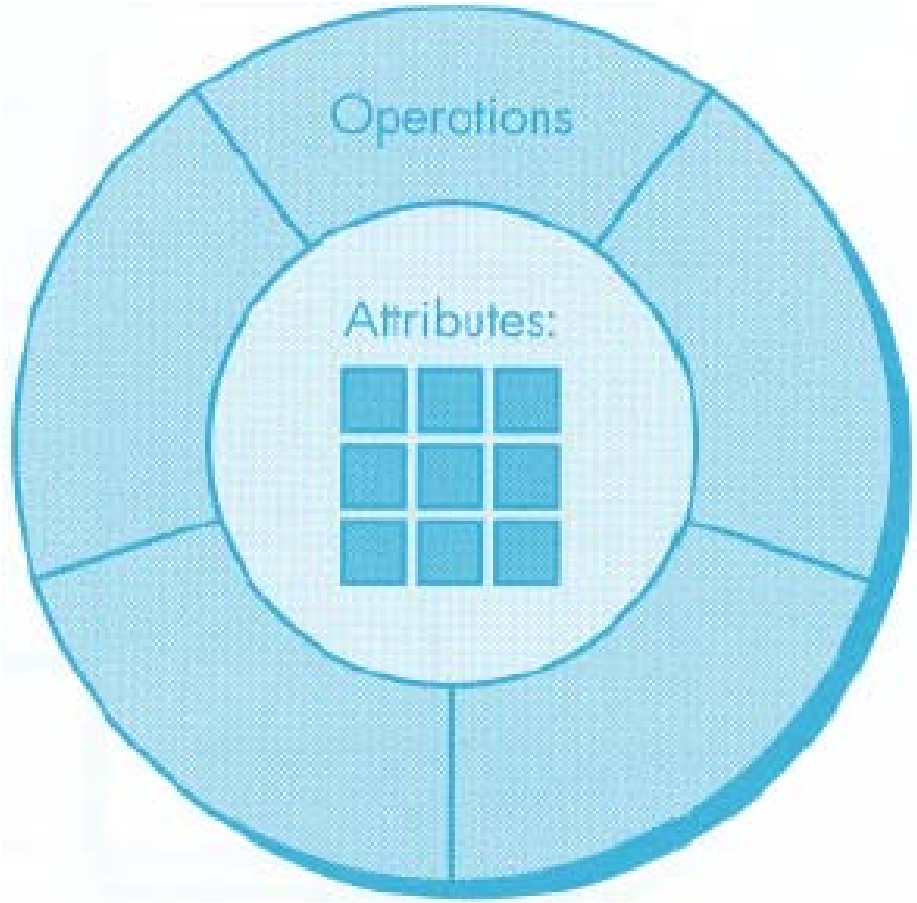
# Example: A Point on the plane *Con't*

Once the model has been built and tested, it is possible to create many objects of this model , in main program.

```
Point point1, point2, point3;
    :
point1.move(50,30);
point1.print();
```

# The Object Model

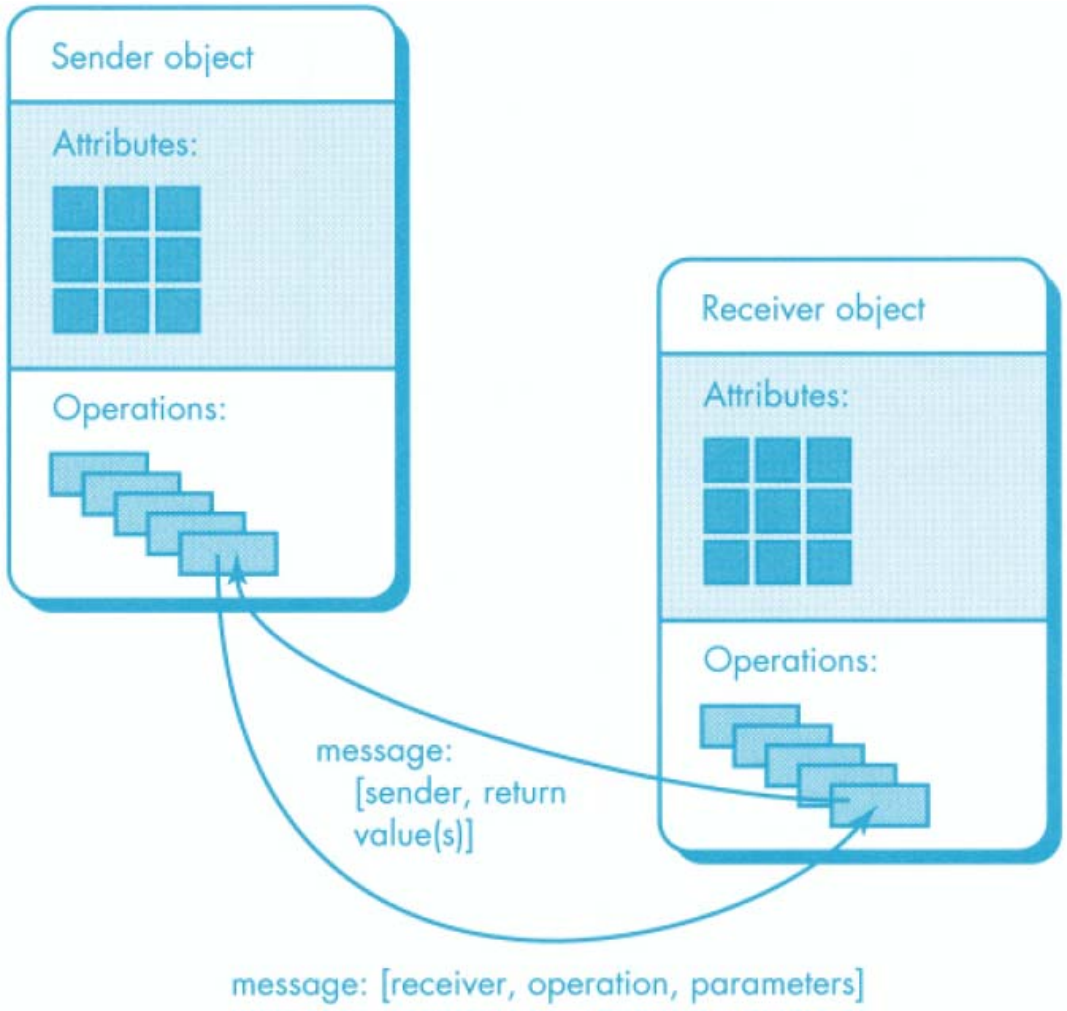



A C++ program typically consists of a number of objects that communicate with each other by calling one another's member functions.

# The Object Model

*Con't*



# OOP *vs*. Procedural Programming



Procedural Programming:

•Procedural languages still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve.

•The programmer must establish the association between the machine model and the model of the problem that is actually being solved.

•The effort required to perform this mapping produces programs that are difficult to write and expensive to maintain. Because the real world thing and their models on the computer are quite different.

# Example: Procedural Programming *Con't*

- Real world thing: student
- Computer model: char *, int, float ...
- It is said that the C language is closer to the computer than the problem.

# OOP *vs*. Procedural Programming *Con't*



Object Oriented Programming

► The object-oriented approach provides tools for the programmer to represent elements in the problem space.

► We refer to the elements in the problem space and their representations in the solution space as "objects."

► The idea is that the program is allowed to adapt itself to the problem by adding new types of objects, so when you read the code describing the solution, you're reading words that also express the problem.

► OOP allows you to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run.

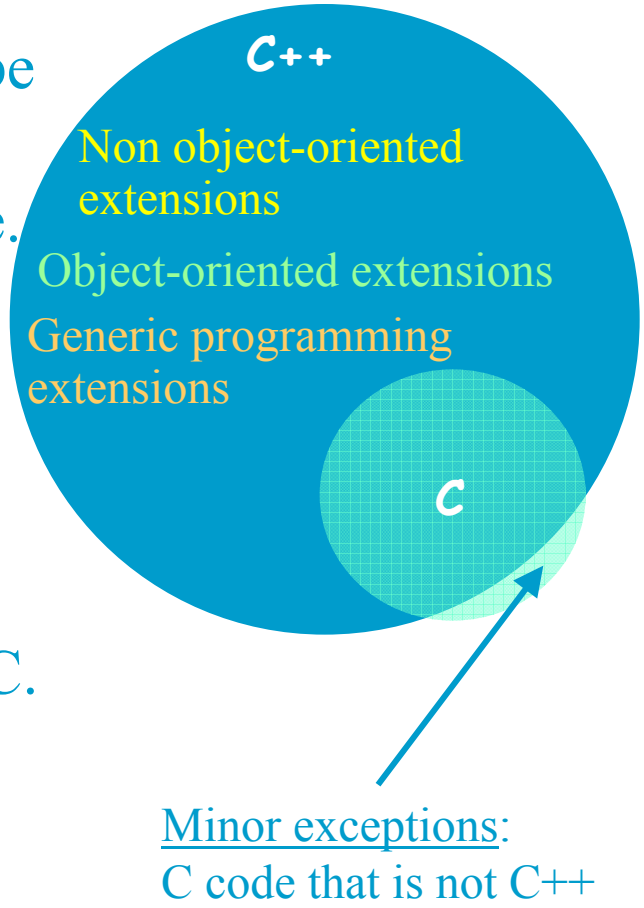
# OOP *vs*. Procedural Programming *Con't*

- Benefits of the object-oriented programming:
  - Readability
  - Understandability
  - Low probability of errors
  - Maintenance
  - Reusability
  - Teamwork

# 2 C++ As a Better C

# C++ As a Better C

- C++ was developed from the C programming language, by adding some features to it. These features can be collected in three groups:
  1. Non-object-oriented features, which can be used in coding phase. These are not involved with the programming technique.
  2. Features which support object-oriented programming.
  3. Features which support generic programming.
- With minor exceptions, C++ is a superset of C.




Minor exceptions:
C code that is not C++

# C++'s Enhancements to C (Non Object-Oriented)

►Caution: The better one knows C, the harder it seems to be to avoid writing C++ in C style, thereby losing some of the potential benefits of C++.

►1. Always keep object-oriented and generic programming techniques in mind.

►2. Always use C++ style coding technique which has many advantages over C style.

►Non object-oriented features of a C++ compiler can be also used in writing procedural programs.

# C++'s Enhancements to C (Non-OO)



- Comment Lines
- /* This is a comment */
- // This is a comment
- C++ allows you to begin a comment with // and use the remainder of the line for comment text.
- This increases readability.

# Declarations and Definitions in C++



►Remember; there is a difference between a declaration and a definition

►A declaration introduces a name – an identifier – to the compiler. It tells the compiler "This function or this variable exists somewhere, and here is what it should look like."

►A definition, on the other hand, says: "Make this variable here" or "Make this function here." It allocates storage for the name.

# Example

```
extern int i;              // Declaration
int i;                     // Definition

struct ComplexT{           // Declaration
    float re,im;
};
ComplexT c1,c2;            // Definition
void func( int, int);      // Declaration     (its body is a definition)
```

► In C, declarations and definitions must occur at the beginning of a block.

► In C++ declarations and definitions can be placed anywhere an executable statement can appear, except that they must appear prior to the point at which they are first used. This improve the readability of the program.

► A variable lives only in the block, in which it was defined. This block is the **scope** of this variable.

# C++'s Enhancements to C (Non-OO)



```
int a=0;
for (int i=0; i < 100; i++){   // i is declared in for loop
        a++;
        int p=12;              // Declaration of p
        ...                    // Scope of p
}                              // End of scope for i and p
```

- Variable i is created at the beginning of the for loop once.
- Variable p is created 100 times.

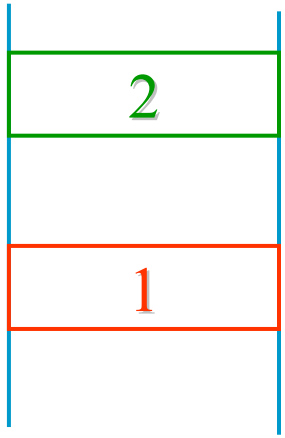
# C++'s Enhancements to C (Non-OO)

- Scope Operator **::**

A definition in a block can hide a definition in an enclosing block or a global name. It is possible to use a hidden global name by using the scope resolution operator ::

```
int y=0;        // Global y
int x=1;        // Global x
void f(){       // Function is a new block
   int x=5;     // Local  x=5, it hides global x
   ::x++;       // Global x=2
   x++;         // Local x=6
   y++;         // Global y=1
}
```

```
int x=1;
void f(){
   int x=2;      // Local  x
   ::x++;        // Global x is 2
}
```



```
int i=1;
main(){
   int i=2;
   {
          int n=i ;
          int i = 3 ;
          cout << i << " " << ::i << endl ;
          cout << n << "\n" ;
    }
  cout << i << " " << ::i << endl;
  return 0 ;
}
```

```
3 1
2
2 1
```

- Like in C, in C++ the same operator may have more than one meaning. The scope operator has also many different tasks.

# inline functions

- In C, macros are defined by using the #define directive of the preprocessor.
- In C++ macros are defined as normal functions. Here the keyword inline is inserted before the declaration of the function.
- Remember the difference between normal functions and macros:
- A normal function is placed in a separate section of code and a call to the function generates a jump to this section of code.
- Before the jump the return address and arguments are saved in memory (usually in stack).

# inline functions *Con't*



- When the function has finished executing, return address and return value are taken from memory and control jumps back from the end of the function to the statement following the function call.
- The advantage of this approach is that the same code can be called (executed) from many different places in the program. This makes it unnecessary to duplicate the function's code every time it is executed.
- There is a disadvantage as well, however.
- The function call itself, and the transfer of the arguments take some time. In a program with many function calls (especially inside loops), these times can add up and decrease the performance.

# inline functions *Con't*

```
#define sq(x) (x*x)
inline int SQ(int x){return (x*x); }
#define max(x,y) (y<x ? x : y)
inline int max(int x,int y){return (y<x ? x : y); }
```

- An inline function is defined using almost the same syntax as an ordinary function. However, instead of placing the function's machine-language code in a separate location, the compiler simply inserts it into the location of the function call. :

```
int j, k, l ;  // Three integers are defined
…….      // Some operations over k and l
j = max( k, l )  ; // inline function max is inserted
j= (k<l ? k : l)
```

# inline functions *Con't*

- The decision to inline a function must be made with some care.
  - If a function is more than a few lines long and is called many times, then inlining it may require much more memory than an ordinary function.
  - It's appropriate to inline a function when it is short, but not otherwise. If a long or complex function is inlined, too much memory will be used and not much time will be saved.

# inline functions *Con't*

- Advantages
  - Debugging
  - Type checking
  - Readable

# Default Function Arguments

- A programmer can give default values to parameters of a function. In calling of the function, if the arguments are not given, default values are used.

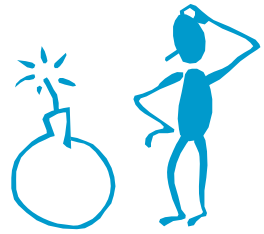
```
int exp(int n,int k=2){
    if(k == 2)
      return (n*n) ;
    else
      return ( exp(n,k-1)*n ) ;
}
```

**exp(i+5)**
// (i+5)* (i+5)
**exp(i+5,3)**
// (i+5)^3

# Example

- In calling a function argument must be given from left to right without skipping any parameter

```
void f(int i, int j=7) ; // right

void g(int i=3, int j) ; // wrong

void h(int i, int j=3,int k=7) ; // right

void m(int i=1, int j=2,int k=3) ; // right

void n(int i=2, int j,int k=3) ; // right ? wrong
```



# Example

void n(int i=1, int j=2,int k=3) ;

- n() → n(1,2,3)
- n(2) → n(2,2,3)
- n(3,4) → n(3,4,3)
- n(5,6,7) →n(5,6,7)

# Function Declarations and Definitions

►C++ uses a stricter type checking.

►In function declarations (prototypes) the data types of the parameters must be included in the parentheses.

```
char grade (int, int, int);    // declaration

int main()
{
   :
}
char grade (int exam_1, int exam_2, int final_exam) // definition
{
  :              // body of function
}
```

# Function Declarations and Definitions

- In C++ a return type must be specified; a missing return type does not default to **int** as is the case in C.
- In C++, a function that has no parameters can have an empty parameter list.

```
int print (void);     /* C style  */
int print();          // C++ style
```

# Reference Operator — &

- This operator provides an alternative name for storage
- There are two usages of the operator

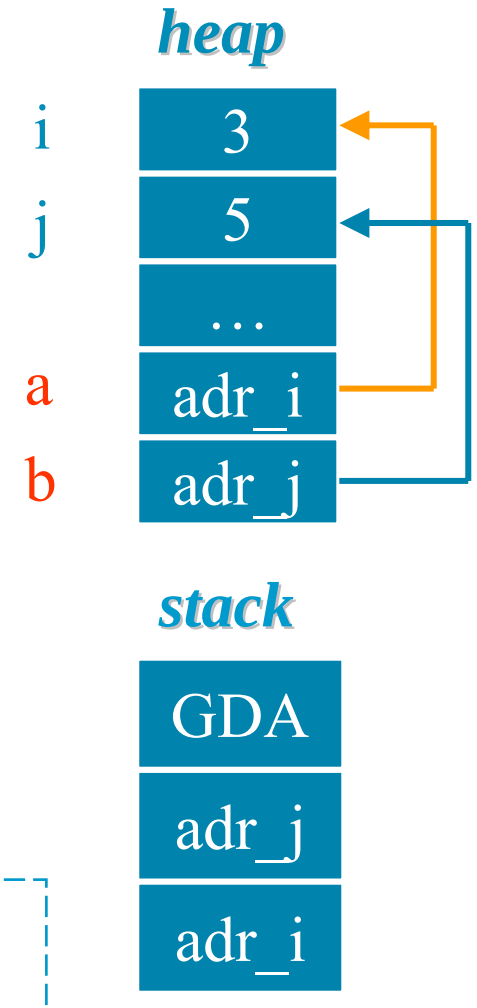
1.
```
int n ;

int& nn = n ;

double a[10] ;

double& last = a[9] ;

const char& new_line = '\n' ;
```

## 2 ▶ Parameters Passing: Consider swap() function

```
void swap(int *a, int *b){
    int temp = *a ;
    *a = *b ;
    *b = temp ;  }
```

```
int main(){
   int i=3,j=5 ;
   swap(&i,&j) ;
   cout << i << " " << j << endl ;
}
```

**5 3**



```
void swap(int& a,int& b){
    int temp = a ;
    a = b ;
    b = temp ;  }
```

```
int main(){
   int i=3,j=5 ;
   swap(i,j) ;
   cout << i << " " << j << endl ;
}
```

```
5 3
```

```
void shift(int& a1,int& a2,int& a3,int& a4){
    int tmp = a1 ;
    a1 = a2 ;
    a2 = a3 ;
    a3 = a4 ;
    a4 = tmp ;
}
```

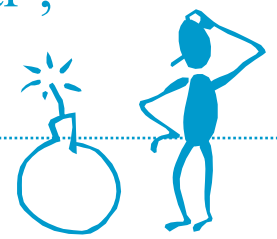
```
int main(){
        int x=1,y=2,z=3,w=4;
        cout << x << y << z << w << endl;
        shift(x,y,z,w) ;
       cout << x << y << z << w << endl;
        return 0 ;
}
```

```
int squareByValue(int a){
        return (a*a) ;
}
```

```
void squareByReference(int& a){
        a *= a ;
}
```

```
int main(){
        int x=2,y=3,z=4 ;
        squareByPointer(&x) ;
        cout << x << endl ;
        squareByReference(y) ;
        cout << y << endl ;
        z = squareByValue(z) ;
        cout << z << endl ;
}
```

```
void squareByPointer(int *aPtr){
        *aPtr = *aPtr**aPtr ;
}
```



```
4
9
16
```

# const Reference



►To prevent the function from changing the parameter accidentally, we pass the argument as constant reference to the function.

```
struct Person{                                  // A structure to define persons
    char name [40];                             // Name filed 40 bytes
    int reg_num;                                // Register number 4 bytes
};                                              // Total: 44 bytes

void print (const Person &k)                    // k is constant reference parameter
{
    cout << "Name: " << k.name << endl;         // name to the screen
    cout << "Num: " << k.reg_num << endl;       // reg_num to the screen
}

int main(){
    Person ahmet;                               // ahmet is a variable of type Person
    strcpy(ahmet.name,"Ahmet Bilir");           // name = "Ahmet Bilir"
    ahmet.reg_num=324;                          // reg_num= 324
    print(ahmet);                               // Function call
    return 0;
}
```

Instead of 44 bytes only 4 bytes (address) are sent to the function.

# Return by reference

►By default in C++, when a function returns a value: return *expression*; *expression* is evaluated and its value is copied into stack. The calling function reads this value from stack and copies it into its variables.

►An alternative to "return by value" is "return by reference", in which the value returned is not copied into stack.

►One result of using "return by reference" is that the function which returns a parameter by reference can be used on the left side of an assignment statement.

```
int&  max( const int a[], int length) {    // Returns an integer reference
    int i=0;                               // indices of the largest element
    for (int j=0 ; j<length ; j++)
        if (a[j] > a[i])    i = j;
    return a[i];                                   // returns reference to a[i]
}
int main()  {
   int array[ ] = {12, -54 , 0 , 123, 63};         // An array with 5 elements
   max(array,5) = 0;                               // write 0 over the largest element
   :
```

# const return parameter

To prevent the calling function from changing the return parameter accidentally, const qualifier can be used.

```
const int&  max( int a[ ], int length)   // Can not be used on the left side of an
{                                        // assignment statement
      int i=0;                           // indices of the largest element
      for (int j=0 ; j<length ; j++)
           if (a[j] > a[i])    i = j;
      return a[i];
}
```

This function can only be on right side of an assignment

```
int main()
{
    int array[ ] = {12, -54 , 0 , 123, 63};      // An array with 5 elements
    int largest;                                 // A variable to hold the largest elem.
    largest = max(array,5);                      // find the largest element
    cout << "Largest element is " << largest << endl;
    return 0;
}
```

# Never return a local variable by reference!

- Since a function that uses "return by reference" returns an actual memory address, it is important that the variable in this memory location remain in existence after the function returns.
- When a function returns, local variables go out of existence and their values are lost.

```
int&  f( ) {                    // Return by reference
    int i;                      // Local variable. Created in stack
    :
    return  i;                  // ERROR! i does not exist anymore.
}
```

Local variables can be returned by their values

```
int  f( ) {                     // Return by value
    int i;                      // Local variable. Created in stack
    :
    return  i;                  // OK.
}
```

# new/delete

►In ANSI C, dynamic memory allocation is normally performed with standard library functions *malloc* and *free*.

►The C++ *new* and *delete* operators enable programs to perform dynamic memory allocation more easily.

►The most basic example of the use of these operators is given below. An int pointer variable is used to point to memory which is allocated by the operator new. This memory is later released by the operator delete.

in C:
```
int *p ;
p = (int *) malloc(N*sizeof(int)) ;
free(p) ;
```

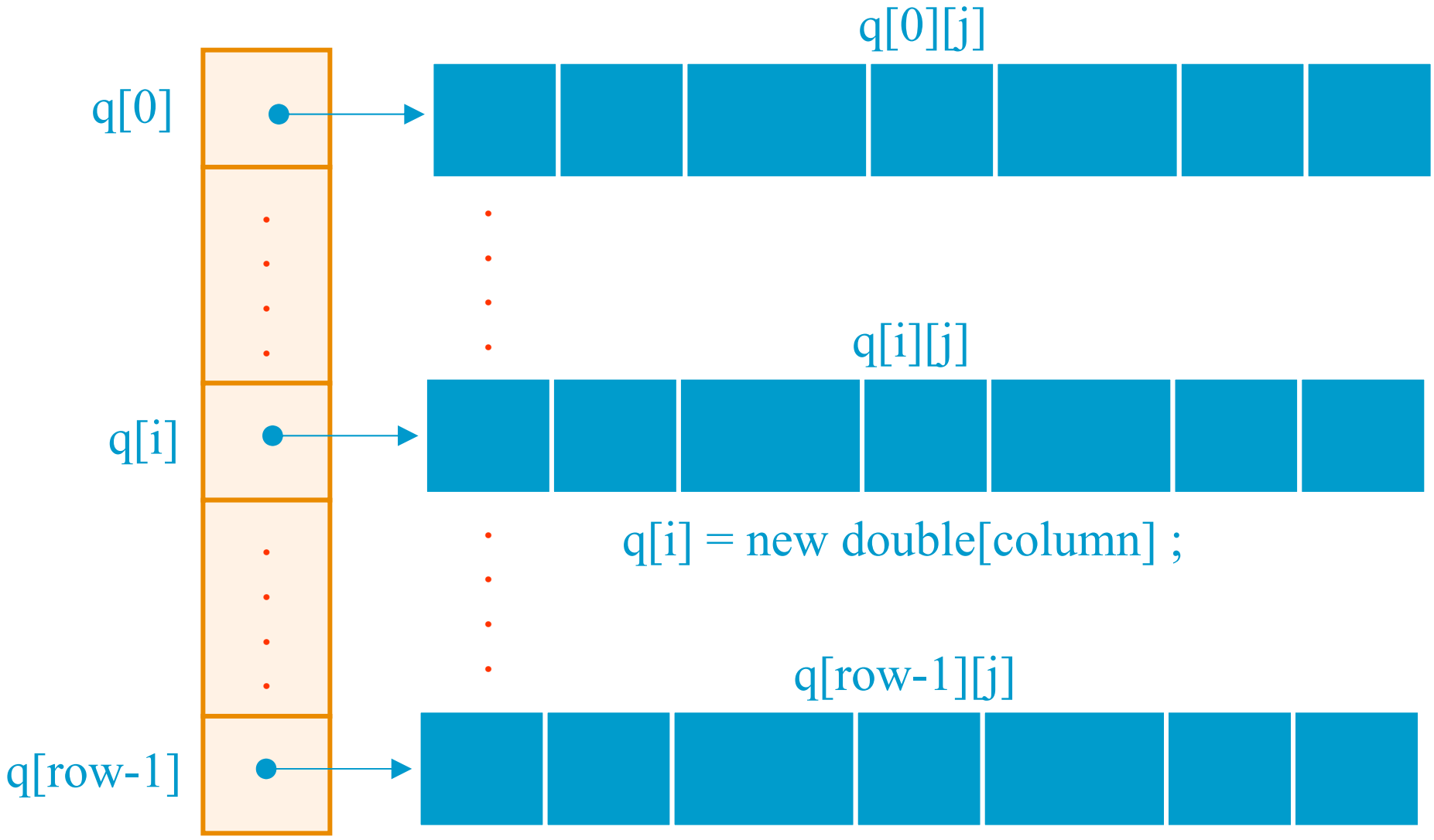
in C++:
```
int *p ;
p = new int[N] ;
delete []p ;
```

```
int *p,*q ;
p = new int[9] ;
q = new int(9) ;
```

## ▶ Two Dimensional Array

1. 
```
double ** q ;
q = new double*[row] ; // matrix size is rowxcolumn
for(int i=0;i<row;i++)
    q[i] = new double[column] ;
.....
for(int i=0;i<row;i++)
   delete []q[i] ;
delete []q ;
```

$i^{th}$ row $j^{th}$ column: q[*i*][*j*]

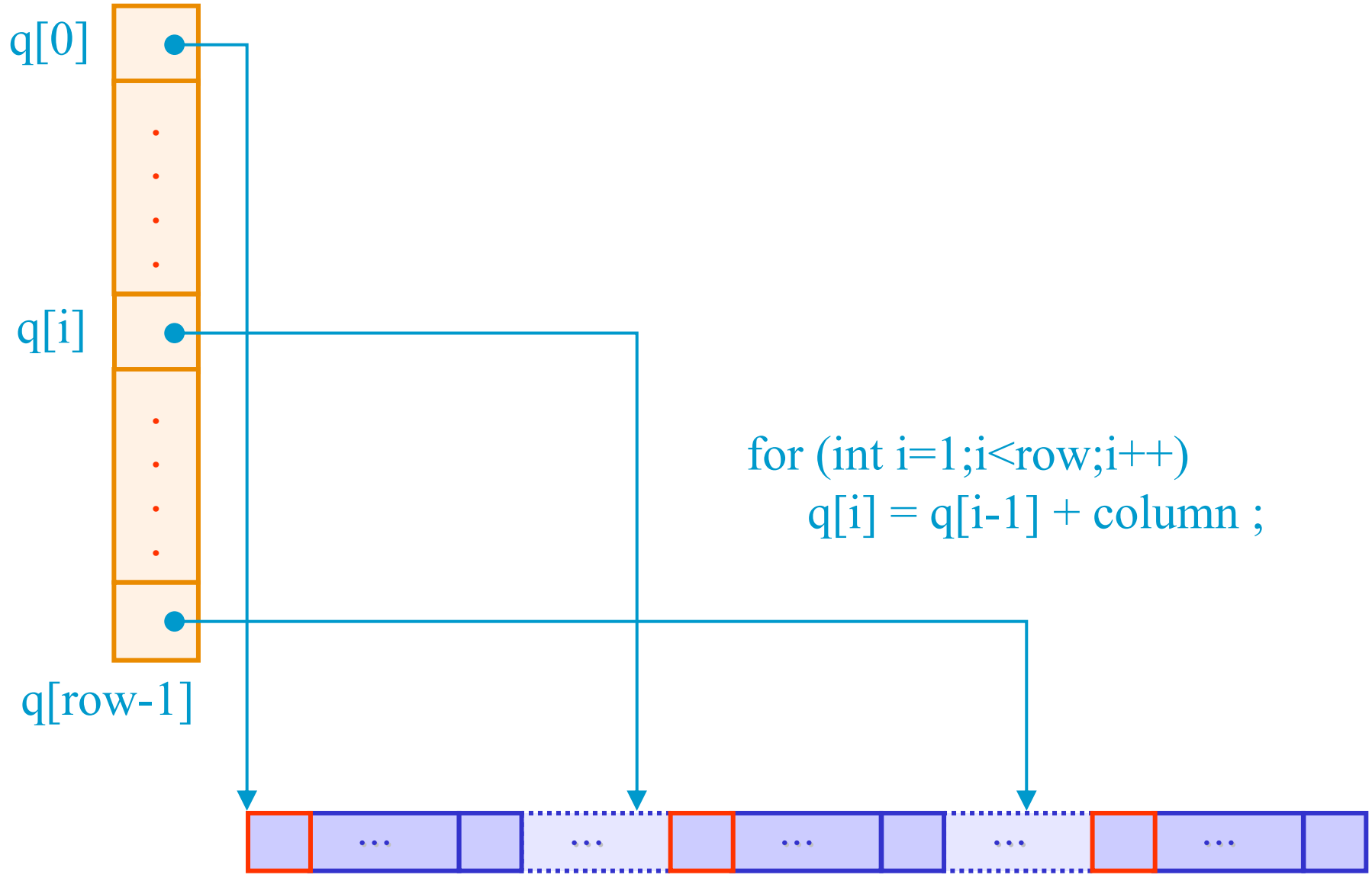

q[0][j]
q[0]
q[i][j]
q[i]
q[i] = new double[column] ;
q[row-1][j]
q[row-1]

## ▶ Two Dimensional Array

2

```
double **q;
p = new double*[row] ; // matrix size is rowxcolumn
q[0] = new double[row*column] ;
for(int i=1;i<row;i++)
    q[i] = q[i-1] + column ;
…..
delete []q[0] ;
delete []q ;
```

$i^{th}$ row $j^{th}$ column: q[*i*][*j*]


q[0]
q[i]
q[row-1]
for (int i=1;i<row;i++)
q[i] = q[i-1] + column ;
q[0] = new double[row*column] ;

```
double ** q ;
memoryAlign  = column % 4;
memoryWidth = ( memoryAlign == 0 ) ?
                        column : (column+4 -memoryAlign)        ;
q[0] = new double[row*memoryWidth] ;
for(int i=0;i<row;i++)
    q[i] = q[i-1] + memoryWidth ;
…..
delete []q[0] ;
delete []q ;
```

# Function Overloading



- Function Overloading

```
double average(const double a[],int size) ;
double average(const int a[],int size) ;
double average(const int a[], const double b[],int size) ;
```

1. 
```
double average(const int a[],int size) {
        double sum = 0.0 ;
       for(int i=0;i<size;i++)   sum += a[i] ;
      return ((double)sum/size) ;
}
```

2. 
```
double average(const double a[],int size) {
        double sum = 0.0 ;
        for(int i=0;i<size;i++)   sum += a[i] ;
        return (sum/size) ;
}
```

3. 
```
double average(const int a[],const double b[],int size) {
        double sum = 0.0 ;
        for(int i=0;i<size;i++)   sum += a[i] + b[i] ;
        return (sum/size) ;
}
```

```
int main() {
        int         w[5]={1,2,3,4,5} ;
        double   x[5]={1.1,2.2,3.3,4.4,5.5} ;
    1   cout << average(w,5) ;
    2   cout << average(x,5) ;
    3   cout << average(w,x,5) ;
        return 0 ;
}
```

# Function Templates



- Function Templates

```
template <typename T>
void printArray(const T *array,const int size){
        for(int i=0;i < size;i++)
            cout << array[i] << " " ;
        cout << endl ;
}
```

```
int main() {
        int         a[3]={1,2,3} ;
        double   b[5]={1.1,2.2,3.3,4.4,5.5} ;
       char       c[7]={‘a’, ‘b’, ‘c’, ‘d’, ‘e’ , ‘f’, ‘g’} ;
  1 printArray(a,3)          ;
  2 printArray(b,5)          ;
  3 printArray(c,7)          ;
  return 0 ;
}
```

```
void printArray(int *array,cont int size){
        for(int i=0;i < size;i++)
            cout << array[i] << "," ;
        cout << endl ;
}
void printArray(char *array,cont int size){
        for(int i=0;i < size;i++)
            cout << array[i] ;
        cout << endl ;
}
```

# Operator Overloading

- In C++ it is also possible to overload the built-in C++ operators such as +, -, = and ++ so that they too invoke different functions, depending on their operands.
- That is, the + in **a**+**b** will add the variables if **a** and **b** are integers, but will call a different function if **a** and **b** are variables of a user defined type.

# Operator Overloading: Rules



- You can't overload operators that don't already exist in C++.
- You can not change numbers of operands. A binary operator (for example +) must always take two operands.
- You can not change the precedence of the operators.

  * comes always before +

- Everything you can do with an overloaded operator you can also do with a function. However, by making your listing more intuitive, overloaded operators make your programs easier to write, read, and maintain.
- Operator overloading is mostly used with objects. We will discuss this topic later more in detail.

# Operator Overloading

►Functions of operators have the name operator and the symbol of the operator. For example the function for the operator + will have the name operator+:

```
struct SComplex{
 float real,img;
};
SComplex operator+(SComplex v1, SComplex v2){
  SComplex result;
  result.real=v1.real+v2.real;
  result.img=v1.img+v2.img;
  return result;
}
```

```
int main(){
  SComplex c1={1,2},c2 ={5,1};
  SComplex c3;
  c3=c1+c2;   // c1+(c2)
}
```

# namespace

►When a program reaches a certain size it's typically broken up into pieces, each of which is built and maintained by a different person or group.

►Since C effectively has a single arena where all the identifier and function names live, this means that all the developers must be careful not to accidentally use the same names in situations where they can conflict.

►The same problem come out if a programmer try to use the same names as the names of library functions.

►Standard C++ has a mechanism to prevent this collision: the namespace keyword. Each set of C++ definitions in a library or program is "wrapped" in a namespace, and if some other definition has an identical name, but is in a different namespace, then there is no collision.

# namespace

```
namespace programmer1{     // programmer1's namespace
    int iflag;             // programmer1's iflag
    void g(int);           // programmer1's g function
    :                      // other variables
}                          // end of namespace

namespace programmer2{     // programmer2's namespace
    int iflag;             // programmer2's iflag
    :
}                          // end of namespace
```

# Accessing Variables

```
programmer1::iflag = 3;          // programmer1's iflag
programmer2::iflag = -345;       // programmer2's iflag
programmer1::g(6);               // programmer1's g function
```

If a variable or function does not belong to any namespace, then it is defined in the global namespace. It can be accessed without a namespace name and scope operator.

This declaration makes it easier to access variables and functions, which are defined in a namespace.

```
using programmer1::iflag;        //  applies to a single item in the namespace
iflag = 3;                       //  programmer1::iflag=3;
programmer2::iflag = -345;
programmer1::g(6);
```

```
using namespace programmer1;     // applies to all elements in the namespace
iflag = 3;                       // programmer1::iflag=3;
g(6);                            // programmer1's function g
programmer2::iflag =  -345;
```

# namespace



```
#include <iostream>
   namespace F {
      float x = 9;
      }
   namespace G {
      using namespace F;
      float y = 2.0;
         namespace INNER_G {
            float z = 10.01;
      }
}
```

```
int main() {
       float x = 19.1;
      using namespace G;
      using namespace G::INNER_G;
      std::cout << "x = " << x << std::endl;
      std::cout << "y = " << y << std::endl;
      std::cout << "z = " << z << std::endl;
      return 0;
}
```

# namespace

```
#include <iostream>
   namespace F {
      float x = 9;
      }
   namespace G {
      using namespace F;
      float y = 2.0;
         namespace INNER_G {
            long x = 5L;
            float z = 10.01;
      }
}
```

```
int main() {
      using namespace G;
      using namespace G::INNER_G;
      std::cout << "x = " << x << std::endl;
      std::cout << "y = " << y << std::endl;
      std::cout << "z = " << z << std::endl;
      return 0;
}
```

# namespace

```
#include <iostream>
   namespace F {
      float x = 9;
      }
   namespace G {
      using namespace F;
      float y = 2.0;
         namespace INNER_G {
            long x = 5L;
            float z = 10.01;
      }
}
```

```
int main() {
      using namespace G;
      std::cout << "x = " << x << std::endl;
      std::cout << "y = " << y << std::endl;
      return 0;
}
```

# namespace

```
#include <iostream>
   namespace F {
      float x = 9;
      }
   namespace G {
            float y = 2.0;
         namespace INNER_G {
            long x = 5L;
            float z = 10.01;
      }
}
```

```
int main() {
      using namespace G;
      std::cout << "x = " << x << std::endl;
      std::cout << "y = " << y << std::endl;
      return 0;
}
```

# namespace

```
#include <iostream>
   namespace F {
      float x = 9;
      }
   namespace G {
            float y = 2.0;
         namespace INNER_G {
            long x = 5L;
            float z = 10.01;
      }
}
```

```
int main() {
      using namespace G::INNER_G;
      std::cout << "x = " << x << std::endl;
      std::cout << "y = " << y << std::endl;
      return 0;
}
```

# Standard C++ Header Files

►In the first versions of C++, mostly '.h' is used as extension for the header files.

►As C++ evolved, different compiler vendors chose different extensions for file names (.hpp, .H , etc.). In addition, various operating systems have different restrictions on file names, in particular on name length. These issues caused source code portability problems.

►To solve these problems, the standard uses a format that allows file names longer than eight characters and eliminates the extension.

►For example, instead of the old style of including iostream.h, which looks like this: #include <iostream.h>, you can now write: #include <iostream>

# Standard C++ Header Files

►The libraries that have been inherited from C are still available with the traditional '**.h**' extension. However, you can also use them with the more modern C++ include style by puting a "**c**" before the name. Thus:

| | | |
|---|---|---|
| #include <stdio.h> | become: | #include <cstdio> |
| #include <stdlib.h> | | #include <cstdlib> |

►In standard C++ headers all declarations and definitions take place in a namespace : **std**

►Today most of C++ compilers support old libraries and header files too. So you can also use the old header files with the extension '.h'. For a high-quality program prefer always the new libraries.

# I/O

- Instead of library functions (printf, scanf), in C++ library objects are used for IO operations.
- When a C++ program includes the **iostream** header, four objects are created and initialized:
  - **cin** handles input from the standard input, the keyboard.
  - **cout** handles output to the standard output, the screen.
  - **cerr** handles unbuffered output to the standard error device, the screen.
  - **clog** handles buffered error messages to the standard error device

# Using `cout` Object

To print a value to the screen, write the word **cout**, followed by the insertion operator (<<).

```
#include<iostream>                    // Header file for the cout object
int main() {
   int i=5;              // integer i is defined, initial value is 5
   float f=4.6;          // floating point number f is defined, 4.6
   std::cout << "Integer Number = " << i << " Real Number= " << f;
   return 0;
}
```

# Using `cin` Object

The predefined cin stream object is used to read data from the standard input device, usually the keyboard. The cin stream uses the >> operator, usually called the "get from" operator.

```
#include<iostream>
using namespace std;  // we don't need std:: anymore
int main() {
   int i,j;  // Two integers are defined
   cout << "Give two numbers \n";  // cursor to the new line
   cin >> i >> j;              // Read i and j from the keyboard
   cout << "Sum= " << i + j << "\n";
   return 0;
}
```

# **std** namespace

```
#include <string>
#include <iostream>
 using namespace std;
 int main() {
   string test;
   while(test.empty() || test.size() <= 5)
   {
    cout << "Type a string longer string. " << endl;
    cin >> test;
   }
```

printf("%s",test.c_str())

# bool Type

The type bool represents boolean (logical) values: true and false

Before **bool** became part of Standard C++, everyone tended to use different techniques in order to produce Boolean-like behavior.

These produced portability problems and could introduce subtle errors.

Because there's a lot of existing code that uses an **int** to represent a flag, the compiler will implicitly convert from an **int** to a **bool** (nonzero values will produce **true** while zero values produce **false**).

Do not prefer to use integers to produce logical values.

```
bool is_greater;          // Boolean variable: is_greater
is_greater = false;       // Assigning a logical value
int a,b;
……….
is_greater = a > b;       // Logical operation
if (is_greater)  ……       // Conditional operation
```

# constant

- In standard C, preprocessor directive #define is used to create constants: #define PI 3.14
- C++ introduces the concept of a named constant that is just like a variable, except that its value cannot be changed.
- The modifier **const** tells the compiler that a name represents a constant:

```
const int MAX = 100;
…
MAX = 5; // Compiler Error!
```

- const can take place before (left) and after (right) the type. They are always (both) allowed and equivalent.

```
int const MAX = 100;  // The same as const int MAX = 100;
```

- Decreases error possibilities.
- To make your programs more readable, use uppercase font for constant identifiers.

# Use of constant–1

Another usage of the keyword const is seen in the declaration of pointers. There are three different cases:

**a)** The data pointed by the pointer is constant, but the pointer itself however may be changed.

```
const char *p = "ABC";
```

p is a pointer variable, which points to chars. The const word may also be written after the type:

```
char const *p = "ABC";
```

Whatever is pointed to by p may not be changed: the chars are declared as const. The pointer p itself however may be changed.

```
*p = 'Z';   // Compiler Error! Because data is constant
p++;    //OK, because the address in the pointer may change.
```

# Use of constant–2

**b)** The pointer itself is a const pointer which may not be changed. Whatever data is pointed to by the pointer may be changed.

```
char * const sp = "ABC"; // Pointer is constant, data may change
*sp = 'Z';          // OK, data is not constant
sp++;               // Compiler Error! Because pointer is constant
```

# Use of constant–3

**c)** Neither the pointer nor what it points to may be changed

The same pointer definition may also be written as follows:

```
char const * const ssp = "ABC";
const char * const ssp = "ABC";
*ssp = 'Z';        // Compiler Error! Because data is constant
ssp++;             // Compiler Error! Because pointer is const
```

►The definition or declaration in which const is used should be read from the variable or function identifier back to the type identifier:

"ssp is a const pointer to const characters"

# Casts

►Traditionally, **C** offers the following *cast* construction:

(*typename*) *expression*

**Example:** f = (float)i / 2;

Following that, C++ initially also supported the *function call style* cast notation:

typename(expression)

**Example:** Converting an integer value to a floating point value

```
int    i=5;
float  f;
f = float(i)/2;
```

►But, these casts are now called *old-style casts*, and they are deprecated. Instead, four *new-style casts* were introduced.

# Casts: static_cast



▶The static_cast<type>(expression) operator is used to convert one type to an acceptable other type.

```
int     i=5;
float  f;
f = static_cast<float>(i)/2;
```

# Casts: const_cast

►The const_cast<type>(expression) operator is used to do away with the const-ness of a (pointer) type.

►In the following example p is a pointer to constant data, and q is a pointer to non-constant data. So the assignment q = p is not allowed.

```
const char *p = "ABC";    // p points to constant data
char       *q;            // data pointed by q may change
q = p;      // Compiler Error! Constant data may change
```

If the programmer wants to do this assignment on purpose then he/she must use the const_cast operator:

```
q = const_cast<char *>(p);
*q = 'X';    // Dangerous?
```

# Casts: reinterpret_cast

The reinterpret_cast<type>(expression) operator is used to reinterpret byte patterns. For example, the individual bytes making up a structure can easily be reached using a reinterpret_cast

```
struct S{                    // A structure
  int i1,i2;                 // made of two integers
};
int main(){
  S   x;                     // x is of type S
  x.i1=1;                    // fields of x are filled
  x.i2=2;
  unsigned char *xp;         // A pointer to unsigned chars
  xp = reinterpret_cast<unsigned char *> (&x);
  for (int j=0; j<8; j++)    // bytes of x on the screen
    std::cout <<  static_cast<int>(*xp++);
  return 0;
}
```

The structure S is made of two integers (2x4=8 bytes). x is a variable of type S. Each byte of x can be reached by using the pointer xp.

# Casts: dynamic_cast

The dynamic_cast<>() operator is used in the context of inheritance and polymorphism. We will see these concepts later. The discussion of this cast is postponed until the section about polymorphism.

- Using the cast-operators is a dangerous habit, as it suppresses the normal type-checking mechanism of the compiler.
- It is suggested to prevent casts if at all possible.
- If circumstances arise in which casts have to be used, document the reasons for their use well in your code, to make double sure that the cast is not the underlying cause for a program to misbehave.

# 3 OO Programming Concepts

# Content

- OOP Concepts
  - Class
    - Encapsulation
    - Information Hiding
  - Inheritance
  - Polymorphism

# OOP Concepts

- When you approach a programming problem in an object-oriented language, you no longer ask how the problem will be divided into functions, but **how it will be divided into objects**.
- Thinking in terms of objects rather than functions has a helpful effect on how easily you can design programs. Because **the real world consists of objects** and there is a close match between objects in the programming sense and objects in the real world.

# What is an Object?

- Many real-world objects have both **a state** (characteristics that can change) and **abilities** (things they can do).
- Real-world object=State (properties)+ Abilities (behavior)
- Programming objects = Data + Functions
- The match between programming objects and real-world objects is the result of combining data and member functions.
- How can we define an object in a C++ program?

# Classes and Objects

- **Class** is a new data type which is used to define objects. A class serves as a plan, or a template. It specifies what data and what functions will be included in objects of that class. Writing a class doesn't create any objects.
- A class is a description of similar objects.
- **Object**s are instances of classes.

# Example

A model (class) to define points in a graphics program.

- Points on a plane must have two properties (states):
  - **x** and **y** coordinates. We can use two integer variables to represent these properties.
- In our program, points should have the following abilities (behavior):
  - Points can move on the plane: **move** function
  - Points can show their coordinates on the screen: **print** function
  - Points can answer the question whether they are on the zero point (0,0) or not: **is_zero** function

# Class Definition: Point

```
class Point {                  // Declaration of Point Class
    int x,y;                   // Properties: x and y coordinates
  public:                      // We will discuss it later
    void move(int, int);       // A function to move the points
    void print();              // to print the coordinates on the screen
    bool is_zero();            // is the point on the zero point(0,0)
};                             // End of class declaration (Don't forget ;)
```

- In our example first data and then the function prototypes are written.
- It is also possible to write them in reverse order.
- Data and functions in a class are called **member**s of the class.
- In our example only the prototypes of the functions are written in the class declaration. The bodies may take place in other parts (in other files) of the program.
- If the body of a function is written in the class declaration, then this function is defined as an inline function (macro).

# Bodies of Member Functions

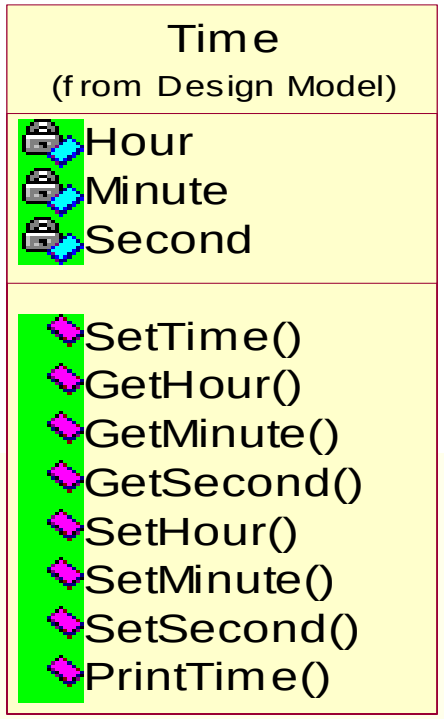
```
// A function to move the points
void Point::move(int new_x, int new_y) {
  x = new_x;          // assigns new value to x coordinate
  y = new_y;          // assigns new value to y coordinate
}
// To print the coordinates on the screen
void Point::print() {
  cout << "X= " << x << ", Y= " << y << endl;
}
// is the point on the zero point(0,0)
bool Point::is_zero() {
  return (x == 0) && (y == 0); // if x=0 & y=0 returns true
}
```

- Now we have a model (template) to define point objects. We can create necessary points (objects) using the model.

```
int main() {
    Point point1, point2;  // 2 object are defined: point1 and point2
    point1.move(100,50); // point1 moves to (100,50)
    point1.print();            // point1's coordinates to the screen
    point1.move(20,65);  // point1 moves to (20,65)
    point1.print();            // point1's coordinates to the screen
    if( point1.is_zero() )  // is point1 on (0,0)?
       cout << "point1 is now on zero point(0,0)" << endl;
    else   cout << "point1 is NOT on zero point(0,0)" << endl;
    point2.move(0,0);      // point2 moves to (0,0)
    if( point2.is_zero() )  // is point2 on (0,0)?
       cout << "point2 is now on zero point(0,0)" << endl;
    else  cout << "point2 is NOT on zero point(0,0)" << endl;
    return 0;
}
```

```
class Time {
   int hour;
   int minute;
   int second ;
 public:
// Get Functions
   int    GetHour(){return hour;} ;
   int    GetMinute(){return minute;} ;
   int    GetSecond (){return second;} ;
// Set Functions
   void SetTime(int h,int m,int s){hour=h;minute=m;second=s;};
   void SetHour(int h){hour= (h>=0 && h<24) ? h : 0;} ;
   void SetMinute(int m){minute= (m>=0 && m<60) ? m : 0;} ;
   void SetSecond(int s){second= (s>=0 && s<60) ? s : 0;} ;
   void PrintTime();
};
```

UML Class Diagram



# C++ Terminology

- A **class** is a grouping of data and functions. A class is very much like a structure type as used in ANSI-C, it is only a pattern to be used to create a variable which can be manipulated in a program.
- An **object** is an instance of a class, which is similar to a variable defined as an instance of a type. An object is what you actually use in a program.
- A **method (member function)** is a function contained within the class. You will find the functions used within a class often referred to as methods in programming literature.
- A **message** is the same thing as a function call. In object oriented programming, we send messages instead of calling functions. For the time being, you can think of them as identical. Later we will see that they are in fact slightly different.

# Conclusion

- Until this slide we have discovered some features of the object-oriented programming and the C++.
- Our programs consist of object as the real world do.
- Classes are living (active) data types which are used to define objects. We can send messages (orders) to objects to enable them to do something.
- Classes include both data and the functions involved with these data (*encapsulation*). As the result:
- Software objects are similar to the real world objects,
- Programs are easy to read and understand,
- It is easy to find errors,
- It supports modularity and teamwork.

# Defining Methods as inline Functions



- In the previous example (Example 3.1), only the prototypes of the member functions are written in the class declaration. The bodies of the methods are defined outside the class.
- It is also possible to write bodies of methods in the class. Such methods are defined as inline functions.
- For example the is_zero method of the Point class can be defined as an inline function as follows:

```
class Point{                    // Declaration of Point Class
   int x,y;                     // Properties: x and y coordinates
 public:
   void move(int, int);         // A function to move the points
   void print();        // to print the coordinates on the screen
   bool is_zero() {  // is the point on the zero point(0,0) inline function
       return (x == 0) && (y == 0);  // the body of is_zero
   }
};
```

# Defining Dynamic Objects

- Classes can be used to define variables like built-in data types (int, float, char etc.) of the compiler.
- For example it is possible to define pointers to objects. In the example below two pointers to objects of type Point are defined.

```
int main()  {
    Point *ptr1 = new Point;  // allocating memory for the object pointed by ptr1
    Point *ptr2 = new Point;  // allocating memory for the object pointed by ptr2
    ptr1->move(50, 50);       // 'move' message to the object pointed by ptr1
    ptr1->print();            //'print' message to the object pointed by ptr1
    ptr2->move(100, 150);     // 'move' message to the object pointed by ptr2
    if( ptr2->is_zero() )     // is the object pointed by ptr2 on zero
        cout << " Object pointed by ptr2 is on zero." << endl;
    else cout << " Object pointed by ptr2 is NOT on zero." << endl;
    delete ptr1;                        // Releasing the memory
    delete ptr2;
    return 0;
}
```

# Defining Array of Objects

- We may define static and dynamic arrays of objects. In the example below we see a static array with ten elements of type Point.
- We will see later how to define dynamic arrays of objects.

```
int main()
{
   Point   array[10];          // defining an array with ten objects
   array[0].move(15, 40);  // 'move' message to the first element (indices 0)
   array[1].move(75, 35); // 'move' message to the second element (indices 1)
      :                          // message to other elements
   for (int i = 0; i < 10; i++)  // 'print' message to all objects in the array
         array[i].print();
   return 0;
}
```

# Controlling Access to Members

- We can divide programmers into two groups: class creators (those who create new data types) and client programmers (the class consumers who use the data types in their applications).
- The goal of the class creator is to build a class that includes all necessary properties and abilities. The class should expose only what's necessary to the client programmer and keeps everything else hidden.
- The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development.
- The first reason for access control is to keep client programmers' hands off portions they shouldn't touch. The hidden parts are only necessary for the internal machinations of the data type but not part of the interface that users need in order to solve their particular problems.

# Controlling Access to Members *Con't*

- The second reason for access control is that, if it's hidden, the client programmer can't use it, which means that the class creator can change the hidden portion at will without worrying about the impact to anyone else.
- This protection also prevents accidentally changes of states of objects.

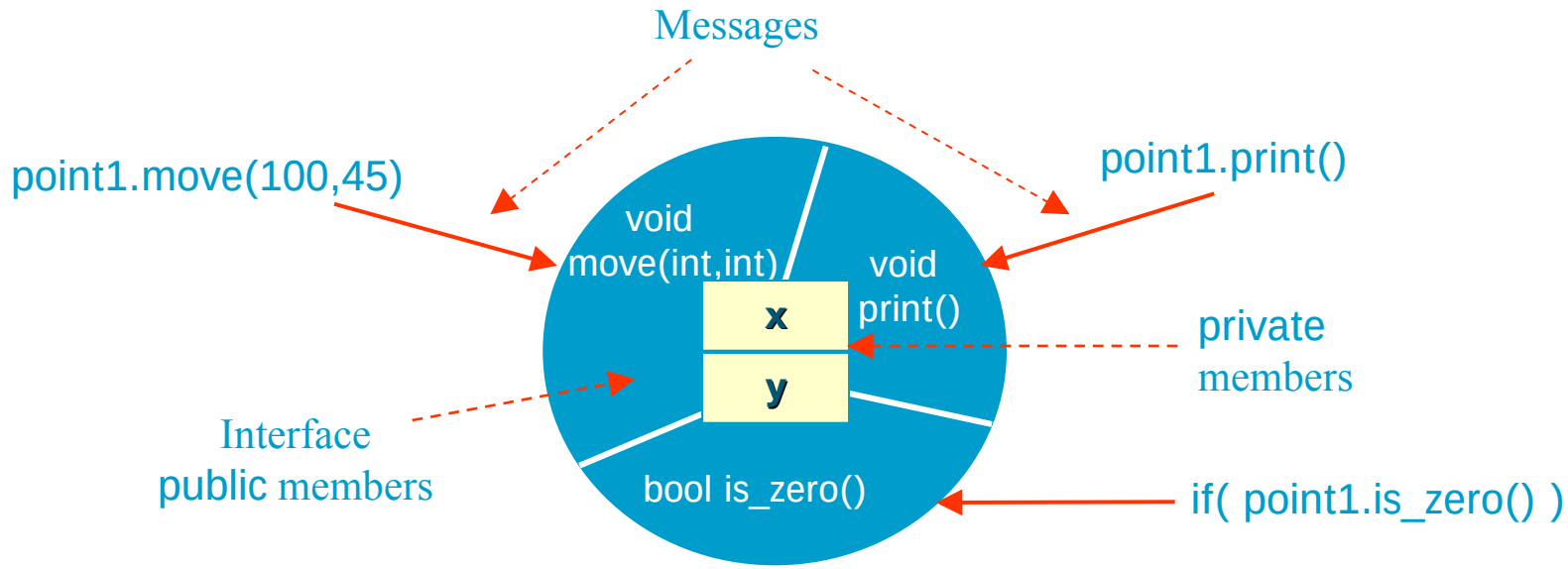
# Controlling Access to Members *Con't*



- The labels public: , private: (and protected: as we will see later) are used to control access to a class' data members and functions.
- Private class members can be accessed only by members of that class.
- Public members may be accessed by any function in the program.
- The default access mode for classes is private: After each label, the mode that was invoked by that label applies until the next label or until the end of class declaration.

# Controlling Access to Members *Con't*

- The primary purpose of public members is to present to the class's clients a view of the services the class provides. This set of services forms the public interface of the class.
- The private members are not accessible to the clients of a class. They form the implementation of the class.



- Example: We modify the move function of the class Point. Clients of this class can not move a point outside a window with a size of 500x300.

```
class Point{                    // Point Class
    int x,y;                    // private members: x and y coordinates
  public:                                 // public members
    bool move(int, int);   // A function to move the points
    void print();          // to print the coordinates on the screen
    bool is_zero();        // is the point on the zero point(0,0)
};
// A function to move the points (0,500 x 0,300)
bool Point::move(int new_x, int new_y) {
   if( new_x > 0 &&  new_x < 500 &&              // if new_x is in 0-500
       new_y > 0 &&  new_y < 300) {      // if new_y is in 0-300
       x = new_x;                                // assigns new value to x coordinate
       y = new_y;                                // assigns new value to y coordinate
       return true;                              // input values are accepted
   }
   return false;                                 // input values are not accepted
}
```

- The new move function returns a boolean value to inform the client programmer whether the input values are accepted or not.
- Here is the main function:

```
int main()  {
     Point p1;        // p1 object is defined
     int x,y;         // Two variables to read some values from the keyboard
     cout << " Give x and y coordinates “;
     cin >> x >> y;       // Read two values from the keyboard
     if( p1.move(x,y) )    // send move message and check the result
        p1.print();        // If result is OK print coordinates on the screen
     else  cout << "\nInput values are not accepted";
}
```

It is not possible to assign a value to x or y directly outside the class.

```
p1.x = -10;    //ERROR! x is private
```

# ***struct*** Keyword in C++

- ***class*** and ***struct*** keywords have very similar meaning in the C++.
- They both are used to build object models.
- The only difference is their default access mode.
- The default access mode for class is ***private***
- The default access mode for struct is ***public***

# Friend Functions and Friend Classes

- A function or an entire class may be declared to be a friend of another class.
- A friend of a class has the right to access all members (private, protected or public) of the class.

```
class A{
  friend class B;            // Class B is a friend of class A
  private:                   // private members of A
     int i;
     float f;
  public:                 // public members of A
     void fonk1(char *c);
};
class B{                     // Class B
    int j;
  public:
    void fonk2(A &s) { cout << s.i; } // B can access private members of A
};
```

In this example, A is not a friend of B. A can not access private members of B.

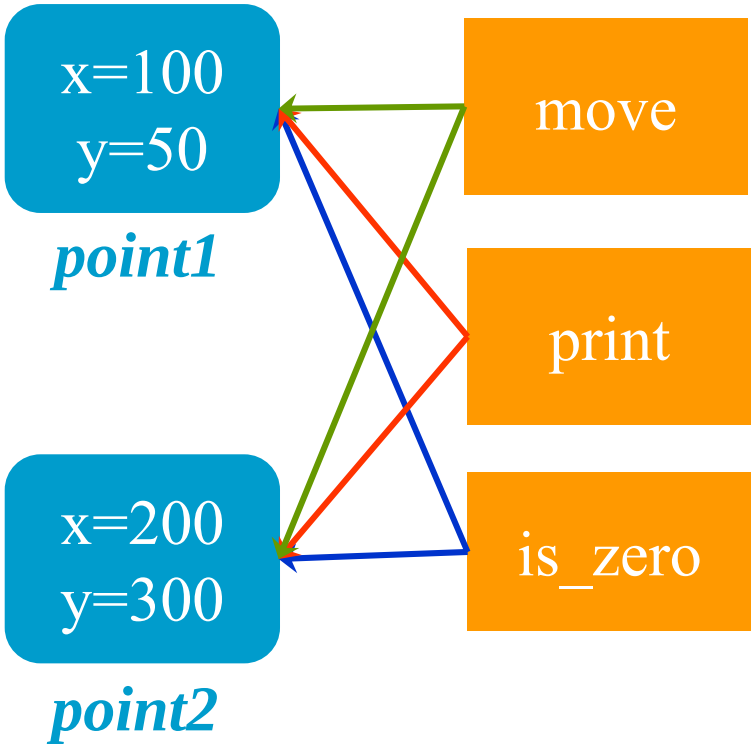
# Friend Functions and Friend Classes *Con't*

- A friend function has the right to access all members (private, protected or public) of the class.

```
class Point{                          // Point Class
    friend void zero(Point &);        // A friend function of Point
     int x,y;                         // private members: x and y coordinates
  public:                             // public members
     bool move(int, int);             // A function to move the points
     void print();                    // to print the coordinates on the screen
     bool is_zero();                  // is the point on the zero point(0,0)
};

// Assigns zero to all coordinates
void zero(Point &p)                   // Not a member of any class
{
   p.x = 0;                           // assign zero to x of p
   p.y = 0;                           // assign zero to y of p
}
```

# *this* Pointer

- Each object has its own data space in the memory of the computer. When an object is defined, memory is allocated only for its data members.
- The code of member functions are created only once. Each object of the same class uses the same function code.




- How does C++ ensure that the proper object is referenced?
- C++ compiler maintains a pointer, called the this pointer.

- A C++ compiler defines an object pointer this. When a member function is called, this pointer contains the address of the object, for which the function is invoked. So member functions can access the data members using the pointer this.
- Programmers also can use this pointer in their programs.
- **Example:** We add a new function to Point class: far_away.

This function will return the address of the object that has the largest distance from (0,0).

```
Point *Point::far_away(Point &p) {
   unsigned long x1 = x*x;                          // x1 = x^2
   unsigned long y1 = y*y;                          // y1 = y^2
   unsigned long x2 = p.x * p.x;
   unsigned long y2 = p.y * p.y;
   if ( (x1+y1) > (x2+y2) ) return this;     // Object returns its address
      else return &p;                          // The address of the incoming object
}
```

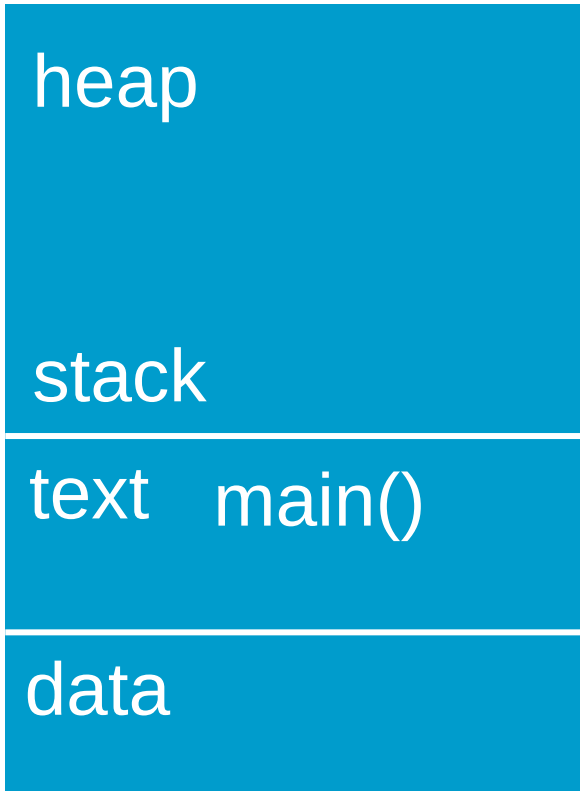
- this pointer can also be used in the methods if a parameter of the method has the same name as one of the members of the class.

```
class Point{                    // Point Class
    int x,y;                    // private members: x and y coordinates
  public:                       // public members
    bool move(int, int);        // A function to move the points
          :                     // other methods are omitted
};
// A function to move the points (0,500 x 0,300)
bool Point::move(int x, int y)      // paramters has the same name as
{                                   // data members x and y
        if( x > 0 &&  x < 500 &&          // if given x is in 0-500
            y > 0 &&  y < 300) {    // if given y is in 0-300
            this->x = x;            // assigns given x value to member x
            this->y = y;            // assigns given y value to memeber y
            return true;            // input values are accepted
        }
        return false;               // input values are not accepted
}
```

# Summary



## Process Model



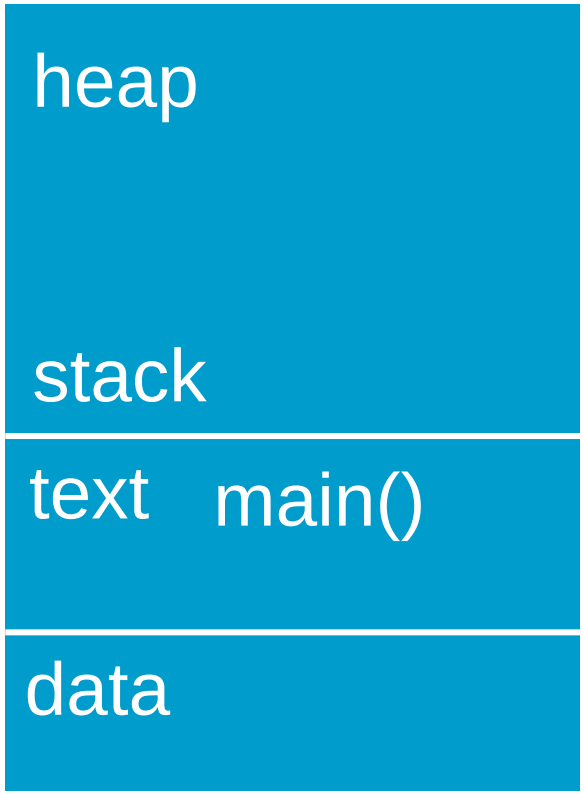

Point o;


o.move(1,1);

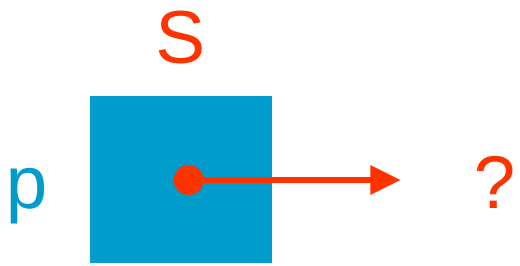
# Summary



## Process Model



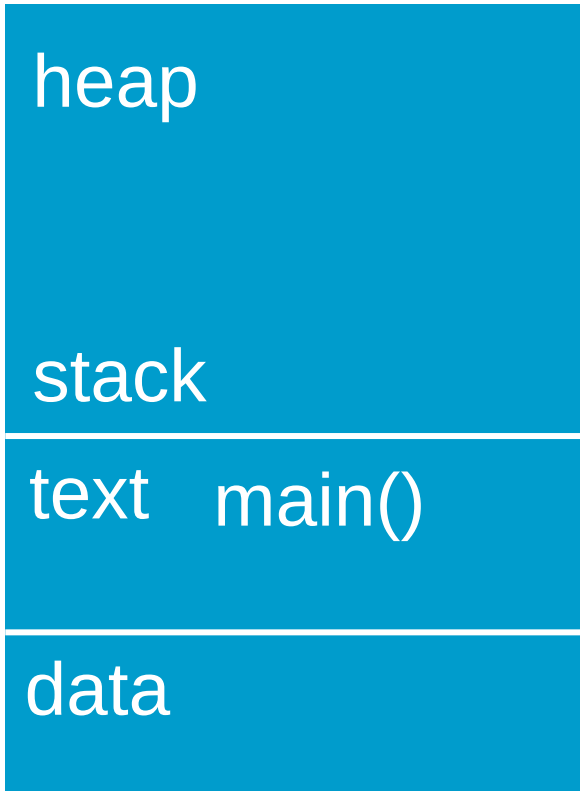

Point *p;



# Summary

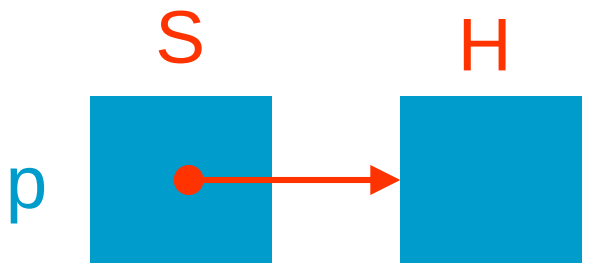


## Process Model




Point *p;

p= new Point();



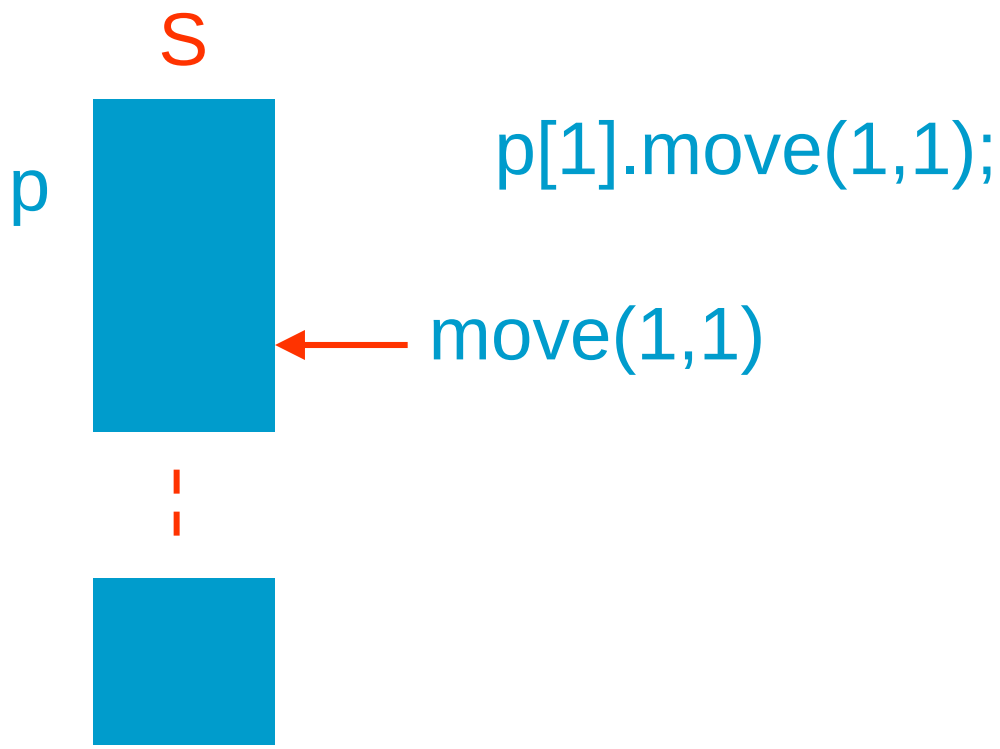

p->move(1,1);

(*p).move(1,1);

# Summary

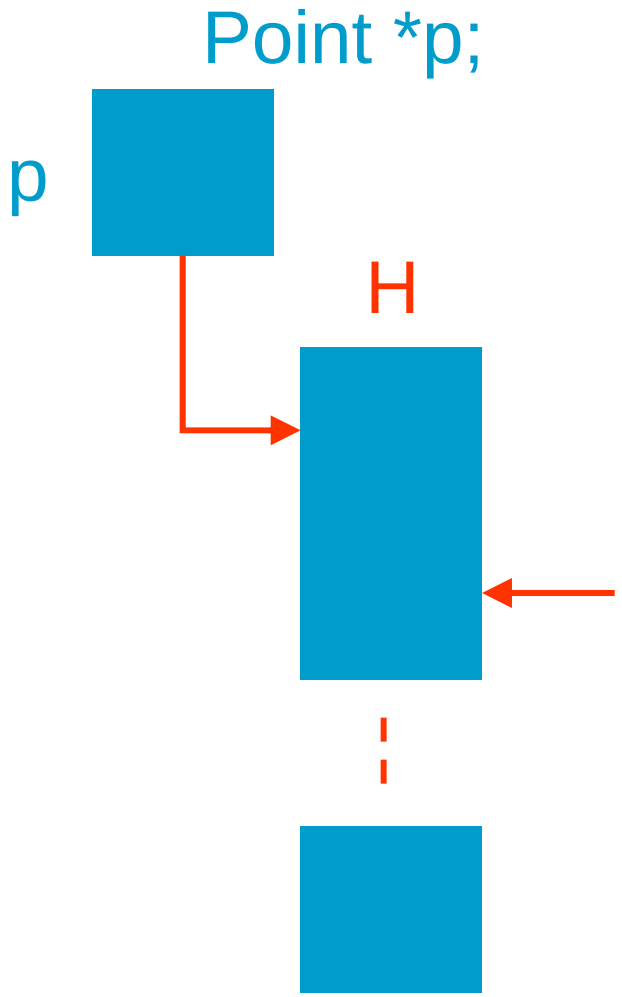


Point p[10];



# Summary






p= new Point[10];

p[1].move(1,1);

(*(p+1)).move(1,1);

(p+1)->move(1,1);

# 4 Initializing and Finalizing Objects

# Content

- Constructors
  - Default Constructor
  - Copy Constructor
- Destructor

# Initializing Objects: Constructors

- The class designer can guarantee initialization of every object by providing a special member function called the constructor.
- The constructor is invoked automatically each time an object of that class is created (instantiated).
- These functions are used to (for example) assign initial values to the data members, open files, establish connection to a remote computer etc.
- The constructor can take parameters as needed, but it cannot have a return value (even not void).

# Initializing Objects: Constructors

- The constructor has the same name as the class itself.
- Constructors are generally public members of a class.
- There are different types of constructors.
- For example, a constructor that defaults all its arguments or requires no arguments, i.e. a constructor that can be invoked with no arguments is called default constructor.
- In this section we will discuss different kinds of constructors.

# Default Constructors

- A constructor that defaults all its arguments or requires no arguments, i.e. a constructor that can be invoked with no arguments.

```
class Point{                          // Declaration Point Class
   int x,y;                           // Properties: x and y coordinates
 public:
   Point();                           // Declaration of the default constructor
   bool move(int, int);               // A function to move points
   void print();                      // to print coordinates on the screen
};

Point::Point() {        // Default Constructor
   cout << "Constructor is called..." << endl;
   x = 0;  // Assigns zero to coordinates
   y = 0;
}

int main() {
   Point p1, p2;                      // Default construct is called 2 times
   Point *pp = new Point;             // Default construct is called once
```

# Constructors with Parameters

- Like other member functions, constructors may also have parameters.
- Users of the class (client programmer) must supply constructors with necessary arguments.

```
class Point{                          // Declaration Point Class
   int x,y;                           // Properties: x and y coordinates
 public:
   Point(int, int);                   // Declaration of the constructor
   bool move(int, int);               // A function to move points
   void print();                      // to print coordinates on the screen
};
```

- This declaration shows that the users of the Point class have to give two integer arguments while defining objects of that class.

# Example: Constructors with Parameters

```
Point::Point(int x_first, int y_first) {
    cout << "Constructor is called..." << endl;
    if ( x_first < 0 )                  // If the given value is negative
        x = 0;                          // Assigns zero to x
    else
        x = x_first;
    if ( y_first < 0 )                  // If the given value is negative
        y = 0;                          // Assigns zero to x
    else
        y = y_first;
}
// -------- Main Program ------------
int main() {
    Point p1(20, 100), p2(-10, 45);         // Construct is called 2 times
    Point *pp = new Point(10, 50);          // Construct is called once
    Point p3;       // ERROR! There is not a default constructor
    :
}
```

# Constructor Parameters with Default Values

- Constructors parameters may have default values

```
class Point{
  public:
     Point(int x_first = 0,  int y_first = 0);
     :
};
Point::Point(int x_first,  int y_first) {
    if ( x_first < 0 )          // If the given value is negative
       x = 0;                   // Assigns zero to x
    else            x = x_first;
    if ( y_first < 0 )          // If the given value is negative
       y = 0;                   // Assigns zero to x
    else    y = y_first;
}
```

- Now, client of the class can create objects

```
Point p1(15,75);                // x=15, y=75
Point p2(100);                  // x=100, y=0
```

- This function can be also used as a default constructor

```
Point p3;    // x=0, y=0
```

# Multiple Constructors

- The rules of function overloading is also valid for constructors. So, a class may have more than one constructor with different type of input parameters.

```
Point::Point() {                                    // Default constructor
   ..............                                   // Body is not important
}

Point::Point(int x_first, int y_first) { // A constructor with parameters
   ..................                               // Body is not important
}
```

- Now, the client programmer can define objects in different ways:

```
Point p1;                 // Default constructor is called
Point p2(30, 10);         // Constructor with parameters is called
```

- The following statement causes an compiler error, because the class does not include a constructor with only one parameter.

```
Point p3(10);             //ERROR! There isn't a constructor with one parameter
```

# Initializing Arrays of Objects

►When an array of objects is created, the default constructor of the class is invoked for each element (object) of the array one time.

```
Point array[10]; // Default constructor is called 10 times
```

►To invoke a constructor with arguments, a list of initial values should be used.

►To invoke a constructor with more than one arguments, its name must be given in the list of initial values, to make the program more readable.

# Initializing Arrays of Objects *Con't*

- // *Constructor*

```
Point(int x_first,  int y_first = 0)  { .... }
// Can be called with one or two args
```

- // *Array of Points*

```
Point array[]= { {10} , {20} , Point(30,40) };
```

- Three objects of type Point has been created and the constructor has been invoked three times with different arguments.

| Objects: | Arguments: |
|---|---|
| array[0] | x_first = 10 , y_first = 0 |
| array[1] | x_first = 20 , y_first = 0 |
| array[2] | x_first = 30 , y_first = 40 |

# Initializing Arrays of Objects *Con't*

- If the class has also a default constructor the programmer may define an array of objects as follows:

  Point array[5]= { {10} , {20} , Point(30,40) };

- Here, an array with 5 elements has been defined, but the list of initial values contains only 3 values, which are sent as arguments to the constructors of the first three elements. For the last two elements, the default constructor is called.
- To call the default constructor for an object, which is not at the end of the array

  Point array[5]= { {10} , {20}, Point() , Point(30,40) };

- Here, for objects array[2] and array[4] the default constructor is invoked.

  Point array[5]= { {10} , {20} , , Point(30,40) }; // ERROR!

# Constructor Initializers

- Instead of assignment statements constructor initializers can be used to initialize data members of an object.
- Specially, to assign initial value to a constant member using the constructor initializer is the only way.
- Consider the class:

```
class C{
    const int CI;
    int x;
 public:
    C( ) {
        x = 0;
        CI = 0;
    }
};
```

```
class C{
    const int CI = 10 ;
    int x;
};
```

# Solution

The solution is to use a constructor initializer.

```
class C{
    const int CI;
    int x;
  public:
    C() : CI(0) {
          x = -2;
    }
};
```

```
class C{
    const int CI;
    int x;
  public:
    C( int a ) : CI(0), x (a)
      { }
};
```

All data members of a class can be initialized by using constructor initializers.

# Destructors

- The destructor is very similar to the constructor except that it is called automatically
  1. when each of the objects goes out of scope or
  2. a dynamic object is deleted from memory by using the delete operator.
- A destructor is characterized as having the same name as the class but with a tilde '~' preceded to the class name.
- A destructor has no return type and receives no parameters.
- A class may have only one destructor.

# Example

```
class String{
    int size;              // Length (number of chars) of the string
    char *contents;        // Contents of the string
 public:
   String(const char *); // Constructor
   void print();           // An ordinary member function
  ~String();               // Destructor
};
```

- Actually, the standard library of C++ contains a **string** class. Programmers don't need to write their own string class. We write this class only to show some concepts.

```
// Constructor : copies the input character array that terminates with a null character
// to the contents of the string
String::String(const char *in_data) {
    cout<< "Constructor has been invoked" << endl;
    size = strlen(in_data);              // strlen is a function of the cstring library
    contents = new char[size +1];        // +1 for null ( '\0' ) character
    strcpy(contents, in_data);           // input_data is copied to the contents
 }
void String::print() {
    cout << contents << " " << size << endl;
}
// Destructor: Memory pointed by contents is given back
String::~String() {
    cout << "Destructor has been invoked" << endl;
    delete[] contents;
}
```

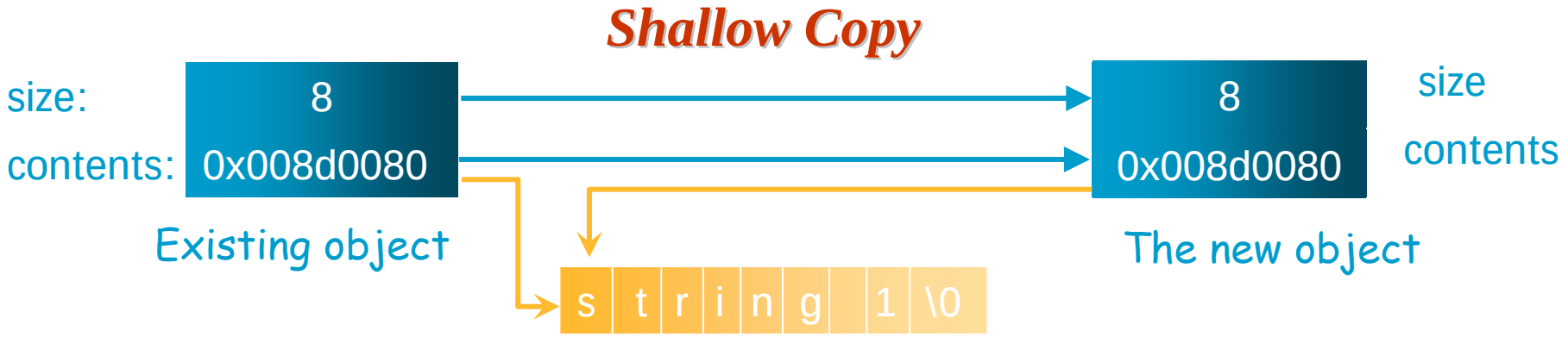
```
int main() {
    String string1("string 1");
    String string2("string 2");
    string1.print();
    string2.print();
    return 0; // destructor is called twice
}
```

# Copy Constructor

- It is a special type of constructors and used to copy the contents of an object to a new object during construction of that new object.
- The type of its input parameter is a reference to objects of the same type. It takes as argument a reference to the object that will be copied into the new object.
- The copy constructor is generated automatically by the compiler if the class author fails to define one.
- If the compiler generates it, it will simply copy the contents of the original into the new object as a byte by byte copy.
- For simple classes with no pointers, that is usually sufficient, but if there is a pointer as a class member so a byte by byte copy would copy the pointer from one to the other and they would both be pointing to the same allocated member.

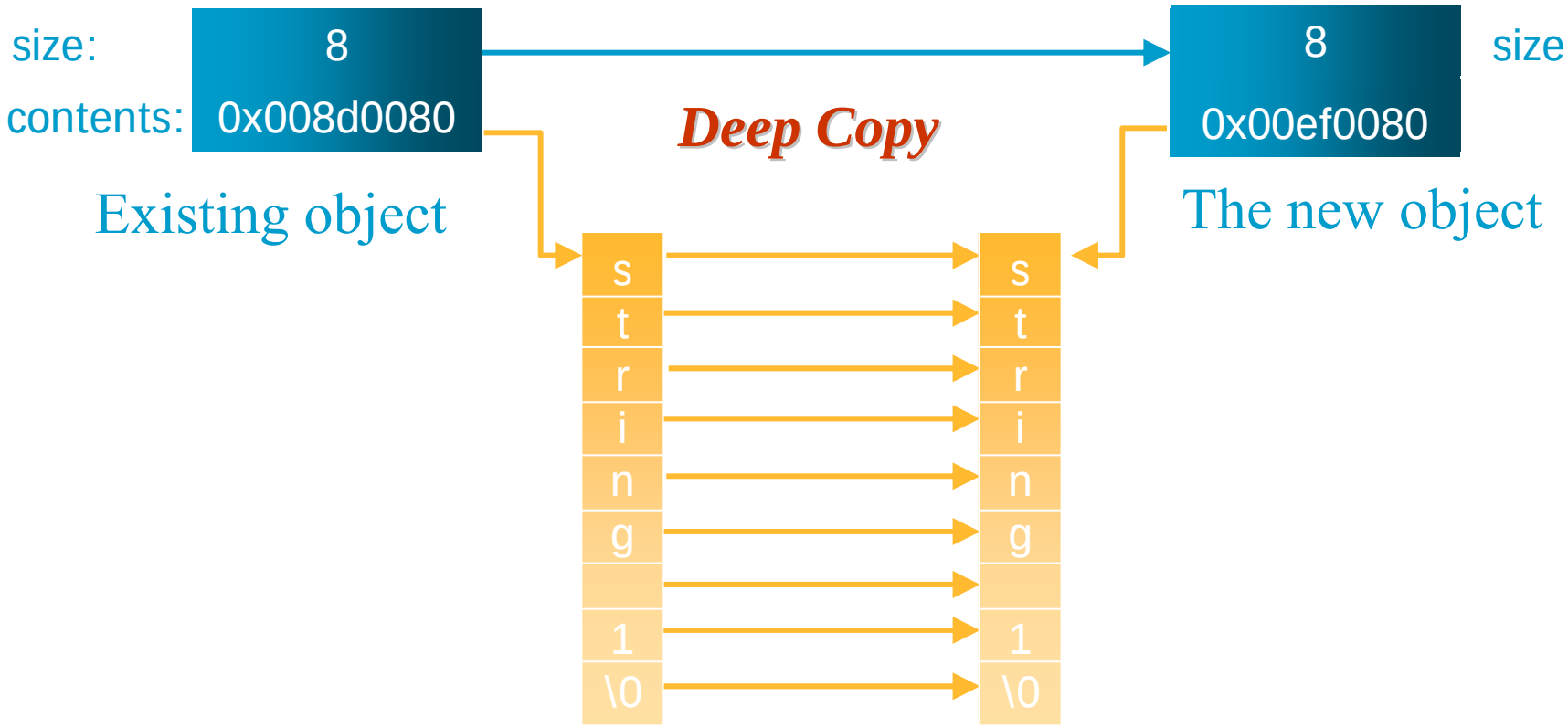
# Copy Constructor *Con't*

- For example the copy constructor, generated by the compiler for the String class will do the following job:



# Copy Constructor *Con't*

- The copy constructor, generated by the compiler can not copy the memory locations pointed by the member pointers.
- The programmer must write its own copy constructor to perform these operations.



# Example: The copy constructor of the String class

```
class String {
    int size;
    char *contents;
 public:
    String(const char *);                    // Constructor
    String(const String &);                  // Copy Constructor
    void print();                            // Prints the string on the screen
    ~String();                               // Destructor
};

String::String(const String &object_in) {    // Copy Constructor
    cout<< "Copy Constructor has been invoked" << endl;
    size = object_in.size;
    contents = new char[size + 1];           //  +1 for null character
    strcpy(contents, object_in.contents);
}

int main() {
    String my_string("string 1");
    my_string.print();
    String other = my_string;                // Copy constructor is invoked
    String more(my_string);                  // Copy constructor is invoked
```

# Constant Objects and Const Member Functions

- The programmer may use the keyword const to specify that an object is not modifiable.
- Any attempt to modify (to change the attributes) directly or indirectly (by calling a function) causes a compiler error.

```
const TComplex cz(0,1);    // constant object
```

- C++ compilers totally disallow any member function calls for const objects. The programmer may declare some functions as const, which do not modify any data of the object. Only const functions can operate on const objects.

```
void print() const     // constant method
{
  cout << "complex number= " << real << ", " << img;
}
```

```
// Constant function: It prints the coordinates on the screen
void Point::print() const
{
    cout << "X= " << x << ", Y= " << y << endl;
}

// -------- Main Program ------------
int main()
{
    const Point cp(10,20);          // constant point
    Point ncp(0,50);                // non-constant point
    cp.print();                     // OK. Const function operates on const object
    cp.move(30,15);                 // ERROR! Non-const function on const object
    ncp.move(100,45);               // OK. ncp is non-const
    return 0;
}
```

- A const method can invoke only other const methods, because a const method is not allowed to alter an object's state either directly or indirectly, that is, by invoking some nonconst method.

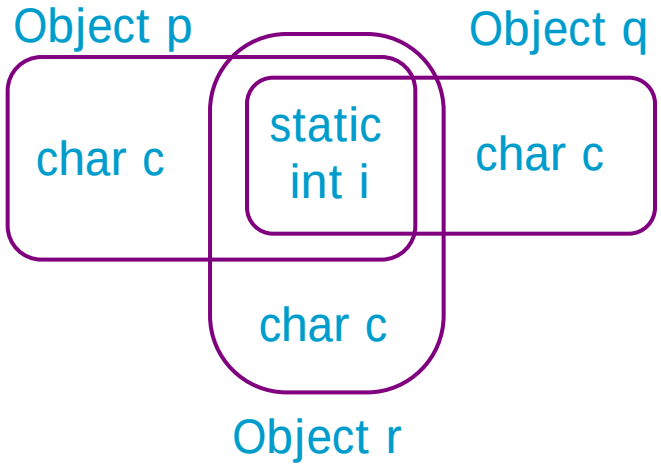
```
class TComplex{
   float real,img;
 public:
   TComplex(float, float);    // constructor
   void print() const;            // const method
   void reset() {real=img=0;}   // non-const method
};
TComplex::TComplex(float r=0,float i=0){
    real=r;
    img=i;
}
void TComplex::print() const {     // const method
     std::cout << “complex number= “ << real << “, “ << img;
}
```

```
int main() {
 const TComplex cz(0,1);    // constant object
 TComplex ncz(1.2,0.5)        // non-constant object
 cz.print();  // OK
 cz.reset(); // Error !!!
 ncz.reset();  // OK
}
```

# *static* Class Members

- Normally, each object of a class has its own copy of all data members of the class.
- In certain cases only one copy of a particular data member should be shared by all objects of a class. A static data member is used for this reason.

```
class A{
   char c;
   static int i;
};

int main()
{
    A   p, q, r;
    :
}
```



# *static* Class Members

- Static data members exist even no objects of that class exist.
- Static data members can be public or private.
- To access public static data when no objects exist use the class name and binary scope resolution operator.

  for example **A::i= 5;**
- To access private static data when no objects exist, a public **static member function** must be provided.
- They must be initialized **once** (and only once) at file scope.

```
class A {
  char c;
  static int count; //  Number of created objects (static data)
 public:
  static void GetCount(){return count;}
  A(){count ++; std::cout<< std::endl << "Constructor “ << count;}
 ~A(){count--; std::cout<< std::endl << "Destructor “ << count;}
};
int A::count=0;     // Allocating memory for number
```

```
int main(){
 std::cout<<"\n Entering 1. BLOCK.....";
 A a,b,c;
  {
    std::cout<<"\n Entering 2. BLOCK.....";
    A d,e;
    std::cout<<"\n Exiting 2. BLOCK.....";
  }
  std::cout<<"\n Exiting 1. BLOCK.....";
}
```

```
Entering 1. BLOCK.....
Constructor 1
Constructor 2
Constructor 3
Entering 2. BLOCK.....
Constructor 4
Constructor 5
Exiting 2. BLOCK.....
Destructor 5
Destructor 4
Exiting 1. BLOCK.....
Destructor 3
Destructor 2
Destructor 1
```

# Passing Objects to Functions as Arguments

- Objects should be passed or returned by reference unless there are compelling reasons to pass or return them by value.
- Passing or returning by value can be especially inefficient in the case of objects. Recall that the object passed or returned by value must be copied into stack and the data may be large, which thus wastes storage. The copying itself takes time.
- If the class contains a copy constructor the compiler uses this function to copy the object into stack.
- We should pass the argument by reference because we don't want an unnecessary copy of it to be created. Then, to prevent the function from accidentally modifying the original object, we make the parameter a const reference.

```
ComplexT & ComplexT::add(const ComplexT& z) {
    ComplexT  result;          // local object
    result.re = re + z.re;
    result.im = im + z.im;
    return result;             // ERROR!
}
```

Remember, local variables can not be returned by reference.

# Avoiding Temporary Objects

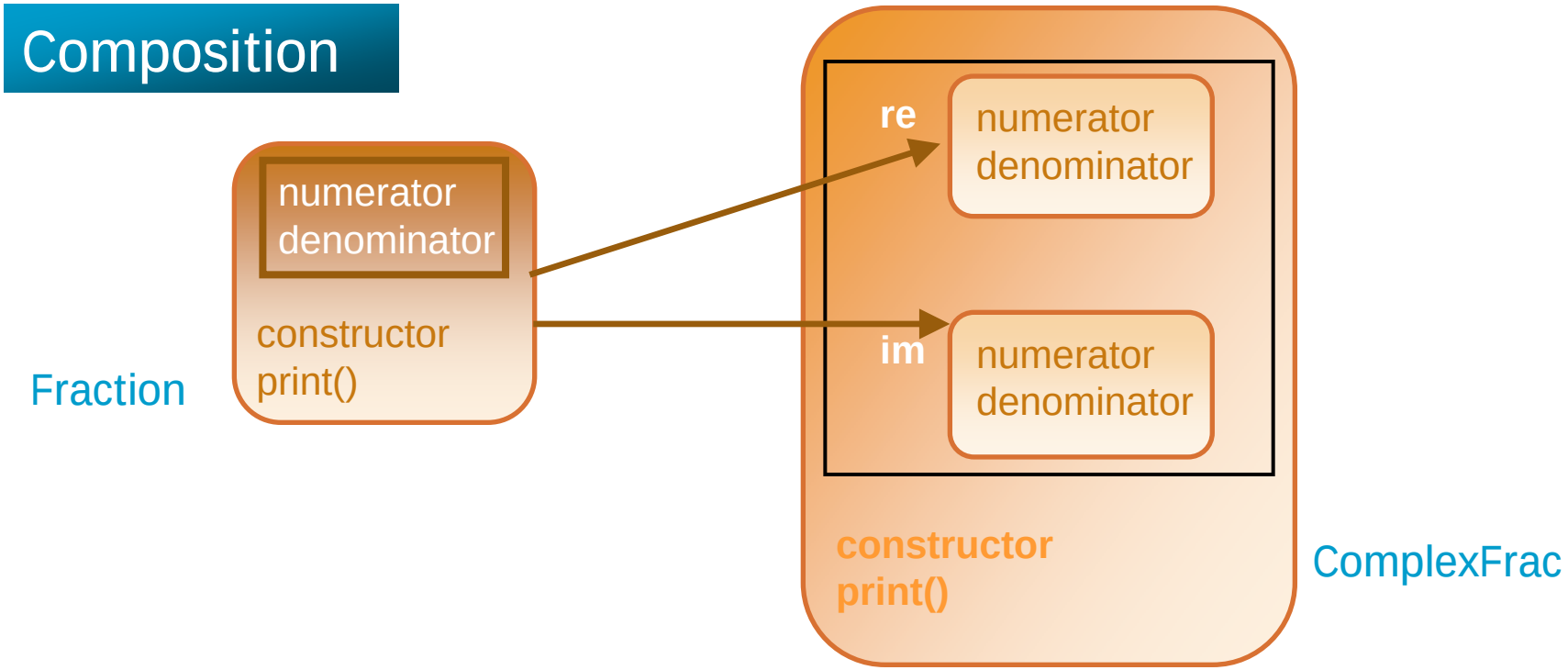
- In the previous example, within the add function a temporary object is defined to add two complex numbers.
- Because of this object, constructor and destructor are called.
- Avoiding the creation of a temporary object within add() saves time and memory space.

```
ComplexT ComplexT::add(const ComplexT& c)  {
    double  re_new,im_new;
    re_new = re + c.re;
    im_new = im + c.im;
    return ComplexT(re_new,im_new);  // Constructor is called
}
```

- The only object that's created is the return value in stack, which is always necessary when returning by value.
- This could be a better approach, if creating and destroying individual member data items is faster than creating and destroying a complete object.

# Nesting Objects: Classes as Members of Other Classes

- A class may include objects of other classes as its data members.
- In the example, a class is designed (ComplexFrac) to define complex numbers. The data members of this class are fractions which are objects of another class (Fraction).



# Composition & Aggregation

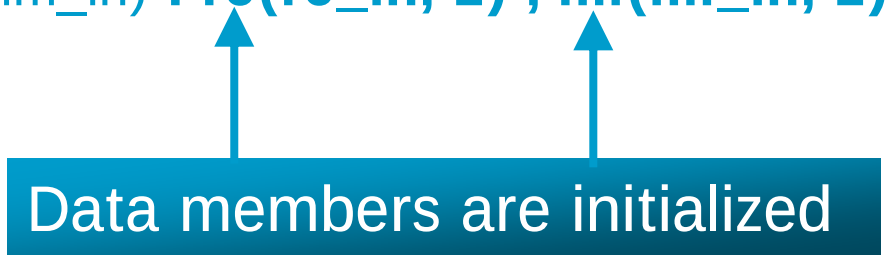
- The relation between Fraction and ComplexFrac is called "**has a relation**". Here, ComplexFrac has a Fraction (actually two Fractions).
- Here, the author of the class ComplexFrac has to supply the constructors of its object members (re , im) with necessary arguments.
- Member objects are constructed in the order in which they are declared and before their enclosing class objects are constructed.

## ▶ Example: A class to define fractions

```
class Fraction {                          // A class to define fractions
    int numerator, denominator;
 public:
    Fraction(int, int);                   // CONSTRUCTOR
    void print() const;
};
Fraction::Fraction(int num, int denom)  {  // CONSTRUCTOR
    numerator = num;
    if (denom==0) denominator = 1;
    else denominator = denom;
          cout << "Constructor of Fraction" << endl;
}
void Fraction::print() const {
    cout << numerator << "/" << denominator << endl;
}
```

## Example: A class to define complex numbers. It contains two objects as members

```
class ComplexFrac {   //  Complex numbers, real and imag. parts are fractions
   Fraction re, im;  //  objects as data members of another class
 public:
   ComplexFrac(int,int);     // Constructor
   void print() const;
 };
ComplexFrac::ComplexFrac(int re_in, int im_in) : re(re_in, 1) , im(im_in, 1)
{
     :
}
void ComplexFrac::print() const  {
    re.print();
    im.print();
}
int main()  {
     ComplexFrac cf(2,5);
     cf.print();
     return 0;
}
```

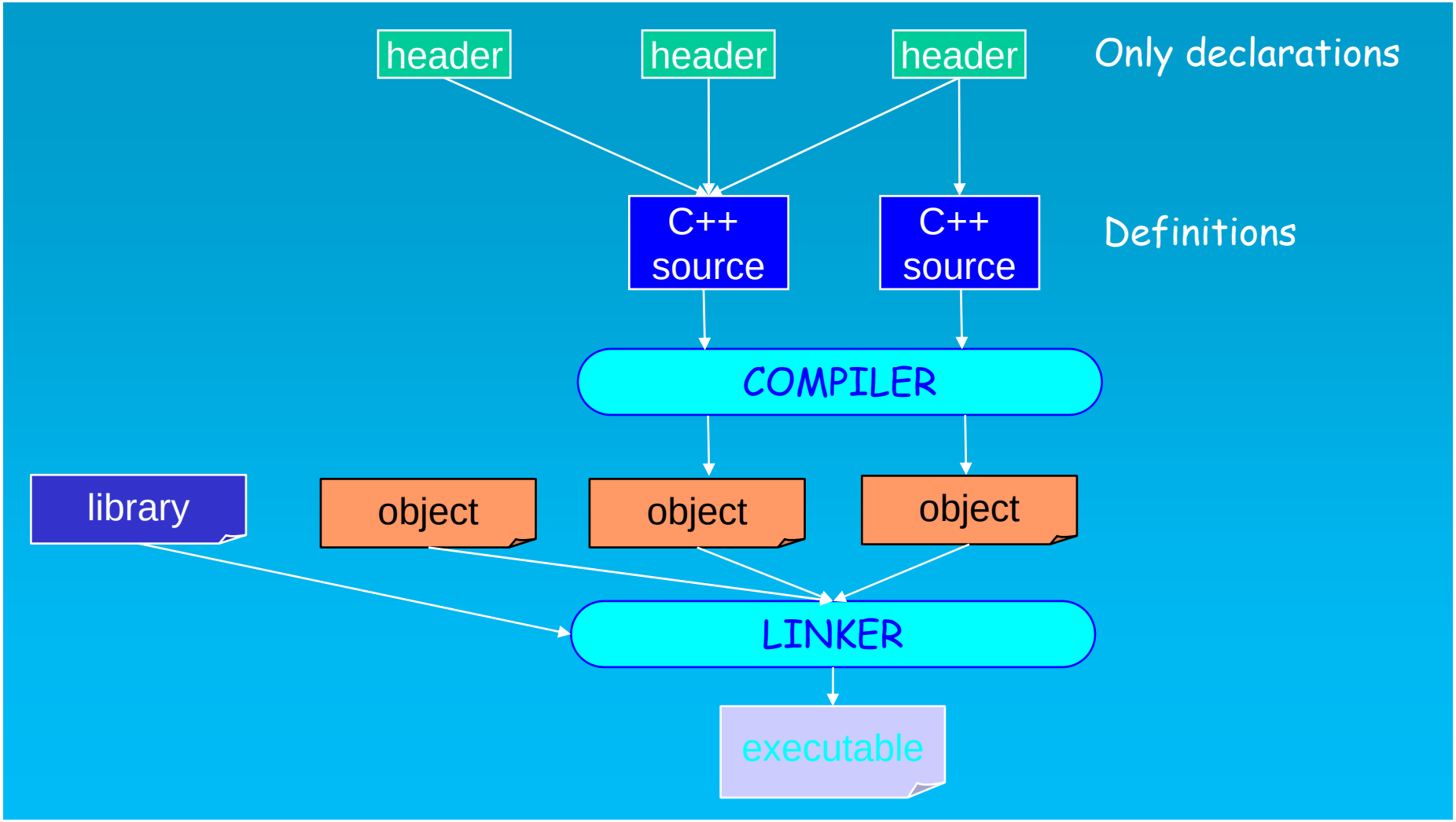



When an object goes out of scope, the destructors are called in reverse order: The enclosing object is destroyed first, then the member (inner) object.

# Working with Multiple Files (Separate Compilation)

- It is a good way to write each class or a collection of related classes in separate files.
- It provides managing the complexity of the software and reusability of classes in new projects.

# Working with Multiple Files



- When using separate compilation you need some way to automatically compile each file and to tell the linker to build all the pieces along with the appropriate libraries and startup code into an executable file.
- The solution, developed on Unix but available everywhere in some form, is a program called ***make***.
- Compiler vendors have also created their own project building tools. These tools ask you which files are in your project and determine all the relationships themselves. These tools use something similar to a makefile, generally called a project file, but the programming environment maintains this file so you don't have to worry about it.
- The configuration and use of project files varies from one development environment to another, so you must find the appropriate documentation on how to use them (although project file tools provided by compiler vendors are usually so simple to use that you can learn them by playing around).
- We will write the example e410.cpp about fractions and complex numbers again. Now we will put the class for fractions and complex numbers in separate files.

# 5 Operator Overloading

# Operator Overloading

- It is possible to overload the built-in C++ operators such as +, >=, and ++ so that they invoke different functions, depending on their operands.
- **a+b** will call one function if **a** and **b** are integers, but will call a different function if **a** and **b** are objects of a class.
- Operator overloading makes your program **easier** to write and to understand.
- Overloading does not actually add any capabilities to C++. Everything you can do with an overloaded operator you can also do with a function.
- However, overloaded operators make your programs easier to write, read, and maintain.

# Operator Overloading

- Operator overloading is only another way of calling a function.
- You have no reason to overload an operator except if it will make the code involving your class easier to write and especially easier to read.
- Remember that code is read much more than it is written

# Limitations

- You can't overload operators that don't already exist in C++. You can overload only the built-in operators.
- You can not overload the following operators

  .
  *
  ->
  ,
  ::
  ?:
  **sizeof**

# Limitations

- The C++ operators can be divided roughly into binary and unary. Binary operators take two arguments. Examples are a+b, a-b, a/b, and so on. Unary operators take only one argument: -a, ++a, a--.
- If a built-in operator is binary, then all overloads of it remain binary. It is also true for unary operators.
- Operator precedence and syntax (number of arguments) cannot be changed through overloading.
- All the operators used in expressions that contain only built-in data types cannot be changed. At least one operand must be of a user defined type (class).

# Overloading the + operator for ComplexT



```
/* A class to define complex numbers */
class TComplex {
    float real,img;
  public:
            :   // Member functions
    TComplex operator+(TComplex&); // header of operator+
function
};
/* The Body of the function for operator + */
TComplex TComplex::operator+(TComplex& z) {
   TComplex result;
   result.real = real + z.real;
   result.img = img + z.img;
   return result;
}
```

```
int main() {
   TComplex z1,z2,z3;
    :    // Other operations
    z3=z1+z2; like z3 = z1.operator+(z2);
}
```

# Overloading the Assignment Operator (=)

- Because assigning an object to another object of the same type is an activity most people expect to be possible, the compiler will automatically create a type::operator=(const type &) if you don’t make one.
- The behavior of this operator is member wise assignment. It assigns (copies) each member of an object to members of another object. (Shallow Copy)
- If this operation is sufficient you don't need to overload the assignment operator. For example, overloading of assignment operator for complex numbers is not necessary.

# Overloading the Assignment Operator (=)

```
void ComplexT::operator=(const ComplexT& z)
{
    re = z.re;
    im = z.im;
}
```

- You don't need to write such an assignment operator function, because the operator provided by the compiler does the same thing.

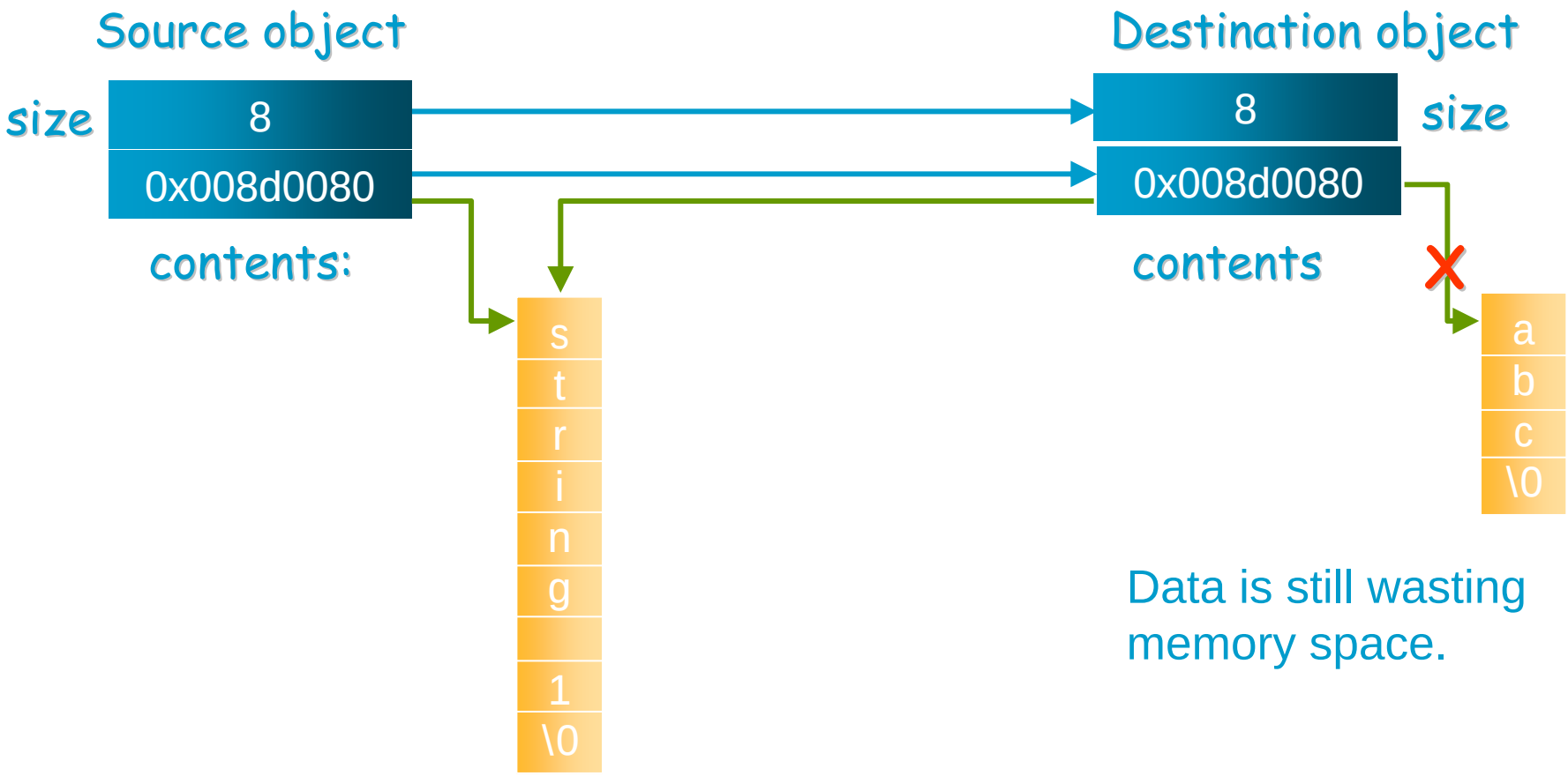
# Overloading the Assignment Operator (=)

- In general, you don't want to let the compiler do this for you.
- With classes of any sophistication (especially if they contain pointers!) you want to explicitly create an operator=.

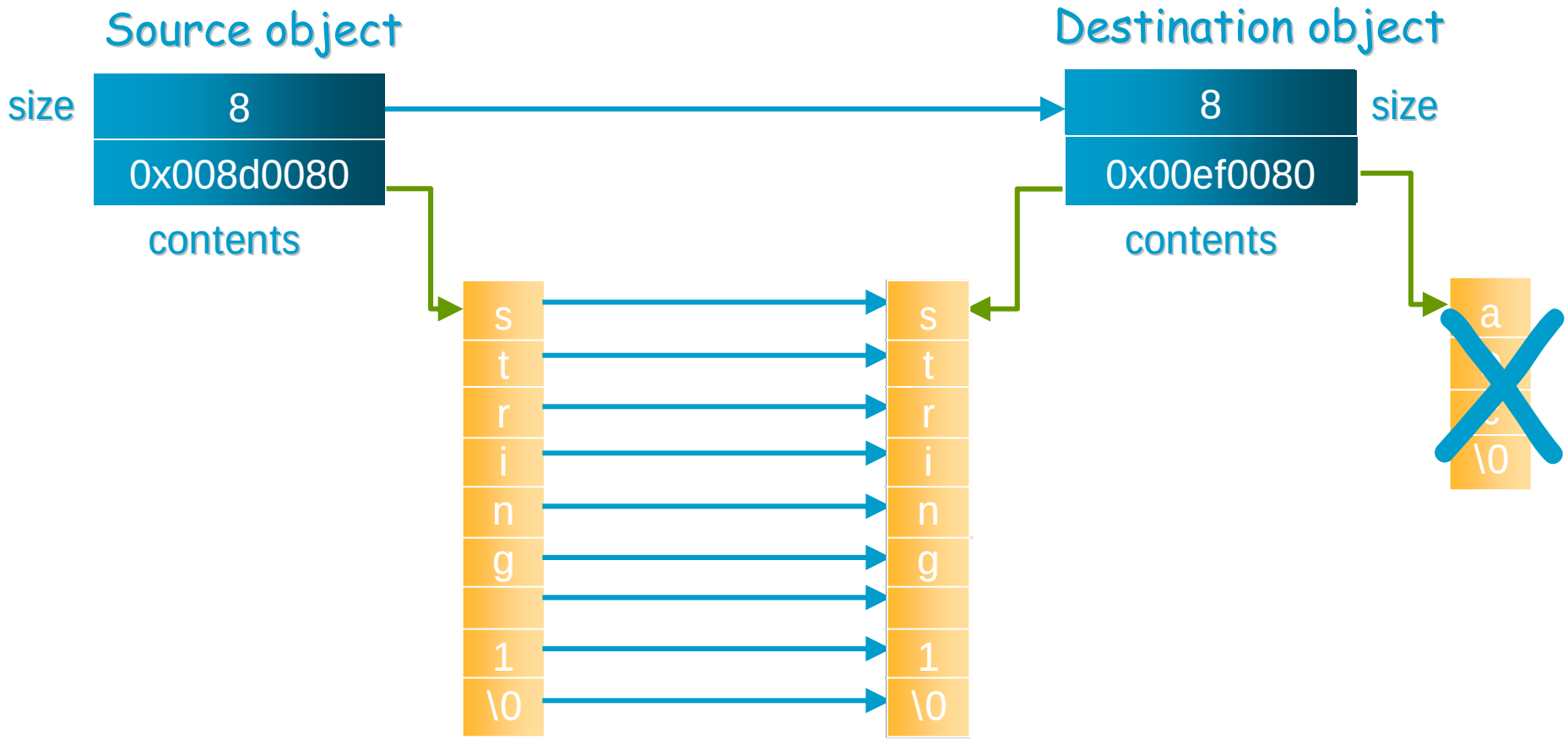
# Example

```
class string {
   int size;
   char *contents;
 public:
   void operator=(const string &);  // assignment operator
   :       // Other methods
};
void string::operator=(const string &s)
{
   size = s.size;
   delete []contents;
   contents = new char[size+1];
   strcpy(contents, s.contents);
}
```

# Operator Provided by the Compiler



# Operator of the Programmer



# Return value of the assignment operator

- When there's a void return value, you can't chain the assignment operator (as in a = b = c ).
- To fix this, the assignment operator must return a reference to the object that called the operator function (its address).

```
// Assignment operator , can be chained as in  a = b = c
const String& String::operator=(const String &in_object)  {
    if (size != in_object.size){ // if the sizes of the source and destination
       size = in_object.size;    // objects are different
       delete [] contents;      // The old contents is deleted
       contents = new char[size+1]; // Memory allocation for the new contents
    }
    strcpy(contents, in_object.contents);
    return *this;          // returns a reference to the object
}
```

# Copy Constructor *vs.* Assignment Operator

- The difference between the assignment operator and the copy constructor is that the copy constructor actually creates a new object before copying data from another object into it, whereas the assignment operator copies data into an already existing object.

# Copy Constructor *vs*. Assignment Operator

- A a;
- A b(a);
- b=a;
- A c=a;

# Overloading Unary Operators

- Unary operators operate on a single operand. Examples are the increment (++) and decrement (--) operators; the unary minus, as in -5; and the logical not (!) operator.
- Unary operators take no arguments, they operate on the object for which they were called. Normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj.

**Example:** We define ++ operator for class ComplexT to increment the real part of the complex number by 0.1 .

```
int main() {
    ComplexT z(1.2, 0.5);
    ++z;  // operator++ function is called
    z.print();
    return 0;
}
```

```
void ComplexT::operator++() {
    re=re+0.1;
}
```

► To be able to assign the incremented value to a new object, the operator function must return a reference to the object.

```
// ++ operator
// increments the real part of a complex number by 0.1
const ComplexT & ComplexT::operator++() {
    re=re+0.1;
    return *this;
}
int main()  {
    ComplexT z1(1.2, 0.5), z2;
    z2 = ++z1; //++ operator is called, incremented value is assigned to z2
    z2.print();
    return 0;
}
```

# Overloading the "[]" Operator



- Same rules apply to all operators. So we don't need to discuss each operator. However, we will examine some interesting operators.
- One of the interesting operators is the subscript operator.
- It can be declared in two different ways:

```
class C {
   returntype & operator [] (paramtype);
        or
   const returntype & operator [] (paramtype) const;
};
```

# Overloading the "[]" Operator

►The first declaration can be used when the overloaded subscript operator modifies the object. The second declaration is used with a const object; in this case, the overloaded subscript operator can access but not modify the object.

If **c** is an object of class C, the expression

**c[i]**

is interpreted as

**c.operator[ ](i)**

►Example: Overloading of the subscript operator for the String class. The operator will be used to access the **i**th character of the string. If **i** is less the zero then the first character and if **i** is greater than the ***size*** of the string the last character will be accessed.

```
// Subscript operator
char & String::operator[](int i) {
    if(i < 0)
       return contents[0];          // return first character
    if(i >= size)
       return contents[size-1];     // return last character
    return contents[i];             // return i th character
}
int main()  {
    String s1("String 1");
    s1[1] = 'p';  // modifies an element of the contents
    s1.print();
    cout << " 5 th character of the string s1 is: " << s1[5] << endl;
    return 0;
}
```

# Overloading the "()" Operator

The function call operator is unique in that it allows any number of arguments.

```
class C{
   returntype  operator ( ) (paramtypes);
};
```

If c is an object of class C, the expression

**c(i, j, k)** is interpreted as

**c.operator( )( i, j, k )**

**Example:** The function call operator is overloaded to print complex numbers on the screen. In this example the function call operator does not take any arguments.

```
// The function call operator without any argument, it prints a complex number
void ComplexT::operator( )( ) const {
    cout << re << " , " << im <<  endl ;
}
```

**Example:** The function call operator is overloaded to copy a part of the contents of a string into a given memory location. In this example the function call operator takes two arguments: the address of the destination memory and the numbers of characters to copy.

```
// The function call operator with two arguments
void String::operator( )( char * dest, int num) const {
    if (num > size) num=size;        // if num is greater the size of the string
    for (int k=0; k < num; k++)  dest[k]=contents[k];
}

int main( ) {
    String s1("Example Program");
    char * c = new char[8];              // Destination memory
    s1(c,7);                             // First 7 letters of string1 are copied into c
    c[7] = '\0';                         // End of string (null) character
    cout << c;
    delete [] c;
    return 0;
}
```

# "Pre" and "post" form of operators ++ and --



►Recall that ++ and -- operators come in “pre” and “post” form.

►If these operators are used with an assignment statement than different forms has different meanings.

```
z2= ++ z1;     // preincrement

z2 = z1++;     // postincrement
```

►The declaration, **operator ++ ( )** with no parameters overloads the **preincrement** operator.

►The declaration, **operator ++ (int)** with a single int parameter overloads the **postincrement** operator. Here, the int parameter serves to distinguish the postincrement form from the preincrement form. This parameter is not used.

# Post-Increment Operator

```
// postincrement operator
ComplexT ComplexT::operator++(int) {
    ComplexT temp;
    temp = *this;          // old value (original objects)
    re= re + 0.1;          // increment the real part
    return temp;           // return old value
}
```

# Pre-Increment Operator

```
// postincrement operator
ComplexT ComplexT::operator++() {
    re= re + 0.1;              // increment the real part
    return *this;              // return old value
}
```

# 6 Inheritance

# Content

- Inheritance
- Reusability in Object-Oriented Programming
- Redefining Members (Name Hiding)
- Overloading vs. Overriding
- Access Control
- Public and Private Inheritance
- Constructor, Destructor and Assignment Operator in Inheritance
- Multiple Inheritance
- Composition *vs* Inheritance

# Inheritance

- Inheritance is one of the ways in object-oriented programming that makes reusability possible.
- Reusability means taking an existing class and using it in a new programming situation.
- By reusing classes, you can reduce the time and effort needed to develop a program, and make software more robust and reliable.

# Inheritance



History

- The earliest approach to reusability was simply rewriting existing code. You have some code that works in an old program, but doesn't do quite what you want in a new project.
- You paste the old code into your new source file, make a few modifications to adapt it to the new environment. Now you must debug the code all over again. Often you're sorry you didn't just write new code.

# Inheritance

- To reduce the bugs introduced by modification of code, programmers attempted to create self-sufficient program elements in the form of functions.
- Function libraries were a step in the right direction, but, functions don’t model the real world very well, because they don’t include important data.
- All too often, functions require modification to work in a new environment.
- But again, the modifications introduce bugs.

# Reusability in Object-Oriented Programming

- A powerful new approach to reusability appears in object-oriented programming is the class library. Because a class more closely models a real-world entity, it needs less modification than functions do to adapt it to a new situation.
- Once a class has been created and tested, it should (ideally) represent a useful unit of code.
- This code can be used in different ways again.

# Reusability in Object-Oriented Programming

1. The simplest way to reuse a class is to just use an object of that class directly. The standard library of the C++ has many useful classes and objects.
   - For example, cin and cout are such built in objects. Another useful class is string , which is used very often in C++ programs.

# Reusability in Object-Oriented Programming

2. The second way to reuse a class is to place an object of that class inside a new class.
   - We call this "creating a member object."
   - Your new class can be made up of any number and type of other objects, in any combination that you need to achieve the functionality desired in your new class.
   - Because you are composing a new class from existing classes, this concept is called composition (or more generally, aggregation). Composition is often referred to as a "has-a" relationship.

# Reusability in Object-Oriented Programming

3. The third way to reuse a class is inheritance, which is described next. Inheritance is referred to as a "is a" or "a kind of" relationship.

# string

- While a character array can be fairly useful, it is quite limited. It's simply a group of characters in memory, but if you want to do anything with it you must manage all the little details.
- The Standard C++ string class is designed to take care of (and hide) all the low-level manipulations of character arrays that were previously required of the C programmer.
- To use strings you include the C++ header file <string>.
- Because of operator overloading, the syntax for using strings is quite intuitive (natural).

# string

```
#include <string>    // Standard header file of C++ (inc. string class)
#include <iostream>
using namespace std;
int main() {
   string s1, s2;                    // Empty strings
   string s3 = "Hello, World.";  // Initialized
   string s4("I am");                // Also initialized
   s2 = "Today";                     // Assigning to a string
   s1 = s3 + " " + s4;                // Combining strings
   s1 += " 20 ";                     // Appending to a string
   cout << s1 + s2 + "!" << endl;
   return 0;
}
```

# string

- The first two strings, s1 and s2, start out empty, while s3 and s4 show two equivalent ways to initialize string objects from character arrays (you can just as easily initialize string objects from other string objects).
- You can assign to any string object using '='. This replaces the previous contents of the string with whatever is on the right-hand side, and you don't have to worry about what happens to the previous contents – that's handled automatically for you.
- To combine strings you simply use the '+' operator, which also allows you to combine character arrays with strings. If you want to append either a string or a character array to another string, you can use the operator '+='.
- Finally, note that cout already knows what to do with strings, so you can just send a string (or an expression that produces a string, which happens with
- s1 + s2 + "!" directly to cout in order to print it.

# Inheritance

- OOP provides a way to modify a class without changing its code.
- This is achieved by using inheritance to derive a new class from the old one.
- The old class (called the ***base class***) is not modified, but the new class (the ***derived class***) can use all the features of the old one and additional features of its own.

# "*is a*" Relationship

- We know that PCs, Macintoshes and Cray are kinds of computers; a worker, a section manager and general manager are kinds of employee.
- If there is a "**kind of**" relation between two objects then we can derive one from other using the inheritance.

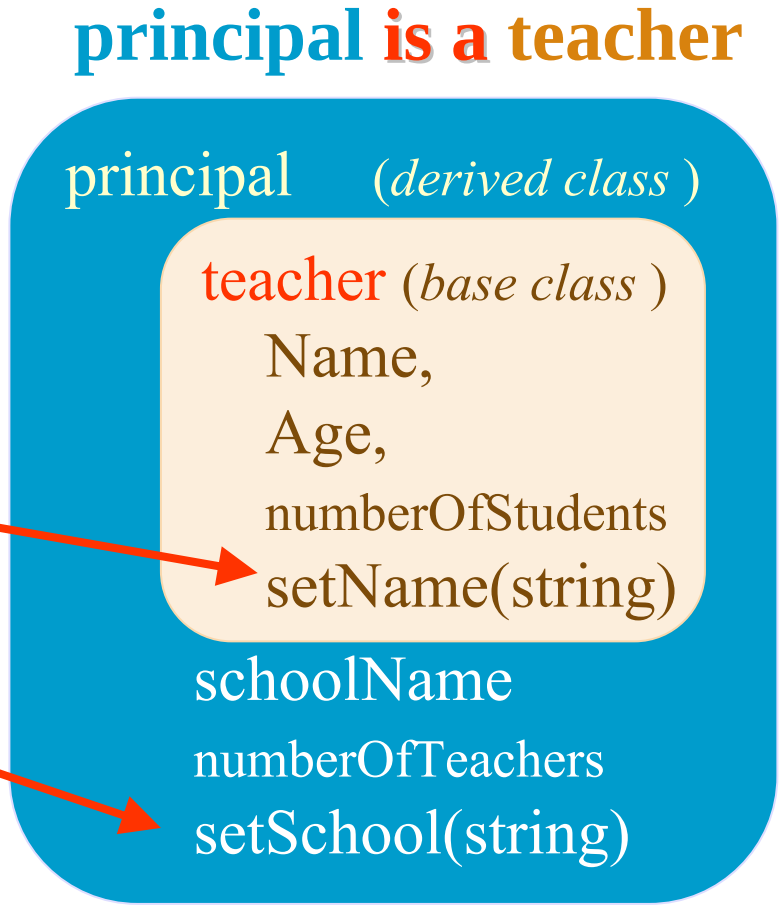
# Inheritance Syntax

- The simplest example of inheritance requires two classes: **a base class** and **a derived class**.
- The base class does not need any special syntax. The derived class, on the other hand, must indicate that it's derived from the base class.
- This is done by placing a colon after the name of the derived class, followed by a keyword such as public and then the base class name.

- Example: Modeling teachers and the principal (director) in a school.
- First, assume that we have a class to define teachers, then we can use this class to model the principal. Because the principal is a teacher.

```
class Teacher {          // Base class
  private:               // means public for derived class members
     string name;
     int age, numberOfStudents;
   public:
     void setName (const string & new_name){ name = new_name; }
};
class Principal : public Teacher {  // Derived class
    string schoolName;              // Additional members
    int numberOfTeachers;
  public:
    void setSchool(const string & s_name){ schoolName = s_name; }
};
```

```
int main()  {
   Teacher t1;
   Principal p1;
   p1.setName(" Principal 1");
   t1.setName(" Teacher 1");
   p1.setSchool(" Elementary School");
   return 0;
}
```



# Redefining Members (Name Hiding)

- Some members (data or function) of the base class may not suitable for the derived class. These members should be redefined in the derived class.
- For example, assume that the Teacher class has a print function that prints properties of teachers on the screen.
- But this function is not sufficient for the class Principal, because principals have more properties to be printed. So the print function must be redefined.

# Redefining Members

```
class Teacher{   // Base class
  protected:
    string name;
    int age, numOfStudents;
  public:
    void setName (const string & new_name) { name = new_name; }
    void print() const;
};

void Teacher::print() const  {    // Print method of Teacher class
   cout << "Name: " << name<< "  Age: " << age << endl;
   cout << "Number of Students: " << numOfStudents << endl;
}
```

```
class Principal : public Teacher{          // Derived class
    string school_name;
    int numOfTeachers;
  public:
    void setSchool(const string & s_name) { school_name = s_name; }
    void print() const;      // Print function of Principal class
};

void Principal::print() const {  // Print method of principal class
   cout << "Name: " << name << "  Age: " << age << endl;
   cout << "Number of Students: " << numOfStudents << endl;
   cout << "Name of the school: " <<  school_name << endl;
}
```

- print() function of the Principal class overrides (hides) the print() function of the Teacher class.

# Redefining Members



►Now the Principal class has two print() functions. The members of the base class can be accessed by using the scope operator (::).

```
void Principal::print() const    {  // Print method of Principal class
    Teacher::print(); // invokes the print function of the teacher class
    cout << "Name of the school: " << school_name << endl;
}
```

# Overloading vs. Overriding

- If you modify the signature and/or the return type of a member function from the base class then the derived class has two member functions with the same name. But this is not overloading, it is overriding.
- If the author of the derived class redefines a member function, it means he or she changes the interface of the base class. In this case the member function of the base class is hidden.

## Example

```
class A{
 public:
   int ia1,ia2;
   void fa1();
   int fa2(int);
};
```

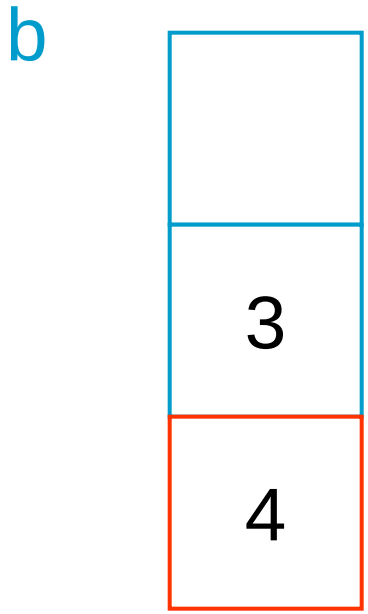
```
class B: public A{
 public:
   float ia1;            // overrides ia1
   float fa1(float);   // overrides fa1
};
```

example14.cpp

```
int main(){
    B b;
    int j=b.fa2(1);
    b.ia1=4;                    // B::ia1
    b.ia2=3;                    // A::ia2   if  ia2 is public in A
    float y=b.fa1(3.14); // B::fa1
    b.fa1();  // ERROR  fa1 function in B hides  the function of A
    b.A::fa1();    // OK
    b.A::ia1=1;   // OK
}
```

# Example






```
b.ia1=4;
b.ia2=3;
```

# Access Control

►Remember, when inheritance is not involved, class member functions have access to anything in the class, whether public or private, but objects of that class have access only to public members.

►Once inheritance enters the picture, other access possibilities arise for derived classes. Member functions of a derived class can access public and protected members of the base class, but not private members. Objects of a derived class can access only public members of the base class.

| Access | Base Class | Derived Class | Object |
|---|---|---|---|
| public | yes | yes | yes |
| protected | yes | yes | no |
| private | yes | no | no |

**Example**

```
class A{
 private:
   int ia1;
 protected:
   int ia2;
public:
   void fa1();
   int fa2(int);
};
```

```
class B: public A{
private:
 float ia1;            // overrides ia1
 public:
float fa1(float);   // overrides fa1
};
```

```
float B::fa1(float f){
    ia1= 2.22 ;
    ia2=static_cast<int>(f*f);
}
```

```
class Teacher {                          // Base class
  private:                               // only members of Teacher can access
    string name;
  protected:                             // Also members of derived classes can
    int age, numOfStudents;
  public:                                // Everyone can access
    void setName (const string & new_name){ name = new_name; }
    void print() const;
};

class Principal : public Teacher {       // Derived class
  private:                               // Default
    string  school_name;
    int numOfTeachers;
  public:
    void setSchool(const string & s_name) { school_name = s_name; }
    void print() const;
    int getAge() const { return age; }  // It works because age is protected
    const string & get_name(){ return name;}// ERROR! name is private
};
```

```
int main()
{
    teacher t1;
    principal p1;
     t1.numberOfStudents=54;
     t1.setName(“Sema Catir");
     p1.setSchool(“Halide Edip Adivar Lisesi");
 }
```

# Protected vs. Private Members

►In general, class data should be private. Public data is open to modification by any function anywhere in the program and should almost always be avoided.

►Protected data is open to modification by functions in any derived class. Anyone can derive one class from another and thus gain access to the base class's protected data. It's safer and more reliable if derived classes can't access base class data directly.

►But in real-time systems, where speed is important, function calls to access private members is a time-consuming process. In such systems data may be defined as protected to make derived classes access data directly and faster.

## Private data: Slow and reliable

```
class A{                              // Base class
  private:
    int i;                            // safe
  public:
    void access(int new_i){    // public interface to access i
        if (new_i > 0 && new_i <= 100)
            i=new_i;
    }
};

class B:public A{                 // Derived class
  private:
    int k;
  public:
    void set(new_i, new_k){
       A::access(new_i);        // reliable but slow
        :
    }
};
```

**Protected data: Fast, author of the derived class is responsible**

```
class A{                          // Base class
  protected:
    int i;          // derived class can access directly
 public:
    :
};

class B:public A{       // Derived class
  private:
    int k;
  public:
    void set(new_i,new_k){
       i=new_i;                        // fast
        :
    }
};
```

# Public Inheritance

►In inheritance, you usually want to make the access specifier public.

```
class Base
  { };
class Derived : public Base {
```

►This is called public inheritance (or sometimes public derivation). The access rights of the members of the base class are not changed.

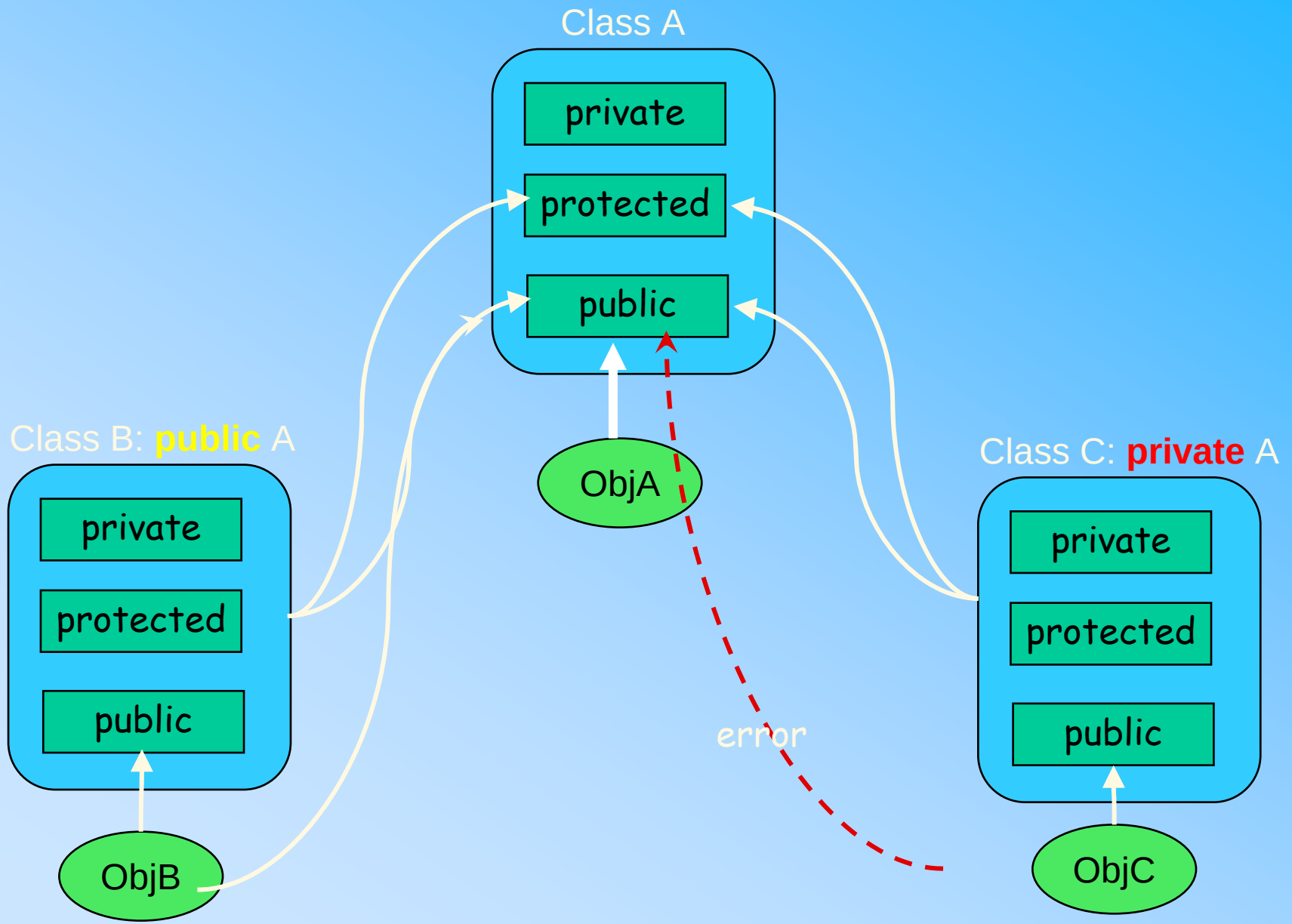
►Objects of the derived class can access public members of the base class.

►Public members of the base class are also public members of the derived class.

# Private Inheritance

```
class Base
  { };
class Derived : private Base {
```

- This is called private inheritance.
- Now public members of the base class are private members of the derived class.
- Objects of the derived class can not access members of the base class.
- Member functions of the derived class can still access public and protected members of the base class.


Class A
private
protected
public
Class B: public A
private
protected
public
ObjA
Class C: private A
private
protected
public
error
ObjB
ObjC

# Redefining Access



- Access specifications of public members of the base class can be redefined in the derived class.
- When you inherit privately, all the public members of the base class become private.
- If you want any of them to be visible, just say their names (no arguments or return values) along with the using keyword in the public section of the derived class:

```
class Base{
   private:
     int k;
   public:
     int i;
     void f();
};
```

```
int main(){
   Base b;
   Derived d;
   b.i=5;    // OK public in Base
   d.i=0;    // ERROR private inheritance
   b.f();    // OK
   d.f();    // OK
   return 0;
};
```

```
class Derived : private Base{  // All members of Base are private now
   int m;
 public:
   Base::f(); // f() is public again
   void fb1();
};
```

```
class Base{
   private:
     int k;
   public:
     int i;
     void f(int);
     bool f(int,float);
};

class Derived : private Base{  // All members of Base are private now
   int m;
 public:
   Base::f(int); // f(int) is public again
   void fb1();
};
```

```
int main(){
   Base b;
   Derived d;
   b.i=5;     // OK public in Base
   d.i=0;     // ERROR private inheritance
   b.f();     // OK
   d.f();     // OK
   return 0;
};
```

# Special Member Functions and Inheritance

►Some functions will need to do different things in the base class and the derived class. They are the overloaded = operator, the destructor, and all constructors.

►Consider a constructor. The base class constructor must create the base class data, and the derived class constructor must create the derived class data.

►Because the derived class and base class constructors create different data, one constructor cannot be used in place of another. Constructor of the base class can not be the constructor of the derived class.

►Similarly, the = operator in the derived class must assign values to derived class data, and the = operator in the base class must assign values to base class data. These are different jobs, so assignment operator of the base class can not be the assignment operator of the derived class.

# Constructors and Inheritance

►When you define an object of a derived class, the base class constructor will be called before the derived class constructor. This is because the base class object is a subobject—a part—of the derived class object, and you need to construct the parts before you can construct the whole.

►If the base class has a constructor that needs arguments, this constructor must be called before the constructor of the derived class.

example15.cpp

```
class Teacher {       // turetilmis sinif
    char *Name;
    int Age,numberOfStudents;
  public:
    Teacher(char *newName){Name=newName;}  // temel sinif kurucusu
};

class Principal : public Teacher{  // turetilmis sinif
    int numberOfTeachers;
  public:
    Principal(char *, int );     // // turetilmis sinif kurucusu
};
```

```
// Constructor of the derived class
// constructor of the base is called before the body of the constructor of the derived class
Principal::Principal(const string & new_name, int numOT):Teacher(new_name)
{
    numOfTeachers = numOT;
}
```

►Remember, the constructor initializer can also be used to initialize members.

```
// Constructor of the derived class
Principal::Principal(const string & new_name, int numOT)
                                :Teacher(new_name), numOfTeachers( numOT)
 { }                                        // body of the constructor is empty

 int main()  {
     Principal p1("Ali Bilir", 20);     // An object of derived class is defined
     return 0;
 }
```

►If the base class has a constructor, which must take some arguments, then the derived class must also have a constructor that calls the constructor of the base with proper arguments.

# Destructors and Inheritance

- Destructors are called automatically.
- When an object of the derived class goes out of scope, the destructors are called in reverse order: The derived object is destroyed first, then the base class object.

```
#include <iostream.h>
class B {
   public:
      B() { cout << "B constructor" << endl; }
     ~B() { cout << "B destructor" << endl; }
};
class C : public B {
  public:
     C() { cout << "C constructor" << endl; }
     ~C() { cout << "C destructor" << endl; }
};
int main(){
   std::cout << "Start" << std::endl;
   C ch;       // create a C object
   std::cout << "End" << std::endl;
}
```

# Example: Constructor Chain

```
#include <iostream.h>
class A {
   private:
      int x;
      float y;
   public:
     A(int i, float f) :
        x(i), y(f)              // initialize A
        { cout << "Constructor A" << endl; }
     void display() {
         cout << intA << ", " << floA << "; ";}
};
```

```
class B : public A {
   private:
      int v;
      float w;
   public:
      B(int i1, float f1, int i2, float f2) :
         A(i1, f1),                      // initialize A
         v(i2), w(f2)           // initialize B
         { cout << "Constructor B" << endl; }
      void display(){
         A::display();
         cout << v << ", " << w << "; ";
         }
};
```

```
class C : public B {
   private:
      int r;
      float s;
   public:
      C(int i1,float f1, int i2,float f2,int i3,float f3) :
         B(i1, f1, i2, f2),          // initialize B
         r(i3), s(f3)                // initialize C
         { cout << "Constructor C" << endl; }
      void display() {
            B::display();
            cout << r << ", " << s;
         }
};
```

```
int main() {
   C c(1, 1.1, 2, 2.2, 3, 3.3);
   cout << "\nData in c = ";
   c.display();
}
```

example19.cpp

# Explanation

- A C class is inherited from a B class, which is in turn inherited from a A class.
- Each class has one int and one float data item.
- The constructor in each class takes enough arguments to initialize the data for the class and all ancestor classes. This means two arguments for the A class constructor, four for B (which must initialize A as well as itself), and six for C (which must initialize A and B as well as itself).
- Each constructor calls the constructor of its base class.

# Explanation

- In main(), we create an object of type C, initialize it to six values, and display it.
- When a constructor starts to execute, it is guaranteed that all the subobjects are created and initialized.
- Incidentally, you can't skip a generation when you call an ancestor constructor in an initialization list. In the following modification of the C constructor:

```
C(int i1, float f1, int i2, float f2, int i3, float f3) :
                A(i1, f1),              // ERROR! can't initialize A
                intC(i3), floC(f3)      // initialize C
  { }
```

the call to A() is illegal because the A class is not the immediate base class of C.

# Explanation: Constructor Chain



- You never need to make explicit destructor calls because there's only one destructor for any class, and it doesn't take any arguments.
- The compiler ensures that all destructors are called, and that means all of the destructors in the entire hierarchy, starting with the most-derived destructor and working back to the root.

# Assignment Operator and Inheritance

- Assignment operator of the base class can not be the assignment operator of the derived class.
- Recall the String example.



```
class String {
  protected:
     int size;
     char *contents;
  public:
     const String & operator=(const String &);   // assignment operator
     :                                            // Other methods
};
```

```
const String & String::operator=(const String &in_object)  {
    size = in_object.size;
    delete[ ] contents;   // delete old contents
    contents = new char[size+1];
    strcpy(contents, in_object.contents);
    return *this;
}
```

▶Example: Class String2 is derived from class String. If an assignment operator is necessary it must be written

```
class String2 : public String { // String2 is derived from String
    int size2;
    char  *contents2;
 public:
    const String2 & operator=(const String2 &);
     :
};

// **** Assignment operator for String2 ****
const String2 & String2::operator=(const String2 &in_object)  {
    size = in_object.size;                          // inherited size
    delete []contents;
    contents= strdup(in_object.contents);
    size2 = in_object.size2;
    delete[ ] contents2;
    contents2 = strdup(in_object.contents2);
    return *this;
}
```

In previous example, data members of String (Base) class must be protected. Otherwise methods of the String2 (Derived) can not access them.

The better way to write the assignment operator of String2 is to call the assignment operator of the String (Base) class.

Now, data members of String (Base) class may be private.

```
//** Assignment operator **
const String2 &  String2::operator=(const String2 & in_object)
{
   String::operator=(in_object);          // call the operator= of String (Base)
   cout<< "Assignment operator of String2 has been invoked" << endl;
   size2 = in_object.size2;
   delete[] contents2;
   contents2 = new char[size2 + 1];
   strcpy(contents2, in_object.contents2);
   return *this;
}
```

In this method the assignment operator of the String is called with an argument of type (String2 &). Actually, the operator of String class expects a parameter of type (String &). This does not cause a compiler error, because as we will se in Section 7, a reference to base class can carry the address of an object of derived class.

# Composition *vs*. Inheritance

►Every time you place instance data in a class, you are creating a "has a" relationship. If there is a class Teacher and one of the data items in this class is the teacher's name, I can say that a Teacher object has a name.

►This sort of relationship is called composition because the Teacher object is composed of these other variables.

►Remember the class ComplexFrac. This class is composed of two Fraction objects.

►Composition in OOP models the real-world situation in which objects are composed of other objects.

# Composition *vs*. Inheritance

- Inheritance in OOP mirrors the concept that we call generalization in the real world. If I model workers, managers and researchers in a factory, I can say that these are all specific types of a more general concept called an employee.
- Every kind of employee has certain features: name, age, ID num, and so on.
- But a manager, in addition to these general features, has a department that he/she manages.
- A researcher has an area on which he/she studies.
- In this example the manager has not an employee.
- The manager is an employee

► You can use composition & inheritance together. The following example shows the creation of a more complex class using both of them.

```
class A {
  int i;
public:
  A(int ii) : i(ii) {}
  ~A() {}
  void f() const {}
};
```

```
class B {
  int i;
public:
  B(int ii) : i(ii) {}
  ~B() {}
  void f() const {}
};
```

```
class C : public B {   // Inheritance, C is B
  A a;   // Composition, C has A
public:
  C(int ii) : B(ii), a(ii) {}
  ~C() {}   // Calls ~A() and ~B()
  void f() const {   // Redefinition
    a.f();
    B::f();
  }
};
```

►C inherits from B and has a member object ("is composed of") of type A. You can see the constructor initializer list contains calls to both the base-class constructor and the member-object constructor.

►The function C::f( ) redefines B::f( ), which it inherits, and also calls the base-class version. In addition, it calls a.f( ).

►Notice that the only time you can talk about redefinition of functions is during inheritance; with a member object you can only manipulate the public interface of the object, not redefine it.

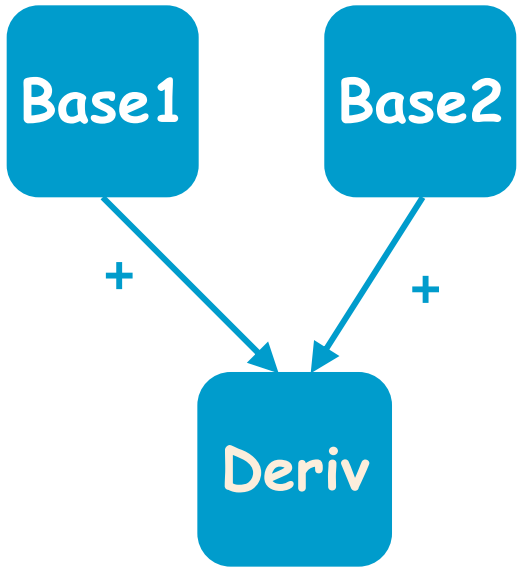
►In addition, calling f( ) for an object of class C would not call a.f( ) if C::f( ) had not been defined, whereas it would call B::f( ).

# Multiple Inheritance

```
class Base1{        // Base 1
 public:
   int a;
   void fa1();
   char *fa2(int);
};
```

```
class Base2{      // Base 2
 public:
   int a;
   char *fa2(int, char*);
   int fc();
};
```

```
class Deriv : public Base1 ,public Base2{
public:
    int a;
    float fa1(float);
    int fb1(int);
};
```




```
int main(){
 Deriv d;
 d.a=4;
 float y=d.fa1(3.14);
 int i=d.fc();
}
```

example20.cpp

char * c=d.fa2(1);
**is not valid.**
In inheritance functions are not overloaded. They are **overridden.**
You have to write

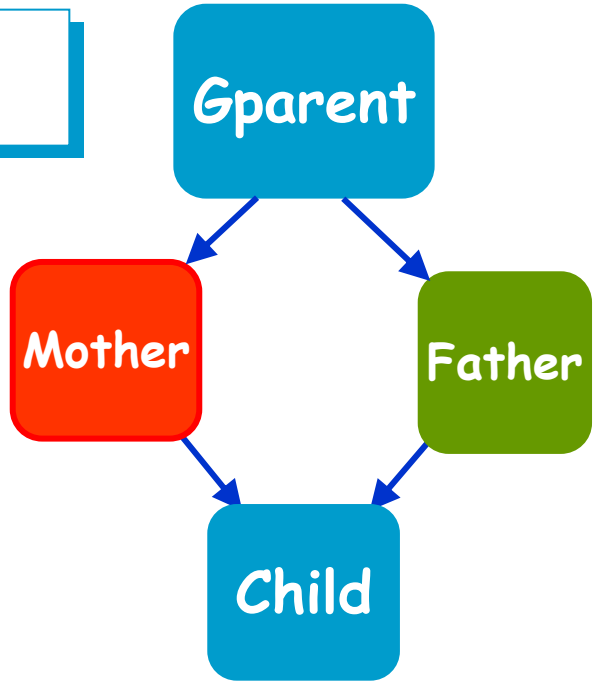
```
char * c=d.Base1::fa2(1);
```

or

```
char * c=d.Base2::fa2(1,"Hello");
```

# Repeated Base Classes

```
class Gparent
   {  };
class Mother : public Gparent
   {  };
class Father : public Gparent
   {  };
class Child : public Mother, public Father
   {  };
```




►Both Mother and Father inherit from Gparent, and Child inherits from both Mother and Father. Recall that each object created through inheritance contains a subobject of the base class. A Mother object and a Father object will contain subobjects of Gparent, and a Child object will contain subobjects of Mother and Father, so a Child object will also contain two Gparent subobjects, one inherited via Mother and one inherited via Father.

►This is a strange situation. There are two subobjects when really there should be one.

# Repeated Base Classes

- Suppose there's a data item in Gparent:

```
class Gparent {
   protected:
      int gdata;
 };
class Child : public Mother, public Father  {
   public:
      void Cfunc() {
         int temp = gdata;  // error: ambiguous
       }
 };
```

- The compiler will complain that the reference to gdata is ambiguous. It doesn't know which version of gdata to access: the one in the Gparent subobject in the Mother subobject or the one in the Gparent subobject in the Father subobject.

# Solution: Virtual Base Classes

►You can fix this using a new keyword, virtual, when deriving Mother and Father from Gparent :

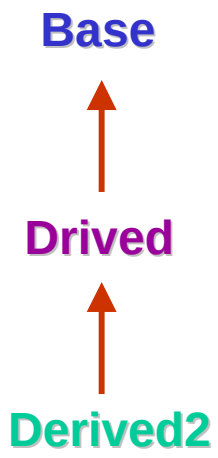
```
class Gparent
   {  };
class Mother : virtual public Gparent
   {  };
class Father : virtual public Gparent
   {  };
class Child : public Mother, public Father
   {  };
```

example21.cpp

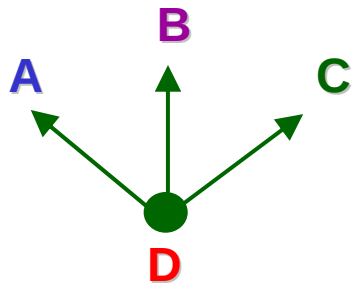
►The virtual keyword tells the compiler to inherit only one subobject from a class into subsequent derived classes. That fixes the ambiguity problem, but other more complicated problems arise that are too complex to delve into here.

►In general, you should avoid multiple inheritance, although if you have considerable experience in C++, you might find reasons to use it in unusual situations.

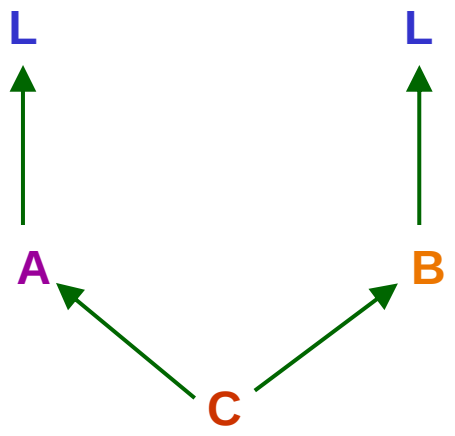
```
class Base
{
   public:
      int a,b,c;
 };
class Derived : public Base
 {
   public:
      int b;
 };
class Derived2 : public Derived
 {
   public:
      int c;
 };
```



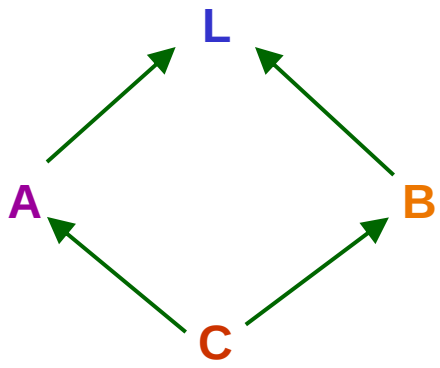
```
class A {
    ...
};
class B {
    ...
};
class C {
    ...
 };
class D : public A, public B, private C {
   ...
 };
```



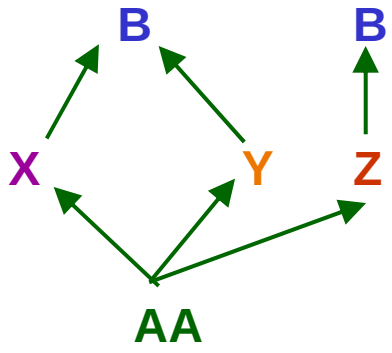
```
class L {
    public:
      int next;
};
class A : public L {
    ...
};
class B : public L {
    ...
};
class C : public A, public B {
   void f() ;
   ...
 };
```



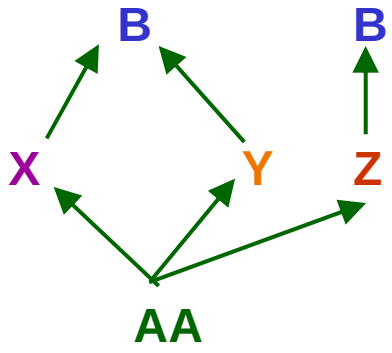
```
class L {
    public:
      int next;
};
class A : virtual public L {
    ...
};
class B : virtual public L {
    ...
};
class C : public A, public B {
   ...
 };
```



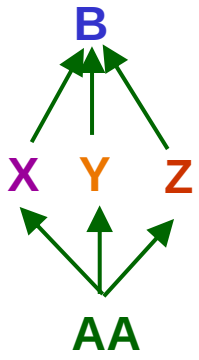
```
class B {
      ...
};
class X : virtual public B {
    ...
};
class Y : virtual public B {
    ...
};
class Z : public B {
   ...
 };
class AA : public X, public Y , public Z {
   ...
 };
```



```
class B {
      ...
};
class X : virtual public B {
    ...
};
class Y : public B {
    ...
};
class Z : public B {
   ...
 };
class AA : public X, public Y , public Z {
   ...
 };
```



```
class B {
      ...
};
class X : virtual public B {
    ...
};
class Y : virtual public B {
    ...
};
class Z : virtual public B {
   ...
 };
class AA : public X, public Y , public Z {
   ...
 };
```



# 7 Object Pointers

# Pointers to Objects

►Objects are stored in memory, so pointers can point to objects just as they can to variables of basic types.

The new Operator:

►The new operator allocates memory of a specific size from the operating system and returns a pointer to its starting point. If it is unable to find space, in returns a 0 pointer.

►When you use new with objects, it does not only allocates memory for the object, it also creates the object in the sense of invoking the object's constructor. This guarantees that the object is correctly initialized, which is vital for avoiding programming errors.

# Pointers to Objects

## The delete Operator

- To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that releases the memory back to the operating system.
- If you create an array with new Type[ ];, you need the brackets when you delete it:

```
int * ptr = new int[10];
:
delete [ ] ptr;
```

Don't forget the brackets when deleting arrays of objects. Using them ensures that all the members of the array are deleted and that the destructor is called for each one. If you forget the brackets, only the first element of the array will be deleted.

# Example

```
class String {
   int size;
   char *contents;
 public:
   String();
   String(const char *);
   String(const String &);
   const String& operator=(const String &);
   void print() const ;
   ~String();
};
```

```
int main()  {
    String *sptr = new String[3];
    String s1("String_1");
    String s2("String_2");
    *sptr = s1;
    *(sptr + 1) = s2;
    sptr->print();
    (sptr+1)->print();
    sptr[1].print();
    delete[] sptr;
    return 0;
}
```

# Linked List of Objects

A class may contain a pointer to objects of its type.
This pointer can be used to build a chain of objects, a linked list.

```
class Teacher {
   friend class Teacher_list;
   string name;
   int age, numOfStudents;
   Teacher * next;
 public:
   Teacher(const string &, int, int);
   void print() const;
   const string& getName() const {
        return name;    }
  ~Teacher()
};
```

```
// linked list for teachers
class Teacher_list{
   Teacher *head;
  public:
   Teacher_list(){head=0;}
   bool append(const string &,int,int);
   bool del(const string &);
   void print() const ;
   ~Teacher_list();
};
```

# Linked List of Objects



►In the previous example the Teacher class must have a pointer to the next object and the list class must be declared as a friend, to enable users of this class building linked lists.

►If this class is written by the same group then it is possible to put such a pointer in the class.

►But usually programmers use ready classes, written by other groups, for example classes from libraries.

►These classes may not have a next pointer.

►To build linked lists of such ready classes the programmer have to define a node class.

►Each object of the node class will hold the addresses of an element of the list.

# Linked List of Objects

```
class Teacher_node{
    friend class Teacher_list;
    Teacher * element;                    // The element of the list
    Teacher_node * next;                  // next node
    Teacher_node(const string &, int, int);      // constructor
    ~Teacher_node();                      // destructor
};
Teacher_node::Teacher_node(const string & n, int a, int nos){
    element = new Teacher(n, a, nos);
    next = 0;
}
Teacher_node::~Teacher_node(){
   delete element;
}
```

# Pointers and Inheritance

- If a class Derived has a public base class Base, then a pointer to Derived can be assigned to a variable of type pointer to Base without use of explicit type conversion. A pointer to Base can carry the address of an object of Derived.
- The opposite conversion, for pointer to Base to pointer to Derived, must be explicit.
- For example, a pointer to Teacher can point to objects of Teacher and to objects of Principal. A principal is a teacher, but a teacher is not always a principal.

# Pointers and Inheritance



```
class Base{

};
class Derived : public Base {

};
Derived d;
Base *bp = &d;                  // implicit conversion
Derived *dp = bp;               // ERROR! Base is not Derived
dp = static_cast<Derived *>(bp);   // explicit conversion
```

# Pointers and Inheritance



- If the class Base is a **private** base of Derived , then the implicit conversion of a Derived* to Base* would not be done.
- Because, in this case a public member of Base can be accessed through a pointer to Base but not through a pointer to Derived.

# Pointers and Inheritance

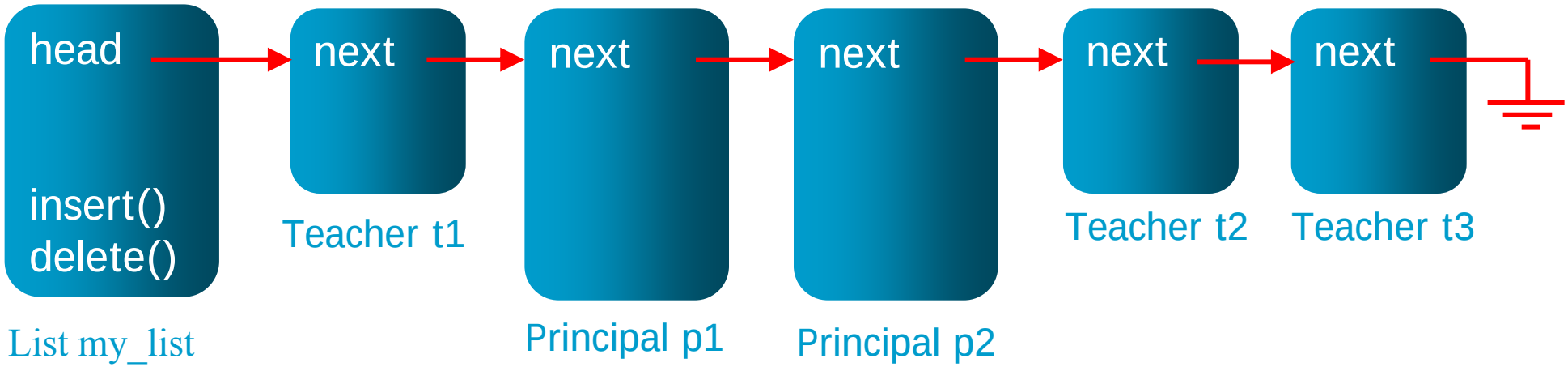


```
class Base{
    int m1;
  public:
    int m2;                          // m2 is a public member of Base
};
class Derived : private Base {  // m2 is not a public member of Derived
   :
};
Derived d;
d.m2 = 5;          // ERROR!   m2 is private member of Derived
Base *bp = &d; // ERROR! private base
bp = static_cast<Base*>(&d);      // ok: explicit conversion
bp->m2 = 5;      // ok
```

# Heterogeneous Linked Lists

- Using the inheritance and pointers, heterogeneous linked lists can be created.
- A list specified in terms of pointers to a base class can hold objects of any class derived from this base class.
- We will discuss heterogeneous lists again, after we have learnt polymorphism.

**Example:** A list of teachers and principals



# 8 Polymorphism

# Content

- Polymorphism
- Virtual Members
- Abstract Class

# Polymorphism

- There are three major concepts in object-oriented programming:
  1. Classes,
  2. Inheritance,
  3. Polymorphism, which is implemented in C++ by virtual functions.
- In real life, there is often a collection of different objects that, given identical instructions (messages), should take different actions. Take teacher and principal, for example.
- Suppose the minister of education wants to send a directive to all personnel: “Print your personal information!” Different kinds of staff (teacher or principal) have to print different information. But the minister doesn’t need to send a different message to each group. One message works for everyone because everyone knows how to print his or her personal information.

# Polymorphism

- Polymorphism means “taking many shapes”. The minister’s single instruction is polymorphic because it looks different to different kinds of personnel.
- Typically, polymorphism occurs in classes that are related by inheritance. In C++, polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- This sounds a little like function overloading, but polymorphism is a different, and much more powerful, mechanism. One difference between overloading and polymorphism has to do with which function to execute when the choice is made.
- With function overloading, the choice is made by the compiler (compile-time). With polymorphism, it’s made while the program is running (run-time).

# Normal Member Functions Accessed with Pointers

```
class Square {       // Base Class
  protected:
    double edge;
  public:
    Square(double e):edge(e){  } //Base class constructor
    double area(){ return( edge * edge ) ; }
};
```

```
class Cube : public Square { // Derived Class
  public:
    Cube(double e):Square(e){} // Derived class cons.
    double area(){ return( 6.0 * edge * edge ) ; }
};
```

```
int main(){
    Square S(2.0)                                    ;
    Cube   C(2.0)                                    ;
    Square *ptr                                      ;
    char    c                                        ;
    cout << "Square or Cube"; cin >> c               ;
    if (c=='s') ptr=&S                               ;
      else      ptr=&C                               ;
    ptr→area();          // which Area ???
}
```

- ptr = &C;
- Remember that it's perfectly all right to assign an address of one type (Derived) to a pointer of another (Base), because pointers to objects of a derived class are type compatible with pointers to objects of the base class.
- Now the question is, when you execute the statement

  ptr->area();

what function is called? Is it Square::area() or Cube::area()?

# Virtual Member Functions Accessed with Pointers

Let's make a single change in the program: Place the keyword **virtual** in front of the declaration of the area() function in the base class.

```
class Square {       // Temel sinif
  protected:
     double edge;
  public:
     Square(double e):edge(e){  }  // temel sinif kurucusu
      virtual double area(){ return( edge * edge ) ; }
};
class Cube : public Square { // Turetilmis sinif
  public:
     Cube(double e):Square(e){} // Turetilmis sinif kurucusu
     double area(){ return( 6.0 * edge * edge ) ; }
};
```

square.cpp

```
int main(){
    Square S(2.0)                                    ;
    Cube   C(8.0)                                    ;
    Square *ptr                                      ;
    char    c                                        ;

    cout << “Square or Cube"; cin >> c               ;
    if (c==‘s') ptr=&S                               ;
       else     ptr=&C                               ;
    ptr→Area();
}
```

# Virtual Member Functions Accessed with Pointers

The function in the base class (Teacher) is executed in both cases. The compiler ignores the **contents** of the pointer ptr and chooses the member function that matches the **type** of the pointer.

Let's make a single change in the program: Place the keyword **virtual** in front of the declaration of the print() function in the base class.

```
class Teacher{                                  // Base class
      string *name;
      int numOfStudents;
  public:
      Teacher(const string &, int);             // Constructor of base
      virtual void print() const;               // A virtual (polymorphic) function
};

class Principal : public Teacher{               // Derived class
      string *SchoolName;
  public:
      Principal(const string &,  int , const string &);
      void print() const;                       // It is also virtual (polymorphic)
};
```

# Late Binding

- Now, different functions are executed, depending on the contents of ptr. Functions are called based on the contents of the pointer ptr, not on the type of the pointer. This is polymorphism at work. I've made print() polymorphic by designating it virtual.
- How does the compiler know what function to compile? In e81.cpp, the compiler has no problem with the expression
- ptr->print();
- It always compiles a call to the print() function in the base class. But in e82.cpp, the compiler doesn't know what class the contents of ptr may be a pointer to. It could be the address of an object of the Teacher class or the Principal class. Which version of print() does the compiler call? In fact, at the time it's compiling the program, the compiler doesn't know what to do, so it arranges for the decision to be deferred until the program is running.

# Late Binding

- At runtime, when the function call is executed, code that the compiler placed in the program finds out the type of the object whose address is in ptr and calls the appropriate print() function: Teacher::print()  or Principal::print(), depending on the class of the object.
- Selecting a function at runtime is called **late binding** or **dynamic binding**. (Binding means connecting the function call to the function.)
- Connecting to functions in the normal way, during compilation, is called *early binding* or *static binding*. Late binding requires a small amount of overhead (the call to the function might take something like 10 percent longer) but provides an enormous increase in power and flexibility.

# How It Works

- Remember that, stored in memory, a normal object—that is, one with no virtual functions—contains only its own data, nothing else.
- When a member function is called for such an object, the compiler passes to the function the address of the object that invoked it. This address is available to the function in the this pointer, which the function uses (usually invisibly) to access the object's data.
- The address in this is generated by the compiler every time a member function is called; it's not stored in the object and does not take up space in memory.
- The ***this*** pointer is the only connection that's necessary between an object and its normal member functions.

# How It Works

- With virtual functions, things are more complicated. When a derived class with virtual functions is specified, the compiler creates a table—an array—of function addresses called the **virtual table**.
- The Teacher and Principal classes each have their own virtual table. There is an entry in each virtual table for every virtual function in the class. Objects of classes with virtual functions contain a pointer to the virtual table of the class. These object are slightly larger than normal objects.
- In the example, when a virtual function is called for an object of Teacher or Principal, the compiler, instead of specifying what function will be called, creates code that will first look at the object's virtual table and then uses this to access the appropriate member function address. Thus, for virtual functions, the object itself determines what function is called, rather than the compiler.

**Example:** Assume that the classes Teacher and Principal contain two virtual functions.

```
class Teacher{                                  // Base class
      string *name;
      int numOfStudents;
  public:
      virtual void read();                      // Virtual function
      virtual void print() const;               // Virtual function
};

class Principal : public Teacher{               // Derived class
     string *SchoolName;
  public:
     void read();                               // Virtual function
     void print() const;                        // Virtual function
};
```

Virtual Table of Teacher

| |
|---|
| &Teacher::read |
| &Teacher::print |

Virtual Table of Principal

| |
|---|
| &Principal::read |
| &Principal::print |

Objects of Teacher and Principal will contain a pointer to their virtual tables.

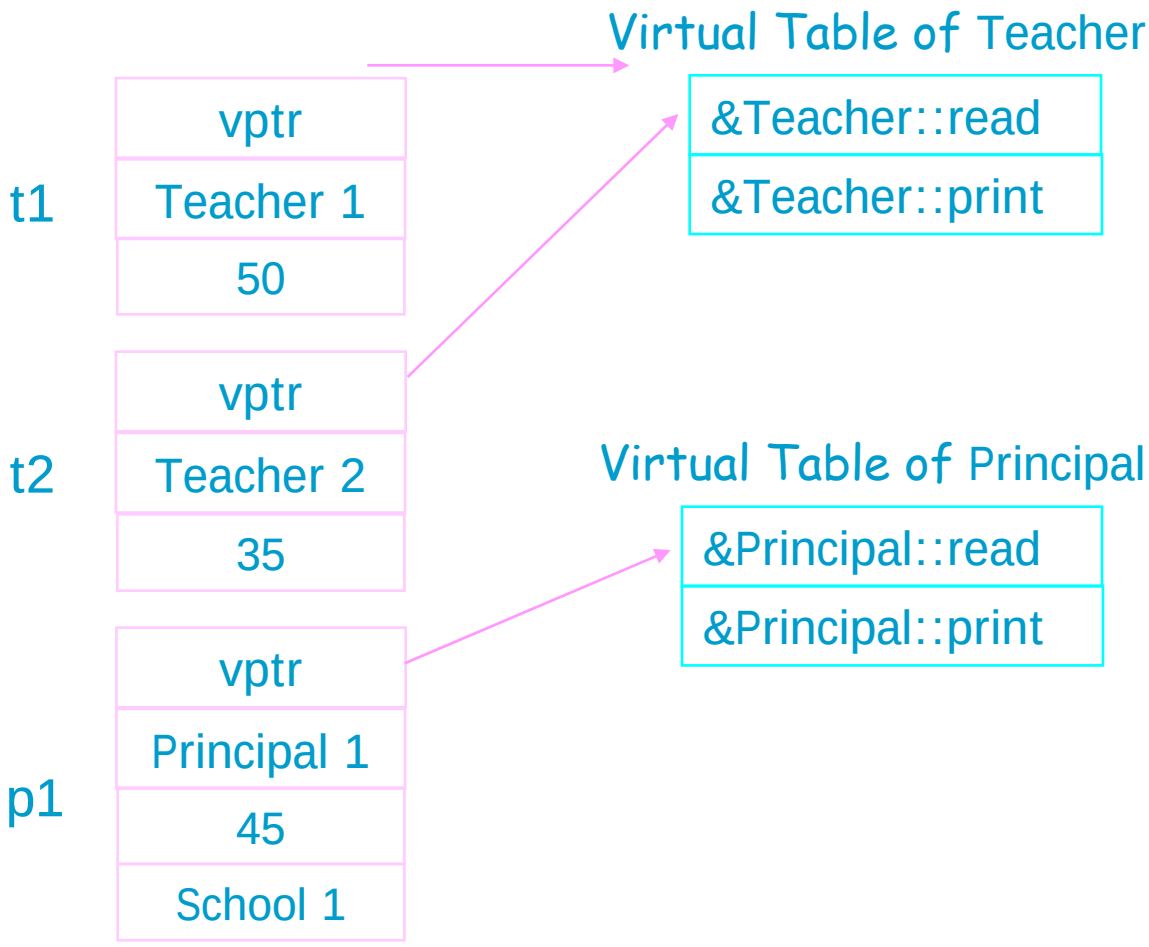
```
int main(){
   Teacher t1("Teacher 1", 50);
   Teacher t2("Teacher 2", 35);
   Principal p1("Principal 1", 45 , "School 1");
   :
}
```

MC68000-like assembly counterpart of the statement ptr->print(); Here ptr contains the address of an object.

```
move.l    ptr, this  ; this to object
movea.l   ptr, a0    ; a0 to object
movea.l   (a0), a1 ; a1<-vptr
jsr       4(a1)      ; jsr print
```

If the print() function would not a virtual function:

```
move.l    ptr, this  ; this to object
jsr       teacher_print
   or
jsr       principal_print
```



# Don’t Try This with Objects

Be aware that the virtual function mechanism works only with pointers to objects and, with references, not with objects themselves.

```
int main(){
  Square S(4);
  Cube C(8);
  S.Area();
  C.Area();
}
```

Calling virtual functions is a time-consuming process, because of indirect call via tables. Don’t declare functions as virtual if it is not necessary.

## Warning

```
class Square {       // Base
  protected:
     double edge;
  public:
     Square(double e):edge(e){  }  // Base Class Constructor
      virtual double Area(){ return( edge * edge ) ; }
};
class Cube : public Square { // Derived Class
  public:
     Cube(double e):Square(e){} // Derived Class Constructor
     double Area(){ return( 6.0 * Square::Area() ) ; }
};
```

*Here,* **Square::Area()** *is not virtual*

# Homogeneous Linked Lists and Polymorphism

Most frequent use of polymorphism is on collections such as linked list:

```
class Square {
  protected:
     double edge;
  public:
     Square(double e):edge(e){  }
     virtual double area(){ return( edge * edge ) ; }
     Sqaure *next ;
};
class Cube : public Square {
  public:
     Cube(double e):Square(e){}
     double area(){ return( 6.0 * edge * edge ) ; }
};
```

```
int main(){
    Circle   c1(50);
    Square s1(40);
    Circle   c2(23);
    Square s2(78);
    Square *listPtr;       // Pointer of the linked list
    /*** Construction of the list ***/
    listPtr=&c1;
    c1.next=&s1;
    s1.next=&c2;
    c2.next=&s2;
    s2.next=0L;
    while (listPtr){ // Printing all elements of the list
        cout << listPtr->Area() << endl ;
        listPtr=listPtr->next;
    }
}
```

# Abstract Classes

► To write polymorphic functions wee need to have derived classes. But sometimes we don't need to create any base class objects, but only derived class objects. The base class exists only as a starting point for deriving other classes.

► This kind of base classes we can call are called an ***abstract class***, which means that no actual objects will be created from it.

► Abstract classes arise in many situations. A factory can make a sports car or a truck or an ambulance, but it can't make a generic vehicle. The factory must know the details about what *kind* of vehicle to make before it can actually make one. Similarly, you'll see sparrows, wrens, and robins flying around, but you won't see any generic birds.

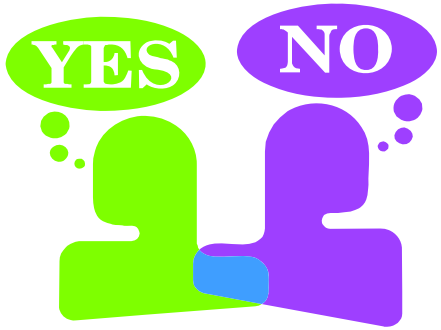
► Actually, a class is an abstract class only in the eyes of humans.

# Pure Virtual Classes

► It would be nice if, having decided to create an abstract base class, I could instruct the compiler to actively *prevent* any class user from ever making an object of that class. This would give me more freedom in designing the base class because I wouldn't need to plan for actual objects of the class, but only for data and functions that would be used by derived classes. There is a way to tell the compiler that a class is abstract: You define at least one ***pure virtual function*** in the class.

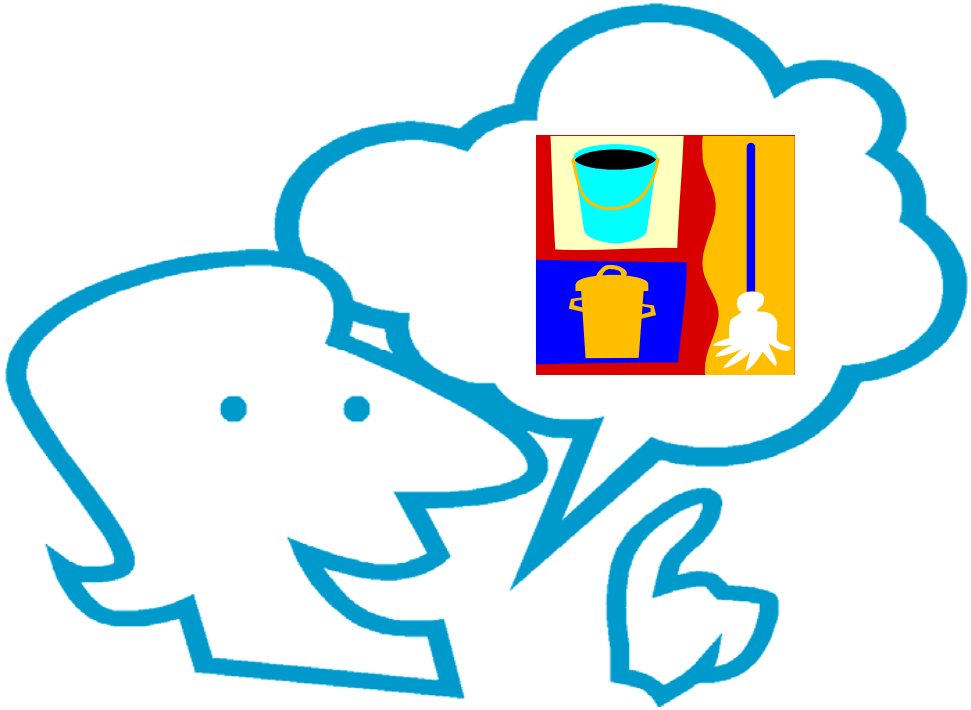
► A pure virtual function is a virtual function with no body. The body of the virtual function in the base class is removed, and the notation **=0** is added to the function declaration.

Are they the same or different?



►Not in the real world, but in our thoughts as an **abstraction** classification.

►A “Cleaning Utensil” does not exist, but specific kinds do!



# Example in Visual C++ 6

```
class CGenericShape{          // Abstract base class
   protected:
      int x,y;
      CGenericShape *next ;
   public:
         CGenericShape(int x_in,int y_in,
                              CGenericShape *nextShape){
                  x=x_in;
                  y=y_in;
                  next = nextShape ;
          }  // Constructor
          CGenericShape* operator++(){return next;}
          virtual void draw(HDC)=0;      // pure virtual function
};
```

```
class CLine:public CGenericShape{       // Line class
  protected:
     int x2,y2;              // End coordinates of line
  public:
     CLine(int x_in,int y_in,int x2_in,int y2_in,
              CGenericShape *nextShape)
              :CGenericShape(x_in,y_in,nextShape){
             x2=x2_in;
             y2=y2_in;
     }
     void draw(HDC hdc){ // virtual draw function
           MoveToEx(hdc,x,y,(LPPOINT) NULL);
           LineTo(hdc,x2,y2); }
};
```

```
class CRectangle:public CLine{  // Rectangle class
  public:
    CRectangle (int x_in,int y_in,int x2_in,int y2_in,
                    CGenericShape *nextShape)
                   :CLine(x_in,y_in,x2_in,y2_in,nextShape)
                   { }
    void draw(HDC hdc){// virtual draw
         Rectangle(hdc,x,y,x2,y2);
    }
};
```

```
class CCircle:public CGenericShape{    // Circle class
  protected:
     int radius;
  public:
     CCircle (int x_cen,int y_cen,int r,
                    CGenericShape *nextShape)
                    :CGenericShape(x_cen,y_cen,nextShape)
     {
       radius=r;
     }
     void draw(HDC hdc) { // virtual draw
         Ellipse(hdc,x-radius,y-radius,x+radius,y+radius);
     }
};
```

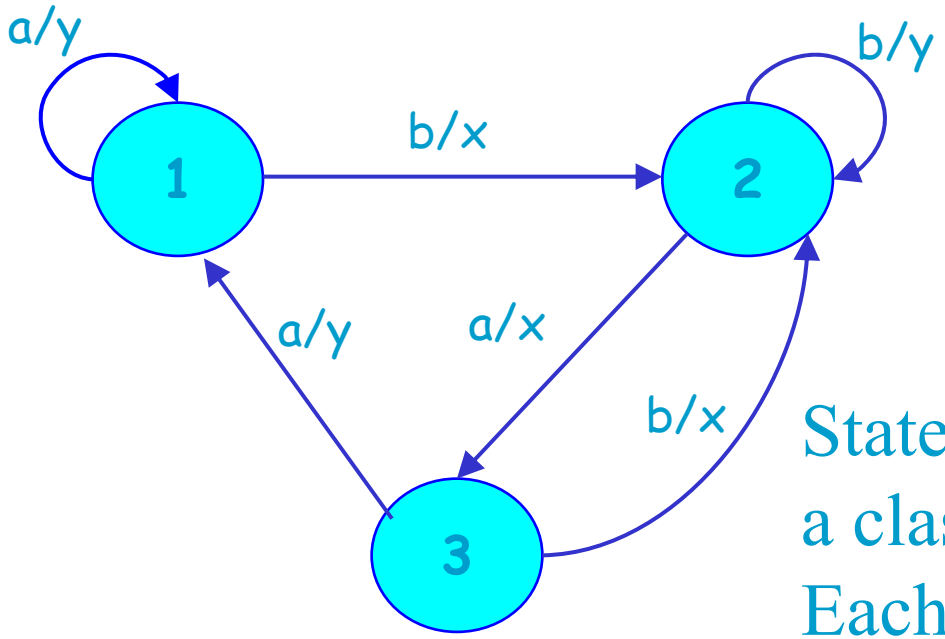
```
void ShowShapes(CGenericShape &shape,HDC hdc)
{
      CGenericShape *p = &shape ;
        // Which draw function will be called?
   while (p!=NULL){
          p->draw(hdc); // It 's unknown at compile-time
          p = ++*p ;
          Sleep(100);
    }
}
```

PolyDraw.dsw

```
PAINTSTRUCT ps;
HDC hdc;

CLine Line1(50,50,150,150,NULL);
CLine Line2(150,50,50,150,&Line1) ;
CCircle Circle1(100,100,20,&Line2);
CCircle Circle2(100,100,50,&Circle1);
CRectangle Rectangle1(50,50,150,150,&Circle2);

switch (message) {
                case WM_PAINT:
                  hdc = BeginPaint (hwnd, &ps);
                      ShowShapes (Rectangle1,hdc);
                   EndPaint (hwnd, &ps);
                return 0;
```

# A Finite State Machine (FSM) Example

State : { 1 , 2 , 3 }
Input : { a, b }, x to exit
Output : { x , y }




States of the FSM are defined using a class structure.
Each state is derived from the same base class.

```
class State{                                                    // Base State (Abstract Class)
   protected:
     State * const next_a, * const next_b;                      // Pointers to next state
     char output;
   public:
     State(State &  a, State &  b):next_a(&a), next_b(&b) { }
     virtual State* transition(char)=0;                         // pure virtual function
};

class State1:public State{                                      // *** State1 ***
  public:
     State1(State &  a, State &  b):State(a, b) { }
     State* transition(char);
};
class State2:public State{                                      // *** State2 ***
  public:
     State2(State &  a, State &  b):State(a, b) { }
     State* transition(char);
};
class State3:public State{                                      // *** State3 ***
  public:
      State3(State &  a, State &  b):State(a, b) { }
      State* transition(char);
};
```

The transition function of each state defines the behavior of the FSM. It takes the input value as argument, examines the input, produces an output value according to the input value and returns the address of the next state.

```
State* State1::transition(char input)
{
    switch(input){
    case 'a':  output = 'y';
                return next_a;
    case 'b': output = 'x';
               return next_b;
    default : cout << endl << "Undefined input";
                cout << endl << "Next State: Unchanged";
              return this;
   }
}
```

# The FSM in our example has three states.

```
class FSM{                                  // Finite State Machine
    State1 s1;
    State2 s2;
    State3 s3;
    State *current;                         // points to the current state
 public:
    FSM() : s1(s1,s2), s2(s3,s2), s3(s1,s2), current(&s1) { }   //Starting state is State1
    void run();
};
void FSM::run()  {
   char in;
   do {
      cout << endl << "Give the input value (a or b;  x: EXIT) ";
      cin >> in;
      if (in != 'x')
         current = current->transition(in);      // Polymorphic function call
      else
         curent = 0;                            // EXIT
    }  while(current);
}
```

The transition function of the current state is called.
Return value of this function determines the next state of the FSM.

# Virtual Constructors?

- Can constructors be virtual?

  **No, they can't be.**

- When you're creating an object, you usually already know what kind of object you're creating and can specify this to the compiler. Thus, there's not a need for virtual constructors.

- Also, an object's constructor sets up its virtual mechanism (the virtual table) in the first place. You don't see the code for this, of course, just as you don't see the code that allocates memory for an object.

- Virtual functions can't even exist until the constructor has finished its job, so **constructors can't be virtual**.

# Virtual Destructors

- Recall that a derived class object typically contains data from both the base class and the derived class.
- To ensure that such data is properly disposed of, it may be essential that destructors for both base and derived classes are called.

# Virtual Destructors

```
class Base {
        public:
                ~Base() { cout << "\nBase destructor"; }
};
class Derived : public Base {
        public:
                ~Derv() { cout << "\nDerived destructor"; }
};
int main(){
        Base* pb = new Derived;
        delete pb;
        cout << endl << "Program terminates.“ << endl ;
}
```

# Virtual Destructors

- But the output is
  Base Destructor
  Program terminates
- In this program bp is a pointer of Base type. So it can point to objects of Base type and Derived type. In the example, bp points to an object of Derived class, but while deleting the pointer only the Base class destructor is called.
- This is the same problem you saw before with ordinary (nondestructor) functions. If a function isn't virtual, only the base class version of the function will be called when it's invoked using a base class pointer, even if the contents of the pointer is the address of a derived class object. Thus in e85.cpp, the Derived class destructor is never called. This could be a problem if this destructor did something important.

To fix this problem, we have to make the base class **destructor virtual**.

```
class Base {
        public:
                virtual ~Base() { cout << "\nBase destructor"; }
};
class Derived : public Base {
        public:
                ~Derv() { cout << "\nDerived destructor"; }
};
int main(){
        Base* pb = new Derived;
        delete pb;
        cout << endl << "Program terminates.“ << endl ;
}
```

# 9 EXCEPTION

# Program Errors

- Kinds of errors with programs
  - Poor logic - bad algorithm
  - Improper syntax - bad implementation
  - Exceptions - Unusual, but predictable problems
- The earlier you find an error, the less it costs to fix it
- Modern compilers find errors early

# Paradigm Shift from C

- In C, the default response to an error is to continue, possibly generating a message
- In C++, the default response to an error is to terminate the program
- C++ programs are more “brittle”, and you have to strive to get them to work correctly
- Can catch all errors and continue as C does

# assert()

- a macro (processed by the precompiler)
  - Returns TRUE if its parameter is TRUE
  - Takes an action if it is FALSE
    - abort the program
    - throw an exception
- If DEBUG is not defined, asserts are collapsed so that they generate no code

# assert() (cont'd)

- When writing your program, if you know something is true, you can use an assert
- If you have a function which is passed a pointer, you can do
  - assert(pTruck);
    - if pTruck is 0, the assertion will fail
- Use of assert can provide the code reader with insight to your train of thought

# assert() (cont'd)

- Assert is only used to find programming errors
- Runtime errors are handled with exceptions
  - DEBUG false => no code generated for assert
  - Animal *pCat = new Cat;
  - assert(pCat); // bad use of assert
  - pCat ->memberFunction();

# assert() (cont'd)

- assert() can be helpful
- Don't overuse it
- Don't forget that it "instruments" your code
  - invalidates unit test when you turn DEBUG off
- Use the debugger to find errors

# Exceptions

- You can fix poor logic (code reviews, debugger)
- You can fix improper syntax (asserts, debugger)
- You have to live with exceptions
  - Run out of resources (memory, disk space)
  - User enters bad data
  - Floppy disk goes bad

# Why are Exceptions Needed?

- The types of problems which cause exceptions (running out of resources, bad disk drive) are found at a low level (say in a device driver)
- The low level code implementer does not know what your application wants to do when the problem occurs, so s/he “throws” the problem “up” to you

# How To Deal With Exceptions

- Crash the program
- Display a message and exit
- Display a message and allow the user to continue
- Correct the problem and continue without disturbing the user

Steinbach's Corollary to Murphy's Law:
"Never test for a system error you don't know how to handle."

# What is a C++ Exception?

- An object
  - passed from the area where the problem occurs
  - passed to the area where the problem is handled
- The type of object determines which exception handler will be used

# Syntax

```
try {
// a block of code which might generate an exception
}
catch(xNoDisk) {
  // the exception handler(tell the user to
  // insert a disk)
}
catch(xNoMemory) {
  // another exception handler for this “try block”
}
```

# The Exception Class

- Define like any other class:

```
class Set {
private:
    int *pData;
public:
    ...
    class  xBadIndex {}; // just like any other class
};
```

# Throwing An Exception

- In your code where you reach an error node:

```
if(memberIndex < 0)

        throw xBadIndex();
```

- Exception processing now looks for a catch block which can handle your thrown object
- If there is no corresponding catch block in the immediate context, the call stack is examined

# The Call Stack

- As your program executes, and functions are called, the return address for each function is stored on a push down stack
- At runtime, the program uses the stack to return to the calling function
- Exception handling uses it to find a catch block

# Passing The Exception

- The exception is passed up the call stack until an appropriate catch block is found
- As the exception is passed up, the destructors for objects on the data stack are called
- There is no going back once the exception is raised

# Handling The Exception

- Once an appropriate catch block is found, the code in the catch block is executed
- Control is then given to the statement after the group of catch blocks
- Only the active handler most recently encountered in the thread of control will be invoked

# Handling The Exception (cont'd)

```
catch (Set::xBadIndex) {
// display an error message
}
catch (Set::xBadData) {
// handle this other exception
}
//control is given back here
```

- If no appropriate catch block is found, and the stack is at main(), the program exits

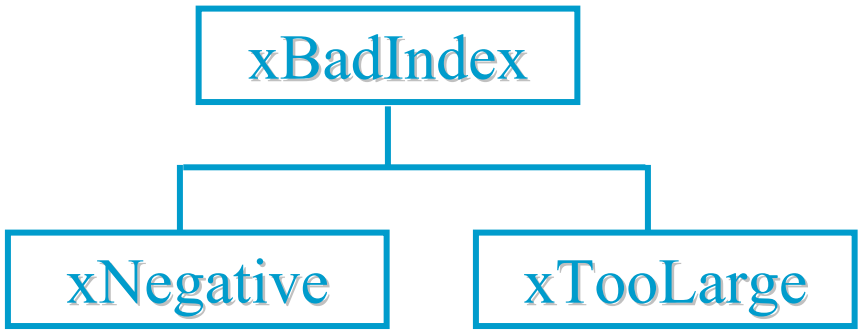
# Default catch Specifications

- Similar to the switch statement

```
catch (Set::xBadIndex)
{ // display an error message }
catch (Set::xBadData)
{ // handle this other exception }
catch (…)
{ // handle any other exception }
```

# Exception Hierarchies

- Exception classes are just like every other class; you can derive classes from them
- So one try/catch block might catch all bad indices, and another might catch only negative bad indices



# Exception Hierarchies (cont'd)

```
class Set {
private:
    int *pData;
public:
    class xBadIndex {};
    class xNegative : public xBadIndex {};
    class xTooLarge: public xBadIndex {};
};
   // throwing xNegative will be
   // caught by xBadIndex, too
```

# Data in Exceptions

- Since Exceptions are just like other classes, they can have data and member functions
- You can pass data along with the exception object
- An example is to pass an error subtype for xBadIndex, you could throw the type of bad index

# Data in Exceptions (Continued)

```
// Add member data,ctor,dtor,accessor method
class xBadIndex {
private:
    int badIndex;
public:
    xBadIndex(int iType):badIndex(iType) {}
    int GetBadIndex () { return badIndex; }
    ~xBadIndex() {}
};
```

# Passing Data In Exceptions

```
// the place in the code where the index is used
if (index <  0)
    throw xBadIndex(index);
if (index > MAX)
    throw xBadIndex(index);
// index is ok
```

# Getting Data From Exceptions

```
catch (Set::xBadIndex &theException)
{
   int badIndex = theException.GetBadIndex();
   if (badIndex < 0 )
      cout << “Set Index “ << badIndex << “ less than 0”;
   else
      cout << “Set Index “ << badIndex << “ too large”;
   cout << endl;
}
```

# Passing Data In Exceptions

```
// the place in the code where the index is used
if (index <  0)
    throw xNegative (index);
if (index > MAX)
    throw xTooLarge(index);
// index is ok
```

# Getting Data From Exceptions

```
catch (Set::xNegative &theException)
{
   int badIndex = theException.GetBadIndex();
  cout << "Set Index " << badIndex << " less than 0";
  cout << endl;
}
```

# Getting Data From Exceptions

```
catch (Set::xTooLarge &theException)
{
   int badIndex = theException.GetBadIndex();
  cout << “Set Index “ << badIndex << “ is too large”;
  cout << endl;
}
```

# Caution

- When you write an exception handler, stay aware of the problem that caused it
- Example: if the exception handler is for an out of memory condition, you shouldn't have statements in your exception object constructor which allocate memory

# Exceptions With Templates

- You can create a single exception for all instances of a template
  - declare the exception outside of the template
- You can create an exception for each instance of the template
  - declare the exception inside the template

# Single Template Exception

```
class xSingleException {};

template <class T>
class Set {
private:
    T *pType;
public:
    Set();
    T& operator[] (int index) const;
};
```

# Each Template Exception

```
template <class T>
class Set {
private:
    T *pType;
public:
    class xEachException {};
    T& operator[] (int index) const;
};
// throw xEachException();
```

# Catching Template Exceptions

- Single Exception (declared outside the template class)

```
catch (xSingleException)
```

- Each Exception (declared inside the template class)

```
catch (Set<int>::xEachException)
```

# Exception Specification

- A function that might throw an exception can warn its users by specifying a list of the exceptions that it can throw.

```
class Zerodivide{/*..*/};
int divide (int, int) throw(Zerodivide);
```

- If your function never throws any exceptions

```
bool equals (int, int) throw();
```

- Note that a function that is declared without an exception specification such as bool equals (int, int); guarantees nothing about its exceptions: It might throw any exception, or it might throw no exceptions.

# Exception Specification

- Exception Specifications Are Enforced At Runtime
- When a function attempts to throw an exception that it is not allowed to throw according to its exception specification, the exception handling mechanism detects the violation and invokes the standard function **unexpected()**.
- The default behavior of **unexpected()** is to call **terminate()**, which terminates the program.
- The default behavior can be altered, nonetheless, by using the function **set_unexpected()**.

# Exception Specification

- Because exception specifications are enforced only at runtime, the compiler might deliberately ignore code that seemingly violates exception specifications.
- Consider the following:

```
int f(); //no exception specification
```

- What if f throws an exception

```
void g(int j) throw()
{
        int result = f();
}
```

# Concordance of Exception Specification

C++ requires exception specification concordance in derived classes. This means that an overriding virtual function in a derived class has to have an exception specification that is at least as restrictive as the exception specification of the overridden function in the base class.

```
class BaseEx{};
class DerivedEx: public BaseEx{};
class OtherEx {};
```

```
class A {
public:
    virtual void f() throw (BaseEx);
    virtual void g() throw (BaseEx);
    virtual void h() throw (DerivedEx);
    virtual void i() throw (DerivedEx);
    virtual void j() throw(BaseEx);
};
```

```
class D: public A {
public:
    void f() throw (DerivedEx); //OK
    void g() throw (OtherEx); //error
    void h() throw (DerivedEx); //OK
    void i() throw (BaseEx); //error
    void j() throw (BaseEx,OtherEx); //error
};
```

# Concordance of Exception Specification

An exception could belong to two groups:

```
class Netfile_err : public Network_err, public File_system_err {
        /* ... */
};
```

Netfile_err can be caught by functions dealing with network exceptions:

```
void f(){
    try {
        / / something
    }
    catch (Network_err& e) {
         // ...
    }
}
```

```
void g() {
     try {
          / / something else
     }
     catch(File_system_err& e) {
          / / ...
     }
}
```

# Exception Matching

```
void f() {
     try {
          throw E() ;
     }
     catch(H) {
          // when do we get here?
     }
}
```

The handler is invoked:

[1] If **H** is the same type as **E**.

[2] If **H** is an unambiguous public base of **E**.

[3] If **H** and **E** are pointer types and [1] or [2] holds for the types to which they refer.

[4] If **H** is a reference and [1] or [2] holds for the type to which **H** refers.

# Resource Management

When a function acquires a resource – that is, it opens a file, allocates some memory from the free store, sets an access control lock, etc., – it is often essential for the future running of the system that the resource be properly released.

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"w") ;
    // use f
    fclose(f) ;
}
```

# Resource Management

Fault-tolerant implementation using try-catch:

```
void use_  file(const char* fn)
{
      FILE* f = fopen(fn,"r") ;
      try {
          // use f
      }
      catch (Ex e) {
           fclose(f) ;
           throw e;
      }
      fclose(f) ;
}
```

```
void f(){
  try {
  …
  use_file(“c:\\dat.txt”);
   …
  }
  catch(SomeEx e){
  }
}
```

# Resource Management

The problem with this solution is that it is verbose, tedious, and potentially expensive.

```
class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a) { p = fopen(n,a) ; }
    File_ptr(FILE* pp) { p = pp; }
    ~File_  ptr() { fclose(p) ; }
    operator FILE*() { return p; }
};
```

```
void use_file(const char* fn) {
        File_ptr f(fn,"r") ;
        // use f
}
```

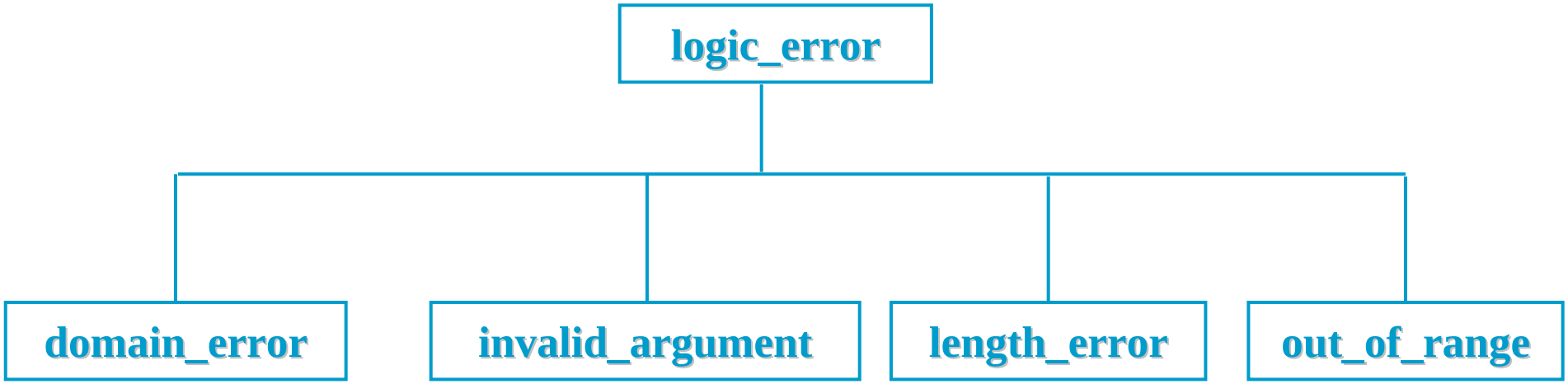
# Standard Exceptions

- The C++ standard includes some predefined exceptions, in <stdexcept>
- The base class is **exception**
  - Subclass **logic_error** is for errors which could have been avoided by writing the program differently
  - Subclass **runtime_error** is for other errors

# Standard Exceptions

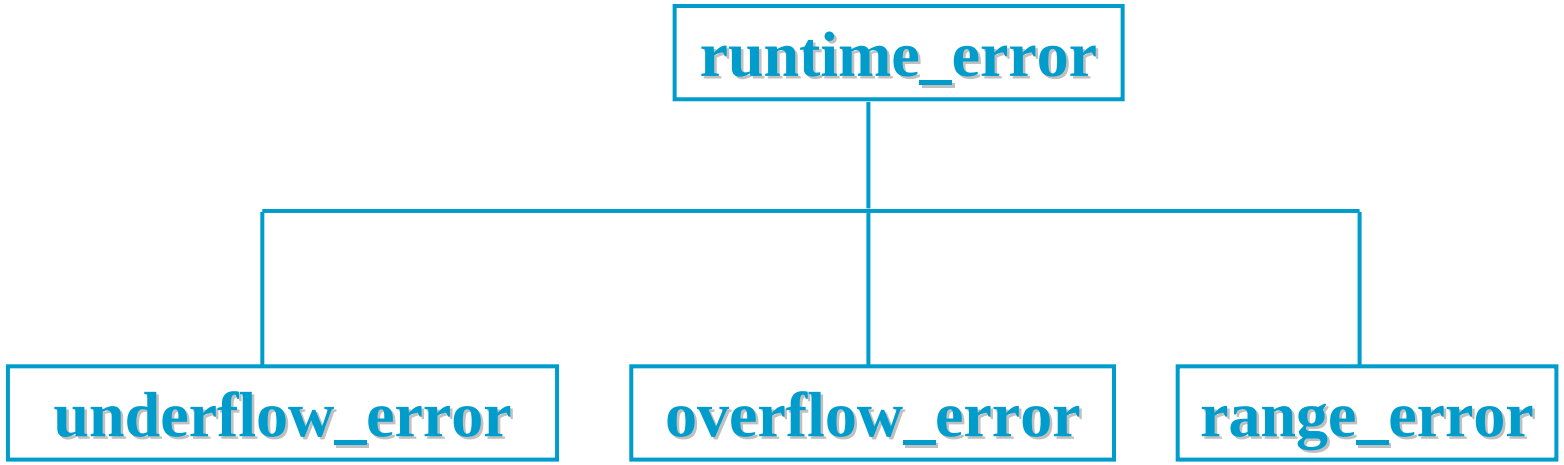
```
class exception {
public:
    exception() throw() ;
    exception(const exception&) throw() ;
    exception& operator=(const exception&) throw() ;
    virtual ~exception() throw() ;
    virtual const char*what() const throw() ;
private:
    / / ...
};
```

# Logic Error Hierarchy



# Runtime Error Hierarchy






The idea is to use one of the specific classes (e.g. range_error) to generate an exception

# Data For Standard Exceptions

```
// standard exceptions allow you to specify
// string information
throw overflow_error(“Doing float division in function div”);

// the exceptions all have the form:
class overflow_error : public runtime_error
{
public:
    overflow_error(const string& what_arg)
        : runtime_error(what_arg) {};
```

# Catching Standard Exceptions

```
catch (overflow_error)
{
    cout << “Overflow error” << endl;
}

catch (exception& e)
{
    cout << typeid(e).name() << “: “ << e.what() << endl;
}
```

# More Standard Exception Data

- catch (exception& e)
  - Catches all classes derived from *exception*
  - If the argument was of type *exception*, it would be converted from the derived class to the exception class
  - The handler gets a *reference to exception* as an argument, so it can look at the object

# RTTI (RunTime Type Information)

- It's one of the more recent additions to C++ and isn't supported by many older implementations. Other implementations may have compiler settings for turning RTTI on and off.
- The intent of RTTI is to provide a standard way for a program to determine the type of object during runtime.
- Many class libraries have already provided ways to do so for their own class objects, but in the absence of built-in support in C++, each vendor's mechanism typically is incompatible with those of other vendors.
- Creating a language standard for RTTI should allow future libraries to be compatible with each other.

# What is RTTI for?

- Suppose you have a hierarchy of classes descended from a common base. You can set a base class pointer to point to an object of any of the classes in this hierarchy. Next, you call a function that, after processing some information, selects one of these classes, creates an object of that type, and returns its address, which gets assigned to a base class pointer.
- How can you tell what kind of object it points to?

# How does it work?

C++ has three components supporting RTTI:

- **dynamic_cast** pointer
  generates a pointer to a derived type from a pointer to a base type, if possible. Otherwise, the operator returns 0, the null pointer.
- **typeid** operator
  returns a value identifying the exact type of an object.
- **type_info** structure
  holds information about a particular type.

**RTTI works only for classes with virtual functions**

# dynamic_cast<>

► The dynamic_cast operator is intended to be the most heavily used RTTI component.

► It doesn't answer the question of what type of object a pointer points to.

► Instead, it answers the question of whether you can safely assign the address of the object to a pointer of a particular type.

```
class Grand {  // has virtual methods} ;
class Superb : public Grand {  ... } ;
class Magnificent : public Superb {  ... } ;

Grand * pg = new Grand;
Grand * ps = new Superb;
Grand * pm = new Magnificent;

Magnificent * p1 = (Magnificent *) pm;      // #1
Magnificent * p2 = (Magnificent *) pg;      // #2
Superb * p3 = (Magnificent *) pm;           // #3
```

**Which of the previous type casts are safe?**

```
Superb pm = dynamic_cast<Superb *>(pg);
```

```
class Grand {
     virtual void speak() ;
} ;
class Superb : public Grand {
      void speak() ;
      virtual void say() ;
} ;
class Magnificent : public Superb {
       char ch ;
       void speak() ;
       void say() ;
} ;
```

```
for (int i = 0; i < 5; i++)
{
    pg = getOne();
    pg->speak();
    ...
```

►However, you can't use this exact approach to invoke the **say**() function; it's not defined for the Grand class.

►However, you can use the **dynamic_cast** operator to see if pg can be type cast to a pointer to Superb.

►This will be true if the object is either type Superb or Magnificent. In either case, you can invoke the **say**() function safely:

```
if (ps = dynamic_cast<Superb *>(pg))
    ps->say();
```

# typeid

- typeid is an operator which allows you to access the type of an object at runtime
- This is useful for pointers to derived classes
- typeid overloads ==, !=, and defines a member function name

```
if(typeid(*carType) == typeid(Ford))
    cout << "This is a Ford" << endl;
```

# typeid().name

```
cout << typeid(*carType).name() << endl;
// If we had said:
// carType = new Ford();
// The output would be:
// Ford
```

- So:

  `cout << typeid(e).name()`

  returns the name of the exception

# e.what()

- The class *exception* has a member function ***what***

```
virtual char* what();
```

- This is inherited by the derived classes
- what() returns the character string specified in the throw statement for the exception

throw overflow_error("Doing float division in function div");

cout << typeid(e).name() << ": " << e.what() << endl;

# Deriving New exception Classes

```
class xBadIndex : public runtime_error {
public:
    xBadIndex(const char *what_arg = “Bad Index”)
        :  runtime_error(what_arg) {}
};
// we inherit the virtual function what
// default supplementary information character string
```

```
template <class T>
class Array{
    private:
      T *data ;
      int Size ;
    public:
      Array(void);
      Array(int);
      class eNegativeIndex{};
      class eOutOfBounds{};
      class eEmptyArray{};
      T& operator[](int) ;
};
```

```
template <class T>
Array<T>::Array(void){
   data = NULL ;
   Size = 0    ;
}
template <class T>
Array<T>::Array(int size){
   Size = size ;
   data = new T[Size] ;
}
```

```
template <class T>
T& Array<T>::operator[](int index){
   if( data == NULL ) throw eEmptyArray() ;
   if(index < 0) throw eNegativeIndex() ;
   if(index >= Size) throw eOutOfBounds() ;
   return data[index] ;
}
```

```
Array<int> a(10) ;
try{
  int b = a[200] ;
}
catch(Array<int>::eEmptyArray){
  cout << "Empty Array" ;
}
catch(Array<int>::eNegativeIndex){
  cout << "Negative Array" ;
}
catch(Array<int>::eOutOfBounds){
  cout << "Out of bounds" ;
}
```

# 10 TEMPLATES

# Kalıp-Parametrik Çok Şekillilik Nedir?

Sınıflardaki fonksiyonların gövdeleri incelendiğinde, yapılan işlemler çoğu zaman, üzerinde işlem yapılan verinin tipinden bağımsızdır. Bu durumda fonksiyonun gövdesi, verinin tipi cinsinden, parametrik olarak ifade edilebilir:

```
int abs(int n) {
    return (n<0) ? -n : n;
}

long abs(long n) {
    return (n<0) ? -n : n;
}

float abs(float n) {
    return (n<0) ? -n : n;
}
```

## C

Her tip için farklı adlarda fonksiyonlar.
örnek mutlak değer fonksiyonları:

**abs**(), **fabs**(), **fabsl**(), **labs**(), **cabs**(), ...

## C++

Fonksiyonlara işlev yükleme bir çözüm olabilir mi?
İşlev yüklenen fonksiyonların gövdeleri değişmiyor !

Gövdeler tekrar ediliyor ⇒ Hata !

## Çözüm

Tipi parametre kabul eden bir yapı : Template

# Fonksiyon Kalıbı Tanımlamak

```
template <class T>
inline T const& max (T const& a, T const& b){
        return a<b?b:a;
 }
int main(){
  int i = 42;
  std::cout << "max(7,i): " << ::max(7,i) << std::endl;
  double f1 = 3.4;
  double f2 = -6.7;
  std::cout << "max(f1,f2): " << ::max(f1,f2) << std::endl;
  std::string s1 = "mathematics"; std::string s2 = "math";
  std::cout << "max(s1,s2): " << ::max(s1,s2) << std::endl;
}
```

```
max(4,7) // Tamam: Her iki argüman int
max(4,4.2) // Hata: ilk T int, ikinci T double

max<double>(4,4.2) // Tamam
max(static_cast<double>(4),4.2) // Tamam
```

# Örnek

```
template <class T>
   void printArray(T *array,const int size){
         for(int i=0;i < size;i++)
              cout << array[i] << " " ;
   cout << endl ;
  }
```

```
int main() {
        int       a[3]={1,2,3} ;
        double    b[5]={1.1,2.2,3.3,4.4,5.5} ;
        char      c[7]={'a', 'b', 'c', 'd', 'e' , 'f', 'g'} ;

        printArray(a,3)       ;
        printArray(b,5)       ;
        printArray(c,7)       ;
   return 0 ;
}
```

```
void printArray(int *array,cont int size){
        for(int i=0;i < size;i++)
            cout << array[i] << " " ;
   cout << endl ;
  }
void printArray(char *array,cont int size){
        for(int i=0;i < size;i++)
            cout << array[i] << "" ;
   cout << endl ;
  }
```

# **template**'in İşleyişi

Gerçekte derleyici template ile verilmiş fonksiyon gövdesi için herhangi bir kod üretmez. Çünkü template ile bazı verilerin tipi parametrik olarak ifade edilmiştir. Verinin tipi ancak bu fonksiyona ilişkin bir çağrı olduğunda ortaya çıkacaktır. Derleyici her farklı tip için yeni bir fonksiyon oluşturacaktır. template yeni fonksiyonun verinin tipine bağlı olarak nasıl oluşturulacağını tanımlamaktadır.



```
int intVar1 = 5;

cout << "abs(" << int << ")=" << abs(intVar1);
```

# **template**'in İşleyişi



- Programı ister template yapısı ile oluşturalım ister de template yapısı olmaksızın oluşturalım, programın bellekte kaplayacağı alan değişmeyecektir.

- Değişen, kaynak kodun boyu olacaktır. template yapısı kullanılarak oluşturulan programın kaynak kodu, daha anlaşılır ve hata denetimi daha yüksek olacaktır. Çünkü template yapısı kullanıldığında değişiklik sadece tek bir fonksiyon gövdesinde yapılacaktır.

# Template Parametresi bir Nesne Olabilir

```
class TComplex {       /* A class to define complex numbers */
    float real,imag;
  public:
            :   // other member functions
    bool operator>(const TComplex&) const ;
};
bool TComplex::operator>(const TComplex& z) const {
   float norm1 = real * real + imag * imag;
   float norm2 = z.real * z.real + z.imag * z.imag;
   if (norm1 > norm2) return true;
      else return false;
}
```

```
template <class T>
const T & max(const T &v1, const T & v2)
{
   if (v1 > v2) return v1;
      else       return v2;
}

int main(){
   int i1=5, i2= -3;
   char c1='D', c2='N';
   float f1=3.05, f2=12.47;
   TComplex z1(1.4,0.6), z2(4.6,-3.8);
   cout << ::max(i1,i2) << endl;
   cout << ::max(c1,c2) << endl;
   cout << ::max(f1,f2) << endl;
   cout << ::max(z1,z2) << endl;
}
```

# Çoklu template Parametreli Argümanlar



```
template <class atype>
int find(const atype *array, atype value, int size) {
    for(int j=0; j<size; j++)
       if(array[j]==value) return j;
    return -1;
}

char chrArr[] = {'a', 'c', 'f', 's', 'u', 'z'};    // array
char ch = 'f'; // value to find
int intArr[] = {1, 3, 5, 9, 11, 13};
int in = 6;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;
```

```
int main()
{
    cout << "\n 'f' in chrArray: index=" << find(chrArr, ch, 6);
    cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
    cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
}
```

# Örnek

```
template <class T>
void swap(T& x, T& y) {
        T temp ;
        temp = x ;
        x = y ;
        y = temp ;
}
char str1[100], str2[100] ;
int i,j ;
TComplex c1,c2;
swap( i , j ) ;
swap( c1 , c2 ) ;
swap( str1[50] , str2[50] ) ;
swap( i , str[25] ) ;
swap( str1 , str2 ) ;
```

hata

```
void swap(char* x, char* y) {
        int max_len ;
        max_len = (strlen(s1)>=strlen(s2)) ? strlen(s1):strlen(s2);
        char* temp = new char[max_len+1];
        strcpy(temp,s1);
         strcpy(s1,s2);
        strcpy(s2,temp);
        delete []temp;
}
```

# Tipsiz Template Parametreleri

```
template <typename T, int VAL>
T addValue (T const& x) {
   return x + VAL;
}
template <double VAT>
double process (double v) {
        return v * VAT;
 }
```

*double'a izin verilmez*

# İşlev Yüklemede Eşleme Önceliği

- Kalıp dışında aynı imzaya sahip fonksiyon
- Kalıp ile tanımlanmış aynı imzaya sahip fonksiyon

```
char str1[100], str2[100] ;
...
swap( str1 , str2 ) ;
```

```
template <class T>
void swap(T& x, T& y) {
        T temp ;
        temp = x ;
        x = y ;
        y = temp ;
}
```

```
void swap(char* x, char* y) {
    ...
}
```

# Çoklu template Parametreli Yapılar



- Template parametre sayısı birden fazla olabilir:

```
template <class atype, class btype>
 btype find(const atype* array, atype value, btype size) {
      for (btype j=0; j<size; j++)
           if(array[j]==value) return j;
      return (btype)-1;
   }
```

- Bu durumda, derleyici sadece farklı dizi tipleri için değil aynı zamanda aranan elemanın farklı tipte olması durumunda da farklı bir kod üretecektir:

```
short int result,si=100;
int invalue=5;
result = find(intArr, invalue,si) ;
```

# Sınıf template Yapısı

```
class Stack {
    int st[MAX];          // array of ints
    int top;              // index number of top of stack
  public:
    Stack();              // constructor
    void push(int var);   // takes int as argument
    int pop();            // returns int value
 };

class LongStack {
    long st[MAX];         // array of longs
    int top;              // index number of top of stack
  public:
    LongStack();          // constructor
    void push(long var); // takes long as argument
    long pop();           // returns long value
 };
```

```
template <class Type,int maxSize>
class Stack{
      Type st[maxSize];         // stack: array of any type
      int top;                  // number of top of stack
   public:
      Stack(){top = 0;}         // constructor
      void push(Type );         // put number on stack
      Type pop();               // take number off stack
};
template<class Type>
void Stack<Type>::push(Type var) // put number on stack
{
   if(top > maxSize-1)          // if stack full,
     throw "Stack is full!";    // throw exception
   st[top++] = var;
}
```

```
template<class Type>
Type Stack<Type>::pop() {        // take number off stack
  if(top <= 0)                   // if stack empty,
      throw "Stack is empty!";   // throw exception
  return st[--top];
}
```

```
int main()
{
  // s1 is object of class Stack<float>
  Stack<float,20> s1;
  // push 2 floats, pop 2 floats
  try{
    s1.push(1111.1);
    s1.push(2222.2);
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
  }
  // exception handler
  catch(const char * msg) {
     cout << msg << endl;
  }
```

```
  // s2 is object of class Stack<long>
  Stack<long,10> s2;
  // push 2 longs, pop 2 longs
  try{
    s2.push(123123123L);
    s2.push(234234234L);
    cout << "1: " << s2.pop() << endl;
    cout << "2: " << s2.pop() << endl;
  }
  // exception handler
  catch(const char * msg) {
     cout << msg << endl;
  }
}     // End of program
```

# Sınıf template Yapısının Farkı



►Template fonksiyonları için template parametresinin ne olacağını **derleyici** çağrı yapılan fonksiyon için imzaya bakarak karar verir.

►Template sınıflar için tanımlandığında template parametresini **programcı** verir.

```
Stack<float,20> s1;
Stack<long,10> s2;
```

```
swap( c1 , c2 ) ;
swap( str1[50] , str2[50] ) ;
```

# Neler Template Parametresi Olamaz?

```
template <typename T> class List { … };
typedef struct { double x, y, z; } Point;
typedef enum { red, green, blue } *ColorPtr;
int main() {
    struct Association { int* p; int* q; };
    List<Assocation*> error1;
    List<ColorPtr> error2;
    List<Point>;
}
```

*enum types*

*local types*

# Static Polymorphism×Dynamic Polymorphism

- Run-time Polymorphism vs. Compile-time Polymorphism
- Run-time Polymorphism:
  - Inheritance & virtual functions
- Compile-time Polymorphism
  - templates

```
class GeoObj {
  public:
    virtual void draw() const = 0;
    virtual Coord center_of_gravity() const = 0;
};
class Circle : public GeoObj {
  public:
    virtual void draw() const;
    virtual Coord center_of_gravity() const;
    …
};
class Line : public GeoObj {
  public:
    virtual void draw() const;
    virtual Coord center_of_gravity() const;
    …
};
```

## Run-time Polymorphism

```
void myDraw (GeoObj const& obj)
{
   obj.draw();
}

int main()
{
   Line l;
   Circle c, c1, c2;

   myDraw(l);          // myDraw(GeoObj&) => Line::draw()
   myDraw(c);          // myDraw(GeoObj&) => Circle::draw()
```

```
// concrete geometric object class Circle
// - not derived from any class
class Circle {
  public:
    void draw() const;
    Coord center_of_gravity() const;
    …
};
// concrete geometric object class Line
// - not derived from any class
class Line {
  public:
    void draw() const;
    Coord center_of_gravity() const;
    …
};
```

## Compile-time Polymorphism

```
template <typename GeoObj>
void myDraw (GeoObj const& obj)
{
   obj.draw();
}

int main()
{
   Line l;
   Circle c, c1, c2;

   myDraw(l);    // myDraw<Line>(GeoObj&)=>Line::draw()
   myDraw(c);    // myDraw<Circle>(GeoObj&)=>Circle::draw()
```

# Advantages & Disadvantages

Dynamic polymorphism in C++:

- Heterogeneous collections are handled elegantly.
- The executable code size is potentially smaller (because only one polymorphic function is needed, whereas distinct template instances must be generated to handle different types).
- Code can be entirely compiled; hence no implementation source must be published (distributing template libraries usually requires distribution of the source code of the template implementations).

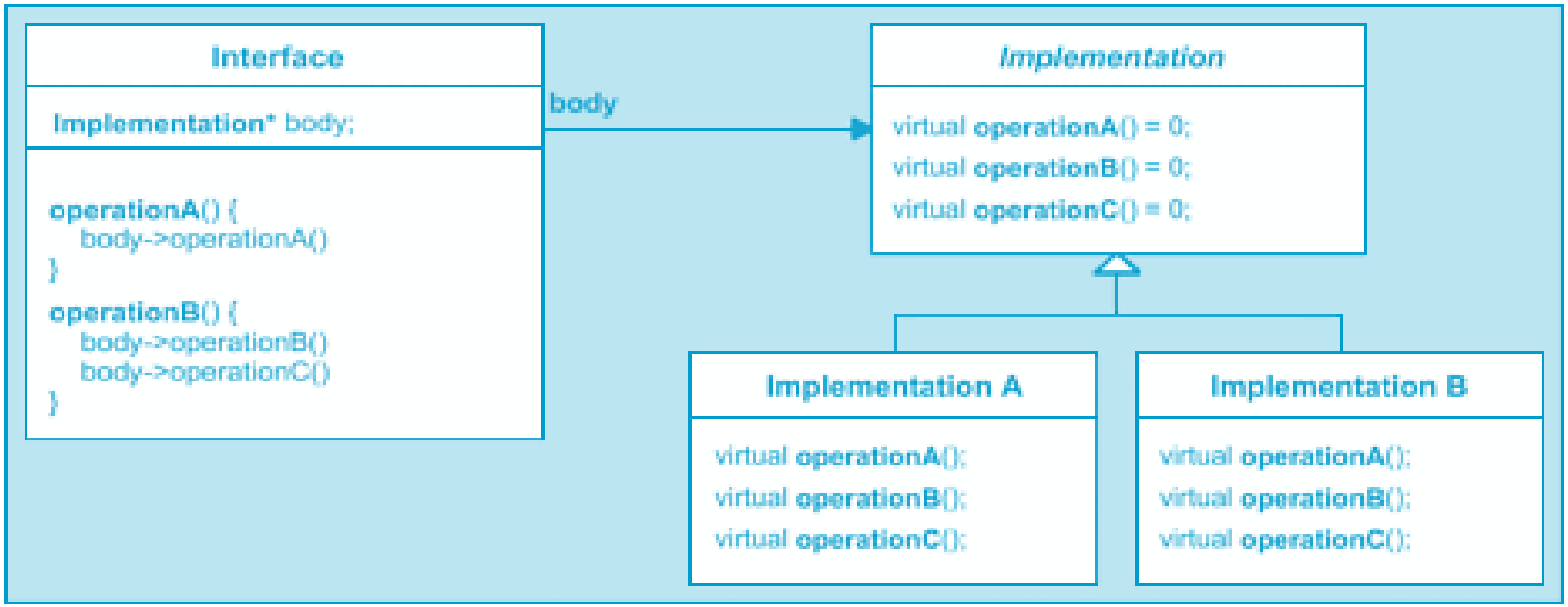
# Advantages & Disadvantages

In contrast, the following can be said about static polymorphism in C++:

- Collections of built-in types are easily implemented. More generally, the interface commonality need not be expressed through a common base class.
- Generated code is potentially faster (because no indirection through pointers is needed a priori and nonvirtual functions can be inlined much more often)
- Concrete types that provide only partial interfaces can still be used if only that part ends up being exercised by the application.

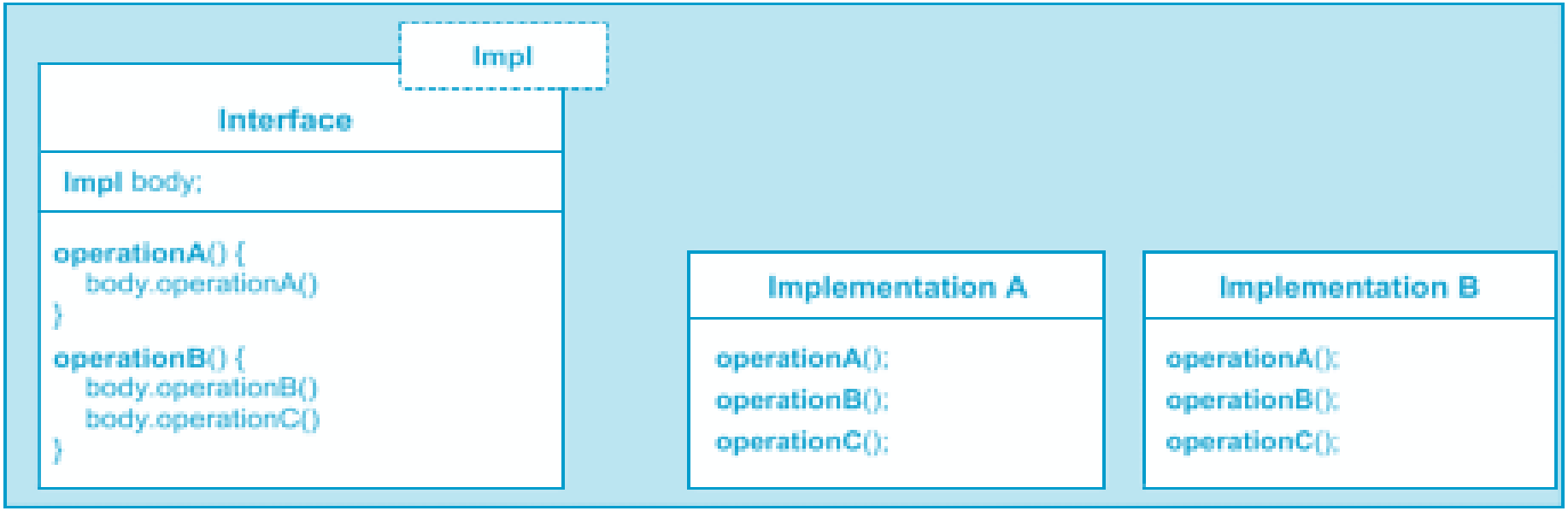
# New Approaches–Design Patterns

- A Design Pattern
  - "Bridge Pattern"
- Inheritance based implementation



# New Approaches–Design Patterns

- Implementation with template



# 11 Generic Programming (*with* STL *in* C++)

# Standard Template Library

Nesneye dayalı programlamada, verinin birincil öneme sahip programlama birimi olduğunu belirtmiştik. Veri, fiziksel yada soyut bir çok büyüklüğü modelleyebilir. Bu model oldukça basit yada karmaşık olabilir. Her nasıl olursa olsun, veri mutlaka bellekte saklanmaktadır ve veriye benzer biçimlerde erişilmektedir. C++, oldukça karmaşık veri tiplerini ve yapılarını oluşturmamıza olanak sağlayan mekanizmalara sahiptir. Genel olarak, programların, bu veri yapılarına belirli bazı biçimlerde eriştiğini biliyoruz:

array, list, stack, queue, vector, map, ...

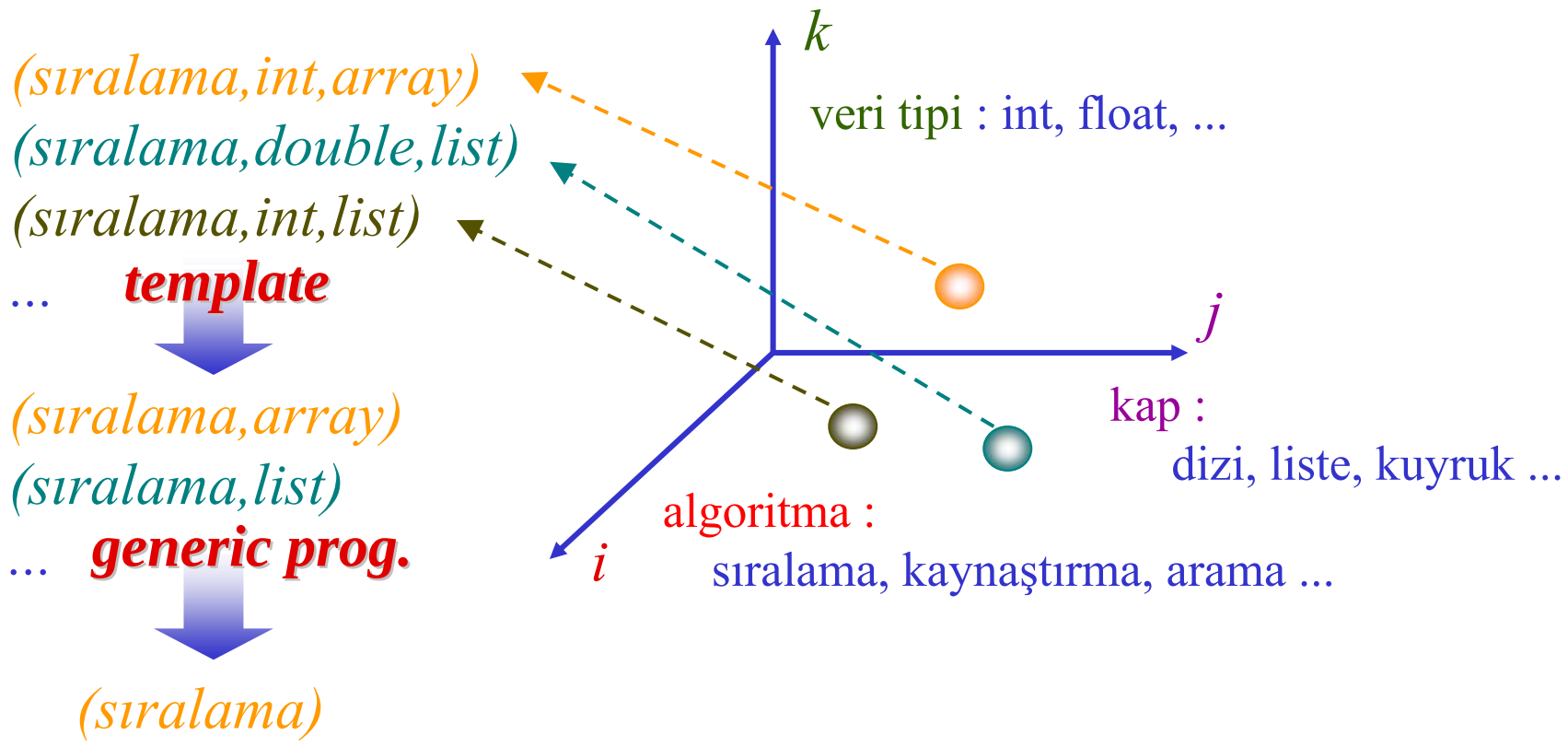
STL kütüphanesi verinin bellekteki organizasyonuna, erişimine ve işlenmesine yönelik çeşitli yöntemler sunmaktadır. Bu bölümde bu yöntemleri inceleyeceğiz.

*Standard Template Library (STL) Hewlett Packard'ın Palo Alto ( California)'daki laboratuarlarında Alexander Stepanov ve Meng Lee tarafından geliştirilmiştir.*

*1970'lerin sonlarında Alexander Stepanov bir kısım algoritmaların veri yapısının nasıl depolandıklarından bağımsız olduklarını gözlemledi. Örneğin, sıralama algoritmalarında sıralanacak sayıların bir dizide mi? yoksa bir listede mi? bulunduğunun bir önemi yoktur. Değişen sadece bir sonraki elemana nasıl erişildiği ile ilgilidir. Stepanov bu ve benzeri algoritmaları inceleyerek, algoritmaları veri yapısından bağımsız olarak performanstan ödün vermeksizin soyutlamayı başarmıştır. Bu fikrini 1985'de Generic ADA dilinde gerçekleştirmiştir. Ancak o dönemde henüz C++'da bir önceki bölümde incelediğimiz Template yapısı bulunmadığı için bu fikrini C++'da ancak 1992 yılında gerçekleştirebilmiştir.*

# Generic Programming

Bir yazılım ürününün bileşenlerini, üç boyutlu uzayda bir nokta olarak düşünebiliriz :



# STL Bileşenleri

STL üç temel bileşenden oluşmaktadır:

- **Algoritma**,
- **Kap (Container):** nesneleri depolamak ve yönetmekten sorumlu nesne,
  - Lineer Kaplar : Vector, Deque, List
  - Asosyatif Kaplar : Set, Map, Multi-set, Multi-map
- **Yineleyici (Iterator)**: algoritmanın farklı tipte kaplarla çalışmasını sağlayacak şekilde erişimin soyutlar.

# Kaplar

| Container | Description | Required Header |
|---|---|---|
| bitset | A set of bits | <bitset> |
| deque | A double-ended queue | <deque> |
| list | A linear list | <list> |
| map | Stores key/value pairs in which each key is associated with only one value | <map> |
| multimap | Stores key/value pairs in which one key may be associated with two or more values | <map> |
| multiset | A set in which each element is not necessarily unique | <set> |
| priority_queue | A priority queue | <queue> |
| queue | A queue | <queue> |
| set | A set in which each element is unique | <set> |
| stack | A stack | <stack> |
| vector | A dynamic array | <vector> |

C++'da sabit boyutlu dizi tanımlamak yürütme zamanında belleğin ya kötü kullanılmasına yada dizi boyunun yetersiz kalmasına neden olmaktadır.

STL kütüphanesindeki **vector** kabı bu sorunları gidermektedir.

STL kütüphanesindeki **list** kabı, bağlantılı liste yapısıdır.

**deque** (D*ouble-Ended QUEue*) kabı, yığın ve kuyruk yapılarının birleşimi olarak düşünülebilir. deque kabı her iki uçtan veri eklemeye ve silmeye olanak sağlamaktadır.

| | | |
|---|---|---|
| **Vector** | Relocating, expandable array | Quick random access (by index number).<br>Slow to insert or erase in the middle.<br>Quick to insert or erase at end. |
| **List** | Doubly linked list<br>Quick access to both ends. | Quick to insert or delete at any location.<br>Slow random access. |
| **Deque** | Like vector, but can be accessed at either end | Quick random access (using index number).<br>Slow to insert or erase in the middle.<br>Quick to insert or erase (push and pop) at either the beginning or the end. |

# Vector

- #include <vector>
- Kurucular
  - Boş: vector<string> object;
  - Belirli sayıda eleman:
    - vector<string> object(5,string("hello")) ;
    - vector<string> container(10)
    - vector<string> object(&container[5], &container[9]);
    - vector<string> object(container) ;

# Vector Member Functions

- Type &vector::back(): returns the reference to the last element
- Type &vector::front():returns the reference to the first element
- vector::iterator vector::begin()
- vector::iterator vector::end()
- vector::clear()
- bool vector::empty()
- vector::iterator vector::erase()
  - erase(pos)
  - erase(first,beyond)

# Vector Member Functions

- vector::insert
  - vector::iterator insert(pos)
  - vector::iterator insert(pos,value)
  - vector::iterator insert(pos,first,beyond)
  - vector::iterator insert(pos,n,value)
- void vector::pop_back()
- void vector::push_back(value)
- vector::resize()
  - resize(n,value)
- vector::swap()
  - vector<int> v1(7),v2(10) ;
  - v1.swap(v2);
- unsigned vector::size()

# Örnek

```
vector<int> v;
cout << v.capacity() << v.size() ;
v.insert(v.end(),3) ;                          v = (3)
cout << v.capacity() << v.size() ;
v.insert (v.begin(), 2, 5);                    v = (5,5,3)
```

```
vector<int> w (4,9);                           w = (9,9,9,9)
w.insert(w.end(), v.begin(), v.end() );        w = (9,9,9,9,5,5,3)
w.swap(v) ;          v = (9,9,9,9,5,5,3)       w=(5,5,3)
```

```
w.erase(w.begin());                            w = (5,3)
w.erase(w.begin(),w.end()) ;
cout << w.empty() ? "Empty" : "not Empty"      Empty
```

```
#define __USE_STL
// STL include files
#include <vector>
#include <list>
```

| | |
|---|---|
| `vector<int> v;` | |
| `v.insert(v.end(),3) ;` | **v = (3)** |
| `v.insert(v.begin(),5) ;` | **v = (5,3)** |
| `cout << v.front() << endl;` | **5** |
| `cout << v.back() ;` | **3** |
| `v.pop_back();` | |
| `cout << v.back() ;` | **5** |

# List

- #include <list>
- Eklenecek eleman sayısı belirli olmadığı durumlarda uygundur
- Kurucular
  - Boş: list<string> object;
  - Belirli sayıda eleman:
    - list<string> object(5,string("hello")) ;
    - list<string> container(10)
    - list<string> object(&container[5], &container[9]);
    - list<string> object(container) ;

# List Member Functions

- Type &list::back(): returns the reference to the last element
- Type &list::front():returns the reference to the first element
- list::iterator list::begin()
- list::iterator list::end()
- list::clear()
- bool list::empty()
- list::iterator list::erase()
  - erase(pos)
  - erase(first,beyond)

# List Member Functions

- list::insert
  - list::iterator insert(pos)
  - list::iterator insert(pos,value)
  - list::iterator insert(pos,first,beyond)
  - list::iterator insert(pos,n,value)
- void list::pop_back()
- void list::push_back(value)
- list::resize()
  - resize(n,value)
- void list<type>::merge(list<type> other)
- void list<type>::remove(value)
- unsigned list::size()


list1.cpp


list2.cpp

# List Member Functions

- list::sort()
- void list::splice(pos,object)


list3.cpp

- void list::unique(): operates on sorted list, removes consecutive identical elements


list4.cpp

# Queue

- #include <queue>
- FIFO (=First In First Out)
- Kurucular
  - Boş: queue<string> object;
  - Kopya Kurucu: queue<string> object(container) ;

# Queue Member Functions

- Type &queue::back(): returns the reference to the last element
- Type &queue::front(): returns the reference to the first element
- bool queue::empty()
- void queue::push(value)
- void queue::pop()

# Priority_Queue

- #include <queue>
- Temel olarak queue ile aynı
- Kuyruğa ekleme belirli bir önceliğe göre yürütülür
- Öncelik: operator<()


priqueue1.cpp


priqueue2.cpp

# Priority_Queue Member Functions

- Type &queu::back(): returns the reference to the last element
- Type &queue::front():returns the reference to the first element
- bool queue::empty()
- void queue::push(value)
- void queue::pop()

# Deque

- #include <deque>
- Head &Tail, Doubly Linked
- deque<string> object
  - deque<string> object(5,string("hello")) ;
  - deque<string> container(10)
  - deque<string> object(&container[5], &container[9]);
  - deque<string> object(container) ;

# Deque Member Functions

- Type &deque::back(): returns the reference to the last element
- Type &deque::front():returns the reference to the first element
- deque::iterator deque::begin()
- deque::iterator deque::end()
- deque::clear()
- bool deque::empty()
- deque::iterator deque::erase()
  - erase(pos)
  - erase(first,beyond)

# Deque Member Functions

- vector::insert
  - deque::iterator insert(pos)
  - deque::iterator insert(pos,value)
  - deque::iterator insert(pos,first,beyond)
  - deque::iterator insert(pos,n,value)
- void deque::pop_back()
- void deque::push_back(value)
- deque::resize()
  - resize(n,value)
- deque::swap()
- unsigned deque::size()

# Asosyatif Kaplar: Set, Multiset, Map, Multimap

- Set sıralı küme oluşturmak için kullanılır.

```
#include <set>
using namespace std;
int main(){
   string names[] = {"Katie", "Robert","Mary", "Amanda", "Marie"};
   set<string> nameSet(names, names+5);// initialize set to array
   set<string>::const_iterator iter;  // iterator to set
   nameSet.insert("Jack");            // insert some more names
   nameSet.insert("Larry");
   nameSet.insert("Robert");          // no effect; already in set
   nameSet.insert("Barry");
   nameSet.erase("Mary");             // erase a name
```

```
    cout << "\nSize=" << nameSet.size() << endl;
    iter = nameSet.begin();              // display members of set
    while( iter != nameSet.end() )
       cout << *iter++ << '\n';
    string searchName;                   // get name from user
    cout << "\nEnter name to search for: ";
    cin >> searchName;     // find matching name in set
    iter = nameSet.find(searchName);
    if( iter == nameSet.end() )
       cout << "The name" << searchName << " is NOT in the set.";
    else
       cout << "The name " << *iter << " IS in the set.";
}
```

```
// set2.cpp set
int main() {
   set<string> city;
   set<string>::iterator iter;
   city.insert("Trabzon");     // insert city names
   city.insert("Adana");
   city.insert("Edirne");
   city.insert("Bursa");
   city.insert(“Istanbul");
   city.insert("Rize");
   city.insert("Antalya");
   city.insert(“Izmir");
   city.insert("Hatay");
   city.insert("Ankara");
   city.insert("Zonguldak");
```

```
iter = city.begin();          // display set
   while( iter != city.end() )
      cout << *iter++ << endl;

   string lower, upper;          // display entries in range
   cout << "\nEnter range (example A Azz): ";
   cin >> lower >> upper;
   iter = city.lower_bound(lower);
   while( iter != city.upper_bound(upper) )
      cout << *iter++ << endl;
}
```

# Map

- #include <map>
- Key/Value pairs
- map<string,int> object
  - pair<string,int>

```
pa[]= {
        pair<string,int>("one",1),
        pair<string,int>("two",2),
        pair<string,int>("three",3),
        pair<string,int>("four",4)
      } ;
```

  - map<string,int> object(&pa[0],&pa[3]);
- object["two"]

# Map Member Functions

- map::insert
  - pair<map::iterator,bool> insert(keyvalue)
  - pair<map::iterator,bool> insert(pos,keyvalue)
  - void insert(first,beyond)
- map::iterator map::lower_bound(key)
- map::iterator map::upper_bound(key)
- pair<map::iterator,map::iterator> map::equal_range(key)
- map::iterator map::find(key)
  - returns map::end() if not found
- unsigned deque::size()

# Map Member Functions

- map::iterator map::begin()
- map::iterator map::end()
- map::clear()
- bool map::empty()
- map::iterator map::erase()
  - erase(keyvalue)
  - erase(pos)
  - erase(first,beyond)

# Örnek

```
int main(){
   map<string,int> city_num;
   city_num["Trabzon"]=61;
    ...
   string city_name;
   cout << "\nEnter a city: ";
   cin >> city_name;
   if (city_num.end()== city_num.find(city_name))
      cout << city_name << " is not in the database" << endl;
   else
      cout << "Number of " << city_name << ": " << city_num[city_name];
}
```

# MultiMap

- #include <map>
- Main difference between map and multimap is that the multimap supports multiple entries of values having the same keys and the same values.

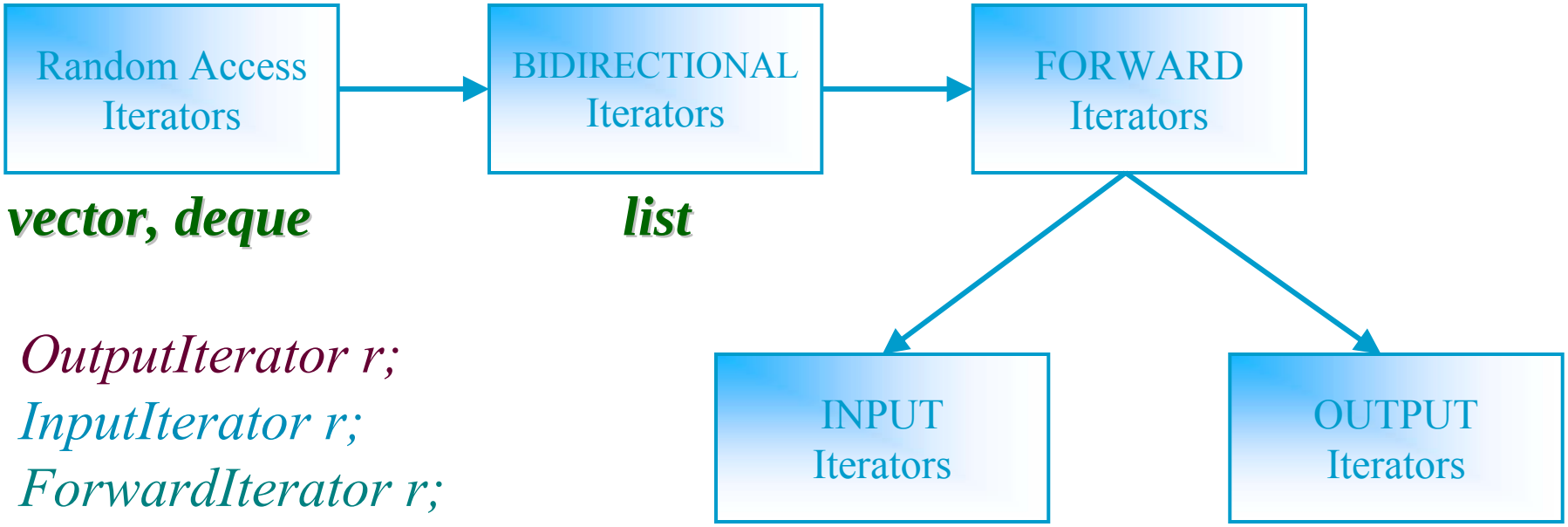
# Özetçe

| İşlem | Yürütülen İşlem |
|---|---|
| a.size() | a.end() – a.begin() |
| a.max_size() | |
| a.empty() | a.size() == 0 |

| İşlem | Dönüş Değeri | Yürütülen İşlem | Uygulanabildiği Kaplar |
|---|---|---|---|
| a.front() | T& | *a.begin() | vector, list, deque |
| a.back() | T& | *a.end() | vector, list, deque |
| a.push_front(x) | void | a.insert(a.begin(),x) | list,deque |
| a.push_back(x) | void | a.insert(a.end(),x) | vector, list,deque |
| a.pop_front() | void | a.erase(a.begin()) | list,deque |
| a.pop_back() | void | a.erase(--a.end()) | list,deque |
| a[n] | T& | *(a.begin()+n) | vector,deque |

# Iterators

Iterators : Genelleştirilmiş İşaretçi




*OutputIterator r;*
*InputIterator r;*
*ForwardIterator r;*
*BidirectionalIterator r;*
*RandomIterator r ;*

# Iterator Capability



| Iterator Capability | Input | Output | Forward | Bidirectional | Random Access |
|---|---|---|---|---|---|
| Dereferencing read | yes | no | yes | yes | yes |
| Dereferencing write | no | yes | yes | yes | yes |
| Fixed and repeatable order | no | no | yes | yes | yes |
| ++i<br>i++ | yes | yes | yes | yes | yes |
| --i<br>i-- | no | no | no | yes | yes |
| i[n] | no | no | no | no | yes |
| i + n | no | no | no | no | yes |
| i - n | no | no | no | no | yes |
| i += n | no | no | no | no | yes |
| i -=n | no | no | no | no | yes |

# Output Iterators

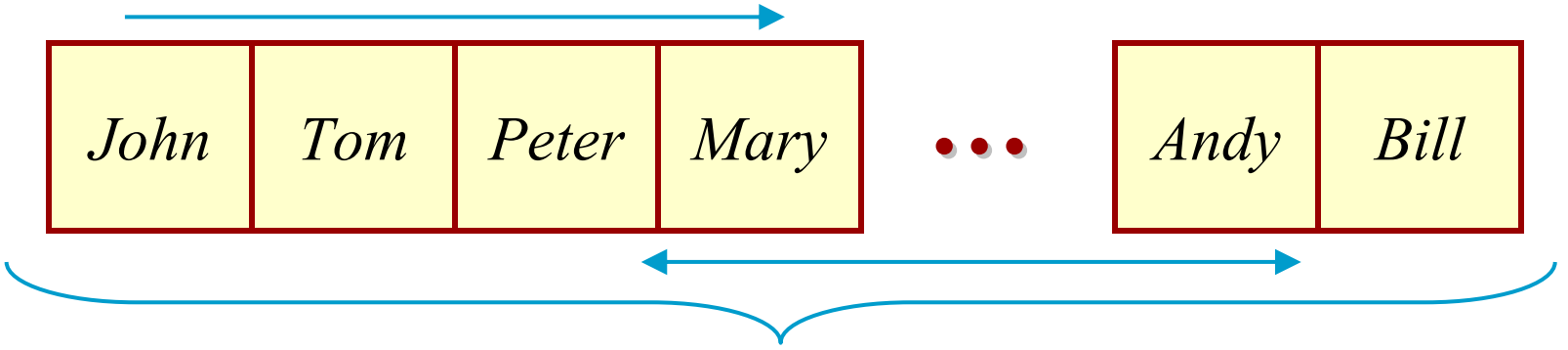
1
```
OutputIterator a ;
...
*a=t ;
t = *a ;          Hata
```

2
```
OutputIterator r ;
...
*r=0 ;
*r=1 ;            Hata
```

3
```
OutputIterator r ;
...
r++ ;
r++ ;             Hata
```

4
```
OutputIterator i,j ;
...
i=j ;
*i++=a ;          Hata
*j=b ;
```

# Forward and Bidirectional Iterators




range[v.begin(),v.end()]

```
list<int> l (1,1) ;
l.push_back(2) ; // list l : 1 2
list<int>::iterator first=l.begin() ;
list<int>::iterator last=l.end() ;
while( last != first){
      -- last ;
      cout << *last << " " ;
}
```

```
template<class ForwardIterator, class T>
ForwardIterator find_linear(ForwardIterator first,
                            ForwardIterator last, T& value){
      while( first != last) if( *first++ == value) return first;
                                  else return last ;
}

vector<int> v(3,1) ;
v.push_back(7);  // vector : 1 1 1 7
vector<int>::iterator i=find_linear(v.begin(), v.end(),7) ;
if(i != v.end() ) cout << *i ;
else cout << "not found!" ;
```

# Bubble Sort

```
template<class Compare>
void bubble_sort(BidirectionalIterator first,
                 BidirectionalIterator last, Compare comp){
 BidirectionalIterator left = first , right = first ;
 right ++ ;
   while( first != last){
       while( right != last ){
              if( comp(*right,*left) )
                 iter_swap(left,right) ;
              right++ ;
              left++;
       }
       last -- ;
       left = first ; right = first ;
   }
}
```

```
list<int> l ;
bubble_sort(l.begin(),l.end(),less<int>()) ;
bubble_sort(l.begin(),l.end(),greater<int>()) ;
```

# Random Access Iterators

```
vector<int> v(1,1) ;
v.push_back(2) ; v.push_back(3) ; v.push_back(4) ; // v : 1 2 3 4
vector<int>::iterator i=v.begin() ;
vector<int>::iterator j=i+2;
cout << *j << " " ;
i += 3 ; cout << *i << "" ;
j = i – 1 ; cout << *j << "" ;
j -= 2 ; cout << *j << "" ;
cout << v[1] << endl ;
(j<i) ? cout << "j < i" : cout << "not j < i" ; cout << endl ;
(j>i) ? cout << "j > i" : cout << "not j > i" ; cout << endl ;
(j>=i) && (j<=i)? cout << "j and i equal" : cout << "j and i not equal > i" ; cout <<
endl ;
i = j ;
j= v.begin();
i = v.end ;
cout << "iterator distance end – begin : " << (i-j) ;
```

# Iterator Operators

- STL provides two functions that return the number of elements between two elements and that jump from one element to any other element in the container:
  - distance()
  - advance()

# distance()

- The distance() function finds the distance between the current position of two iterators.

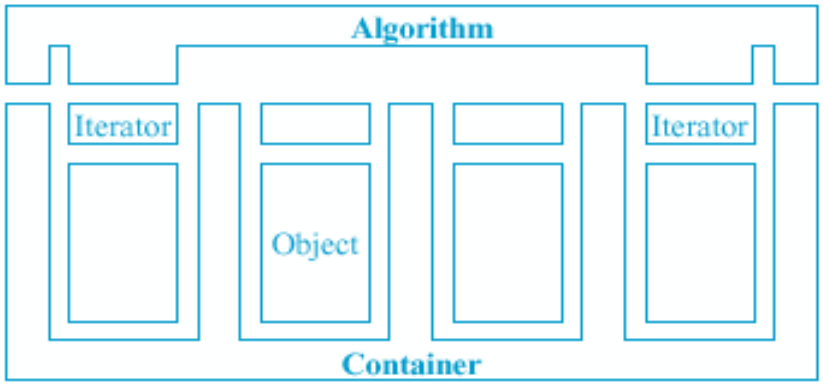
```
template<class RandomAccessIterator>
iterator_traits<RandomAccessIterator>::difference_type
distance(RandomAccessIterator first, RandomAccessIterator
  last) {
      return last – first;
}
template<class InputIterator>
iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last) {
iterator_traits<InputIterator>::difference_type n = 0;
      while (first++ != last) ++n;
       return n;
}
```

# advance()

- So far, we have seen how we can move iterators forward and backward by using the increment and decrement operators, respectively. We can also move random access iterators several steps at a time using the addition and subtraction functions. Other types of iterators, however, do not have the addition and subtraction functions.
- The STL provides the advance() function to move any iterator—except the output iterators—several steps at a time:

```
template<class InputIterator, class Distance>
void advance(InputIterator& ii, Distance& n) {
        while (n--) ++ii;
}
template<class BidirectionalIterator, class Distance>
void advance(BidirectionalIterator & bi, Distance& n) {
        if (n >= 0) while (n--) ++bi;
       else while (n++) --bi;
}
template<class RandomAccessIterator, class Distance>
void advance(RandomAccessIterator& ri, Distance& n) {
        ri += n;
}
```

# Designing Generic Algorithm




implementing an algorithm such as computing the maximum value in a sequence can be done without knowing the details of how values are stored in that sequence:

```
template <class Iterator>
Iterator max_element (Iterator beg,   // refers to start of collection
                      Iterator end)   // refers to end of collection
{
   …
}
```

# Binary Search for Integer Array

```
const int * binary_search(const int * array, int n, int x){
    const int *lo = array, *hi = array + n , *mid ;
    while( lo != hi ) {
            mid = lo + (hi-lo)/2 ;
            if( x == *mid ) return mid ;
            if( x < *mid ) hi = mid ;
            else lo = mid + 1 ;
    }
  return 0 ;
}
```

# Binary Search—Template Solution (Form-1)



```
template<class T>
const T * binary_search(const T * array, int n, T& x){
    const T *lo = array, *hi = array + n , *mid ;
    while( lo != hi ) {
            mid = lo + (hi-lo)/2 ;
            if( x == *mid ) return mid ;
            if( x < *mid ) hi = mid ;
            else lo = mid + 1 ;
    }
  return 0 ;
}
```

# Binary Search—Template Solution (Form-2)



```
template<class T>
const T * binary_search(T * first,T * last, T& x){
    const T *lo = first, *hi = last , *mid ;
    while( lo != hi ) {
            mid = lo + (hi-lo)/2 ;
            if( x == *mid ) return mid ;
            if( x < *mid ) hi = mid ;
            else lo = mid + 1 ;
    }
  return last ;
}
```

# Generic Binary Search

```
template<class RandomAccessIterator,class T>
const T * binary_search(RandomAccessIterator first,
                        RandomAccessIterator last, T& value){
 RandomAccessIterator not_found = last, mid ;
 RandomAccessIterator lo= first, hi=last ;
    while( lo != hi ) {
          mid = lo + (hi-lo)/2 ;
          if( value == *mid ) return mid ;
          if( value < *mid ) hi = mid ;
          else lo = mid + 1 ;
    }
  return not_found ;
}
```

# STL Algorithms

#include <algorithm>

| Algorithm | Purpose |
|---|---|
| find | Returns first element equivalent to a specified value |
| count | Counts the number of elements that have a specified value |
| equal | Compares the contents of two containers and returns true if all corresponding elements are equal |
| search | Looks for a sequence of values in one container that correspond with the same sequence in another container |
| copy | Copies a sequence of values from one container to another (or to a different location in the same container) |
| swap | Exchanges a value in one location with a value in another |
| iter_swap | Exchanges a sequence of values in one location with a sequence of values in another location |
| fill | Copies a value into a sequence of locations |
| sort | Sorts the values in a container according to a specified ordering |
| merge | Combines two sorted ranges of elements to make a larger sorted range |
| accumulate | Returns the sum of the elements in a given range |
| for_each | Executes a specified function for each element in the container |

# find()

- The find() algorithm looks for the first element in a container that has a specified value.
- find() example program shows how this looks when we're trying to find a value in an array of int's.

# Example

```
#include <iostream>
#include <algorithm>                    //for find()
int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };
int main() {
   int* ptr;
   ptr = find(arr, arr+8, 33);          //find first 33
   cout << “First object with value 33 found at offset ”
            << (ptr-arr) << endl;
   return 0;
}
```

# count()

- count() counts how many elements in a container have a specified value and returns this number.

```
#include <iostream>
#include <algorithm>       //for count()
int arr[] = { 33, 22, 33, 44, 33, 55, 66, 77 };

int main(){
    int n = count(arr, arr+8, 33);  //count number of 33's
    cout << "There are " << n << " 33's in arr." << endl;
    return 0;
}
```

# count_if()

- size_t count_if(InputIterator first, InputIterator last, Predicate predicate)

```
#include <vector>
#include <algorithm>  //for count_if()
int a[] = { 1, 2, 3, 4, 3, 4, 2, 1, 3 };
class Odd {
    public:
        bool operator()(int val){ return val&1 ; }
};
int main(){
    std::vector<int> iv(a,a+9) ;
    std::cout << count_if(iv.begin(),iv.end(),Odd()) ;
    return 0 ;
}
```

# equal()

- bool equal(InputIterator first, InputIterator last,
  InputIterator otherFirst)
- bool equal(InputIterator first, InputIterator last,
  InputIterator otherFirst,Predicate predicate)

```
class CaseString {
    public:
        bool operator()(string const &first,string const &second){
            return !strcasecmp(first.c_str(),second.c_str()) ;
        }
};
int main(){
    string
   first[]={"Alpha","bravo","Charley","echo","Delta","golf"},
   second[]={"alpha","Bravo","charley","Echo","delta","Golf"} ;
    std::string *last = first + sizeof(first)/sizeof(std::string) ;
    cout << (equal(first,last,second)?"Equal":"Not equal") ;
    cout << (equal(first,last,second,CaseString())?"Equal":
                                                    "Not equal") ;
    return 0 ;
}
```

# fill(),fill_n()

- void fill(ForwardIterator first,ForwardIterator last,
  Type const &value)

```
vector<int> iv(8) ;
fill(iv.begin(),iv.end(),8) ;
```

- void fill_n(ForwardIterator first,Size n,
  Type const &value)

```
vector<int> iv(8) ;
fill_n(iv.begin()+2,4,8) ;
```

# sort()

- You can guess what the sort() algorithm does.

  Here's an example:

```
#include <iostream>
#include <algorithm>
int arr[] = {45, 2, 22, -17, 0, -30, 25, 55};
int main(){
   sort(arr, arr+8);            //sort the numbers
   for(int j=0; j<8; j++)       //display sorted array
      cout << arr[j] << ' ';
   return 0;
}
```

# search()

- Some algorithms operate on two containers at once. For instance, while the find() algorithm looks for a specified value in a single container, the search() algorithm looks for a sequence of values, specified by one container, within another container.

```
int source[] =  { 11, 44, 33, 11, 22, 33, 11, 22, 44 };
int pattern[] = { 11, 22, 33 };
int main(){
 int* ptr;
 ptr = search(source, source+9, pattern, pattern+3);
 if(ptr == source+9)   cout << "No match found\n";
 else                  cout << "Match at " << (ptr - source) ;
  return 0;
}
```

# binary_search()

- #include <algorithm>
  - bool binary_search(ForwardIterator first, ForwardIterator last,Type const &value)
  - bool binary_search(ForwardIterator first, ForwardIterator last,Type const &value,Comparator comp)

# merge()



```
#include <iostream>
#include <algorithm>          //for merge()
using namespace std;
int src1[] = { 2, 3, 4, 6, 8 };
int src2[] = { 1, 3, 5 };
int dest[8];
int main(){                   //merge src1 and src2 into dest
      merge(src1, src1+5, src2, src2+3, dest);
      for (int j=0; j<8; j++)     //display dest
           cout << dest[j] << ' ';
      cout << endl;
      return 0;
}
```

# accumulate()

- #include <numeric>
  - Type accumulate(InputIterator first, InputIterator last,Type init)
    operator+() is applied to all elements and the result is returned
  - Type accumulate(InputIterator first, InputIterator last, Type init,BinaryOperation op)
    binary operator op() is applied to all elements

```
#include <iostream>
#include <numeric>
#include <vector>

int main(){
    int ia[]={1,2,3,4} ;
    std::vector<int> iv(ia,ia+4) ;

    cout << accumulate(iv.begin(),iv.end(),int()) << std::endl ;
    cout << accumulate(iv.begin(),iv.end(),int(1),multiplies<int>())
         << endl ;
    system("pause") ;
    return 0 ;
}
```

# adjacent_difference()

- #include <numeric>
  - OutputIterator adjacent_difference(InputIterator first, InputIterator last,OutputOperator result)
  - OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputOperator result,BinaryOperation op)

```
#include <iostream>
#include <numeric>
#include <vector>
int main(){
    int ia[]={1,3,7,23} ;
    std::vector<int> iv(ia,ia+4) ;
    std::vector<int> ov(iv.size()) ;
    adjacent_difference(iv.begin(),iv.end(),ov.begin()) ;
    copy(ov.begin(),ov.end(),std::ostream_iterator<int>(cout," ")) ;
    std::cout << std::endl ;

   adjacent_difference(iv.begin(),iv.end(),ov.begin(),minus<int>()) ;
    copy(ov.begin(),ov.end(),ostream_iterator<int>(cout," ")) ;
    system("pause") ;
    return 0 ;
}
```

# copy(), copy_backward()

- #include <algorithm>
  - OutputIterator copy(InputIterator first, InputIterator last, OutputIterator destination)
  - BidirectionalIterator copy(InputIterator first, InputIterator last, BidirectionalIterator last2)

# for_each

- Function for_each(ForwardIterator first,
  ForwardIterator last,Function func)

```
void lowerCase(char &c){
    c = static_cast<char>(tolower(c)) ;
}
void capitalizedOutput(std::string const &str){
    char *tmp = strcpy(new char[str.size()+1],str.c_str()) ;
    std::for_each(tmp+1,tmp+str.size(),lowerCase) ;

    tmp[0] = toupper(*tmp) ;
    std::cout << tmp << " " ;
    delete []tmp;
}
```

foreach1.cpp

```
int main(){
    std::string
        sarr[] =
        {
          "alpha", "BRAVO", "charley", "ECHO", "delta",
          "FOXTROT", "golf", "HOTEL"
        },
        *last = sarr + sizeof(sarr) / sizeof(std::string) ;
    void (*f)(std::string const&) ;
    f = std::for_each(sarr,last,capitalizedOutput) ;
    std::cout << std::endl ;
    f("alpha") ;
    std::cout << std::endl ;
    system("pause") ;
    return 0 ;
}
```

# Another Example

```
class Show{
        int d_count ;
    public:
        void operator()(std::string &str){
            for_each(str.begin(),str.end(),lowerCase) ;
            str[0] =toupper(str[0]);
            std::cout << ++d_count << " " << str << "; " ;
        }
        int getCount() const{
           return d_count ;
        }
} ;
```

foreach2.cpp

```
int main(){
    std::string
        sarr[] =  {
                    "alpha", "BRAVO", "charley", "ECHO", "delta",
                    "FOXTROT", "golf", "HOTEL"
                  },
                  *last = sarr + sizeof(sarr) / sizeof(std::string) ;
    cout << for_each(sarr,last,Show()).getCount() << endl ;
    system("pause") ;
    return 0 ;
}
```

# transform()



```
int _3_n_plus_1(int n) {
    return (n&1) ? 3*n+1 : n/2 ;
}
```

```
void show(int n) {
        std::cout << n << " " ;
 }
```

```
int main(){
    int iArr[] = { 5,2,23,76,33,44} ;
    std::for_each(iArr,iArr+6,show) ; std::cout << std::endl ;
    std::transform(iArr,iArr+6,iArr,_3_n_plus_1) ;
    std::for_each(iArr,iArr+6,show) ;
    system("pause") ;
    return 0 ;
}
```

transform.cpp

# Predicates in <functional>

- When the type of the return value of a unary function object is bool, the function is called a unary predicate. A binary function object that returns a bool value is called a binary predicate.
- The Standard C++ Library defines several common predicates in <functional>

**TABLE .** PREDICATES DEFINED IN `<functional>`

| *Function* | *Type* | *Description* |
|---|---|---|
| `equal_to` | binary | `arg1 == arg2` |
| `not_equal_to` | binary | `arg1 != arg2` |
| `greater` | binary | `arg1 > arg2` |
| `greater_equal` | binary | `arg1 >= arg2` |
| `less` | binary | `arg1 < arg2` |
| `less_equal` | binary | `arg1 <= arg2` |
| `logical_and` | binary | `arg1 && arg2` |
| `logical_or` | binary | `arg1 \|\| arg2` |
| `logical_not` | unary | `!arg1` |

```
template<class T>
class equal_to : binary_function<T, T, bool> {
bool operator()(T& arg1, T& arg2) const { return arg1 == arg2; }
};
```

# 12 STREAMS

# Streams



►A ***stream*** is a general name given to a flow of data in an input/output situation. For this reason, streams in C++ are often called ***iostreams***.

►An **iostream** can be represented by an object of a particular class.

►For example, you've already seen numerous examples of the ***cin*** and ***cout*** stream objects used for input and output.

# Advantages of Streams

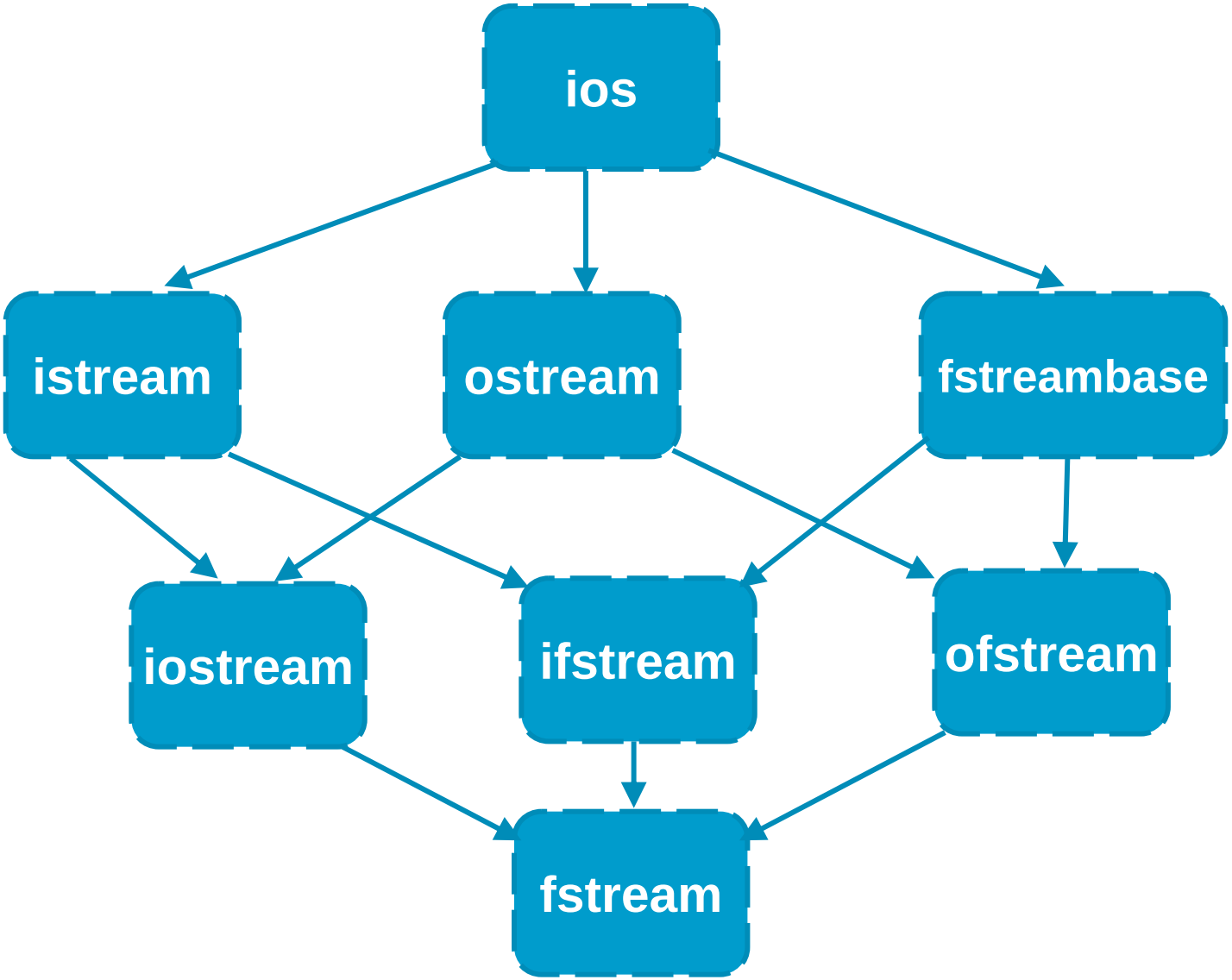


►Old-fashioned C programmers may wonder what advantages there are to using the stream classes for I/O instead of traditional C functions such as printf() and scanf() and—for files—fprintf(), fscanf(), and so on.

►One reason is that the stream classes are less prone to errors. If you've ever used a %d formatting character when you should have used a %f in printf(), you'll appreciate this. There are no such formatting characters in streams, because each object already knows how to display itself. This removes a major source of program bugs.

►Second, you can overload existing operators and functions, such as the insertion (<<) and extraction (>>) operators, to work with classes you create. This makes your classes work in the same way as the built-in types, which again makes programming easier and more error free (not to mention more aesthetically satisfying).

# Stream Class Hierarchy



# Stream Class Hierarchy

- The **ios** class is the base class for the **iostream** hierarchy.
  - –contains many constants and member functions common to input and output operations of all kinds.
  - –also contains a pointer to the **streambuf** class, which contains the actual memory buffer into which data is read or written and the low-level routines for handling this data.

# Stream Class Hierarchy

- The **istream** and **ostream** classes are derived from **ios** and are dedicated to input and output, respectively.
- The **istream** class contains such member functions as get(), getline(), read(), and the extraction (») operators, whereas **ostream** contains put() and write() and the insertion («) operators.
- The **iostream** class is derived from both **istream** and **ostream** by multiple inheritance.
  - used with devices, such as disk files, that may be opened for both input and output at the same time.

# Stream Class Hierarchy

- The **ifstream** class is used for creating input file objects
- The **ofstream** class is used for creating input file objects is used for creating output file objects.
- To create a read/write file the **fstream** class should be used.

# ios

- The ios class is the grand daddy of all the stream classes and contains the majority of the features you need to operate C++ streams.
- The three most important features are
  - the formatting flags,
  - the error-status bits,
  - the file operation mode.

We'll look at formatting flags and error-status bits now.

# Formatting Flags

Formatting flags are a set of enum definitions in ios. They act as on/off switches that specify choices for various aspects of input and output format and operation.

| | |
|---|---|
| skipws | Skip (ignore) whitespace on input. |
| left | Left adjust output. |
| right | Right adjust output. |
| dec | Convert to decimal. |
| oct | Convert to octal. |
| hex | Convert to hexadecimal. |
| showbase | Use base indicator on output (0 for octal, 0x for hex). |
| showpoint | Show decimal point on output. |
| uppercase | Use uppercase X, E, and hex output letters ABCDEF. |
| showpos | Display '+' before positive integers. |
| scientific | Use exponential format on floating-point output [9.1234E2]. |
| fixed | Use fixed format on floating-point output [912.34]. |
| unitbuf | Flush all streams after insertion. |

# Formatting Flags



- There are several ways to set the formatting flags, and different flags can be set in different ways. Because they are members of the ios class, flags must usually be preceded by the name ios and the scope-resolution operator (e.g., ios::skipws). All the flags can be set using the **setf**() and **unsetf**() ios member functions.
- For example,

```
cout.setf(ios::left); //left justify output text
cout >> "This text is left-justified";
cout.unsetf(ios::left); //return to default
                        //(right justified)
```

- Many formatting flags can be set using manipulators, so let's look at them now.

# Manipulators



- Manipulators are formatting instructions inserted directly into a stream.
- You've seen examples before, such as the manipulator ***endl***, which sends a new line to the stream and flushes it:

```
cout << "To each his own." << endl;
```

- There is also used the setiosflags() manipulator:

```
cout << setiosflags(ios::fixed)  // use fixed decimal point
     << setiosflags(ios::showpoint) //always show decimal point
     << var;
```

# No-argument ios Manipulators

| | |
|---|---|
| ws | Turn on whitespace skipping on input |
| dec | Convert to decimal |
| oct | Convert to octal |
| hex | Convert to hexadecimal |
| endl | Insert new line and flush the output stream |
| ends | Insert null character to terminate an output string |
| flush | Flush the output stream |
| lock | Lock file handle |
| unlock | Unlock file handle |

You insert these manipulators directly into the stream. e.g., to output var in hexadecimal format, you can say

```
cout << hex << var;
```

# ios Manipulators with Arguments

- Manipulators that take arguments affect only the next item in the stream.
- For example, if you use ***setw*** to set the width of the field in which one number is displayed, you'll need to use it again for the next number.

| | | |
|---|---|---|
| setw() | field width (int) | Set field width for output |
| setfill() | fill character (int) | Set fill character for output (default is a space) |
| setprecision() | precision (int) | Set precision (number of digits displayed) |
| setiosflags() | formatting flags (long) | Set specified flags |
| resetiosflags() | formatting flags (long) | Clear specified flags |

# Functions

- The ios class contains a number of functions that you can use to set the formatting flags and perform other tasks.
- Most of these functions are shown below:

| | |
|---|---|
| ch=fill() | Return the fill character (fills unused part of field; default is space). |
| fill(ch) | Set the fill character. |
| p=precision() | Get the precision (number of digits displayed for floating point). |
| precision(p) | Set the precision. |
| w=width() | Get the current field width (in characters). |
| width(w) | Set the current field width. |
| setf(flags) | Set specified formatting flags (e.g., ios::left). |
| unsetf(flags) | Unset specified formatting flags. |

►These functions are called for specific stream objects using the normal dot operator. For example, to set the field width to 14, you can say

```
cout.width(14);
```

►Similarly, the following statement sets the fill character to an asterisk (as for check printing):

```
cout.fill('*');
```

►You can use several functions to manipulate the ios formatting flags directly.

For example, to set left justification, use

```
cout.setf(ios::left);
```

To restore right justification, use

```
cout.unsetf(ios::left);
```

# istream

The istream class, which is derived from ios, performs input-specific activities.

***istream*** functions:

| | |
|---|---|
| **>>** | Formatted extraction for all basic (and overloaded) types. |
| **get(ch)** | Extract one character into ch. |
| **get(str)** | Extract characters into array str, until '\0'. |
| **get(str, MAX)** | Extract up to MAX characters into array. |
| **get(str, DELIM)** | Extract characters into array str until specified delimiter (typically '\n'). Leave delimiting char in stream. |

# *istream* Functions

| | |
|---|---|
| get(str, MAX, DELIM) | Extract characters into array str until MAX characters or the DELIM character. Leave delimiting char in stream |
| getline(str, MAX, DELIM) | Extract characters into array str until MAX characters or the DELIM character. Extract delimiting character |
| putback(ch) | Insert last character read back into input stream |
| ignore(MAX, DELIM) | Extract and discard up to MAX characters until (and including) the specified delimiter (typically '\n') |
| peek(ch) | Read one character, leave it in stream |
| count = gcount() | Return number of characters read by a (immediately preceding) call to get(), getline(), or read() |
| read(str, MAX) | For files. Extract up to MAX characters into str until EOF |
| seekg(position) | Sets distance (in bytes) of file pointer from start of file |
| seekg(position, seek_dir) | Sets distance (in bytes) of file pointer from specified place in file: seek_dir can be ios::beg, ios::cur, ios::end |
| position = tellg(pos) | Return position (in bytes) of file pointer from start of file |

# ostream

The ostream class handles output or insertion activities.

**ostream functions:**

| | |
|---|---|
| << | Formatted insertion for all basic (and overloaded) types. |
| put(ch) | Insert character ch into stream. |
| flush() | Flush buffer contents and insert new line. |
| write(str, SIZE) | Insert SIZE characters from array str into file. |
| seekp(position) | Sets distance in bytes of file pointer from start of file. |
| seekp(position, seek_dir) | Set distance in bytes of file pointer from specified place in file. seek_dir can be ios::beg, ios::cur, or ios::end. |
| position = tellp() | Return position of file pointer, in bytes. |

# Ostream and _withassign Classes



►The iostream class, which is derived from both istream and ostream, acts only as a base class from which other classes, specifically iostream_withassign, can be derived.

►It has no functions of its own (except constructors and destructors). Classes derived from iostream can perform both input and output.

►There are three _withassign classes:

istream_withassign, derived from istream
ostream_withassign, derived from ostream
iostream_withassign, derived from iostream

►These _withassign classes are much like those they're derived from except they include overloaded assignment operators so their objects can be copied.

## Predefined Stream Objects

| Objects Name | Class | Used for |
|---|---|---|
| cin | istream_withassign | Keyboard input |
| cout | ostream_withassign | Normal screen output |
| cerr | ostream_withassign | Error output |
| clog | ostream_withassign | Log output |

The cerr object is often used for error messages and program diagnostics. Output sent to cerr is displayed immediately, rather than being buffered, as output sent to cout is. Also, output to cerr cannot be redirected. For these reasons, you have a better chance of seeing a final output message from cerr if your program dies prematurely. Another object, clog, is similar to cerr in that it is not redirected, but its output is buffered, whereas cerr's is not.

## Stream Errors

What happens if a user enters the string "nine" instead of the integer 9, or pushes ENTER without entering anything? What happens if there's a hardware failure? We'll explore such problems in this session. Many of the techniques you'll see here are applicable to file I/O as well.

# Error-Status Bits

The stream error-status bits (error byte) are an ios member that report errors that occurred in an input or output operation.

| | |
|---|---|
| goodbit | No errors (no bits set, value = 0). |
| eofbit | Reached end of file. |
| failbit | Operation failed (user error, premature EOF). |
| badbit | Invalid operation (no associated streambuf). |
| hardfail | Unrecoverable error. |

Various ios functions can be used to read (and even set) these error bits.

| | |
|---|---|
| int = eof(); | Returns true if EOF bit set. |
| int = fail(); | Returns true if fail bit or bad bit or hard-fail bit set. |
| int = bad(); | Returns true if bad bit or hard-fail bit set. |
| int = good(); | Returns true if everything OK; no bits set. |
| clear(int=0); | With no argument, clears all error bits; otherwise sets specified bits, as in clear(ios::failbit). |

inp.cpp

```
#include <iostream>
int main() {
    int i;
    char ok=0;
    while(!ok) {                          // cycle until input OK
        cout << "\nEnter an integer: ";
        cin >> i;
        if( cin.good() ) ok=1;                  // if no errors
        else {
          cin.clear();                    // clear the error bits
          cout << "Incorrect input";
          cin.ignore(20, '\n');                // remove newline
        }
    }
    cout << "integer is " << i; // error-free integer
}
```

# No-Input Input



►Whitespace characters, such as TAB, ENTER , and '\n', are normally ignored (skipped) when inputting numbers. This can have some undesirable side effects. For example, users, prompted to enter a number, may simply press the key without typing any digits. Pressing ENTER causes the cursor to drop down to the next line while the stream continues to wait for the number.

►What's wrong with the cursor dropping to the next line?

–First, inexperienced users, seeing no acknowledgment when they press , may assume the computer is broken.

–Second, pressing repeatedly normally causes the cursor to drop lower and lower until the entire screen begins to scroll upward.

►Thus it's important to be able to tell the input stream *not* to ignore whitespace. This is done by clearing the skipws flag:

```
cout << "\nEnter an integer: ";
cin.unsetf(ios::skipws);        // don't ignore whitespace
cin >> i;
if( cin.good() )
   {
   // no error
   }
  // error
```

Now if the user types without any digits, failbit will be set and an error will be generated. The program can then tell the user what to do or reposition the cursor so the screen does not scroll.

# Disk File I/O with Streams



► Disk files require a different set of classes than files used with the keyboard and screen. These are ifstream for input, fstream for input and output, and ofstream for output. Objects of these classes can be associated with disk files and you can use their member functions to read and write to the files.

► The ifstream, ofstream, and fstream classes are declared in the FSTREAM.H file.

► This file also includes the IOSTREAM.H header file, so there is no need to include it explicitly;

► FSTREAM.H takes care of all stream I/O.

```
#include <fstream.h>                 // for file I/O
int main(){
   char ch = 'x';                    // character
   int j = 77;                       // integer
   double d = 6.02;                  // floating point
   char str1[] = "Kafka";            // strings
   char str2[] = "Proust";           // (no embedded spaces)
   ofstream outfile("fdata.txt");    // create ofstream object
   outfile << ch                     // insert (write) data
           << j   << ' '             // needs space between numbers
           << d
          << str1 << ' '             // needs space between strings
           << str2;
}
```

Here the program defines an object called outfile to be a member of the ofstream class. At the same time, it initializes the object to the file name FDATA.TXT. This initialization sets aside various resources for the file, and accesses or *opens* the file of that name on the disk. If the file doesn't exist, it is created. If it does exist, it is truncated and the new data replaces the old. The outfile object acts much as cout did in previous programs, so the insertion operator (<<) is used to output variables of any basic type to the file. This works because the insertion operator is appropriately overloaded in ostream, from which ofstream is derived.

When the program terminates, the outfile object goes out of scope. This calls its destructor, which closes the file, so you don't need to close the file explicitly.

You must separate numbers (such as 77 and 6.02) with nonnumeric characters. Because numbers are stored as a sequence of characters rather than as a fixed-length field, this is the only way the extraction operator will know, when the data is read back from the file, where one number stops and the next one begins. Second, strings must be separated with whitespace for the same reason. This implies that strings cannot contain embedded blanks. In this example, I use the space character (" ") for both kinds of delimiters. Characters need no delimiters, because they have a fixed length.

# Reading Data

Any program can read the file generated by previous program by using an ifstream object that is initialized to the name of the file. The file is automatically opened when the object is created. The program can then read from it using the extraction (>>) operator.

```
// reads formatted output from a file, using >>
#include <fstream.h>
const int MAX = 80;
int main(){
   char ch;                          // empty variables
   int j;
   double d;
   char str1[MAX];
   char str2[MAX];
   ifstream infile("fdata.txt");      // create ifstream object
   infile >> ch >> j >> d >> str1 >> str2;  // extract data from it
   cout << ch << endl                 // display the data
        << j << endl
        << d << endl
        << str1 << endl
        << str2 << endl;
}
```

# Detecting End-OF-File



► Objects derived from ios contain error-status bits that can be checked to determine the results of operations. When you read a file little by little, you will eventually encounter an end-of-file condition. The EOF is a signal sent to the program from the hardware when there is no more data to read. The following construction can be used to check for this:

```
while( !infile.eof() )   // until eof encountered
```

► However, checking specifically for an eofbit means that I won't detect the other error bits, such as the failbit and badbit, which may also occur, although more rarely. To do this, I could change the loop condition:

```
while( infile.good() )   // until any error encountered
```

► But even more simply, I can test the stream directly

```
while( infile )          // until any error encountered
```

Any stream object, such as infile, has a value that can be tested for the usual error conditions, including EOF. If any such condition is true, the object returns a zero value.

► If everything is going well, the object returns a nonzero value. This value is actually a pointer, but the "address" returned has no significance except to be tested for a zero or nonzero value.

# Binary I/O

You can write a few numbers to disk using formatted I/O, but if you're storing a large amount of numerical data, it's more efficient to use binary I/O in which numbers are stored as they are in the computer's RAM memory rather than as strings of characters. In binary I/O an integer is always stored in 2 bytes, whereas its text version might be 12345, requiring 5 bytes. Similarly, a float is always stored in 4 bytes, whereas its formatted version might be 6.02314e13, requiring 10 bytes.

The next example shows how an array of integers is written to disk and then read back into memory using binary format. I use two new functions: write(), a member of ofstream, and read(), a member of ifstream. These functions think about data in terms of bytes (type char). They don't care how the data is formatted, they simply transfer a buffer full of bytes from and to a disk file. The parameters to write() and read() are the address of the data buffer and its length. The address must be cast to type char, and the length is the length in bytes (characters), *not* the number of data items in the buffer.

# Example

```
#include <fstream.h>          // for file streams
const int MAX = 100;          // number of ints
int buff[MAX];                // buffer for integers
int main() {
    int j;
    for(j=0; j<MAX; j++)   // fill buffer with data
    buff[j] = j; // (0, 1, 2, ...)
    ofstream os("edata.dat", ios::binary);     // create output stream
    os.write( (char*)buff, MAX*sizeof(int) );  // write to it
    os.close(); // must close it
    for(j=0; j<MAX; j++)      // erase buffer
       buff[j] = 0;
    ifstream is("edata.dat", ios::binary);        // create input stream
    is.read( (char*)buff, MAX*sizeof(int) );   // read from it
    for(j=0; j<MAX; j++) // check data
       if( buff[j] != j )  std::cerr << "\nData is incorrect";
          else std::cout << "\nData is correct";
}
```

# Writing an Object to Disk

When writing an object, you generally want to use binary mode. This writes the same bit configuration to disk that was stored in memory and ensures that numerical data contained in objects is handled properly.

```
#include <fstream.h>            // for file streams
class person {                  // class of persons
  protected:
    char name[40];              // person's name
    int age;                    // person's age
  public:
    void getData(void) {        // get person's data
        std::cout << "Enter name: "; cin >> name;
        std::cout << "Enter age: "; cin >> age;
     }
};
```

```
int main() {
   person pers;                    // create a person
   pers.getData();                 // get data for person
   ofstream outfile("PERSON.DAT", ios::binary);
   outfile.write( (char*)&pers, sizeof(pers) );  // write to it
 }
```

## Reading an Object from Disk

```
#include <fstream.h>               // for file streams
class person {                     // class of persons
   protected:
      char name[40];               // person's name
      int age;                     // person's age
   public:
      void showData(void) {        // display person's data
         std::cout << "\n   Name: " << name;
         std::cout << "\n   Age: " << age;
      }
   };
```

```
int main() {
   person pers;                    // create person variable
   ifstream infile("PERSON.DAT", ios::binary); // create stream
   infile.read( (char*)&pers, sizeof(pers) );  // read stream
   pers.showData();                            // display person
}
```

To work correctly, programs that read and write objects to files, must be working on the same class of objects. Objects of class person in these programs are exactly 42 bytes long, with the first 40 occupied by a string representing the person's name and the last 2 containing an int representing the person's age.

Notice, however, that although the person classes in both programs have the same data, they may have different member functions. The first includes the single function getData(), whereas the second has only showData(). It doesn't matter what member functions you use, because members functions are not written to disk along with the object's data. The data must have the same format, but inconsistencies in the member functions have no effect. This is true only in simple classes that don't use virtual functions.

# I/O with Multiple Objects

```
#include <fstream.h>              // for file streams
class person {                    // class of persons
   protected:
      char name[40];              // person's name
      int age;                    // person's age
   public:
      void getData() {            // get person's data
         cout << "\n   Enter name: "; cin >> name;
         cout << "   Enter age: "; cin >> age;
      }
      void showData() {           // display person's data
         cout << "\n   Name: " << name;
         cout << "\n   Age: " << age;
      }
};
```

objfile.cpp

```
int main(){
   char ch;
   person pers;                    // create person object
   fstream file;                   // create input/output file
   file.open("PERSON.DAT", ios::out | ios::binary ); // open for append
   do{                             // data from user to file
      cout << "\nEnter person's data:";
      pers.getData();              // get one person's data
      file.write( (char*)&pers, sizeof(pers) ); // write to file
      cout << "Enter another person (y/n)? ";
      cin >> ch;
   } while(ch=='y');               // quit on 'n'
   file.close();                   // reset to start of file
   file.open("PERSON.DAT", ios::in | ios::binary );
    file.read( (char*)&pers, sizeof(pers) ); // read first person
   while( !file.eof() )            // quit on EOF
    {
      cout << "\nPerson:";         // display person
      pers.showData();
      file.read( (char*)&pers, sizeof(pers) );  // read another
    }                              // person
}
```

# Reacting to Errors

The next program shows how errors are most conveniently handled. All disk operations are checked after they are performed. If an error has occurred, a message is printed and the program terminates. We will use the technique, discussed earlier, of checking the return value from the object itself to determine its error status. The program opens an output stream object, writes an entire array of integers to it with a single call to write(), and closes the object. Then it opens an input stream object and reads the array of integers with a call to read().



```
#include <fstream> // for file streams
#include <process> // for exit()
const int MAX = 1000;
int buff[MAX];
int main(){
  for(int j=0; j<MAX; j++) buff[ j ] = j; // fill buffer with data
  ofstream os; // create output stream
  os.open("edata.dat", ios::trunc | ios::binary); // open it
  if(!os) { cerr << "\nCould not open output file"; exit(1); }
  std::cout << "\nWriting..."; // write buffer to it
  os.write( (char*)buff, MAX*sizeof(int) );
  if(!os) { cerr << "\nCould not write to file"; exit(1); }
  os.close(); // must close it
}
```

```
  for(j=0; j<MAX; j++) buff[ j ] = 0; // clear buffer
  ifstream is; // create input stream
  is.open("edata.dat", ios::binary);
  if(!is) { std::cerr << "\nCould not open input file"; exit(1); }
  std::cout << "\nReading...";
  is.read( (char*)buff, MAX*sizeof(int) ); // read file
  if(!is) { std::cerr << "\nCould not read from file"; exit(1); }
  for(j=0; j<MAX; j++) // check data
    if( buff[j] != j ) { std::cerr << "\nData is incorrect"; exit(1); }
  std::cout << "\nData is correct";
}
```

## Analyzing Errors

In the previous example, we determined whether an error occurred in an I/O operation by examining the return value of the entire stream object.

```
if(!is)
   // error occurred
```

However, it's also possible, using the ios error-status bits, to find out more specific information about a file I/O error.

```
#include <fstream.h>          // for file functions
int main(){
   ifstream file;
   file.open("GROUP.DAT", ios::nocreate);
   if( !file )
      cout << endl <<"Can't open GROUP.DAT";
   else
      cout << endl << "File opened successfully.";
   cout << endl << "file = " << file;
   cout << endl << "Error state = " << file.rdstate();
   cout << endl << "good() = " << file.good();
   cout << endl << "eof() = " << file.eof();
   cout << endl << "fail() = " << file.fail();
   cout << endl << "bad() = " << file.bad();
   file.close();
}
```

This program first checks the value of the object file. If its value is zero, the file probably could not be opened because it didn't exist. Here's the output of the program when that's the case:

```
Can't open GROUP.DAT
file = 0x1c730000
Error state = 4
good() = 0
eof() = 0
fail() = 4
bad() = 4
```

The error state returned by rdstate() is 4. This is the bit that indicates the file doesn't exist; it's set to 1. The other bits are all set to 0. The good() function returns 1 (true) only when no bits are set, so it returns 0 (false). I'm not at EOF, so eof() returns 0. The fail() and bad() functions return nonzero because an error occurred.

In a serious program, some or all of these functions should be used after every I/O operation to ensure that things have gone as expected.

# File Pointers

Each file object has associated with it two integer values called the *get pointer* and the *put pointer*. These are also called the *current get position* and the *current put position*, or—if it's clear which one is meant—simply the *current position*. These values specify the byte number in the file where writing or reading will take place

There are times when you must take control of the file pointers yourself so that you can read from or write to an arbitrary location in the file. The seekg() and tellg() functions allow you to set and examine the get pointer, and the seekp() and tellp() functions perform the same actions on the put pointer.

```
// seeks particular person in file
#include <fstream.h> // for file streams
class person { // class of persons
  protected:
    char name[40]; // person's name
    int age; // person's age
  public:
    void showData() { // display person's data
      cout << "\n Name: " << name; cout << "\n Age: " << age;
    }
};
```

```
int main(){
    person pers; // create person object
    ifstream infile; // create input file
    infile.open("PERSON.DAT", ios::binary); // open file
    infile.seekg(0, ios::end); // go to 0 bytes from end
    int endposition = infile.tellg(); // find where we are
    int n = endposition / sizeof(person); // number of persons
    cout << endl << "There are " << n << " persons in file";
    cout << endl << "Enter person number: "; cin >> n;
    int position = (n-1) * sizeof(person); // number times size
    infile.seekg(position); // bytes from begin
    infile.read( (char*)&pers, sizeof(pers) ); // read one person
    pers.showData(); // display the person
}
```

Here's the output from the program, assuming that the PERSON.DAT file contains 3 persons:

```
There are 3 persons in file
Enter person number: 2
   Name: Rainier
   Age: 21
```

# File I/O Using Member Functions

So far, we've let the main() function handle the details of file I/O. This is nice for demonstrations, but in real object-oriented programs, it's natural to include file I/O operations as member functions of the class.



In the next example, we will add member functions, diskOut() and diskIn() to the person class. These functions allow a person object to write itself to disk and read itself back in.

Simplifying assumptions: First, all objects of the class will be stored in the same file, called PERSON.DAT. Second, new objects are always appended to the end of the file. An argument to the diskIn() function allows me to read the data for any person in the file. To prevent attempts to read data beyond the end of the file, I include a static member function, diskCount(), that returns the number of persons stored in the file.

```
#include <fstream.h> // for file streams
class person {// class of persons
  protected:
    char name[40]; // person's name
    int age; // person's age
  public:
    void getData(){ // get person's data
    cout << "\n Enter name: "; cin >> name; cout << " Enter age: "; cin >> age;}
  void showData(){ // display person's data
        cout << "\n Name: " << name; cout << "\n Age: " << age; }
  void diskIn(int ); // read from file
  void diskOut(); // write to file
  static int diskCount(); // return number of  persons in file
};

void person::diskIn(int pn){ // read person number pn  from file
  ifstream infile; // make stream
  infile.open("PERSON.DAT", ios::binary); // open it
  infile.seekg( pn*sizeof(person) ); // move file ptr
  infile.read( (char*)this, sizeof(*this) ); // read one person
}
```

```
void person::diskOut()              // write person to end of file
{
  ofstream outfile;                 // make stream
  outfile.open("PERSON.DAT", ios::app | ios::binary);  // open it
  outfile.write( (char*)this, sizeof(*this) ); // write to it
}

int person::diskCount()             // return number of persons  in file
{
  ifstream infile;
  infile.open("PERSON.DAT", ios::binary);
  infile.seekg(0, ios::end);        // go to 0 bytes from end
  return infile.tellg() / sizeof(person); // calculate number of persons
}
```

```
int main(void){
   person p;                          // make an empty person
   char ch;
   do{                                // save persons to disk
      cout << "\nEnter data for person:";
      p.getData();                    // get data
      p.diskOut();                    // write to disk
      cout << "Do another (y/n)? ";
      cin >> ch;
   }while(ch=='y');                   // until user enters 'n'
   int n = person::diskCount();       // how many persons in file?
   cout << "\nThere are " << n << " persons in file";
   for(int j=0; j<n; j++) {  // for each one,
      cout << "\nPerson #" << (j+1);
      p.diskIn(j);                    // read person from disk
      p.showData();                   // display person
   }
}
```

# Overloading the « and » Operators

In this session I'll show how to overload the extraction and insertion operators. This is a powerful feature of C++. It lets you treat I/O for user-defined data types in the same way as for basic types such as int and double. For example, if you have an object of class TComplex called c1, you can display it with the statement

cout << c1; just as if it were a basic data type.

You can overload the extraction and insertion operators so they work with the display and keyboard (cout and cin). With a little more care, you can also overload them so they work with disk files as well.

```
#include<iostream>
class TComplex {
    float   real,img;
    friend std::istream& operator >>(std::istream&, TComplex&);
    friend std::ostream& operator <<(std::ostream&, const TComplex&);
  public:
    TComplex(float rl=0,float ig=0){real=rl;img=ig;}
    TComplex operator+(const TComplex&);
};
```

```
istream& operator >>(istream& stream, TComplex& z){  // Overloading >>
   cout << "Enter real part:";
   stream >> z.real;
   cout << "Enter imaginer part:";
   stream >> z.img;
   return stream;
}
ostream& operator <<(ostream& stream, const TComplex & z){
   stream << "( " << z.real << " , " << z.img << " ) \n";
   return stream;
}
TComplex TComplex::operator+(const TComplex & z){    // Operator +
  return TComplex (real+z.real , img+z.img);
}
int main(){
  TComplex z1,z2,z3;
  std::cin >> z1;
  std::cin >> z2;
  z3=z1+z2;
  std::cout << " Result=" << z3;
}
```

inout.cpp

# Overloading for Files

The next example shows how the << and >> operators can be overloaded so they work with both file I/O and cout and cin.

```
#include<fstream>
class TComplex {
    float real,img;
    friend istream& operator >>(istream&, TComplex&);
    friend ostream& operator <<(ostream&, const TComplex&);
 public:
    TComplex(float rl=0,float ig=0){real=rl;img=ig;}
};
istream& operator >>(istream& stream, TComplex &z){
  char dummy;
  stream >> dummy >> z.real;
  stream >> dummy >> z.img >> dummy;
  return stream;
}
ostream& operator <<(ostream& stream, const TComplex & z){
                stream << "(" << z.real << " , " << z.img << ") \n";
                return stream;
};
```

```
int main(){
  char ch;
  TComplex z1;
  ofstream ofile;                  // create and open
  ofile.open("complex.dat");  // output stream
  do {  std::cout << "\nEnter Complex Number:(real,img)";
        cin >> z1;                  // get complex number from user
        ofile << z1;                // write it to output str
        std::cout << "Do another (y/n)? ";   std::cin >> ch;
   } while(ch != 'n');
   ofile.close();                  // close output stream
   std::ifstream ifile;            // create and open
   ifile.open("complex.dat");  // input stream
   std:.cout << "\nContents of disk file is:";
   while(!ifile.eof()){
     ifile >> z1;  // read complex number from stream
     if (ifile)
       std::cout << "\nComplex Number = " << z1;   // display complex number
   }
}
```

fileio.cpp

# Overloading for Binary I/O

So far, you've seen examples of overloading operator<<() and operator>>() for formatted I/O. They also can be overloaded to perform binary I/O. This may be a more efficient way to store information, especially if your object contains much numerical data.

```
#include <fstream.h> // for file streams
class person {// class of persons
  protected:
      char name[40]; // person's name
      int age; // person's age
  public:
   void getData(){ // get data from keyboard
           cout << "\n Enter name: "; cin.getline(name, 40);
           cout << " Enter age: "; cin >> age;
   }
   void putData(){ // display data on screen
           cout << "\n Name = " << name; cout << "\n Age = " << age;
   }
   friend istream& operator >> (istream& s, person& d);
   friend ostream& operator << (ostream& s, person& d);
```

```
        void persin(istream& s){
          s.read( (char*)this, sizeof(*this) );
        }
        void persout(ostream& s) // write our data to file
        {
          s.write( (char*)this, sizeof(*this) );
        }
};   // end of class definiton
istream& operator >> (istream& s, person& d) {
        d.persin(s);
        return s;
}
ostream& operator << (ostream& s, person& d){
        d.persout(s);
        return s;
}
```

```
int main(){
        person pers1, pers2, pers3, pers4;
        cout << "\nPerson 1";
        pers1.getData();  // get data for pers1
        cout << "\nPerson 2";
        pers2.getData(); // get data for pers2
        outfile("PERSON.DAT", ios::binary);
        outfile << pers1 << pers2; // write to file
        outfile.close();
        ifstream infile("PERSON.DAT", ios::binary);
        infile >> pers3 >> pers4; // read from file into
        cout << "\nPerson 3";     // pers3 and pers4
        pers3.putData();  // display new objects
        cout << "\nPerson 4";
        pers4.putData();
}
```